MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

EFFICIENT DEEP NEURAL NETWORK ARCHITECTURES FOR CLASSIFICATION OF EMOTIONAL REACTIONS TO ARTISTIC PAINTINGS

Autor: Arturo Garrido Contreras

Director: Raquel Caro Carretero

Madrid

Agosto de 2019

AUTHORIZATION FOR DIGITALIZATION, STORAGE AND DISSEMINATION IN THE NETWORK OF END-OF-DEGREE PROJECTS, MASTER PROJECTS, DISSERTATIONS OR BACHILLERATO REPORTS

1. Declaration of authorship and accreditation thereof.

The author Mr. /Ms. Arturo Garrido Contreras

HEREBY DECLARES that he/she owns the intellectual property rights regarding the piece of work: Efficient Deep Neural Network Architectures for Classification of Emotional Reactions to Artistic Paintings

that this is an original piece of work, and that he/she holds the status of author, in the sense granted by the Intellectual Property Law.

2. Subject matter and purpose of this assignment.

With the aim of disseminating the aforementioned piece of work as widely as possible using the University's Institutional Repository the author hereby **GRANTS** Comillas Pontifical University, on a royalty-free and non-exclusive basis, for the maximum legal term and with universal scope, the digitization, archiving, reproduction, distribution and public communication rights, including the right to make it electronically available, as described in the Intellectual Property Law. Transformation rights are assigned solely for the purposes described in a) of the following section.

3. Transfer and access terms

Without prejudice to the ownership of the work, which remains with its author, the transfer of rights covered by this license enables:

a) Transform it in order to adapt it to any technology suitable for sharing it online, as well as including metadata to register the piece of work and include "watermarks" or any other security or protection system.

b) Reproduce it in any digital medium in order to be included on an electronic database, including the right to reproduce and store the work on servers for the purposes of guaranteeing its security, maintaining it and preserving its format.

c) Communicate it, by default, by means of an institutional open archive, which has open and cost-free online access.

d) Any other way of access (restricted, embargoed, closed) shall be explicitly requested and requires that good cause be demonstrated.

e) Assign these pieces of work a Creative Commons license by default.

f) Assign these pieces of work a HANDLE (*persistent* URL). by default.

4. Copyright.

The author, as the owner of a piece of work, has the right to:

a) Have his/her name clearly identified by the University as the author

b) Communicate and publish the work in the version assigned and in other subsequent versions using any medium.

c) Request that the work be withdrawn from the repository for just cause.

d) Receive reliable communication of any claims third parties may make in relation to the work and, in particular, any claims relating to its intellectual property rights.

5. Duties of the author.

The author agrees to:

a) Guarantee that the commitment undertaken by means of this official document does not infringe any third party rights, regardless of whether they relate to industrial or intellectual property or any other type.

b) Guarantee that the content of the work does not infringe any third party honor, privacy or image rights.
c) Take responsibility for all claims and liability, including compensation for any damages, which may be brought against the University by third parties who believe that their rights and interests have been infringed by the assignment.
d) Take responsibility in the event that the institutions are found guilty of a rights infringement regarding the work subject to assignment.

6. Institutional Repository purposes and functioning.

The work shall be made available to the users so that they may use it in a fair and respectful way with regards to the copyright, according to the allowances given in the relevant legislation, and for study or research purposes, or any other legal use. With this aim in mind, the University undertakes the following duties and reserves the following powers:

a) The University shall inform the archive users of the permitted uses; however, it shall not guarantee or take any responsibility for any other subsequent ways the work may be used by users, which are non-compliant with the legislation in force. Any subsequent use, beyond private copying, shall require the source to be cited and authorship to be recognized, as well as the guarantee not to use it to gain commercial profit or carry out any derivative works.
b) The University shall not review the content of the works, which shall at all times fall under the exclusive responsibility of the author and it shall not be obligated to take part in lawsuits on behalf of the author in the event of any infringement of intellectual property rights deriving from storing and archiving the works. The author hereby waives any claim against the University due to any way the users may use the works that is not in keeping with the legislation in force.
c) The University shall adopt the necessary measures to safeguard the work in the future.
d) The University reserves the right to withdraw the work, after notifying the author, in sufficiently justified cases, or in the event of third party claims.

Madrid, on 23 of August, 2019

HEREBY ACCEPTS

Signed.....

Reasons for requesting the restricted, closed or embargoed access to the work in the Institution's Repository

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Efficient Deep Neural Network Architectures for Classification of Emotional Reactions to Artistic Paintings

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2018/2019 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Arturo Garrido Contreras Fecha: 23 / 08 / 19

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Raquel Caro Carretero Fecha: 23 / 08 / 2019

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

EFFICIENT DEEP NEURAL NETWORK ARCHITECTURES FOR CLASSIFICATION OF EMOTIONAL REACTIONS TO ARTISTIC PAINTINGS

Autor: Arturo Garrido Contreras

Director: Raquel Caro Carretero

Madrid

Agosto de 2019

ARQUITECTURAS EFICIENTES DE REDES NEURONALES PROFUNDAS PARA LA CLASIFICACIÓN DE REACCIONES EMOCIONALES A CUADROS ARTÍSTICOS

Autor: Garrido Contreras, Arturo

Director: Caro Carretero, Raquel

Entidad Colaboradora: Stanford University. ChezMana.

RESUMEN DEL PROYECTO

INTRODUCCIÓN E IMPORTANCIA DEL PROYECTO

Este proyecto ha sido desarrollado como parte de la asignatura de Deep Learning (CS230) en la Universidad de Stanford. La universidad ha suministrado créditos de AWS para ejecutar los modelos debido a un acuerdo existente entre las dos entidades. El proyecto fue propuesto por ChezMana **[1]**, una startup con sede en Palo Alto, que también proporcionó la base de datos de pinturas artísticas digitalizadas.

Clasificar obras de arte en función de la emoción transmitida ha sido un desafío incluso con las técnicas de aprendizaje profundo más avanzadas. Las dificultades provienen del carácter subjetivo de las emociones y el hecho de que tanto características globales (color, textura, etc.) como locales (una persona sonriendo) de una imagen pueden evocar emociones. El estudio de la emoción con aprendizaje profundo ha sido menos explorado que otras aplicaciones de "Computer Vision" debido a los desafíos mencionados. Sin embargo, el progreso en este campo podría tener muchas aplicaciones en educación, interacción hombre-máquina, y medios y entretenimiento **[2] [3].**

Este documento se centra en dos áreas. El objetivo principal consiste en construir un clasificador que prediga las emociones evocadas por las pinturas artísticas. Se exploran diferentes enfoques y arquitecturas para evaluar y comparar su desempeño. Debido a la cantidad limitada de datos etiquetados de manera fiable, se utilizarán Redes Neuronales Generativas Adversarias (Generative Adversarial Networks, GANs) para generar nuevos cuadros. Los modelos generativos son mucho más complejos y difíciles de entrenar que los modelos de clasificación, pero se utilizarán arquitecturas existentes en la literatura para tratar de obtener resultados razonables e interesantes. El objetivo de generar nuevas obras de arte es utilizarlas para entrenar el clasificador de emociones, para mejorar su rendimiento. Sin embargo, es un desafío interesante en sí mismo.

ESTADO DEL ARTE

- Clasificación

Las Redes Neuronales Convolucionales (CNNs de aquí en adelante) representan la arquitectura más popular en problemas de reconocimiento de imagen. La principal ventaja que proporcionan las CNNs es una gran reducción en el número de parámetros necesarios en la red en comparación con las redes neuronales clásicas. Funcionan bien para

problemas que tienen características que no dependen de la posición, como el reconocimiento de objetos en las imágenes.

Las CNNs analizan diferentes regiones locales, en lugar de la imagen completa a la vez, con el uso de un filtro o núcleo, como se muestra en la **Figura 1**. Hay una serie de hiperparámetros que definen el funcionamiento de una convolución, especialmente el "padding" y "stride" **[4].**

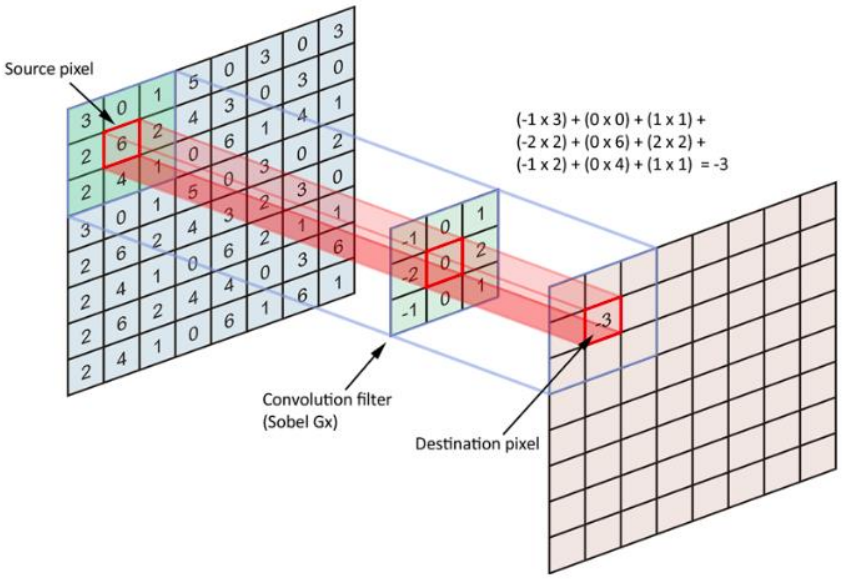


Figura 1. Convolución. Fuente: ***[5]***

Hay algunas arquitecturas de CNNs que funcionan bien en una variedad de tareas de Computer Vision. Algunos ejemplos son:

AlexNet

La red llamada "AlexNet" fue presentada por Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton **[6]**. Esta red neuronal, con 60M de parámetros y 650,000 neuronas, logró los mejores resultados en el concurso ImageNet LSVRC-2010 2010, que consiste en clasificar 1.2M de imágenes de alta resolución en 1000 categorías diferentes.

La arquitectura consta de cinco capas convolucionales, algunas de las cuales son seguidas por capas "max-pooling" **[4]** y tres "fully connected layers" **[4]**, con una capa "softmax" final.

VGG-16

Presentada por Karen Simonyan y Andrew Zisserman **[7]**, con la participación de Google DeepMind. La arquitectura consta de trece capas convolucionales, cinco capas capas "max-pooling", dos "fully connected layers" y una "softmax" final. La innovación con respecto a los modelos anteriores consiste en una arquitectura mucho más profunda, que es posible debido al uso de filtros de convolución muy pequeños (3x3), resultando en alrededor de 133M parámetros.

- Redes Neuronales Generativas Adversarias (GANs)

Hasta hace poco, los éxitos más importantes en el aprendizaje profundo han involucrado modelos discriminatorios, que consisten en mapear un input de alta dimensión (imagen, texto, audio, etc.) a una clase. Los modelos generativos profundos han tenido menos impacto, debido a la dificultad de aplicar los modelos y métodos existentes a tareas generativas.

El principio básico de las GANs consiste en tener dos redes que se entrenen compitiendo entre sí. Una de ellas es el **Generador**, que será la que crea muestras de datos que simulan provenir de la misma distribución que los datos de entrenamiento. Por otro lado, el **Discriminador**, recibe como entrada una muestra y la clasifica como real (procedente de la distribución de probabilidad real) o falsa (procedente de una distribución diferente) **[8].** En cada etapa de entrenamiento, tanto el generador como el discriminador actualizan sus parámetros para reducir sus respectivas funciones de coste. Estas funciones de coste se muestran en la **Ecuación 1** y la **Ecuación 2**.

$$J^{(D)} = \underbrace{-\frac{1}{m_{real}}\sum_{i=1}^{m_{real}} y_{real}^{(i)}.\log(D(x^{(i)}))}_{\text{cross-entropy 1: "D should correctly label real data as 1"}} \underbrace{-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}(1-y_{gen}^{(i)}).\log(1-D(G(z^{(i)})))}_{\text{cross-entropy 2: "D should correctly label generated data as 0"}}$$

Ecuación 1. Función de coste del discriminador. Fuente: ***[9]***

$$J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D(G(z^{(i)})))$$

Ecuación 2. Función de coste del generador. Fuente: ***[9]***

Estos son solo dos ejemplos de posibles funciones de coste a utilizar. La literatura actual ofrece muchas más alternativas tanto para el error del generador, como para el del discriminador **[10].**

Hay algunas arquitecturas de GANs que han demostrado funcionar bien en problemas muy variados. La más utilizada es DCGAN **[11]**, cuyo generador pasa por 4 capas de "deconvolución", como se muestra en la **Figura 2**.

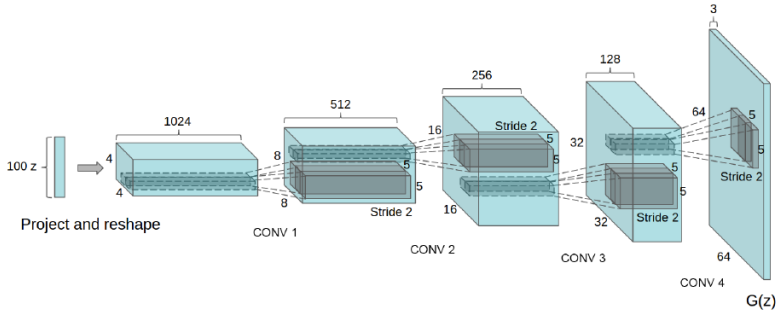


Figura 2. Arquitectura del generador de DCGAN. Fuente: ***[11]***

DATOS

Este trabajo utiliza un conjunto de datos de 40,000 imágenes artísticas, proporcionadas por el patrocinador del proyecto **[1]**, de las cuales solo 3,000 están etiquetadas en función de la emoción transmitida. Había 16 clases de emociones distintas, algunas de ellas muy difíciles de distinguir entre sí, como se puede observar en la **Figura 3**.

Figura 3. Optimismo (izquierda) y Alegría (derecho) son un ejemplo de la similitud de las emociones asignadas. Fuente: elaboración propia.

Por lo tanto, se tomó la decisión de agrupar las clases en 4 categorías principales:

• Categoría 0: Tristeza, Miedo, Reuplsión, Enfado.

• Categoría 1: Neutro, NA.

• Categoría 2: Lujuria, Envidia, Sorpresa.

• Categoría 3: Optimismo, Alegría, Amor.

SOLUCIÓN

Una práctica común al afrontar un problema de aprendizaje profundo es empezar con un modelo básico para obtener los resultados de referencia **[12]**. A continuación, trabajar iterativamente, creando modelos cada vez más complejos y ajustando los hiperparámetros en el proceso. Este es el enfoque seguido tanto para la tarea de clasificación como para los modelos de GANs.

- Clasificación
    - ShallNet

Este modelo de referencia consiste en una arquitectura CNN de 3 capas construida desde cero. En este punto del estudio, solo un pequeño porcentaje de los datos estaba etiquetado en función de las emociones. Se tomó la decisión de entrenar el modelo solo con cuadros de tipo "retrato", para facilitar la tarea de esta arquitectura simple. Los resultados del modelo (porcentaje de predicciones correctas) se muestran en la **Tabla 1**.

    - VGG16 + TL versión 1

Entrenar una arquitectura convolucional desde cero con tan pocos puntos de datos puede ser un desafío, tal y como lo demuestran los resultados del modelo anterior. Por lo tanto, se utilizó Transfer Learning **[13]** en la siguiente arquitectura, una técnica consistente en

utilizar una red neuronal que ya ha sido entrenada con otros datos como punto de partida, y reentrenarla con los datos de interés.

Se utilizó la arquitectura VGG16 **[14]**, entrenada previamente con el conjunto de datos imageNet **[6]** de 15 millones de imágenes pertenecientes a aproximadamente 22,000 categorías. El potencial de este enfoque era incierto, ya que el modelo había sido entrenado previamente con imágenes reales, no en pinturas artísticas.

La última capa de "max-pooling" de la red VGG16 se reemplazó con una capa de "global average pooling", y se utilizó una sola "fully connected layer" en lugar de dos, con una capa softmax al final. Solo estas dos últimas capas fueron entrenadas, y el resto de los parámetros del modelo VGG16 permanecieron "congelados". Este modelo recibe el nombre de "VGG16 + TL versión 1". Los resultados se muestran en la **Tabla 1.**

A continuación, se utilizaron Mapas de Activación de Clase (CAM) como parte del proceso de análisis de errores, para informar los modelos futuros. Este análisis, que se muestra en la **Figura 4**, revela que aunque algunas emociones son activadas por características locales (por ejemplo, por una cara), el modelo también está reaccionando frente a regiones del cuadro que no deberían evocar una emoción (ruido).

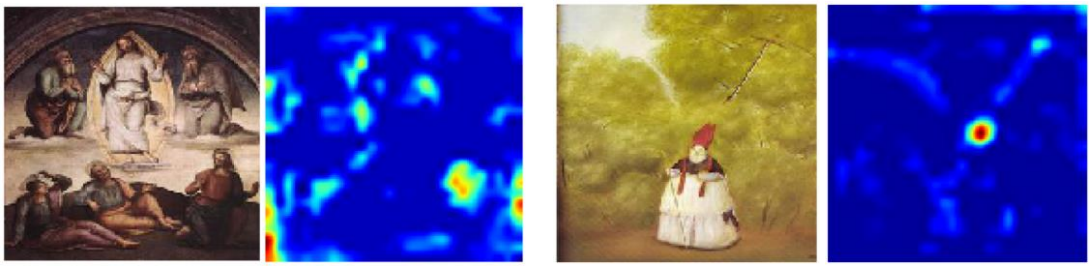


Figura 4. Mapas de Activación de Clase de distintos cuadros. Fuente: elaboración propia.

- VGG16 + TL versión 2

Tras el análisis de los CAMs, se decidió filtrar los datos nuevamente y utilizar únicamente "retratos", debido al hecho de que los retratos tienen características locales más consistentes, por lo que el modelo no debería mostrar tanto ruido como antes. Se construyó el mismo modelo VGG16 para evaluar su desempeño con este nuevo conjunto de datos. Este modelo se denomina "VGG16 + TL versión 2". Los resultados se muestran en la **Tabla 1**.

- CV + FC

El análisis de los CAMs mostrado anteriormente evidencia que las características locales (una cara, un objeto, etc.) no son suficientes para definir la emoción transmitida por un cuadro. Por lo tanto, se para el siguiente modelo, se adoptó un enfoque fundamentalmente diferente, impulsado por la investigación realizada sobre el estudio de la emoción y las imágenes.

Se definieron 13 características extraídas "a mano", basadas en el color (6 características), el brillo (1 característica) y la textura (6 características) de la imagen, lo que permite que el modelo asocie características globales a una emoción particular. Las características de textura se basan en el "Perlin noise" **[15]** que forma la escala Hard-Soft (HS) presentada

por **[16]**. Para medir el brillo de la imagen se utiliza un método estándar, calculando la raíz cuadrada de una combinación lineal del error cuadrático medio de cada canal de color **[16]**. Para las características basadas en el color, se utiliza el método de **[17]**: primero se extraen los 5 centroides de color superiores de la imagen, y a continuación se comparan con los mapas de color estándar que se crean para cada clase de emoción. Estas características se muestran en la **Figura 5**.

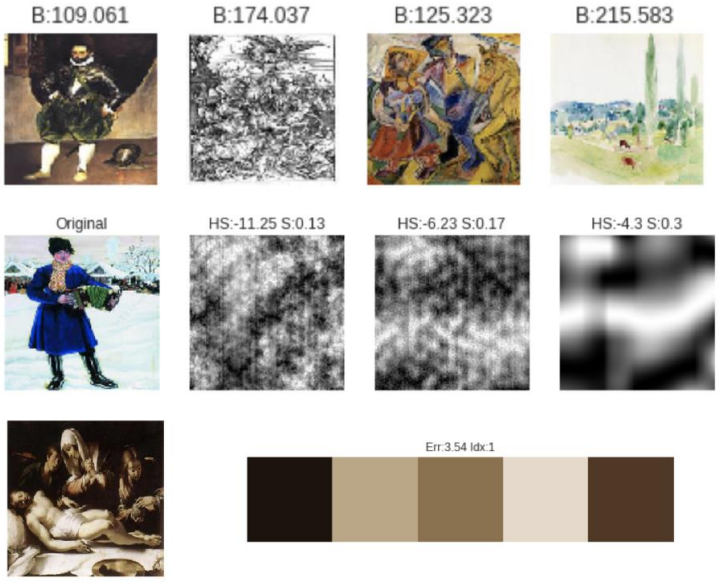


Figura 5. Características extraídas "a mano". De arriba a abajo: brillo percibido donde B es el valor de brillo; Similitud estructural de la textura, donde HS es el valor Hard-Soft y S es la similitud estructural de la imagen de entrada al modelo con nuestros mapas de textura; Distancia del mapa de color, donde se obtienen los 5 colores primarios de la imagen de entrada al modelo y se calculan las distancias a nuestros mapas de colores de referencia. Fuente: elaboración propia.

Una vez que se definen las características, se construyó un modelo simple. Este modelo, denominado "CV + FC", consiste en una capa de 4 neuronas seguida de una capa softmax. Los resultados, más exitosos de lo esperado, también se muestran en la **Tabla 1**.

- Redes Neuronales Generativas Adversarias (GANs)

Para la tarea generativa, el conjunto de datos se limitó a "retratos", con alrededor de 7k muestras en total (ya que no necesitan estar etiquetados). Al trabajar solo con retratos, el generador tiene más opciones de ser capaz de crear imágenes realistas.

  - BasicGAN

Este modelo de referencia posee la siguiente arquitectura:

• Generador: red poco profunda con tres capas "fully-connected". Recibe un código aleatorio de 100 dimensiones y genera una matriz de dimensión 224 * 224 * 3.
• Discriminador: red poco profunda, con dos capas "fully-connected", más una capa con una salida sigmoide (ya que se trata de un problema de clasificación binaria).

Después de entrenar durante 175 épocas, se crea una gráfica de la evolución del error del generador y el discriminador. En un juego generativo-adversario, el error de cada miembro debería ser muy irregular, ya que ambos mejoran constantemente y hacen que

la tarea del otro sea más complicada. Sin embargo, como se ve en la **Figura 6**, en nuestro modelo el discriminador converge rápidamente mientras que el error del generador tiene más ruido. Esto significa que el generador no logra engañar al discriminador la mayoría de las veces (ralentizando el aprendizaje del discriminador), por lo que en la próxima iteración, una arquitectura más compleja del generador es el principal requisito.

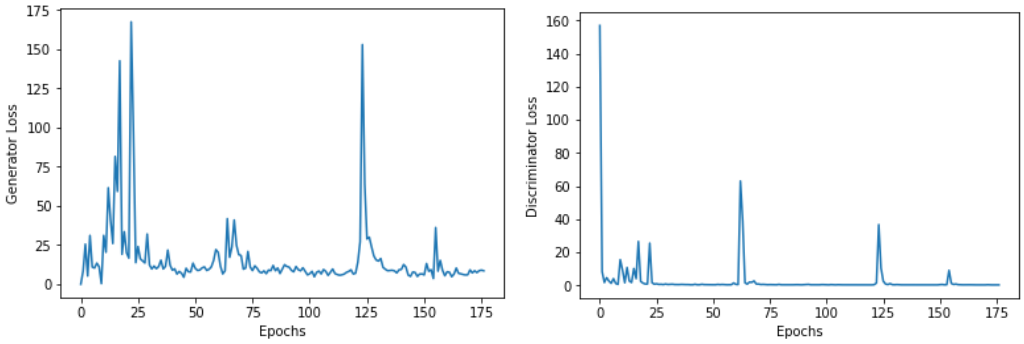


Figura 6. Error del generador (izquierda) y discriminador (derecha). Fuente: elaboración propia.

Algunas imágenes generadas por el generador se pueden ver en la **Figura 8**. A pesar del ruido esperado, hay un patrón (en la mayor parte de las imágenes generadas) consistente en una forma más brillante en el centro de la imagen. Dado que la GAN ha sido entrenadoa con retratos, este patrón es razonable, ya que en el centro de un retrato hay una cara, que es más brillante que el resto de la imagen.

- GANDeconv-Conv

Para este nuevo modelo, se diseñó una arquitectura más compleja.

• El generador tiene 5 capas convolucionales, seguidas de una última capa con función de activación *tanh*.
• El discriminador tiene 3 capas convolucionales, seguidas de una "max-pooling layer", y una "fully connected layer" al final.

Esta vez, los errores del generador y el discriminador, que se muestran en la **Figura 7**, muestran que el discriminador tarda más en converger. Esto sugiere que el generador hace que sea más difícil para el discriminador detectar imágenes falsas. Sin embargo, cuando ambos convergen, la mayoría de las imágenes generadas por el generador son etiquetadas como falsas por el discriminador, que es superior de nuevo. Por lo tanto, se requieren arquitecturas aún más complejas para el generador, o un mayor tiempo de entrenamiento.

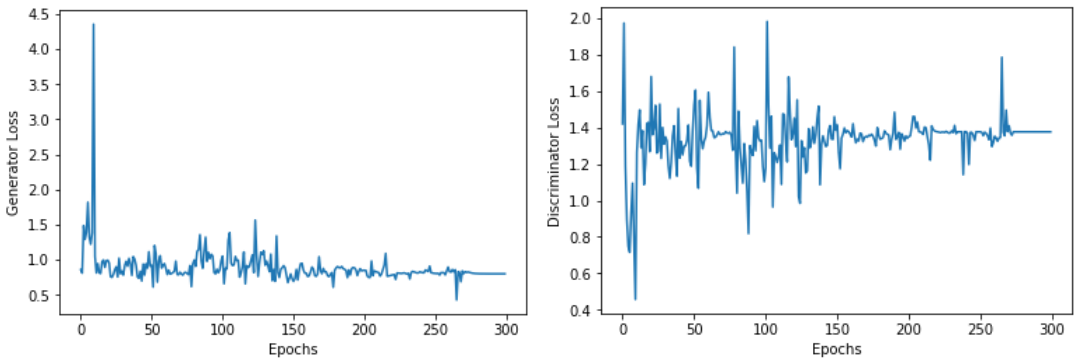


Figure 7. Error del generador (izquierda) y discriminador (derecha). Fuente: elaboración propia

- DCGAN

Finalmente, se utilizó un modelo de la literatura. La arquitectura DCGAN es robusta en un gran número de casos distintos.

El código, accesible en **[18]**, ha sido personalizado para acomodar la dimensión de las imágenes de nuestro conjunto de datos. El principal problema encontrado fue la potencia computacional necesaria para entrenar dicho modelo. Las s disponibles para este proyecto solo permitieron la ejecución de un número reducido de épocas en un tiempo razonable.

Aunque el generador no llega a producir pinturas realistas, debido a las pocas épocas de entrenamiento, las imágenes mostraron un brillo similar a una cara en el centro de la imagen. Como se puede ver en la **Figura 9**, se aprecia incluso la forma de un cuello.

RESULTADOS

- Clasificación

Como resumen, la **Tabla 1** muestra los resultados obtenidos por los diferentes modelos. La precisión se define como el porcentaje de predicciones correctas. Nótese que hay 4 clases posibles, por lo que un clasificador "aleatorio" tendría una precisión del 25%.

Nombre del Modelo	Precisión en el Conjunto de Entrenamiento	Precisión en el Conjunto de Test
ShallNet	29.0%	27.6%
VGG16+TL version 1	45.1%	39.3%
VGG16+TL version 1	56.1%	57.2%
CV+FC	55.2%	56.3%

Table 1. Comparación de la precisión de los cuatro modelos de clasificación. Fuente: elaboración propia.

Aunque no hay modelos realizando una tarea exactamente similar disponibles en la literatura para usar como punto de referencia, **[19]** alcanza una precisión del 59% en una tarea comparable. Dicha tarea consiste la clasificación en función de la emoción transmitida de imágenes reales, lo cual lo facilita, pero en una de 9 clases posibles (en lugar de 4), lo cual lo complica. Por otro lado, usan un conjunto de datos mucho más grande. De esta manera, los resultados obtenidos en este documento se consideran muy satisfactorios.

- Redes Neuronales Generativas Adversarias (GANs)

Los resultados finales no son lo suficientemente realistas como para ser utilizados como datos de entrenamiento para la tarea de clasificación, lo cual era el objetivo inicial. Sin embargo, se generaron algunos patrones interesantes, que pueden informar proyectos en el futuro.

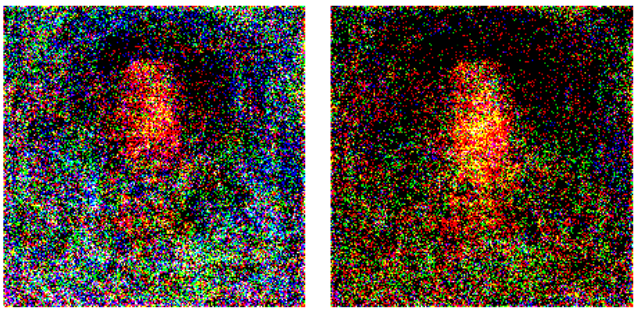


Figura 8. Imágenes generadas por el generador de BasicGAN. Se observa una forma brillante en el centro de la imagen, que se puede asociar a una cara. Fuente: elaboración propia.

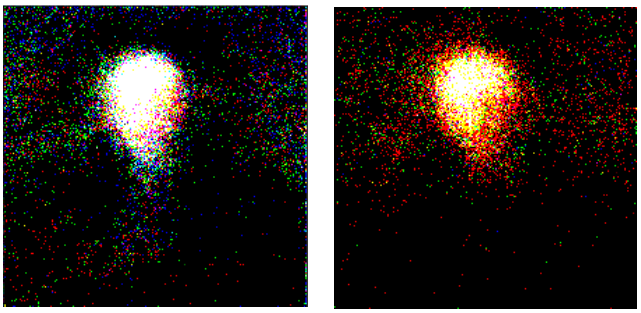


Figura 94. Imágenes generadas por el generador de DCGAN después de un número reducido de épocas de entrenamiento. Fuente: elaboración propia.

CONCLUSIONES

En la tarea de clasificación, la limitación de datos fue el principal determinante del desempeño de los modelos. Fue este problema el que provocó que el modelo pre-entrenado proporcionara resultados significativamente mejores que la red poco profunda, y el que permitió que el modelo CV + FC de características de extraídas a mano proporcionara prácticamente los mejores resultados.

Como trabajo futuro, la prioridad número uno es obtener un conjunto de datos fiable y extenso. Se trata de una tarea complicada, ya que es difícil asignar etiquetas de emoción a obras de arte de forma consistente, debido a la subjetividad asociada.

Otra línea de trabajo futuro consistiría en combinar modelos que integren características locales y globales. Los modelos de arquitectura profunda proporcionaron resultados significativamente buenos debido a su capacidad para identificar características locales que pueden evocar emociones (una cara sonriente, una pose de oración, etc.). Sin embargo, su rendimiento está limitado por la importancia de las características globales (color, textura, brillo, etc.). Este efecto se hizo evidente con el sorprendente desempeño del modelo de características extraídas “a mano”, CV + FC. Por lo tanto, un modelo que sea capaz de identificar características tanto globales como locales, definitivamente proporcionaría los mejores resultados.

Para las GANs, la conclusión principal extraída de este estudio es la importancia de una arquitectura suficientemente compleja del generador. Sin embargo, una arquitectura

compleja para una tarea generativa requiere mucha potencia computacional para entrenar, la cual no estaba disponible para este proyecto. Como línea de trabajo futuro, más recursos permitirían unos mejores resultados al modelo DCGAN, que ha demostrado ser robusto en múltiples aplicaciones.

La otra conclusión relevante respecto al empleo de GANs es la importancia de utilizar un conjunto de imágenes con una estructura similar. En este proyecto, este hecho se traduce en la obtención de mejores resultados entrenando los modelos solo con retratos, frente a entrenarlos con todos los tipos de pinturas (a pesar de que suponga un conjunto de datos de entrenamiento más extenso).

BIBLIOGRAFÍA

[1] https://www.chezmana.com/. Creating a marketplace for your artworks.

[2] S. Malik, Sungchul Kim, and Eunyee Koh. Perceptual similarity ranking

[3] McDuff, D., et al. "A cross-platform real-time multi-face expression recognition toolkit." Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems [Association for Computing Machinery (ACM), 2016].

[4] Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

[5] https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks

[7] Simonyan, K. &. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv*.

[8] Simonyan, K. &. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv*.

[9] Katanforoosh, K. (s.f.). Generative Adversarial Networks. *Stanford University, CS230*, (pág. 18).

[10] Lucic, K. Are GANs created equal, a large case study. *Advances in neural information processing systems*. 2018.

[11] Alec Radford, L. M. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv*.

[12] Ng, A. (2018). Machine Learning Strategsy. *CS230 lecture. Stanford University.*

[13] Torrey, L. (2010). Transfer Learning. *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques.*

[14] Hassan, M. u. (2018). VGG16 – Convolutional Network for Classification and Detection.

[15] Perlin, K. (2002). Improving Noise. *Association for Computing Machinery*.

[16] R. M. H. Nguyen and M. S. Brown. Why you should forget luminance conversion and do something better. In 2017 IEEE Conference on Computer Vision and Pattern Recognition. (CVPR), pages 5920–5928, July 2017.

[17] Liu, S. a. (2018). Texture-aware emotional color transfer between images. *IEEE*.

[18] https://github.com/carpedm20/DCGAN-tensorflow

[19] You, Q. L. (2016). Building a large scale dataset for image emotion recognition: The fine print and the benchmark. *Thirtieth AAAI Conference on Artificial Intelligence*.

EFFICIENT DEEP NEURAL NETWORK ARCHITECTURES FOR CLASSIFICATION OF EMOTIONAL REACTIONS TO ARTISTIC PAINTINGS

Author: Garrido Contreras, Arturo

Directors: Caro Carretero, Raquel

Collaborating Entity: Stanford University. ChezMana.

PROJECT SUMMARY

INTRODUCTION AND PROJECT RELEVANCE

This project was developed as part of the Deep Learning (CS 230) course at Stanford University. The university provided AWS credits to run the models due to an existing partnership between the two entities. The project was proposed by ChezMana **[1]**, a Palo Alto based startup, that also provided the dataset.

Classifying art into emotion categories has been challenging for researchers even with state-of-the-art deep learning. The difficulties stem from subjective labeling, different user perspectives, and the fact that emotions can be invoked by global and local features of an image. The study of emotion with deep learning has been less explored than other computer vision applications due to the aforementioned challenges. However, progress in this field could have many applications in education, human-robot interaction and media and entertainment **[2][3]**.

This document is focused on two areas. The main goal consists on constructing a classifier that predicts emotion classes evoked by art images. Different approaches and architectures are explored to asses and compare their performance. Due to the limited amount of reliably labeled data, Generative Adversarial Networks are used to generate novel artworks. Generative models are much more complex and difficult to train than classification models, but existing architectures will be leveraged to try to get reasonable and insightful results. The goal of generating novel artworks is intended as a data-generation technique, that will be fed to the emotion classifier to improve its performance. However, it is a relevant challenge in and off itself.

STATE OF THE ART

- Classification problem

Convolutional Neural Networks (CNN hereon) represent the most popular architecture in Computer Vision. The main advantage that CNNs provide is a huge reduction in the number of parameters needed in the network compared to classic Neural Networks. They work well for problems that have non spatially dependent features, such as object recognition in images.

CNNs look at different local regions at a time as opposed to the whole image at once, with the use of a *filter* or *kernel*, as shown in **Figure 1**. There is a number of hyperparameters that define the functioning of a convolution, notably padding and stride **[4]**.

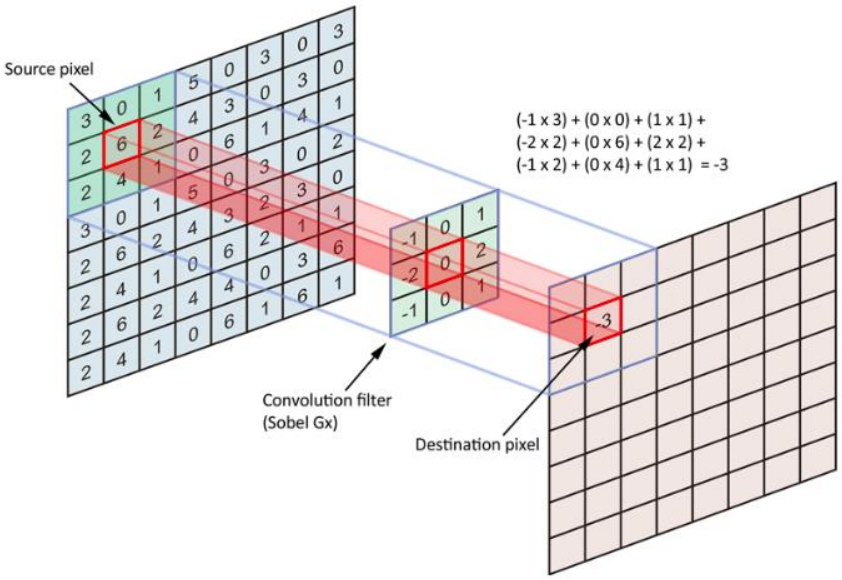


Figure 1. Convolution. Source: ***[5]***

There are some CNN architectures that are known to perform well in a variety of Computer Vision tasks, such as:

AlexNet

The so-called AlexNet architecture was introduced by Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton **[6]**. This neural network, with 60M parameters and 650,000 neurons, achieved the best performance in the 2010 ImageNet LSVRC-2010 contest, consisting on classifying 1.2M high-resolution images into 1000 different categories.

The architecture consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax.

VGG-16

Introduced by Karen Simonyan and Andrew Zisserman **[7]**, with affiliation to Google DeepMind. The architecture consists of thirteen convolutional layers, five max-pooling layers, two fully-connected layers and a final 1000-way softmax. The innovation with respect to previous models, consists on a much larger architecture depth, which is feasible due to the use of very small (3x3) convolution filters, resulting in around 133M parameters.

- Generative Adversarial Networks

Up until recently, the most important successes in deep learning have involved discriminative models, which consist on mapping a high-dimension input (image, text, audio, etc.) to a class label. Deep generative models have had less impact, due to the difficulty of applying the existing models and methods to generative tasks.

The base principle of Generative Adversarial Networks (GANs hereon) is to have two networks that are trained competing against each other. One of them is the **Generator**, which will be the one creating samples of data that are intended to come from the same distribution as the training data. On the other hand, the **Discriminator**, receives as input a sample and classifies it as real (coming from the real probability distribution) or fake (coming from a different distribution) **[8]**. In one training step, two gradient steps take place: the generator updating its parameters to reduce its cost function, and the discriminator updating its parameters to reduce its cost function. These cost functions are shown in **Equation 1** and **Equation 2**.

$$J^{(D)} = \underbrace{-\frac{1}{m_{real}}\sum_{i=1}^{m_{real}} y_{real}^{(i)}.\log(D(x^{(i)}))}_{\text{cross-entropy 1: "D should correctly label real data as 1"}} \underbrace{-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}(1-y_{gen}^{(i)}).\log(1-D(G(z^{(i)})))}_{\text{cross-entropy 2: "D should correctly label generated data as 0"}}$$

Equation 1. Cost function of the Discriminator ***[9]***

$$J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D(G(z^{(i)})))$$

Equation 2. Cost function of the Generator. Source: ***[9]***

These are just two examples of potential cost functions to use. The current literature offers a lot more forms for the generator loss, as well as for the discriminator's **[10]**.

There are some GAN architectures that have been proven to work well on a number of use cases. The most extensively used one is DCGAN **[11]**, whose Generator goes through 4 deconvolution layers, as shown on **Figure 2**.

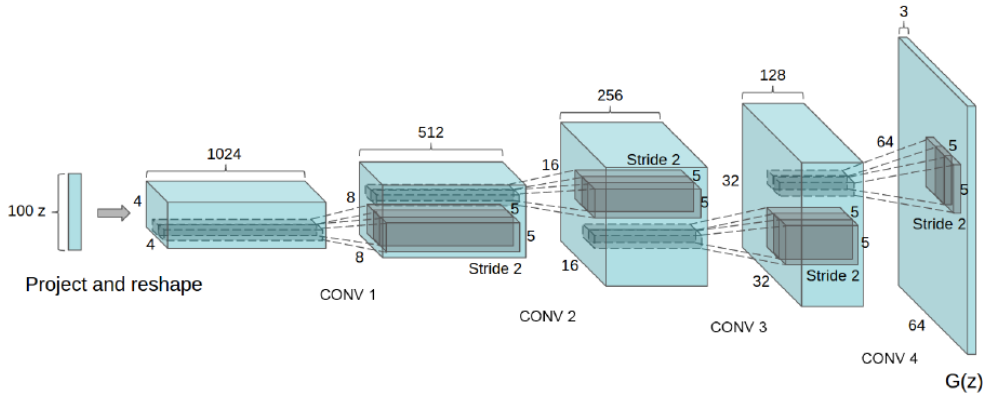


Figure 2. Generator architecture of DCGAN. Source: ***[11]***

DATASET

This work uses a dataset of 40,000 artistic images, provided by the project sponsor **[1]**, out of which only 3,000 were labeled by emotion. There were 16 different classes for the emotion label, some of them very difficult to distinguish from each other, as can be seen on **Figure 3**.

Figure 3. Optimism (left) and Joy (right) are an example of the inconsistency of the provided labels. Source: own elaboration.

Therefore, a decision was made to group the classes into 4 main categories:

- Category 0: Sadness, Fear, Disgust, Anger.
- Category 1: Neutral, NA.
- Category 2: Lust, Envy, Surprise.
- Category 3: Optimism, Joy, Love.

SOLUTION

A good practice when approaching a deep learning problem, is to start with a basic model to obtain the baseline results **[12]**. Then, work iteratively, creating more and more complex models and tuning the hyperparameters in the process. This approach was followed for both the classification and GAN models.

- Classification
    - ShallNet

This simple baseline model consists of a 3-layer CNN architecture built from scratch. At this point of the study, only a small percentage of the data was labeled by emotions. A decision was made to train the model only with "portrait" paintings, in order to make the task easier for this simple architecture. The accuracy results (percentage of correct predictions) are shown in **Table 1**.

    - VGG16+TL version 1

As evidenced by the results of the previous model, training a convolutional architecture from scratch with such few datapoints can be challenging. Therefore, Transfer Learning **[13]** was used in the next architecture.

The VGG16 architecture **[14]** was used, pretrained with the imageNet dataset **[6]** of 15 million images belonging to roughly 22,000 categories. The potential of this approach was unclear, since the model was pretrained on real images, not artistic paintings.

The last maxpooling layer of the VGG16 network was replaced with a global average pooling layer, and one fully-connected layer was used rather than two, with a softmax layer at the end. Only these last two layers were trained, and the rest of the parameters of the VGG16 were "frozen". This model was named "VGG16 + TL version 1". Its accuracy results are shown in **Table 1**.

Class Activation Maps (CAMs) were used as part of the error analysis process, in order to inform the future models. This analysis, shown in **Figure 4**, revealed that although some emotions are activated by local features that may be expected to do so (for example, by a face), the model is also overfitting noise, since some of the high gradient regions should not provoke any emotional reaction.

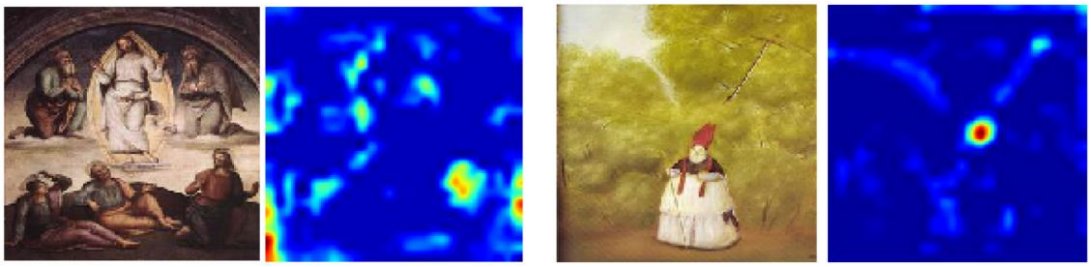


Figure 4. Class Activation Maps from different paintings. Source: own elaboration

- VGG16+TL version 2

Driven by the CAMs, the data was filtered again by portraits, due to the fact that portraits have more consistent local features, so the model should not overfit on as much noise as previously. The same VGG16 model was built, to asses its performance with this new dataset. This model was named "VGG16 + TL version 2". Results are shown in **Table 1**.

- CV+FC

Still driven by the CAMs analysis, it was clear that local features (a face, an object, etc.) are not enough to define the emotion transmitted by a painting. Therefore, a fundamentally different approach, driven by the research conducted on the study of emotion and images, was taken.

13 hand-engineered features were defined, based on the color (6 features), brightness (1 feature) and texture (6 features) of the image, allowing the model to associate global features to a particular emotion. The texture features are based on Perlin noise **[15]** which forms the Hard-Soft (HS) scale presented by **[16]**. We use a standard method to measure image brightness by calculating the square root of a linear combination of the RMS values for each color channel from **[16]**. For color features we used the method in **[17]**: we first extract the top 5 color centroids from the example image, then compare them to the standard color maps we created for each emotion class. These features are shown in **Figure 5**.

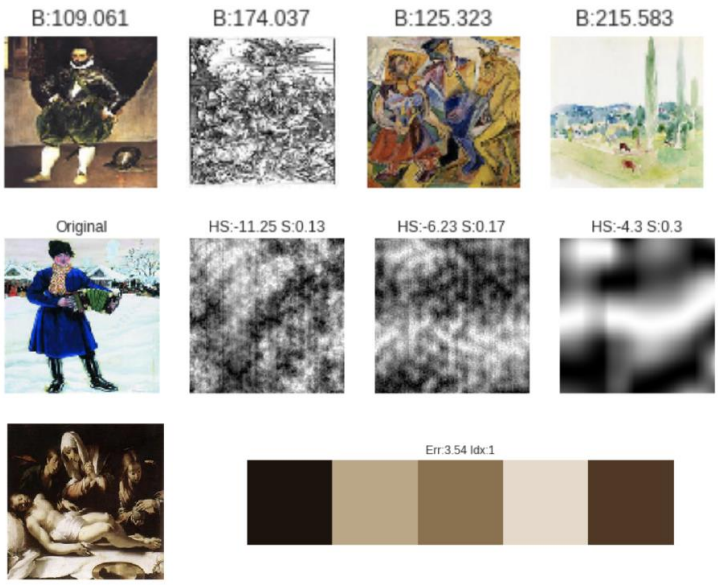


Figure 5. Hand engineered features, from top to bottom: Perceived Brightness where B is the brightness value, Texture Similarity Structure where HS is the Hard-Soft value and S is the structural similarity of the input image to our texture maps, Color Map Distance where the 5 primary colors of the input image are extracted and the distances to our landmark color maps are calculated. Source: own elaboration.

Once the features are defined, a simple model was built. This model, named "CV+FC", consists of a 4-neuron layer followed by a softmax layer. The results are also shown in **Table 1**.

- Generative Adversarial Networks

For the generative task, the dataset was limited to "portraits", with around 7k samples in total (since they do not need to be labeled). By working with only portraits, the generator should do a better job at creating realistic images.

  - BasicGAN

This baseline model had the following architecture:

- Generator: shallow network with three Fully Connected layers. It receives a 100-dimension random code and generates an array of dimension 224 * 224 * 3.
- Discriminator: shallow network, with two Fully Connected layers, plus a dense layer with a sigmoid output (since it is a binary classification problem).

After training for 175 epochs, the cost of the generator and discriminator were printed. In a generative-adversarial game, the cost of each member should be very "noisy", since both of them are constantly getting better and making the task more difficult for each other. However, as seen in **Figure 6**, in our model the discriminator converges fast while the generator cost has more noise. This means that the generator fails to trick the discriminator most of the times (slowing the learning of the discriminator), so in the next iteration, a more complex architecture of the generator is the primary requirement.

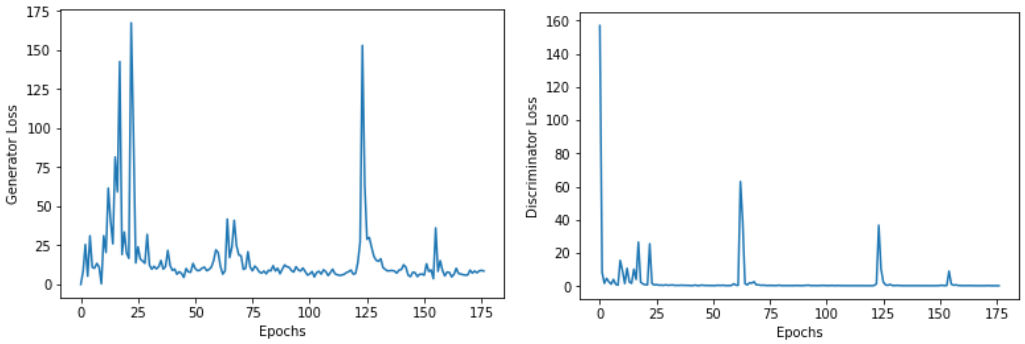


Figure 6. Cost of the generator (left) and discriminator (right). Source: own elaboration.

Some images generated by the generator can be seen in **Figure 8**. Despite the expected noise, there is a pattern (consistent across generated images) of a brighter shape in the center of the image. Since the GAN was trained with portraits, this pattern is reasonable, as at the center of a portrait there is a face, which is brighter than the rest of the image.

- GANDeconv-Conv

For this new model, a more complex architecture was designed.

- The generator has 5 convolutional layers, followed by a last layer with a *tanh* activation function.
- The discriminator has 3 convolutional layers, followed by a max-pooling layer, and a fully connected layer in the end.

This time, the costs of the generator and discriminator, shown in **Figure 9**, show that the discriminator takes longer to converge. This suggests that the generator is making it more difficult for the discriminator to detect fake images. However, when both converge, most of the images generated by the generator are labeled as fake by the discriminator, which is superior again. Either more training or more complex architectures for the generator are required still.

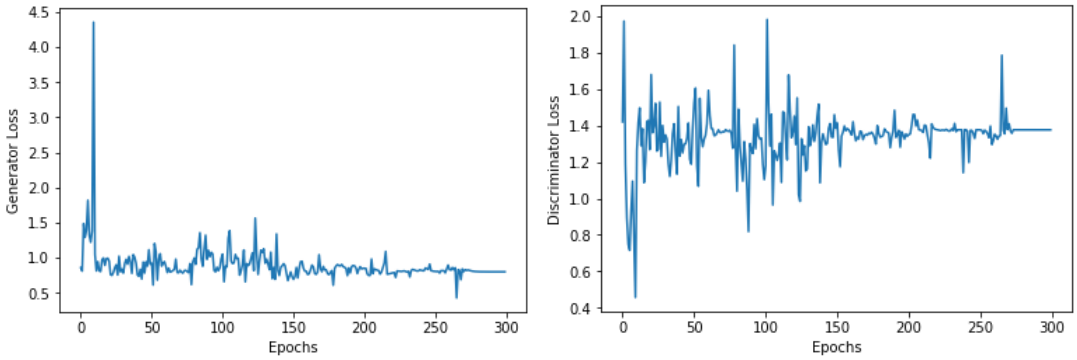


Figure 7. Cost of the generator (left) and discriminator (right). Source: own elaboration.

- DCGAN

Finally, a model from the literature was used. The DCGAN architecture is robust in terms of stability in most settings.

Its code, accessible in **[18]**, was customized in order to accommodate the dimension of the images from our dataset. The main problem encountered was the computational

power needed to train such a model. The GPUs available for this project only allowed for the execution of very few epochs within a reasonable amount of time.

Although the generator did not produce realistic paintings, due to the few epochs it was trained for, the images did show a face-like brightness in the middle of the image. As can be seen in **Figure 9**, even a collar is noticeable.

RESULTS

- Classification

As a summary, **Table 1** shows the performance results of the different models. Accuracy is defined as the percentage of correct predictions. Note that there are 4 possible categories, so a "random" classifier would have an accuracy of 25%.

Model Name	Training Set Accuracy	Test Set Accuracy
ShallNet	29.0%	27.6%
VGG16+TL version 1	45.1%	39.3%
VGG16+TL version 1	56.1%	57.2%
CV+FC	55.2%	56.3%

Table 1. Performance comparison of the four classification models. Source: own elaboration.

Although there is not an exactly similar task available in the literature to use as a benchmark, **[19]** reaches an accuracy of 59% in a comparable problem. They classify the emotion of real images, which makes the task easier, but into 9 potential classes (instead of 4), which makes it more difficult. On the other hand, they use a much larger dataset. Therefore, the obtained results in this document are very satisfactory.

- Generative Adversarial Networks

Although the end results are not good enough to be used as training data for the classification task, which was the initial goal, some interesting patterns were generated.

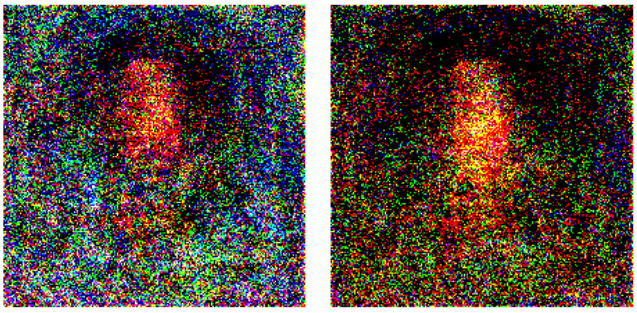


Figure 8. Image generated by the Generator of BasicGAN. A brighter shape is seen in the center of the image can be associated to a face. Source: own elaboration.

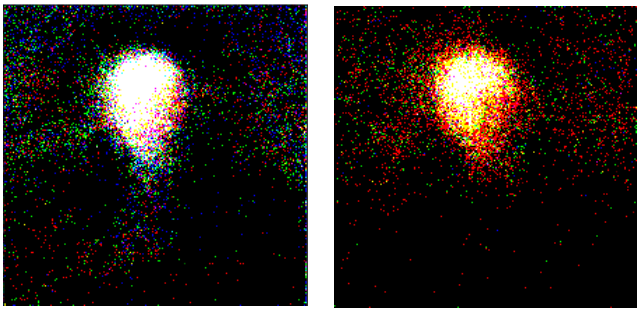


Figure 94. Images generated by DCGAN after a few epochs. Source: own elaboration.

CONCLUSIONS

In the classification task, the limitation of data was the main driver of the model's performance. It was this issue that caused the pre-trained model to provide significantly better results than the shallow network, and that allowed the CV+FC model of hand-engineered features to provide almost the best results.

As future work, the number one priority is to obtain a reliable and extended dataset. This task is very challenging, as it is difficult to provide consistent emotion labels to artworks, due to the subjectivity of the task.

Another line of future work would consist of combining models integrating both local and global features. The deep architecture models provided significantly good results due to their ability to identify local features that may evoke emotions (a smiling face, a praying pose, etc.). However, their performance is limited by the role of global features (color, texture, brightness, etc.) in evoking emotions. This became evident with the surprisingly good performance of the CV+FC hand-engineered model. Therefore, a model that is able to identify both global and local features, would definitely provide top results.

For the GANs, the main conclusion extracted from this study is the importance of a sophisticated enough architecture of the generator. However, a complex architecture for a generative task requires a lot of computing power to train, which was not available for this project. As a line of future work, more resources would allow to train the DCGAN model, which has proven to be robust for multiple applications, to a higher level.

Another relevant conclusion is the importance of images with a similar structure in the training set. In the case of this project, this translated into portraits providing much better results than when combining all types of artworks (landscapes, scenes, etc.).

BIBLIOGRAPHY

[1] https://www.chezmana.com/. Creating a marketplace for your artworks.

[2] S. Malik, Sungchul Kim, and Eunyee Koh. Perceptual similarity ranking

[3] McDuff, D., et al. "A cross-platform real-time multi-face expression recognition toolkit." Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems [Association for Computing Machinery (ACM), 2016].

[4] Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

[5] https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks

[7] Simonyan, K. &. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv*.

[8] Simonyan, K. &. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv*.

[9] Katanforoosh, K. (s.f.). Generative Adversarial Networks. *Stanford University, CS230*, (pág. 18).

[10] Lucic, K. Are GANs created equal, a large case study. *Advances in neural information processing systems*. 2018.

[11] Alec Radford, L. M. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv*.

[12] Ng, A. (2018). Machine Learning Strategsy. *CS230 lecture. Stanford University.*

[13] Torrey, L. (2010). Transfer Learning. *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques.*

[14] Hassan, M. u. (2018). VGG16 – Convolutional Network for Classification and Detection.

[15] Perlin, K. (2002). Improving Noise. *Association for Computing Machinery*.

[16] R. M. H. Nguyen and M. S. Brown. Why you should forget luminance conversion and do something better. In 2017 IEEE Conference on Computer Vision and Pattern Recognition. (CVPR), pages 5920–5928, July 2017.

[17] Liu, S. a. (2018). Texture-aware emotional color transfer between images. *IEEE*.

[18] https://github.com/carpedm20/DCGAN-tensorflow

[19] You, Q. L. (2016). Building a large scale dataset for image emotion recognition: The fine print and the benchmark. *Thirtieth AAAI Conference on Artificial Intelligence*.

Table of Contents

1. Introduction 2

1.1. A Psychology Perspective 2

1.2. A Computer Science Perspective 3

1.3. Project Relevance 3

1.4. Scope 4

2. State of the Technology 5

2.1. Classification problem 5

2.1.1. Convolutional Neural Networks basics 5

2.1.2. CNN architectures for Computer Vision 10

2.1.3. Interpretability of CNNs 12

2.2. GANs 13

2.2.1. Motivation 13

2.2.2. Generator vs Discriminator 15

2.2.3. Training GANs 17

2.2.4. State of the Art Architectures 21

2.2.5. Notable Results 22

3. Dataset and Features 25

3.1. Dataset Description 25

3.2. Dataset Cleaning and Manipulation 25

4. Implemented Models and Analysis of Results 29

4.1. Classification 29

4.1.1. ShallNet – A shallow CNN 29

4.1.2. VGG16+TL version 1 31

4.1.3. VGG16+TL version 2 36

4.1.4. CV+FC 37

4.2. GANs 44

4.2.1. BasicGAN 44

4.2.2. GANDeconv-Conv 47

4.2.3. DCGAN 49

5. Conclusions and Future Work 50

5.1. Classification 50

5.2. GANs 51

BIBLIOGRAPHY 52

ANNEXES A-E

1. Introduction

Classifying art into emotion categories has been challenging for researchers even with state-of-the-art deep learning. The difficulties stem from subjective labeling, different user perspectives, and that emotions can be invoked by global and local features of an image. The study of emotion with deep learning has been less explored than other computer vision applications due to the aforementioned challenges. However, progress in this field could have many applications in education, human-robot interaction and media and entertainment **[1][2]**.

1.1. A Psychology Perspective

Daniel Berlyne's "new experimental aesthetics" represented a turning point in the study of emotional responses to art **[3][4]**. He defined a class of stimulus features known as "collative variables", such as complexity, novelty, uncertainty and conflict, that drive the viewer's emotional reactions. All these features share two main similarities: they involve comparing pieces of information (expected information with incoming information) and they have the potential to increase arousal. Therefore, if art can be explained in terms of these objective features, then an emotional reaction to such art could theoretically be measured and predicted.

On the other hand, modern emotion psychology states that emotions are human's psychological mechanisms to deal with "fundamental life tasks" **[5].** This approach has given birth to a multitude of theories and models regarding the functions, nature and processes of emotions **[6]**. *Appraisal theories of emotion* are considered a leading perspective on the understanding of the elicitation and differentiation of emotions. These theories revolve around the principle that it is not the event itself that causes an emotional experience, but the personal evaluation of the event. This results in a more subjective and occurrence-specific approach: in the same situation, the same person will have different emotions at different times **[7]**.

Finally, recent studies have shown that personal inclinations such as color preferences depend on factors such as gender **[8]** or educational background **[9]**.

1.2. A Computer Science Perspective

In recent years, with the increasing use of digital photography technology by the general public and the ubiquity of social networks, the number of available images has exploded into yet unseen numbers. Huge image collections can be accessed through the Internet, and professional image databases for research purposes grow by thousands of images per day.

In parallel to this explosion of available data, the advent of deep convolutional neural networks has allowed computers to finally "solve" the object classification problem. In the ImageNet Large-Scale Visual Recognition Challenge **[10]**, algorithms are tasked with identifying which of 1000 possible classes of object is present in a particular image. Inception-Resnet, a powerful Convolutional Neural Network which will be discussed in detail in **Section 2**, has a top-5 classification error of just 3.08% on the ImageNet test set **[11]**.

These advances have raised a new challenge among the deep learning community: is it possible to make use of these convolutional architectures to carry out more abstract classification problems? When a person looks at an image, in addition to identifying the objects, she might experience an emotional response. Is it possible for a deep network to "learn" this expected reaction? This field of research, sometimes referred to as *Artificial Emotional Intelligence*, inherently has a number of challenges due to its subjectivity. Unlike the case of object recognition (and most supervised-learning problems), in which there is an objective source of truth for which objects are or are not contained in a photo, there is no way to "prove" that a particular photo evokes excitement or fear or sadness.

1.3. Project Relevance

The range of applications of emotional reaction predictions is not as broad and straightforward as that of object detection.

However, this field will have an increasingly relevant role as machine-learning based technologies become prevalent in the workplace and in people's life. Emotions play an important role in human interactions, so systems that understand and respond to emotions have the potential to provide a much better user experience and user interface **[2]**. For this

reason, *Artificial Emotional Intelligence* is gaining importance in the field of human-computer interaction.

In addition, media and entertainment companies are also leveraging this technology. As an example, Chez Mana **[12]**, a Palo Alto based startup that sponsored this work and provided the datasets, allows companies to choose the emotions that they want to transmit in a specific advertising campaign (for example, a bank may want artwork that transmits confidence and reliability). Their technology matches these selected emotions to unique images designed by their artists and connects the companies with these artists to get custom artwork.

1.4. Scope

This document is focused on two areas. The main goal consists on constructing a classifier that predicts emotion classes of art images. Different approaches and architectures are explored to asses and compare their performance. Due to the limited amount of reliable data, Generative Adversarial Networks are used to generate novel artworks. Generative models are much more complex and difficult to train than classification models, but existing architectures will be leveraged to try to get reasonable and insightful results. The goal of generating novel artworks is intended as a data-expansion technique, that will reinforce the emotion prediction classifier. However, it is a big challenge in and off itself.

2. State of the technology

The literature review was carried out in two main areas: emotion classification (of both photographs and artistic paintings) and image generation through Generative Adversarial Networks (GANs hereon).

2.1. Classification problem

2.1.1. Convolutional Neural Networks basics

Convolutional Neural Networks (CNNs hereon) represent the most popular architecture in Computer Vision. The name comes from the convolution linear operator used with matrices. During the past decade, CNNs have provided revolutionary results in different fields related to pattern recognition, such as computer vision and speech recognition.

The main advantage that CNNs provide is a huge reduction in the number of parameters needed in the network compared to classic Neural Networks. They work well for problems that have non spatially dependent features, such as object recognition in images (i.e. the position of the object will vary from image to image) **[13]**.

Elements of a CNN

Let's assume that the input to the network is an image of size 64x64x3 (64 by 64 pixels and 3 RGB[1] channels). If we think of a traditional neural network that receives raw pixels as input, connecting this input layer into a single neuron of the hidden layer would require 64x64x3 = 12,288 parameters. If we wanted the hidden layer to be of the same size of the image, that would translate into 12,288 x 12,288 = 150M parameters in only one layer. This approach makes the problem too computationally expensive even for a very low-resolution image.

Instead of a full connection, CNNs look at different local regions at a time as opposed to the whole image at once, with the use of a *filter* or *kernel*, as shown in **Figure 1**. If the kernel used is of size 7x7x3, then the total number of parameters for this first layer will be 147.

[1] RGB (red, green, and blue) refers to a system for representing the colors to be used on a computer display. Red, green, and blue can be combined in various proportions to obtain any color in the visible spectrum. Levels of R, G, and B can each range from 0 to 100 percent of full intensity.

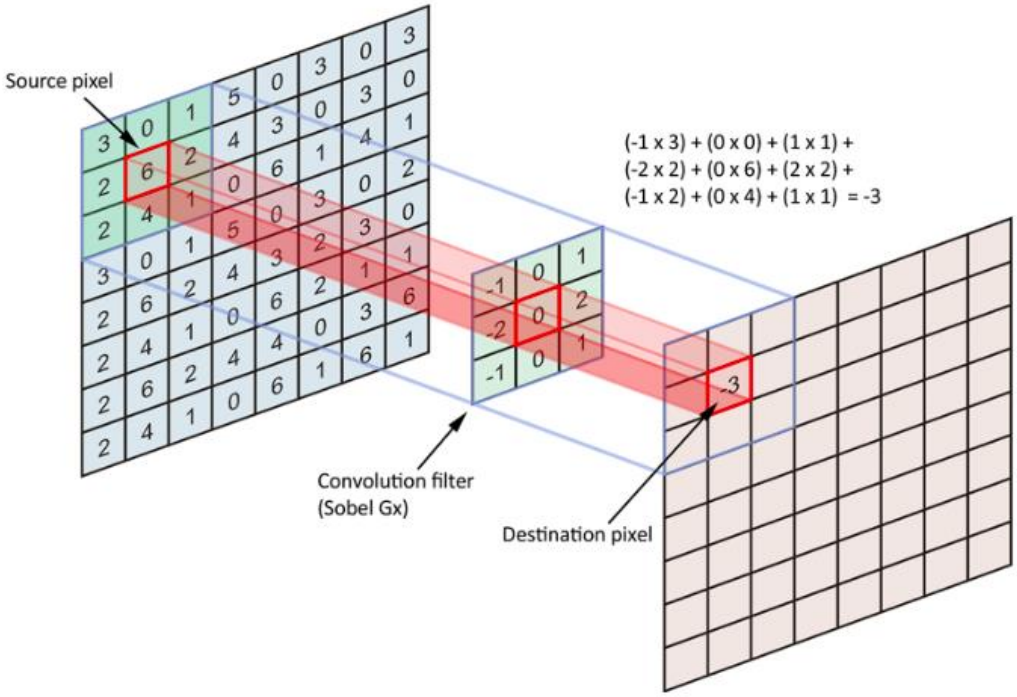


*Figure 1. Convolution. Source: image taken from **[14]***

This filter or kernel will "slide" through the image. It will start in the upper left corner and compute the first hidden neuron of the next layer. Then, it will slide one (or more) pixel to the right, and compute the next hidden neuron, etc. A simplified example is shown in **Figure 2a** through **Figure 2d**. The convolution is completed once all pixels have been covered.

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		

*Figure 2a. Step 1. Source: image taken from **[14]***

1	1x1	1x0	0x1	0
0	1x0	1x1	1x0	0
0	0x1	1x0	1x1	1
0	0	1	1	0
0	1	1	0	0

4	3	

*Figure 2b. Step 2. Source: image taken from **[14]***

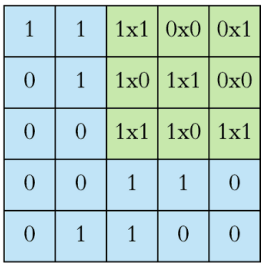


*Figure 3c. Step 3. Source: image taken from **[14]***

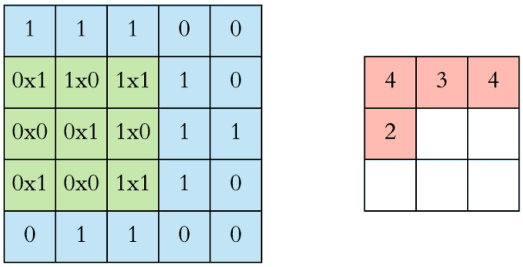


*Figure 2d. Step 4. Source: image taken from **[14]***

Notice that the for the sake of simplicity figures above represent both an input and a kernel in 2 dimensions (only one RGB channel). In practice, both the image and the kernel will likely have 3 dimensions.

Padding and Stride

The output of a convolutional layer will be affected by both the shape of the input (size of the image in Computer Vision), the shape of the kernel, the padding and the stride.

- *Padding*

From the previous figures, it can be noticed that a problem associated with a convolution layer is the loss of information on the borders of the image. In **Figure 2a** through **Figure 2d**, the central pixel will take part in all the calculations. However, the upper-left pixel only takes part on the first calculation. An easy and efficient way to avoid this issue is using zero-padding, sometimes referred to as simply padding **[15]**.

Padding is defined as the number of zeros concatenated at the beginning and at the end of an axis.

- *Stride*

  If the kernel slides one pixel at a time, the size of the output of the layer may be too large. Also, this output will likely present some overlap. To avoid this, we can move the kernel more than one pixel at a time. This is referred to as Stride.

  Stride is defined as the distance between two consecutive positions of the kernel along an axis.

Figure 4 shows a convolution with an input of size 5 x 5 and a kernel has a size of 3 x 3. **Figure 5** on the other hand shows a similar convolution but this time with padding = 1, and stride = 2.

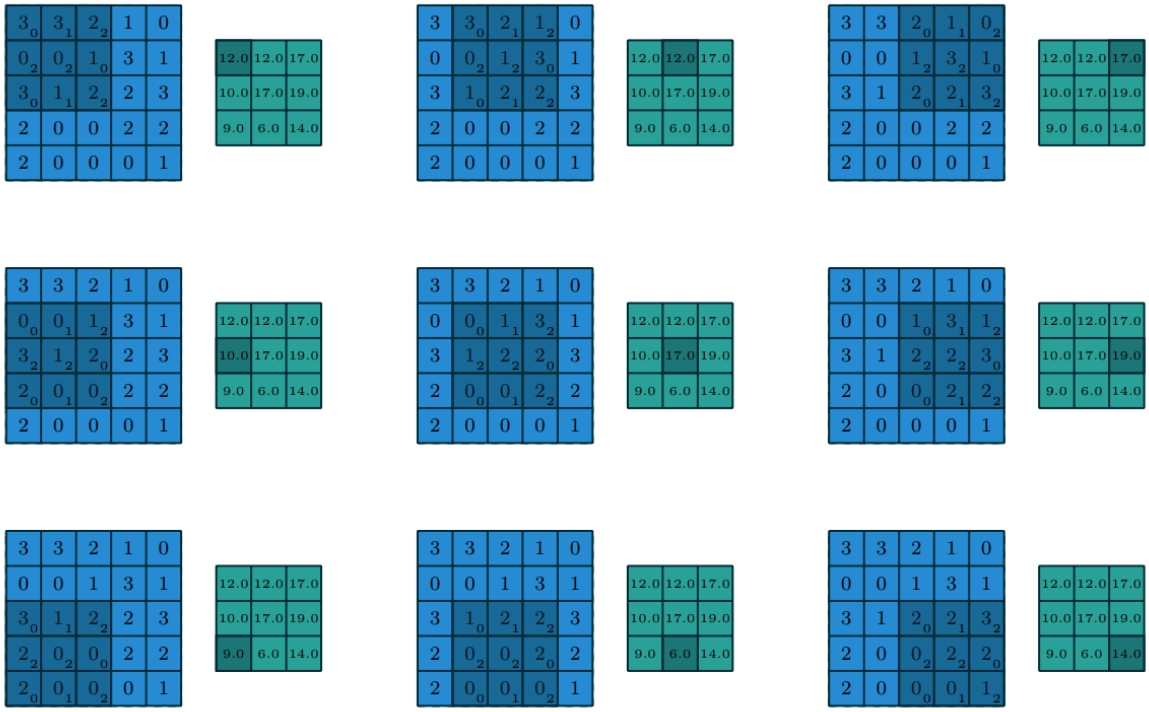


*Figure 4. Computing the output of a convolution with no padding or stride. Source: image taken from **[15]***

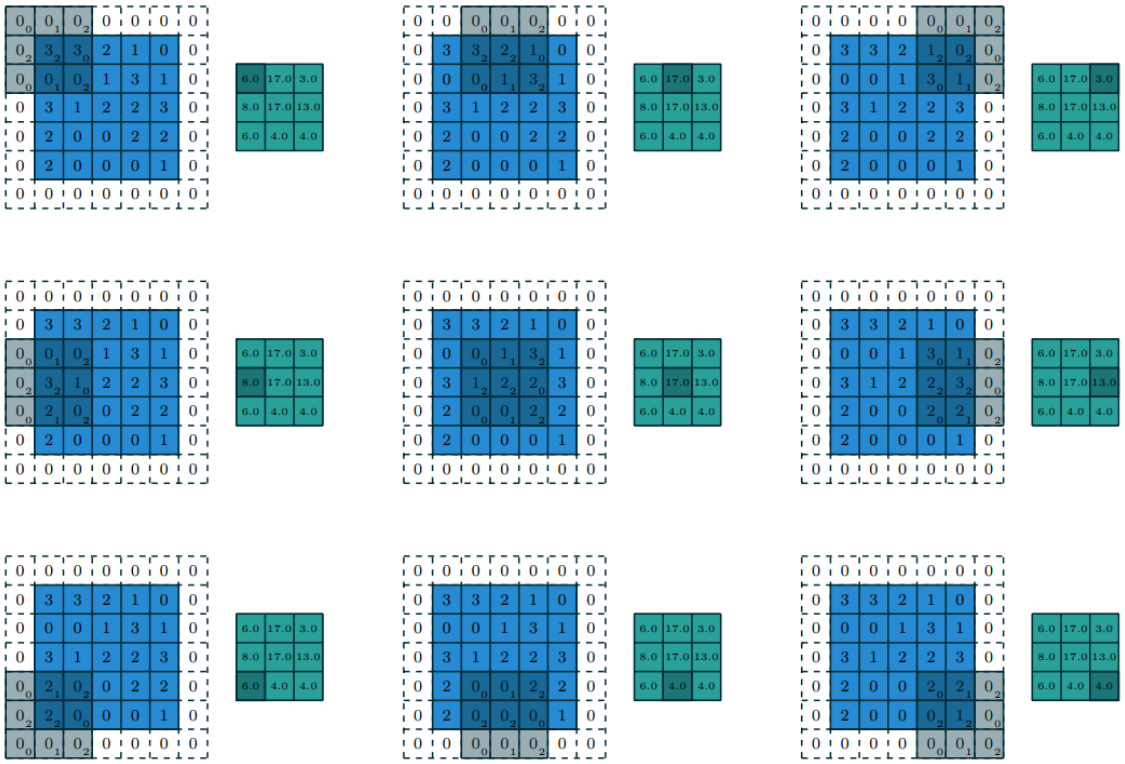


*Figure 5. Computing the output of a convolution with p = 1 and s = 2. Source: image taken from **[15]***

Pooling layers

The purpose of a pooling layer is to reduce the dimension of further layers **[16]**. In the domain of image processing, this can be seen as reducing the resolution of the image.

Max-pooling is one of the most common pooling methods. As shown in **Figure 6**, it consists of dividing the input into sub-regions and taking the maximum of each region. A common size of the partitions is 2x2, which is the one used in the figure below.

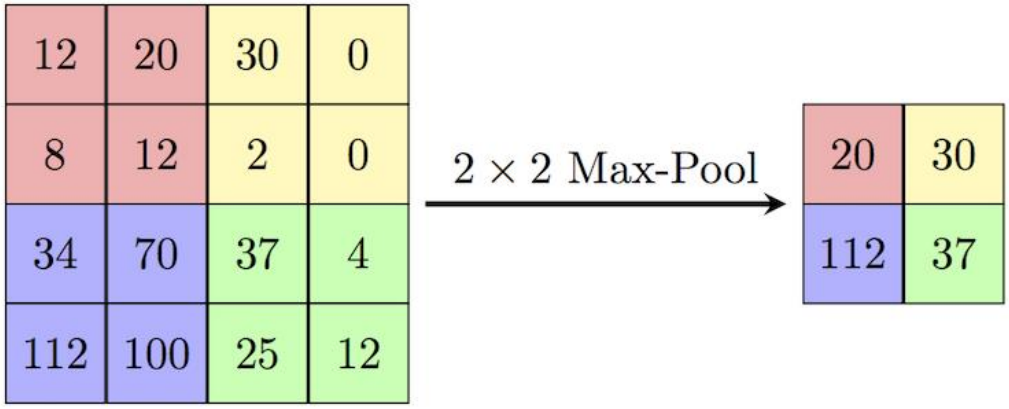


*Figure 6. Max Pooling. Source: image taken from **[13]***

Average-pooling is also commonly used. It takes the average of all the elements in the partition.

Fully connected layers

A fully-connected layer (FCL) refers to a layer similar to those in a traditional neural network. In a FCL, each node of the previous layer is connected to every node of the next layer. In order to have a fully connected layer, the output of the last convolutional layer is *flattened* (turned into a vector instead of a matrix).

As a final note, the size of the filter (denoted f), the padding (denoted d), the stride (denoted s) and the number of filters are all hyperparameters of the model.

Figure 7 shows the architecture of LeNet-5, a frequently used CNN model. This architecture alternates *convolutions* with *subsampling* (or average pooling). Then, it flattens the output of the last subsampling and has 2 full connections and a gaussian connexion.

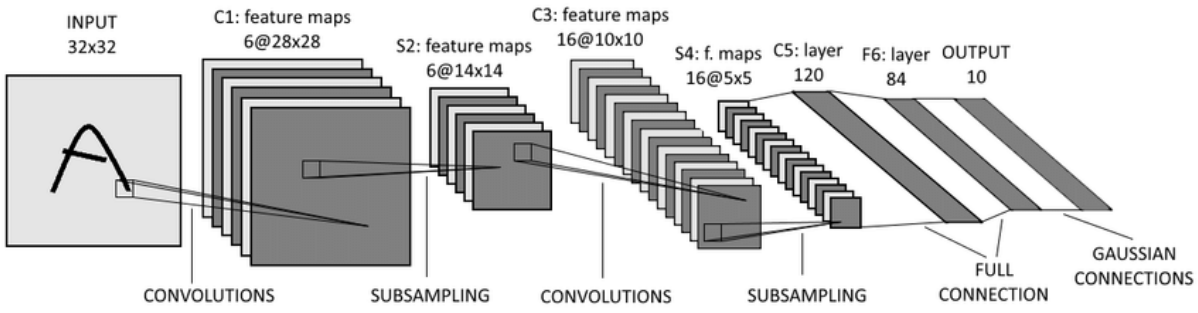


*Figure 7. Architecture of LeNet-5. Source: image taken from **[13]***

2.1.2. CNN architectures for Computer Vision

AlexNet

The so-called AlexNet architecture was introduced by Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton **[10]**. This neural network, with 60M parameters and 650,000 neurons, achieved the best performance in the 2010 ImageNet LSVRC-2010 contest, consisting on classifying 1.2M high-resolution images into 1000 different categories.

The architecture, consisting of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax is shown in **Figure 8**.

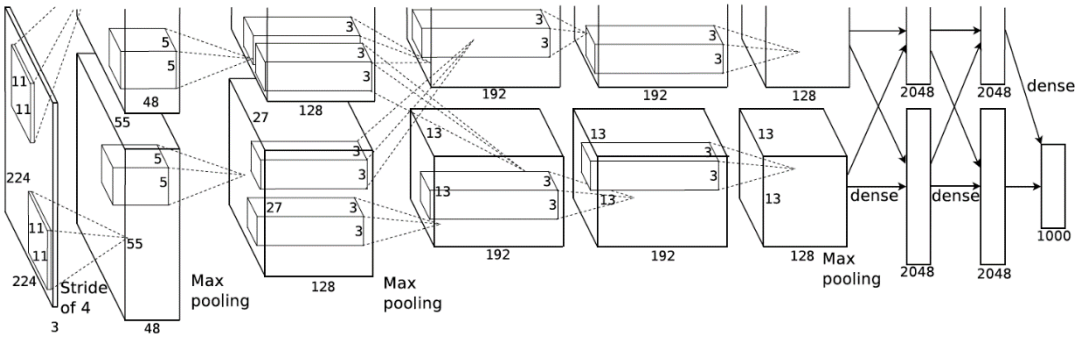


*Figure 8. AlexNet architecture. Source: image taken from **[10]***

This architecture has been used in one of the most influential studies in visual emotion analysis **[17]**, providing accurate results. The approach taken consists of pre-training the network with the ImageNet dataset (for the image classification task), and then fine tuning it with an image emotion recognition dataset. This publication also introduced the largest image emotion dataset publicly available.

VGG-16

Introduced by Karen Simonyan and Andrew Zisserman **[18]**, with affiliation to Google DeepMind. The architecture, shown in **Figure 9** and **Figure 10**, consists of thirteen convolutional layers, five max-pooling layers, two fully-connected layers and a final 1000-way softmax. The innovation with respect to previous models, consists on a much larger architecture depth, which is feasible due to the use of very small (3x3) convolution filters, resulting in around 133M parameters.

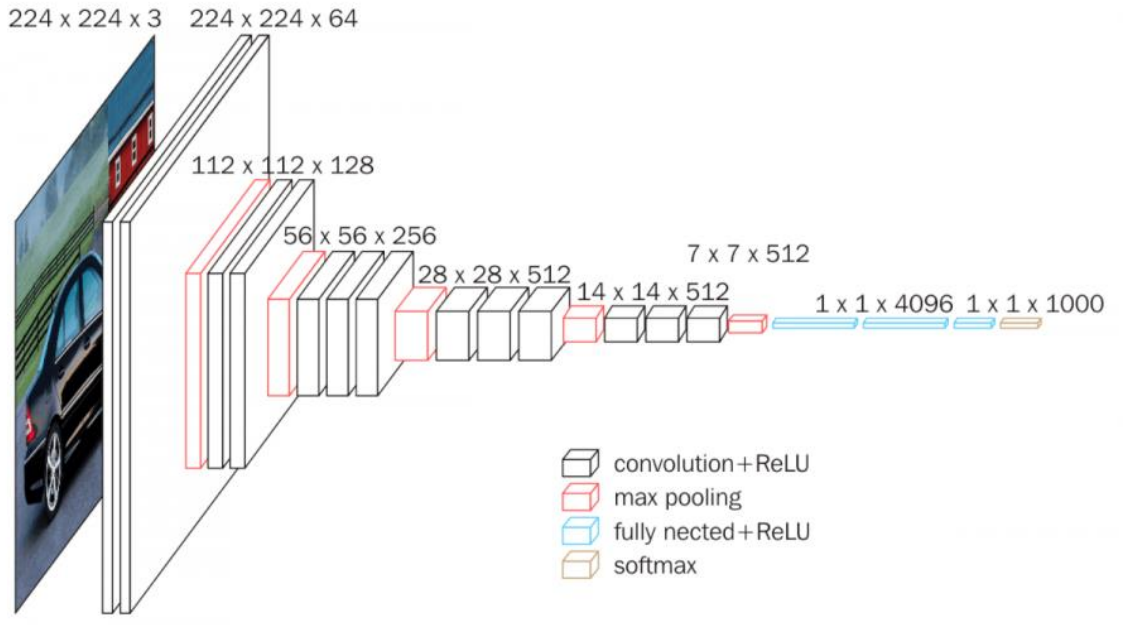


*Figure 9. VGG16 architecture (3D view). Source: image taken from **[19]***

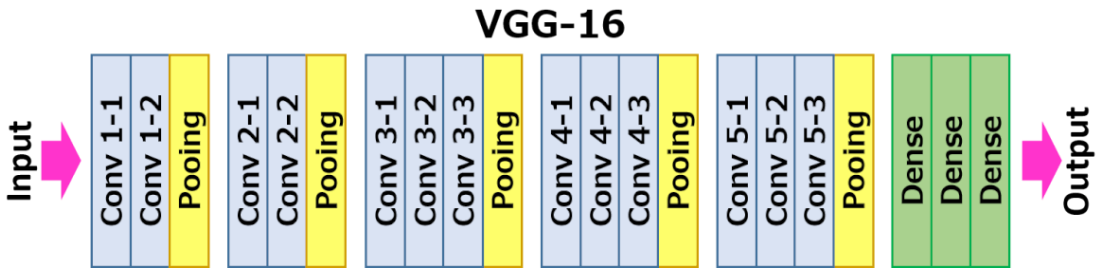


*Figure 10. VGG16 architecture. Source: image taken from **[19]***

A common variation of this architecture is the so-called VGG-19, which includes 19 layers with parameters (instead of only 16).

There are two main drawbacks to these architectures:

- A large amount of computational power is needed to train it

- The weights themselves are quite large

2.1.3. Interpretability of CNNs

In the rise of the surprising performance of CNNs when classifying images, there is not a clear understanding of their functioning and why they perform so well. A lot of efforts have been carried out to understanding what drives the classification decision of deep learning model **[20]**.

In **[21]** the authors introduce the use of Class Activations Maps (CAMs hereon) as a method to obtaining an insight into the functioning of intermediate feature layers and the operation of the classifier. The main role of this strategy is a diagnostic one, allowing Machine Learning researchers to better asses how they can improve their models.

CAMs allow us to spot the specific regions of an image that activated a specific class in our model, or in other words, which regions in the image were relevant to a specific class. Take **Figure 11** as an example. Say we are building a CNN that allows us to classify an image into 100 classes. If one of the classes is "dog", and we want to classify the image on the left, the class activation map for the "dog" class will look like the image on the right. This means that it is both the head of the dog and its lower back that are driving the CNN to classify this image as a dog. On the contrary, if we were looking at the CAM for the class "person", the image would have a concentration of red and yellow on the area of the head of the kid.

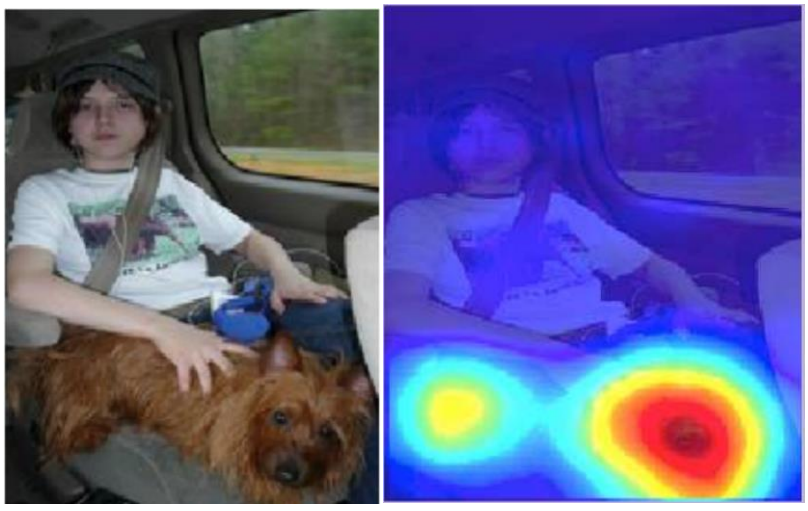


*Figure 11. Class Activaiton Map for the class "dog". Source: image taken from **[22]***

The mathematical principle, shown in **Figure 12**, is very simple. It consists on computing how much of the final score on the class "dog" (which represents the probability that the classifier assigns to the image being a dog) comes from the initial pixels of the input image.

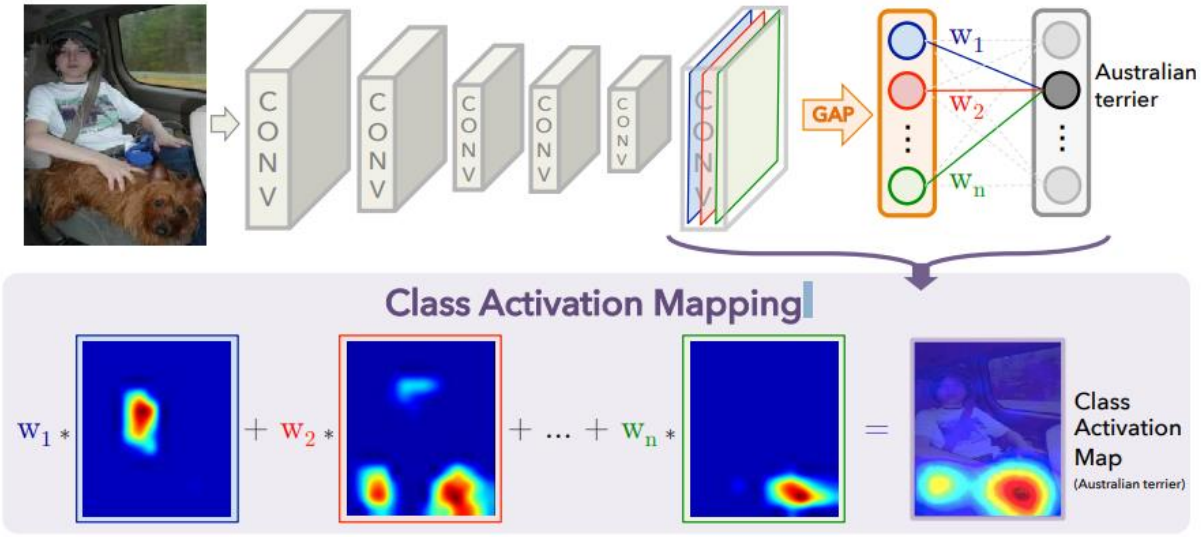


*Figure 12. Computation of a Class Activation Map. Source: image taken from **[22]***

2.2. GANs

2.2.1. Motivation

Up until recently, the most important successes in deep learning have involved discriminative models, which consist on mapping a high-dimension input (image, text, audio, etc.) to a class label. Deep generative models have had less impact, due to the difficulty of applying the existing models and methods to generative tasks.

Although their applications do not seem straightforward, there are different reasons for which studying generative models is useful **[23]**:

- Training and sampling from generative models allows us to put into evaluate and put into practice our ability to represent and manipulate high-dimensional probability distributions, which are important objects in a wide variety of applied math and engineering domains.

- Generative models can be leveraged in reinforcement learning problems. There are two high-level categories of approaches to reinforcement learning: model-free and model-based, with the latter containing a generative model. In this use cases, the role of generative models is to simulate possible futures that can be fed to the reinforcement learning algorithm for training. For a recent example of such a model, see **[24]**.
- Generative models can be trained with missing data and can provide predictions on inputs that are missing data. A particular use case in which this property can be applied is semi-supervised learning, where labels from a large percentage of the training examples are missing. While modern deep learning algorithms require very precisely labeled data in order to generalize well outside of training examples, semi-supervised learning allows to reduce the number of labels. GANs can be used for semi-supervised learning **[25]**.
- In some Machine Learning problems, a single output may have multiple correct answers, also known as multi-modal outputs. Some traditional algorithms to train machine learning models, such as minimizing the mean squared error (MSE) between the correct output and the model's output, are not able to train models that can produce multiple different correct answers. However, GANs enable ML models to work with these multi-modal outputs. A relevant example is using a model to predict the next frame of a video sequence **[26]**. **Figure 13** shows how the MLE model provides an answer where all potential solutions that can be right are overlapped. On the other hand, the the generative-adversarial model understands that there are many possible outputs, each of which must be recognizable as a realistic image.

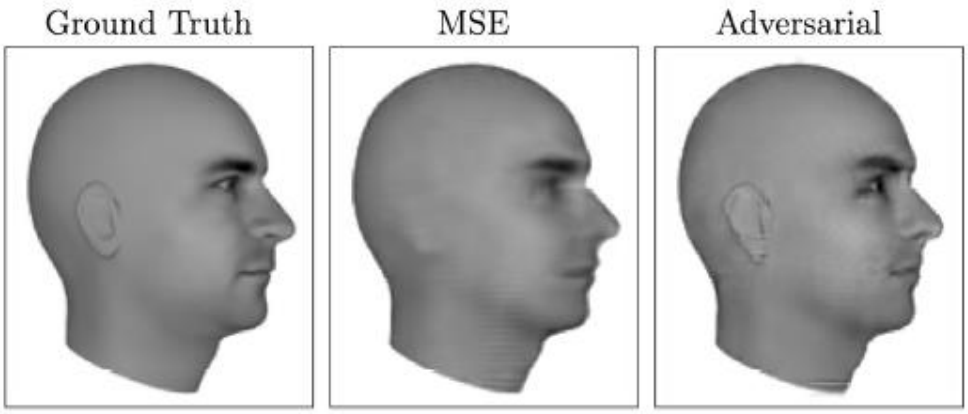


*Figure 13. Predicting the next frame. Source: image taken from **[23]***

- Finally, a lot of tasks may require the generation of realistic images from a specific probability distribution. An example is taking a low-resolution image and generating a high-resolution counterpart.

*Figure 14. Obtaining the high-resolution counterpart from a low-resolution Image. Source: image taken from **[27]***

In **Figure 14**, The leftmost image is an original high-resolution image. It is then down sampled to make a low-resolution image, and different methods are used to attempt to recover the high-resolution image.

2.2.2. Generator vs Discriminator

The base principle of Generative Adversarial models is to have two networks that are trained competing against each other. One of them is the **Generator**, which will be the

one creating samples of data that are intended to come from the same distribution as the training data. On the other hand, the **Discriminator**, receives as input a sample and classifies it as real (coming from the real probability distribution) or fake (coming from a different distribution).

A recurrently used metaphor **[23]** consists of thinking of the generator as a counterfeiter, whose goal is to create fake money that appears to be realistic enough. On the other hand, the discriminator is like the police, whose goal is to allow legitimate money to be used and catch counterfeit money. To succeed in this game, the counterfeiter must learn to make money that is indistinguishable from genuine money. This is translated into the generator network having to learn to create samples that are drawn from the same distribution as the training data. **Figure 15** illustrates this process.

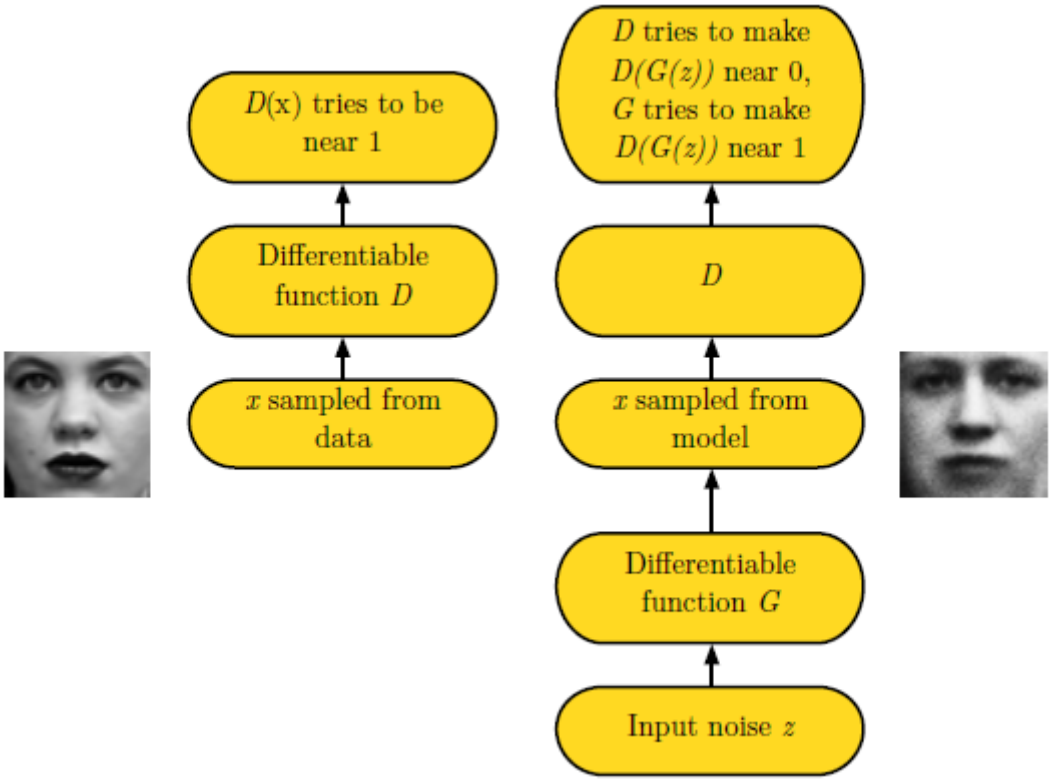


*Figure 15. The GAN framework. Source: image taken from **[23]***

In the Computer Vision field, a common application is to get the generator to create images of a specific type. As an example, in **[28]**, authors use a GAN to create realistic images of birds.

Both the Generator and Discriminator can be seen as a function that must be differentiable with respect to both the parameters and the input. Therefore, the generator can be described as a function G that takes z as input and uses $\theta^{(G)}$ as parameters. On the other hand, the discriminator is a function D that takes x as input and uses $\theta^{(D)}$ as parameters.

Let's have a closer look at each of the components.

Discriminator:

The discriminator's job consists of a binary classification problem, where there are two potential outputs: real and fake. As such, any common network architecture for classification problems may be used. In the next section, we will study its cost function.

Generator

Typically, a deep neural network will be used for the generator. When **z**, the input of the generator, is sampled from some distribution, G(**z**), where G() is the generator function, will provide a sample **x** drawn from the probability model that the generator is approximating. There is a lot of freedom in terms of which architecture to use for G. The main requirement is that G has to be differentiable.

2.2.3. Training GANs

As previously mentioned, both networks (Generator and Discriminator) are trained in parallel for their respective goals. As such, in one training step, two batches of data will be sampled: a batch *x* of training data from the dataset, and a batch *z* of values drawn from the generator model's prior **[23]**. Then, two gradient steps take place. One will be the generator updating its parameters to reduce its cost function, and the other one will be the discriminator updating its parameters to reduce its cost function.

To better understand the process, lets have a look at the cost functions of each member:

Cost functions

For all the different GANs currently used, the cost function of the discriminator is always the same. On the other hand, the cost of the generator varies more often.

- Discriminator's Cost Function

The discriminator will use a standard cross-entropy cost that is used for binary classifiers with a sigmoid output.

$$J^{(D)} = \underbrace{-\frac{1}{m_{real}}\sum_{i=1}^{m_{real}} y_{real}^{(i)}.\log(D(x^{(i)}))}_{\substack{\text{cross-entropy 1:} \\ \text{“D should correctly label real data as 1”}}} \underbrace{-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}(1-y_{gen}^{(i)}).\log(1-D(G(z^{(i)})))}_{\substack{\text{cross-entropy 2:} \\ \text{“D should correctly label generated data as 0”}}}$$

*Equation 1. Cost function of the Discriminator. Source: equation taken from **[29]***

The difference with a traditional cross-entropy cost is that this cost function is trained using two different batches of data. The first batch, which contains samples from the real dataset (real images in the case of Computer Vision) is of the form $(x^{(i)}, {y_{real}}^{(i)})$. The second batch, that is coming from the generator, contains samples of the form $(G(z^{(i)}), {y_{gen}}^{(i)})$, where G is the generator function, that outputs a sample.

In **Equation 1**, ${y_{real}}^{(i)}$ is always 1, as these samples come from the real dataset. Similarly, ${y_{gen}}^{(i)}$ is always 0, as these samples are fake and come from the generator.

- Generator’s Cost Function

As has been mentioned, the generator can have different cost functions. Since the goal of the generator is to “fool” the discriminator, the most straightforward version consists of assigning it the opposite cost function as the discriminator.

$$J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} \log(1-D(G(z^{(i)})))$$

*Equation 2. Cost function of the Generator. Source: equation taken from **[29]***

Note that in this case, there are no ${y_{real}}^{(i)}$ and ${y_{gen}}^{(i)}$ components, as was the case with the discriminator’s cost function. This is because the generator only receives the *z* batch. If we take the cross-entropy 2 of **Equation 1**, set ${y_{gen}}^{(i)} = 0$, and change the sign, we get to the discriminator’s cost function (as expected).

The first modification that can be introduced for the generator cost function consists of using a non-saturated cost, as shown in **Figure 16**.

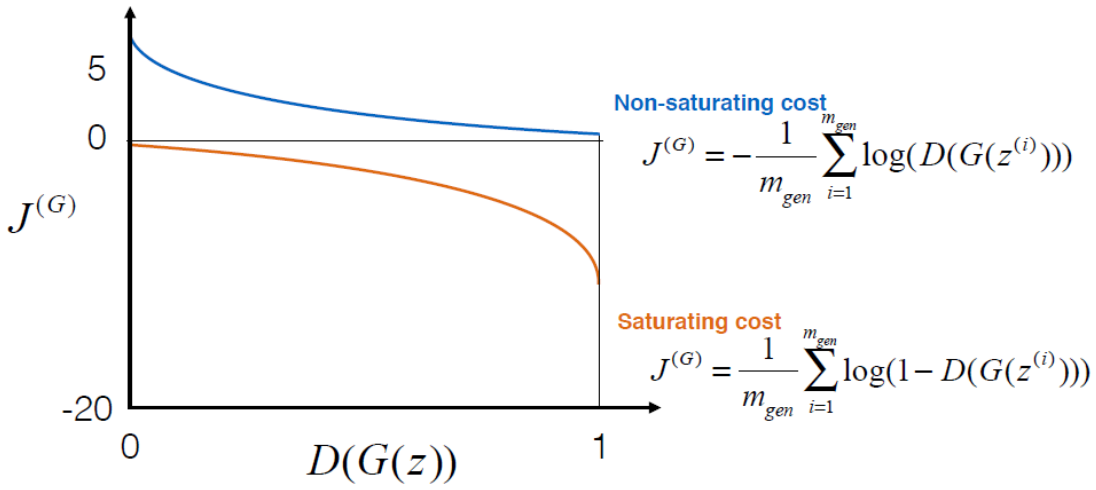


*Figure 16. Non-saturating cost of the generator. Source: image taken from **[29]***

As we can see in **Figure 16**, if the output of the discriminator $D(G(z))$ is close to 0, which means that the discriminator is correctly classifying the output of the generator as a fake sample, the gradient of the cost of the generator is also close to 0. This is not what we want. Early on, the generator will perform badly, and the discriminator will most of the times "catch" its outputs as fake, so we will be in the area where $D(G(z))$ is close to 0. We want to train the generator as fast as possible, and with a cost function close to 0, the training will be slow.

For this reason, we prefer the non-saturating cost from **Figure 16**, which was introduced in **[23]**. Although we will be optimizing for the same thing, as shown on **Equation 3**, the shape of this cost function provides a faster training of the generator early on.

$$\min\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D(G(z^{(i)})))\right] \Leftrightarrow \max\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)})))\right] \Leftrightarrow \min\left[-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)})))\right]$$

*Equation 3. Equivalence between saturating and non-saturating cost functions of the generator. Source: equation taken from **[29]***

These are just two examples of potential cost functions to use. The current literature offers a lot more forms for the generator loss, as well as for the discriminator's, as shown on **Table 1**.

GAN	Discriminator Loss	Generator Loss
MM GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{GAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$
NS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{NSGAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{NSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[\log(D(\hat{x}))]$
WGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} = -\mathbb{E}_{x\sim p_d}[D(x)] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
WGAN GP	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGANGP}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_g}[(\|\|\nabla D(\alpha x + (1-\alpha\hat{x})\|\|_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGANGP}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
LS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{LSGAN}} = -\mathbb{E}_{x\sim p_d}[(D(x)-1)^2] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{LSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[(D(\hat{x}-1)^2]$
DRAGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{DRAGAN}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_d+\mathcal{N}(0,c)}[(\|\|\nabla D(\hat{x})\|\|_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{DRAGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1-D(\hat{x}))]$
BEGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{BEGAN}} = \mathbb{E}_{x\sim p_d}[\|\|x-\mathrm{AE}(x)\|\|_1] - k_t\mathbb{E}_{\hat{x}\sim p_g}[\|\|\hat{x}-\mathrm{AE}(\hat{x})\|\|_1]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{BEGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\|\|\hat{x}-\mathrm{AE}(\hat{x})\|\|_1]$

*Table 1. Generator and discriminator loss functions. Source: table taken from **[30]***

The main difference among the functions shown above resides on whether the discriminator outputs a probability (MM GAN, NS GAN, DRAGAN) or its output is unbounded (WGAN, WGAN GP, LS GAN, BEGAN); whether the gradient penalty is present (WGAN GP, GRAGAN) and where is it evaluated.

Frequency of training

There is one last aspect to explore regarding the training process of GANs, and it is that of the frequency of training of each participant **[23]**.

The generator and discriminator will be trained by competing against each other: the G trying to fool the D, and the D trying not to be fooled by the G. This means that ideally they should have similar levels of performance. If, for example, the D is way better than the G and never gets fooled, there will be an unbalance.

As a general rule, the discriminator is trained more often than the generator, for two main reasons:

- The discriminator sets the upper bound of how good the generator can be
- The generator learns faster when the discriminator is a better performer, because of the shape of non-saturating costs shown on **Figure 16**.

2.2.4. State-of-the-Art Architectures

Generative models are much more difficult to train and converge than discriminative models **[28]**. However, there are some GAN architectures that have been proven to work well on certain use cases. For that reason, it can be very valuable to use well-known models from the literature rather than building a model from the ground up.

Good-performing architectures include:

DCGAN

Introduced in 2015 by Alec Radford and Soumith Chintala, from Facebook's AI research division **[14]**, this CNN-like architecture was developed with the goal of improving unsupervised learning with CNNs, such as clustering data and leveraging the clusters for classification scores.

This new structure provides top-of-the-class stability in its training in most settings.

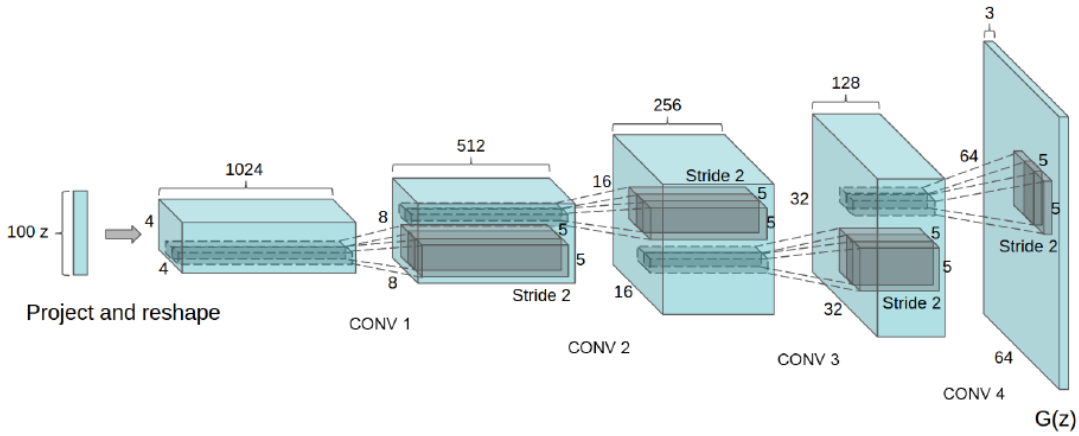


*Figure 17. Generator architecture of DCGAN. Source: image taken from **[31]***

StackGAN

Synthesizing high-quality images from text descriptions is a challenging problem in computer vision. Introduced in 2016, StackGans generate 256x256 photo-realistic images conditioned on text descriptions **[32]**.

Their innovative approach consists on breaking down this difficult problem into two components, carried out by two different networks:

- Stage-I GAN: sketches a low-resolution image from the text description
- Stage-II GAN: takes the Stage-I results and the text descriptions, and generates a high-resolution output.

As reference, the architecture of the StackGAN is shown on **Figure 18**. However, it uses techniques such as conditioning augmentation, compression and spatial replication, etc. which exceed the scope of this document.

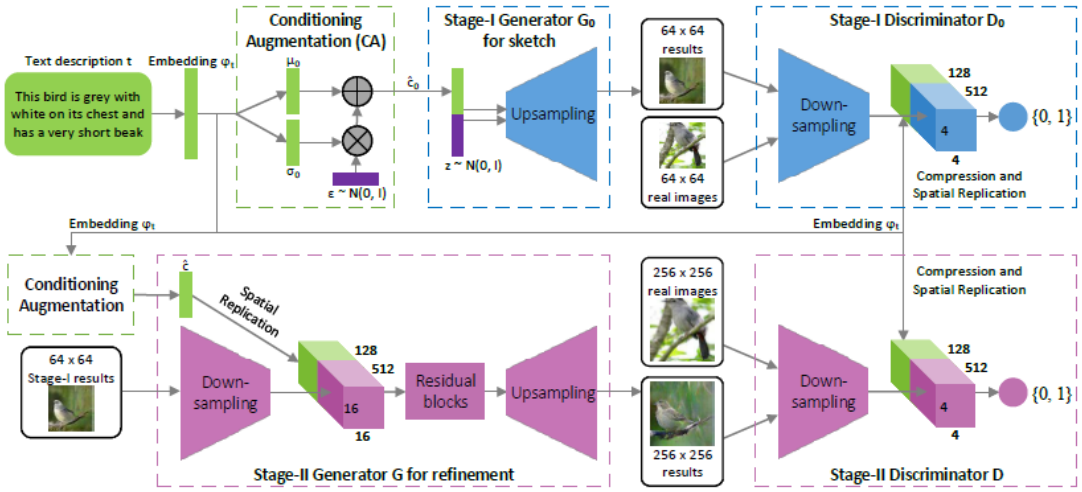


*Figure 18. StackGAN architecture. Source: image taken from **[32]***

2.2.5. Notable Results

This section is aimed at showcasing the potential of GANs when a converging architecture is found.

Operation on codes

As previously explained, the generator is fed a random vector ***z*** and it outputs, in the case of computer vision, a generated image ***x***. In **[31]**, the authors prove that if an arithmetic operation is carried out between vectors in the Z space, these will result in a similar operation in the image space X.

An example is shown in **Figure 19**. Say z_1 is a code that when fed to generator G, results in the creation of the image of a man with glasses; z_2 generates a man without glasses

and z_3 generates a woman without glasses. When we perform the operation $z_1 - z_2 + z_3$, and we run the result through the generator, it results in a woman with glasses, which is the logical result of the operation "man with glasses" – "man without glasses" + "woman without glasses".

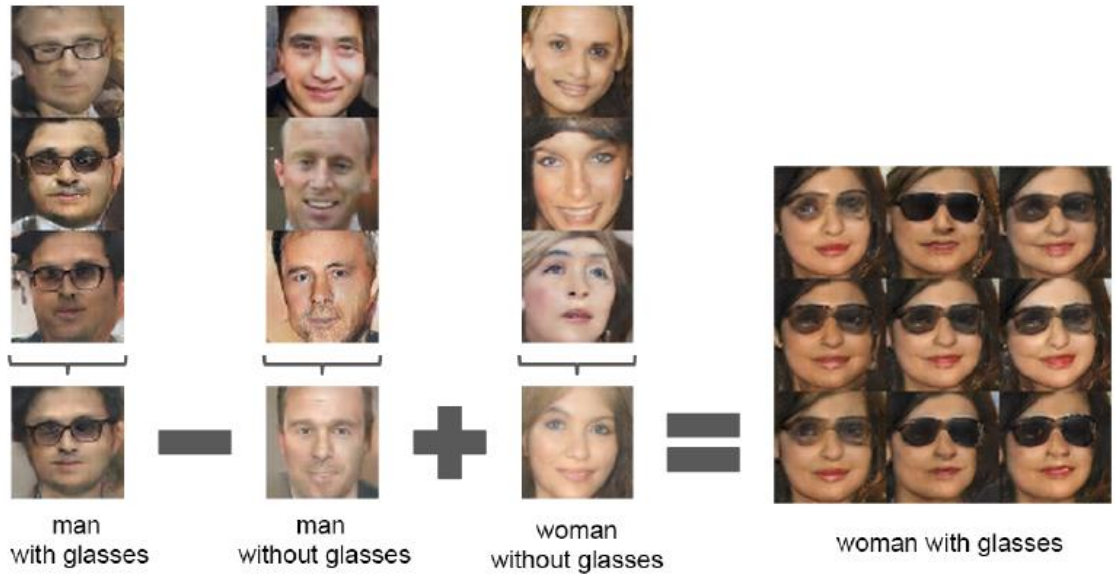


*Figure 19. Vector arithmetic for visual concepts. Source: image taken from **[31]***

Image Generation

Generative Adversarial Networks face challenges in generating high quality images. In **[28]**, the authors use a StackGAN architecture to generate photo-realistic high-quality images. Some results are shown in **Figure 20**.

*Figure 20. High-quality images generated by a StackGAN. Source: image taken from **[28]***

Super-resolution Image

In **[27]**, the authors present SRGAN, a generative adversarial network for image super-resolution. This is the first model capable of inferring photo-realistic images with a 4X upscaling factor.

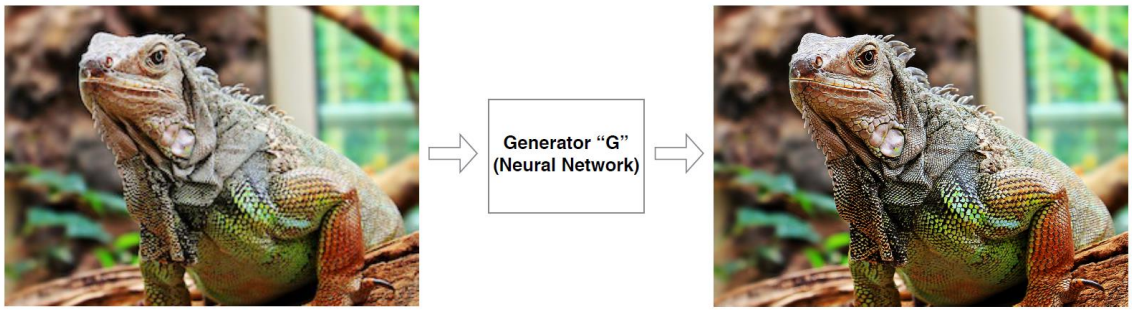


*Figure 21. Super-resolution image (right) from a lower resolution photograph (left). Source: image taken from **[27]***

3. Dataset and Features

3.1. Dataset description

This work uses a dataset of 40,000 artistic images, provided by the project sponsor **[12]**. Most of the artwork is medieval with highly subjective labels. In theory, each image contains the following features, most of them categorical:

- Title
- Style: realism, romanticism, impressionism…
- Artist
- Genre: landscape, portrait, religious painting…
- Tag: any explanatory information of the painting
- Emotion: neutral, optimism, joy, sadness…

However, further analysis showed that only 3,000 of the 40,000 artistic images were labeled by emotion, which is the label of interest in the context of the project. The number of classes for each categorical distribution is shown in **Table 2**.

Categorical Feature	Number of classes
Style	73
Artist	398
Genre	37
Tag	1362
Emotion	16

Table 2. Classes for each categorical variable. Source: own elaboration

3.2. Data cleaning and manipulation

The provided dataset was not in workable form, as it consisted on a xlsx file containing a list of URLs pointing to the corresponding image. All of the data cleaning and data analysis work, which was a big component of the project, is presented in **Annex A**.

As described in **Section 1.1**, a painting may transmit different emotions to different people, so there are no ground truth labels for our data as there would be for a traditional classification task. Therefore, the consistency of the labels is very important. With this intuition, a qualitative analysis of the images was carried out to assess the label consistency, which revealed that the rich number of emotions resulted in overlapping categories. An example is shown in **Figure 22**, where the image on the left represents optimism, while the image on the right represents joy. Given that these images would be difficult to classify as such by a human observer (which can be seen as the ground truth), the models would not provide good results.

Figure 22. Optimism (left) and Joy (right) are an example of the inconsistency of the provided labels. Source: own elaboration.

Therefore, the provided 16 emotion classes are grouped into 4 main categories. Data from each of the four categories is shown in **Figure 23**.

Figure 23. Four grouped classes from left to right: 0:[Sadness, Fear, Disgust, Anger], 1:[Neutral, NA], 2:[Lust, Envy, Surprise], 3:[Optimism, Joy, Love]. Source: own elaboration.

After some unsuccessful preliminary models, the data was filtered to only portraits. This decision was motivated by error analysis using Class Activation Maps, which will be

explained in depth in **Section 2.1.3**. This resulted in only 500 labeled artworks for the classifier, which is too small of a dataset to fit a deep learning model. Therefore, this small dataset was augmented to 2,000 using flips and rotations, but not blur because it would affect the texture of the painting, which is an important feature to detect the emotion, as it will be explained in **Section 4.1.4**.

Another learning extracted from the qualitative analysis is that, while some emotion labels seem to be driven by objects present in the painting (e.g. a painting with a baby labeled as "Optimism"), others are defined by global features such as the color, texture, or brightness (e.g. a dark painting of a landscape being labeled as "Sadness"). This effect is shown in **Figure 24**. This observation drove the decision to build different models focusing on both local and global features, as will be explained in **Section 4.1.4**.

Figure 24. Labels for the first two images from the left driven by local features (classes 3 and 1 respectively) vs. two images on the right driven by global features (classes 1 and 3 respectively). Source: own elaboration.

For GANs, all unlabeled artworks were used. Notice that, as it does not consist on a supervised learning project, no labels are needed. As will be explained in **Section 4.2**, in order to obtain more meaningful results, we will train different models with different subsets of the data.

As the data is provided as an xlsx file containing a list of URLs pointing to the corresponding image, a pipeline to obtain this images in workable form must be built. To do so, every image was downloaded with its name being the "base64 encoding" of the painting's title. After having downloaded the whole dataset in such a format, each code was decoded, and each title was matched with the specific painting to add the rest of the features (genre and emotion). As an example, the painting with the title: 'The Beethoven Frieze: The Longing for Happiness Finds Repose in Poetry. Right wall' was downloaded with the following image name:

VGhlIEJlZXRob3ZlbiBGcmllemU6IFRoZSBMb25naW5nIGZvciBIYXBwaW5lc3Mg
RmluZHMgUmVwb3NlIGluIFBvZXRyeS4gUmlnaHQgd2FsbA==

Then, that code would be decoded back to the original title. Having the title of the painting, the value of the rest of the features were looked up in the original xlsx file and added to the element. Again, see **Annex A** for more detail on the process.

Finally, in order to train a model, it is a requirement that the input images have a standardized size. As shown in **Annex A**, function **resize_things(things, dir_in, dir_out)** is created to resize all images into 244 x 224 x 3.

4. Implemented Models and Analysis of Results

The performance results of each model informed the architecture of the next model. Therefore, this section will present the different models in the order in which they were implemented during the study, commenting the results and limitations of each. First, we will focus on the classification problem. Then, on the art generation with GANs.

As a preliminary note, the models were written in Jupyter Notebooks[2], included in **Annexes A through E**. The main reason to do this was because it allows for an easier execution on cloud infrastructure. **AWS** was the infrastructure provider chosen to run the models.

4.1. Classification

One of the best practices when approaching a deep learning problem, is to start with a basic model to obtain the baseline results **[33]**. Then, work iteratively, creating more and more complex models and tuning the hyperparameters in the process. This makes it easier to make sure that your model is advancing in the right direction.

Following this approach, we started with a shallow CNN to have some baseline results, and then created more complex models on an iterative manner.

4.1.1. ShallNet – A shallow CNN

At the beginning of the study, a smaller subset of data was provided by the project sponsor **[12]** while they worked on labeling and preparing the full dataset. Out of this subset, only around 700 images had emotion labels.

This was an added obstacle that contributed to the decision to start with a simpler model. The goals of this first model was to utilize this subset of data to obtain insights that would

[2] The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text

drive the implementation of more complex models down the line. The main insights we were seeking to find were:

- How does the model fit the images? Does it "look" at local features (i.e. the objects in the painting) or global features (i.e. the color and brightness).
- Does the model generalize well, or is it overfitting the training examples without really identifying emotions?

With this intuition, a 3-layer CNN model, that was named "ShallNet", was built from scratch, as can be seen on **Annex E**.

Since the dataset contained only 700 images with 12 possible emotion categories, it would be complicated to obtain a robust model. Therefore, a decision was made to group all of these emotion categories into two high-level groups:

- Positive: optimism, joy, love
- Negative / neutral: sadness, fear, disgust, anger, neutral, lust, envy, surprise, Nan

Once this grouping was complete, the model was built. As can be seen on **Page 7** of **Annex E**, it consists of 2 convolutional layers (each of them followed by a maxpool layer) and a fully connected layer. The Tensorflow[3] library was used to build this model, due to the easiness and speed of implementation it offers for convolutional models.

The initial experiments did not provide good results, even in terms of the accuracy in the training set, the main reason being thought to be that there was a large variety of types of paintings (the Genre feature) and not enough datapoints. Therefore, another design decision was made: to only use a subset of paintings that had a similar enough form. There were 94 "portrait" paintings, so this category was chosen. See **Pages 2-3** of **Annex E** for more information on the data filtering process.

The new iteration with this subset of data provided the results on **Table 3**. The results are considerably positive. However, notice that there was a larger proportion of datapoints with the "negative" label (around 70%), so the model is partially overfitting towards this label (although not completely, as it was demonstrated by the follow-up error analysis).

[3] TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks.

Figure 25 shows the evolution of the cost function, which shows a rapid descent and later stabilization. The model was let to run for 500 epochs, but no significant improvement was observed after epoch 200.

Data	Accuracy[4]
Training dataset (70 paintings)	81.4%
Test Set	83.3%

Table 3. Accuracy results for ShallowNet. Source: own elaboration ***(Annex E)***

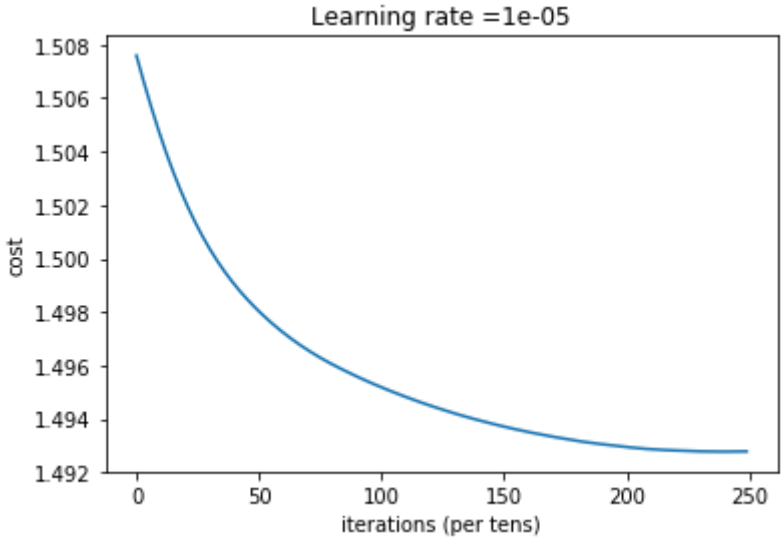


Figure 25. Cost computed per epoch. Source: own elaboration ***(Annex E)***

This model had its clear limitations from the beginning, but its implementation was a good warmup exercise to get familiar with potential issues that may arise throughout the study. One of the intuitions extracted is that, even with the full dataset, it would be complicated to train a deep learning model from scratch. This informed our next architecture.

4.1.2. VGG16+TL version 1

Because of the difficulty to train a deep learning model from scratch with such few datapoints, Transfer Learning was used in the next architecture.

[4] The accuracy is computed as the percentage of correct predictions. Since it is a binary classification problem, a classifier that assigns classes at random would have a 50% accuracy, which can be seen as the baseline.

Transfer Learning is defined as the improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned **[34]**. In other words, it consists of using as a baseline a machine learning model that has already been trained on a large dataset. Then, this model is tuned with the dataset of interest. This method will provide good results as long as the original dataset in which the model was pre-trained and the dataset of interest are similar enough.

This is a common practice in Computer Vision. The initial layers of the neural network work as a "feature extractor", which means that they detect features such as an eye, a nose and a mouth. The later layers of the network, if it is trained as an object detector, will put these features together to determine it is a human face. Therefore, there is value in having a pretrained network that already extracts features, and then training the last layers of the network to do whatever classification needed.

The VGG16 architecture **[19]** was used, pretrained with the imageNet dataset **[10]** of 15 million images belonging to roughly 22,000 categories. This approach still raised some questions, namely:

- Artistic paintings can be very different from the real images from imageNet, so the pretrained network may not be as effective in extracting features.
- The emotion of a painting does not depend solely on local features.

We decided to "freeze" (i.e. not train) the parameters of the VGG16, replaced the last maxpooling layer with a global average pooling layer, and used one FC layer rather than two with a softmax on top. The resulting architecture is shown in **Table 4**, where only the two last layers were trained**.** We named this network "VGG16 + TL version 1".

For this architecture, we used the Keras[5] library rather than Tensorflow, as it provides access to a pre-trained VGG16 through a simple function. See **Pages 7-9** of **Annex D** for extended information.[6]

We refined the hyperparameters of the Adam Optimizer **[35]** used in our model: learning rate, β_1, β_2, ε, etc. This fine-tuning was driven by the model's performance, as well as

[5] Keras is an open-source neural-network library written in Python. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible.
[6] Notice that in **Annex D**, the printed results do not correspond to the best performing iteration and are only shown to prove the correct functioning of the code.

Class Activation Maps, which will be explained shortly. The evolution of the accuracy is shown in **Table 4.**

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
global_average_pooling2d_1 (	(None, 512)	0
FC (Dense)	(None, 512)	262656
SM (Dense)	(None, 1)	513

Figure 26. Architecture of VGG16 + TL version 1. Source: own elaboration ***(Annex D)****.*

Iteration	Cross Validation set accuracy
#1	31.5%
#2	36.6%
#3	38.1%
#4	39.3%

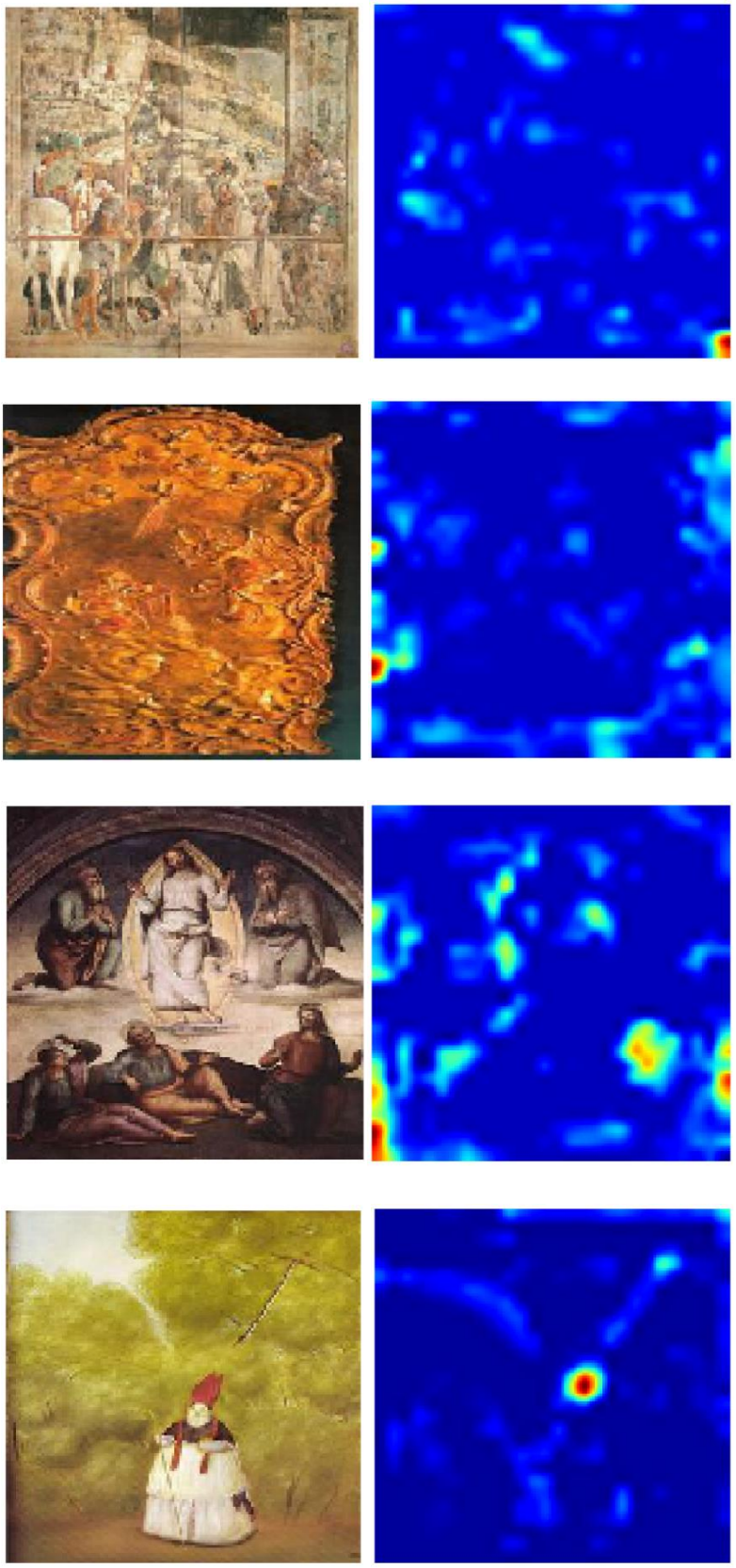
Table 4. Performance of the different iterations on the "VGG16 + TL version 1 model"[7]. Source: own elaboration ***(Annex D)****.*

Iteration number 4 used the following hyperparameters:

- Learning rate = 0.1
- β_1 = 0.9
- β_2 = 0.999
- ε = None
- Decay **[36]** = 0.0

When reflecting about the performance and limitations of this model in order to inform the next architecture iteration, we decided to use Class Activations Maps, explained in **Section 2.1.3.** This analysis was performed for a variety of paintings, as can be seen in **Figure 27**, some of them with multiple people, some with one person, and some with no people at all.

[7] Remember that the paintings are classified into one of 4 possible classes. Therefore, a model assigning classes at random would have an accuracy of around 25%



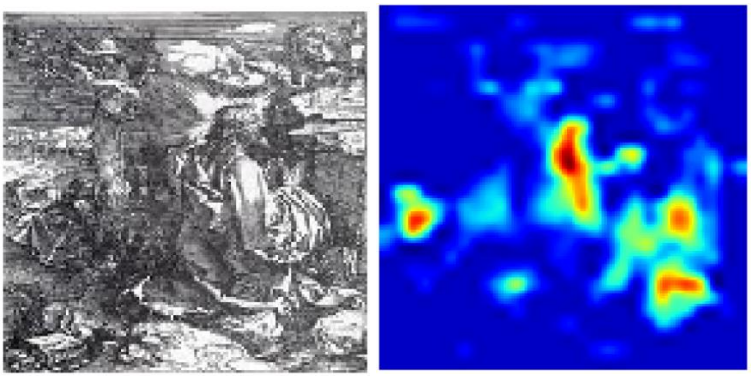


Figure 27. Class Activation Maps from different paintings. Source: own elaboration ***(Annex D)***

This analysis shows that, although some emotions are activated by local features that may be expected to do so (for example, by a face), the model is also overfitting noise, since some of the high gradient regions do not provoke any emotional reaction.

4.1.3. VGG16+TL version 2

The observations from the CAMS drove to the conclusion that labels are often driven by global features, such as color and texture, rather than local features. Therefore, the decision to filter the data by to only portraits, as explained in **Section 3.2**. This decision was driven by the fact that portraits have more consistent features, so the model should not overfit on as much noise as previously. We decided to rebuild a model using transfer learning from VGG16 again, and to train it only with this filtered data. This model is named VGG16+TL version 2.

Again, the hyper parameters were fine-tuned in a similar fashion to with the previous models. Here, we also experimented with "unfreezing" more parameters (in other words, training more layers), rather than just those in the last Fully Connected layer. However, after an extensive trial and error procedure, no other configuration performed better than training only the last layer, so we came back to the original structure.

Table 5 shows the accuracy results of the different iterations of this model.

Iteration	Cross Validation set accuracy
#1	51.8%
#2	54.5%
#3	56.1%
#4	57.2%

Table 5. Performance of the different iterations on the "VGG16 + TL version 2 model". Source: own elaboration ***(Annex D)****.*

This is a considerable improvement with respect to the previous model, demonstrating that it is easier to associate a feature from a portrait to an emotion. However, the error analysis through CAMs also revealed that the model was partially overfitting to noise, so there is still room for improvement.

At this point, a decision was made to explore alternative approaches to emotion classification. This exploration resulted in the model presented below.

4.1.4. CV+FC

Further research (which was presented in **Section 1**) was conducted regarding the study of emotion and images, both from Computer Vision and psychology fields.

There is a line of research which, instead of focusing on deep end-to-end models (as the ones created up to this point), explores hand-engineered features motivated by Computer Vision theory **[37]**. These features, based on the color, brightness and texture of the image, allow the model to associate global features to a particular emotion, which was one of the challenges we identified from our last models. With this intuition, supported by **[38]** and **[39]**, we decided to hand-engineer 13 different features that may be related to the transmission of emotions, including:

- 6 features representing the texture of the image
- 6 features representing the color of the image
- 1 feature representing the brightness

Let's have a closer look at the relevance of each parameter.

Color

Different colors make images evoke different emotions on people. For this reason, artists usually use different color combinations to express different emotions when creating their works of art.

[38] uses an innovative approach to color transfer between images, by extracting a "color palette" with the main colors from the reference image. We use this "color extraction" method to use this color map to determine the emotion that the image transmits.

The idea is to use a k-means clustering **[40]** where each pixel, which is an array of 3 elements (RGB color code) is a data point. See function "extractTopKColors(img_in, k)" in **Page 18** of **Annex D** for more detailed information. We decide to use 5 clusters **[36]**, which will provide the 5 "predominant" colors of the image. As an example, when the algorithm is given the painting on the left of **Figure 28**, it will return the color map on the right.

*Figure 28. Color map of a painting. Source: own elaboration **(Annex D)**.*

To test whether 5 main colors were enough to represent the color map of an image, we decided to print the same painting using this 5-color palette. The result, shown in **Figure 29**, indicates that the color palette provides a robust representation.

*Figure 29. Original painting (left) and main-colors-based painting (right). Source: own elaboration **(Annex D)**.*

Once this method to extract the main colors of a painting was developed, the strategy, motivated by **[38]** consisted on obtaining a set of standard color maps with at least one from each emotion category (note that, as explained in **Section 3**, there are 4 main emotion categories). As can be seen in **Annex D**, we extracted 6 color maps, for the following emotions: joy, love, fear, anger, sadness, surprise. These reference color maps are shown in **Figures 30-35**.

*Figure 30. Reference color map for "joy". Source: own elaboration **(Annex D)**.*

*Figure 31. Reference color map for "surprise". Source: own elaboration **(Annex D)**.*

Figure 32. Reference color map for "love". Source: own elaboration ***(Annex D)****.*

Figure 33. Reference color map for "fear". Source: own elaboration ***(Annex D)****.*

Figure 34. Reference color map for "anger". Source: own elaboration ***(Annex D)****.*

Figure 35. Reference color map for "sadness". Source: own elaboration ***(Annex D)****.*

Then, for a painting, the 6 color-related features are extracted as follows:

- Extracting the top 5 color centroids from the painting
- Computing the Euclidean distance between the example image and each of the 6 reference colors maps.
- Each of these Euclidean distances is an input for the model.

Texture

Although color transfer methods can transfer visual appearance between images by changing colors, they are usually not enough in and off themselves to change an image to meet an accurate emotion **[38]**. For this reason, emotion-transfer methods have started incorporating the texture of an image as an important factor to determine the emotion.

For this reason, we decided to include features about the texture of an image in our model. Similarly to the process with colors, the texture of an image will be conveyed to the model as the distance to a number of reference textures.

Regarding the reference textures, Perlin-noise-generated images were used **[44]**. This images will depend on a series of so-called "Perlin parameters" (number of octaves, frequency, persistence, lacunarity) that determine the visual appearance of the image **[39]**. The reference texture images are shown in **Figure 36**, where the Hard-Soft scale **[39]** was calculated. It can be seen that these reference images cover a large enough range of H-S values.

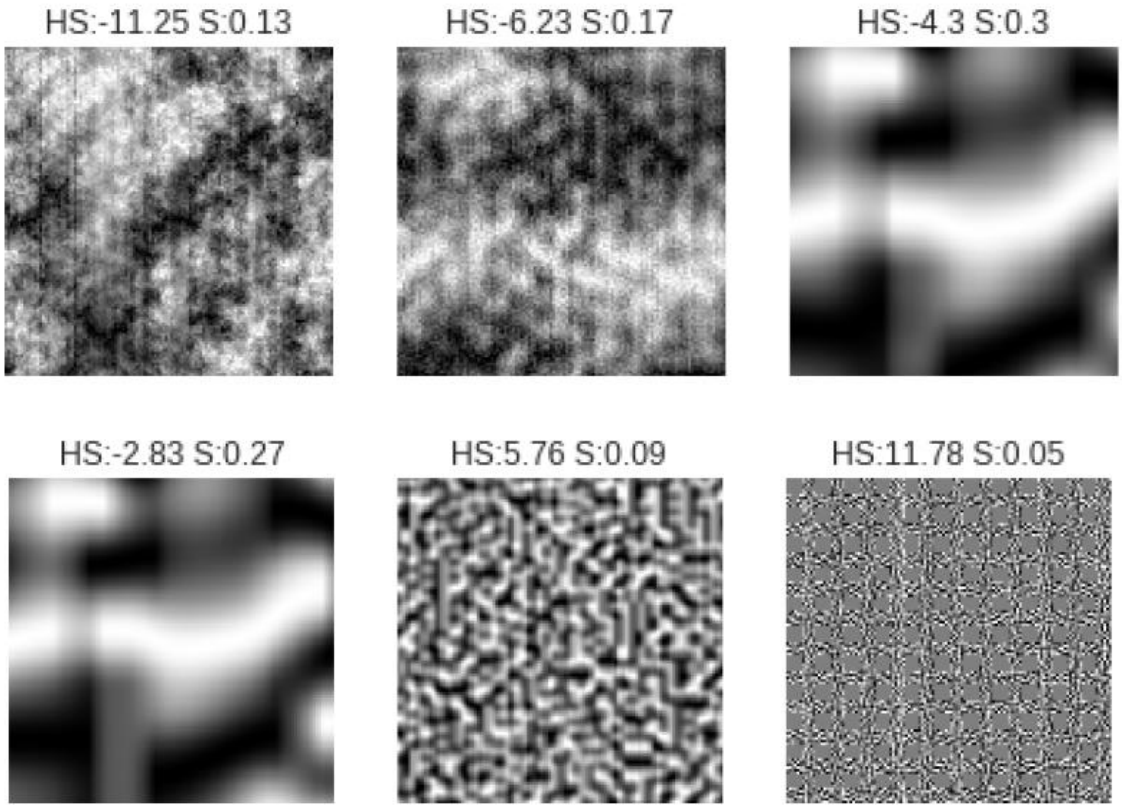


*Figure 36. Reference images for texture. Source: own elaboration **(Annex D),** based on the theory from **[39]***

Once the reference images have been established, the Structural Similarity (SSIM) Index between the original image to be classified and each of the six references are calculated. The SSIM index is a method for measuring the similarity between two images **[39]**. These six values will be used as inputs for the model.

Brightness

The final global feature that affects emotion, hence being included in the model, is brightness. A standard method is used to compute the image brightness, by calculating the square root of a linear combination of the RMS values for each color channel **[41]**.

Once the features are defined, the model architecture was built. Due to the reduced dimension of the input (13 features, as opposed to the 224 x 224 x 3 of the end-to-end deep learning model that receives the image as input), a shallow network was built. This model was named "CV+FC" (as it was based on Computer Vision theory and only contained Fully Connected layers). The architecture, shown in **Table 6**, is very simple, consisting on a 4-neuron layer followed by a softmax layer.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 13)	0
FC (Dense)	(None, 4)	56
SM (Dense)	(None, 1)	5

Total params: 61
Trainable params: 61
Non-trainable params: 0

*Table 6. Architecture of the CV+FC model. Source: own elaboration **(Annex D)**.*

After 10 epochs, the model returned surprisingly good results, as shown on **Table 7**. This demonstrated the relevance of global features when determining the emotion of an image.

Model Name	Accuracy
CV + FC	56.3%

*Table 7. Accuracy of the CV+FC model. Source: own elaboration **(Annex D)**.*

As a summary, **Table 8** shows the results of the best final model for all four architectures.

Model Name	Training Set Accuracy	Test Set Accuracy
ShallNet[8]	29.0%	27.6%
VGG16+TL version 1	45.1%	39.3%
VGG16+TL version 1	56.1%	57.2%
CV+FC	55.2%	56.3%

*Table 8. Comparison of the performance of the four models. Source: own elaboration **(Annex D)**.*

"ShallNet" performs worse since there is too few data to train a model from scratch. The VGG-16 pretrained models provide a very competitive accuracy, generalizing well to the test set. This demonstrates the value of utilizing a pretrained model when the amount of data is not abundant. Finally, the CV+FC model provides an accuracy almost as good as the VGG-16 models, also generalizing well to the test set. This result, better than expected, is a useful reminder that end-to-end deep learning models do not always perform best, and human input (such as the color, texture, and brightness of an image) can be extremely valuable.

[8] Note that the ShallNet model was initially trained grouping the emotions into only two categories (**Section 4.1.1.**). However, for the sake of consistency, when compering it to the other models, it was trained with the original grouping into four emotion categories.

4.2. GANs

Following the same approach as for the Classification task, we work iteratively, starting with a simple model and then creating more and more complex models and tuning the hyperparameters in the process.

A very simple architecture for both the generator and discriminator was created as a starting point. Then a more complex model was designed, driven by the learnigns from the initial model. Finally, a model extracted from the literature was utilized.

4.2.1. BasicGAN

The main advantage when working with GANs is that, as explained in **Section 3**, there is no need for the data to be labeled. This radically incremented the size of our dataset (since only 3k of the 40k images were labeled by emotion).

The baseline model, named "BasicGAN" (**Annex B**) was as follows:

- Generator: shallow network with three Fully Connected layers. It receives a 100-dimension random code (*z* in **Section 2.2**), and generates an array of dimension 224 * 224 * 3, which is the size of the image, through 3 Fully Connected layers.
- Discriminator: shallow network, with two Fully Connected layers, plus a dense layer with a sigmoid output (since it is a binary classification problem).

Due to the simplicity of the model, we limited the dataset to "portraits", with around 7k samples in total. By working with only portraits, the generator should do a better job at creating consistent images.

This model was trained for 175 epochs. The values used for the hyperparameters were the default ones:

- Batch size = 200
- Learning rate = 0.001

The Tensorflow library was used (**Annex B**).

The cost of both the generator and discriminator were printed, as shown in **Figures 37, 38**. These plots provide an insight on which of the two is being the limiting component in terms of learning (i.e. if the discriminator has a high cost and is not learning, it will slow down the learning of the generator).

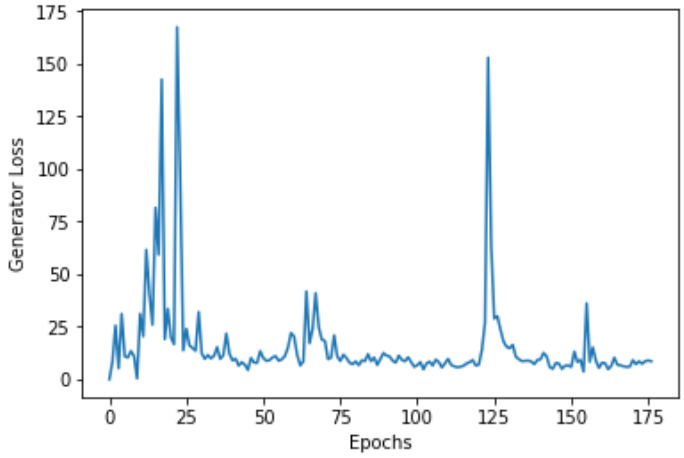


Figure 37. Generator Loss. Source: own elaboration ***(Annex B).***

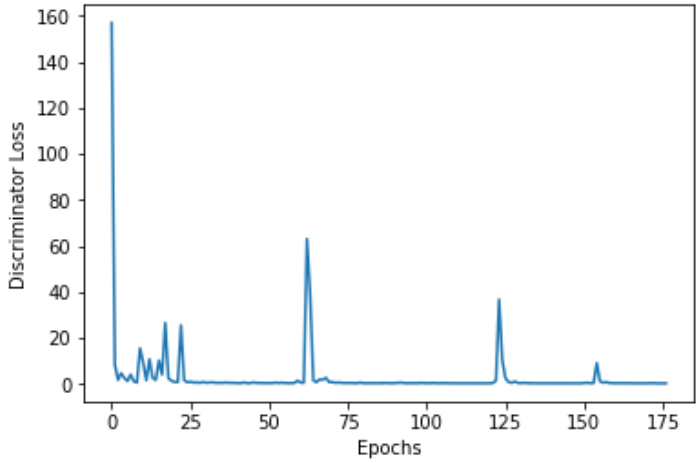


Figure 38. Discriminator Loss. Source: own elaboration ***(Annex B).***

As expected, these plots show that whenever the discriminator advances a "learning step" (the cost function peaks), this produces a "learning step" on the generator as well, as its task becomes more difficult. As an example, shortly before epoch 125 the discriminator's cost rises and is then reduced, and consequently, in epoch 125, the cost of the generator goes up.

Although these results provide some insight, they show that the model is clearly not learning well, since most of the time the costs are stable. In a generative-adversarial game, the cost of each member should be very "noisy", since both of them are constantly getting better and making the task more difficult for each other. Furthermore, the discriminator converges fast while the generator cost has more noise. This can be explained by the fact that the generator fails to trick the discriminator most of the times. We tried tuning the learning rates of both generator and discriminator, as well as training the generator more often, but the results were similar. Therefore, for the next iteration, a more complex architecture of the generator was the primary requirement.

Although there were no expectations for such a simple generator to create interpretable artworks, the general images were explored to try to identify any patterns. **Figure 39** shows two of the images that were generated. Despite the expected noise, there is a pattern (consistent across generated images) of a brighter shape in the center of the image. Since the GAN was trained with portraits, this pattern is reasonable, as at the center of a portrait there is a face, which is brighter than the rest of the image. The generator is trying to replicate this.

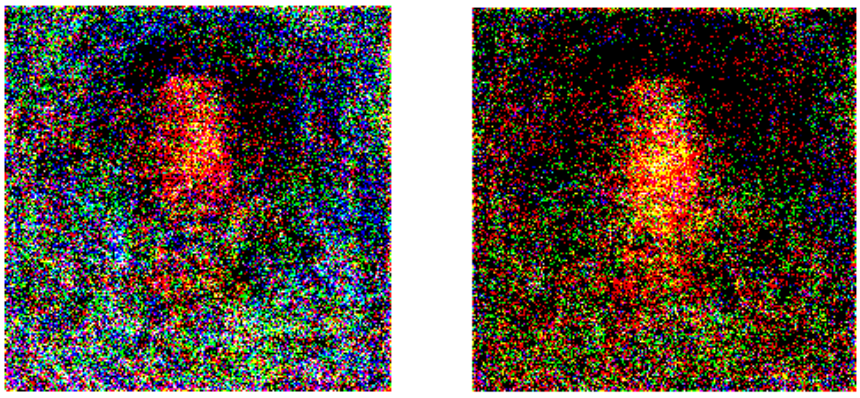


Figure 39. Image generated by the Generator. A brighter shape is seen in the center of the image can be associated to a face. Source: own elaboration ***(Annex B)****.*

The random noise at each pixel makes it clear that a convolutional architecture for the generator is needed, as it will provide more consistency throughout the image. This insight drove the development of the next model.

4.2.2. GANDeconv-Conv

Driven by the insights of the baseline model, in this new iteration both the generator and discriminator were designed with convolutions. We call this model "GANDeconv-Conv", and it is shown in **Annex C**. After some experimentation with the hyperparameters (notably the number of layers), the architectures of the generator and discriminator are as follows:

- The generator has 5 convolutional layers, followed by last tanh layer to make sure the output is standardized.
- The discriminator has 3 convolutional layers, followed by a max-pooling layer, and a fully connected layer in the end.

First of all, the models are trained with all artworks. Again, this makes it difficult for the generator to create consistent images that can beat the discriminator's filter. Some of the images generated by the generator are shown in **Figure 40**. Some patterns can be inferred: person-like shadows on the left image, and landscape-like colors on the right image (blue sky at the top, darker colors at the bottom). However, the images remain mostly noise-driven.

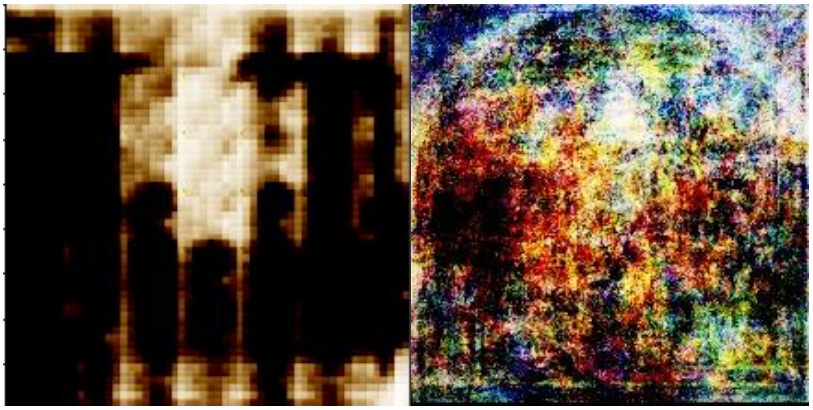


*Figure 40. Images generated by the Generator when trained with all types of artworks. Source: own elaboration (**Annex C**)*

Therefore, a filtering is carried out again to focus only on portraits, as it may make the task of the generator easier.

Again, the cost of both the generator and discriminator were printed, as shown in **Figures 41, 42**. The hope is that the discriminator will show more noise, indicating that it is learning and the generator is making its tasks more difficult progressively.

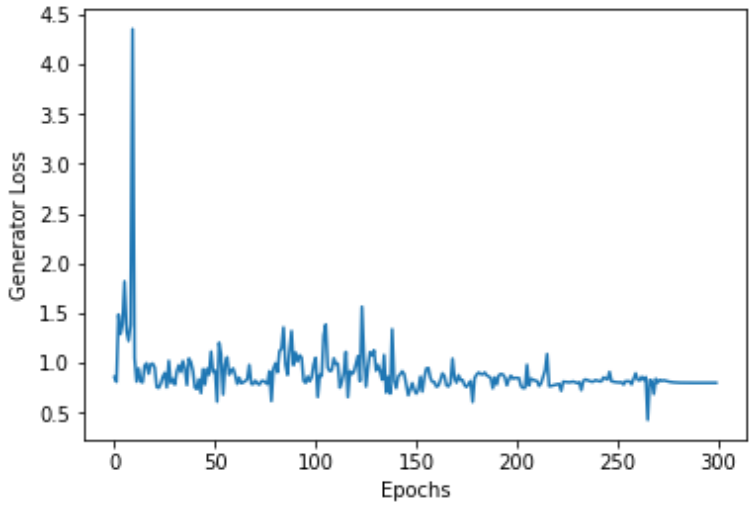


Figure 41. Generator Loss. Source: own elaboration ***(Annex C)****.*

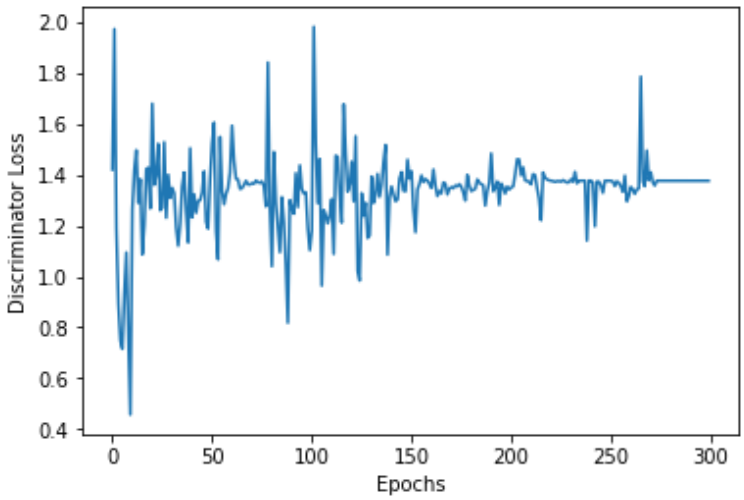


Figure 42. Discriminator Loss. Source: own elaboration ***(Annex C)****.*

For our GANDeconvConv model, as observed above, the cost functions are different from the previous case in that the discriminator takes much longer to converge. This suggests that the generator is making it more difficult for the discriminator to detect fake images.

However, when both converge, most of the images generated by the generator are labeled as fake by the discriminator, which is superior again. Either more training or more complex architectures for the generator are required still.

4.2.3. DCGAN

Finally, a model from the literature was used. The DCGAN architecture, introduced in **Section 2.2.4**, excels in terms of stability in training in most settings.

Its code, accessible in **[42]** was customized in order to accommodate the dimension of the images from our dataset. The main problem encountered was the computational power needed to train such a model. The GPUs available for this project only allowed for the execution of very few epochs within a reasonable amount of time.

From the beginning, this model was trained exclusively with portraits, in an effort to obtain realistic images from the generator. Since only very few epochs could be run because of the limited resources, the generator did not produce realistic paintings. However, the images did show a face-like brightness in the middle of the image, and even a collar is noticeable, as can be seen in **Figure 43**.

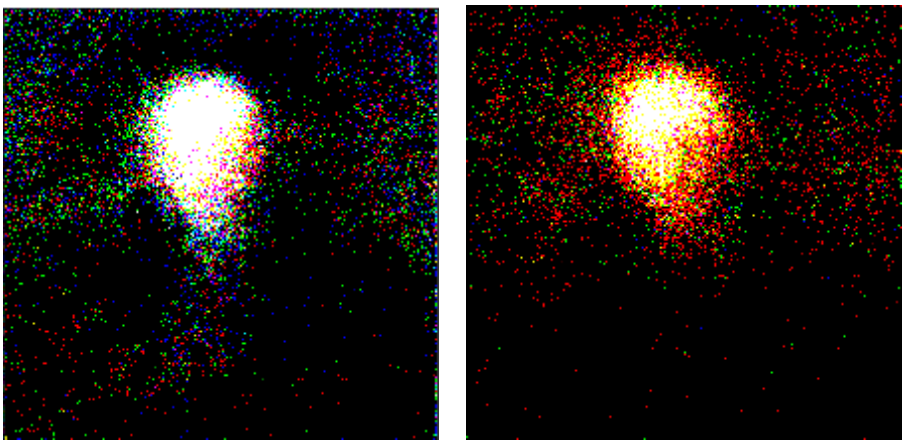


*Figure 43. Images generated by DCGAN after a few epochs. Source: own elaboration, with the baseline code of **[44]***

5. Conclusions and future work

5.3. Classification

In the classification task, the limitation of data was the main driver of the model's performance. It was this issue that caused the pre-trained model to provide significantly better results than the shallow network, and that allowed the CV+FC model of hand-engineered features to provide almost the best results.

Figure 44 provides a graphical representation of this intuition. This serves as a high-level explanation, where the exact convergence values are by no means accurate.

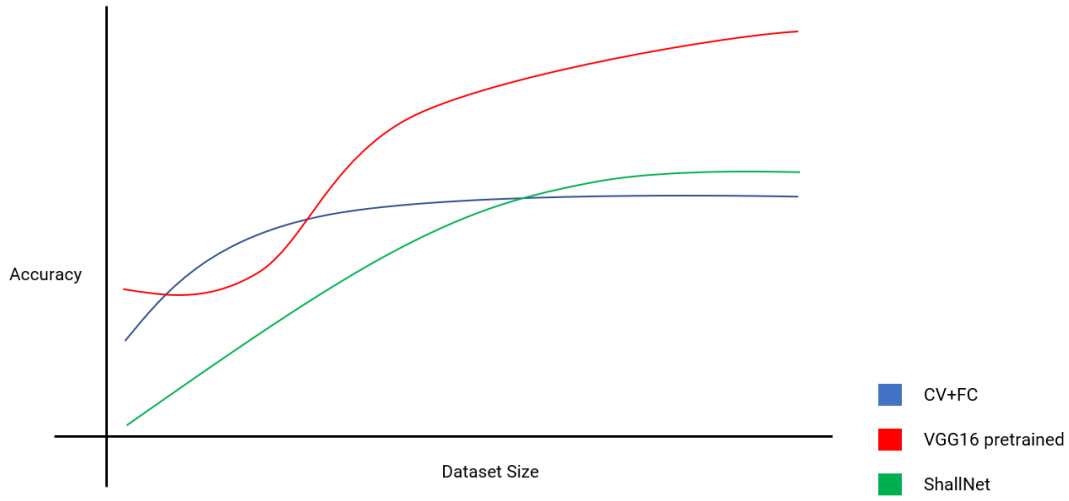


Figure 44. Intuition of the performance of each model based on available data. Generated for the sake of intuition. Source: own elaboration.

As future work, the number one priority is to obtain a reliable and extended dataset. The main challenge with artistic data is that it is more complicated for multiple people to provide consistent labelling in terms of emotions, since the emotion transmitted by a specific artwork can vary from person to person. This issue renders popular labeling tools such as Amazon's Mechanical Turk[9] useless.

The suggested approach is to provide a reduced set of human labelers with a standardized training on how to observe and interpret works of art. Then, a "qualification test" should be conducted with the same set of images for the labelers to identify the subset that

[9] Amazon Mechanical Turk (MTurk) is a crowdsourcing marketplace that makes it easier for individuals and businesses to outsource their processes and jobs to a distributed workforce who can perform these tasks virtually.

provides consistent labeling. Finally, this subset of human labelers should label a larger set of artworks in terms of the transmitted emotion.

Another line of future work would consist of combining models integrating both local and global features. The deep architecture models provided significantly good results due to their ability to identify local features that may evoke emotions (a smiling face, a praying pose, etc.). However, their performance is limited by the role of global features (color, texture, brightness, etc.) in evoking emotions. This became evident with the surprisingly good performance of the CV+FC hand-engineered model. Therefore, a model that is able to identify both global and local features, would definitely provide top results.

5.4. GANs

As it had been identified in the literature research in **Section 2.2**, creating generative models is a very difficult task, where the randomness associated to fine-tuning hyperparameters by hand plays a big role. The main conclusion extracted from this study is the importance of a sophisticated enough architecture of the generator. If the generator is not able to generate realistic images, the discriminator will not be "fooled" often.

However, a complex architecture for the generator requires a lot of computing power to train, which was not available for this project. As a line of future work, more resources would allow to train the DCGAN model, which has proven to be robust for multiple applications, to a higher level.

Another relevant conclusion is the importance of images with a similar structure in the training set. In the case of this project, this translated into portraits providing much better results than when combining all types of artworks (landscapes, scenes, etc.). This is aligned with the state-of-the art results in the field of image generation, which has proven to be more powerful for example for the generation of human-like faces **[43]**.

BIBLIOGRAPHY

[1] S. Malik, Sungchul Kim, and Eunyee Koh. Perceptual similarity ranking

[2] McDuff, D., et al. "A cross-platform real-time multi-face expression recognition toolkit." Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems [Association for Computing Machinery (ACM), 2016].

[3] Berlyne, D. E. (1971a). Aesthetics and psychobiology. New York: Appleton-Century-Crofts.

[4] Berlyne, D. E. (Ed.). (1974). Studies in the new experimental aesthetics: Steps toward an objective psychology of aesthetic appreciation. Washington, DC: Hemisphere.

[5] Ekman, Paul. "An argument for basic emotions." Cognition & emotion 6.3-4 (1992): 169-200.

[6] Silvia, P. J., & Warburton, J. B. (2006). Positive and negative affect: Bridging states and traits.

[7] Silvia, Paul J. "Emotional responses to art: From collation and arousal to cognition and emotion." Review of general psychology 9.4 (2005): 342-357.

[8] Hurlbert, Anya C., and Yazhu Ling. "Biological components of sex differences in color preference." Current biology 17.16 (2007): R623-R625.

[9] Hanafy, Ihab Mahmoud, and Reham Sanad. "Colour preferences according to educational background." Procedia-Social and Behavioral Sciences 205 (2015): 437-444.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks

[11] C. Szegedy, S. Ioffe, and V. Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning.

[12] https://www.chezmana.com/. Creating a marketplace for your artworks.

[13] ALBAWI, Saad; MOHAMMED, Tareq Abed; AL-ZAWI, Saad. Understanding of a convolutional neural network. En 2017 International Conference on Engineering and Technology (ICET). IEEE, 2017. p. 1-6.

[14] https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

[15] Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

[16] Scherer, Dominik, Andreas Müller, and Sven Behnke. "Evaluation of pooling operations in convolutional architectures for object recognition." International conference on artificial neural networks. Springer, Berlin, Heidelberg, 2010.

[17] You, Q. L. (2016). Building a large scale dataset for image emotion recognition: The fine print and the benchmark. *Thirtieth AAAI Conference on Artificial Intelligence*.

[18] Simonyan, K. &. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv*.

[19] Hassan, M. u. (2018). VGG16 – Convolutional Network for Classification and Detection.

[20] Simonyan, K. (2014). Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps.

[21] Zeiler, M. D. (2014). Visualizing and understanding convolutional networks. *European conference on computer vision*.

[22] Zhou, B. (2016). Learning deep features for discriminative localization. Proceedings of the IEEE conference on computer vision and pattern recognition.

[23] Goodfellow, I. (2016). NIPS 2016 Tutorial: Generative Adversarial Networks. *OpenAI*.

[24] Finn, C. I. (2016). Unsupervised learning for physical interaction through video prediction. *Advances in neural information processing systems*.

[25] Springenberg, J. T. (2015). Unsupervised and semi-supervised learning with categorical generative adversarial networks. *arXiv preprint arXiv*.

[26] Lotter, W. G. (2015). Unsupervised learning of visual structure using predictive generative networks. *arXiv preprint arXiv*.

[27] Ledig, Christian, et al. "Photo-realistic single image super-resolution using a generative adversarial network." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[28] Zhang, Han, et al. "Stackgan++: Realistic image synthesis with stacked generative adversarial networks." IEEE transactions on pattern analysis and machine intelligence 41.8 (2018): 1947-1962.

[29] Katanforoosh, K. (s.f.). Generative Adversarial Networks. *Stanford University, CS230*, (pág. 18).

[30] Lucic, K. Are GANs created equal, a large case study. *Advances in neural information processing systems*. 2018.

[31] Alec Radford, L. M. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv*.

[32] Han Zhang, T. X. (2016). StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks. *ICCV*.

[33] Ng, A. (2018). Machine Learning Strategsy. *CS230 lecture. Stanford University.*

[34] Torrey, L. (2010). Transfer Learning. *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques.*

[35] Kingma, D. P. (2014). Adam: A method for stochastic optimization.

[36] Krogh, A. a. (1992). A simple weight decay can improve generalization. *Advances in neural information processing systems*.

[37] Kiritchenko, S. M. (2018). Wikiart emotions: an annotated dataset of emotions evoked by art. *LREC*.

[38] Liu, S. a. (2018). Texture-aware emotional color transfer between images. *IEEE*.

[39] Lucassen, Marcel P., Theo Gevers, and Arjan Gijsenij. "Texture affects color emotion." Color Research & Application 36.6 (2011): 426-436.

[40] Wagstaff, Kiri, et al. "Constrained k-means clustering with background knowledge." Icml. Vol. 1. 2001.

[41] R. M. H. Nguyen and M. S. Brown. Why you should forget luminance conversion and do something better. In 2017 IEEE Conference on Computer Vision and Pattern Recognition. (CVPR), pages 5920–5928, July 2017.

[42] https://github.com/carpedm20/DCGAN-tensorflow

[43] Rani Horev. “Style-based GANs – Generating and Tuning Realistic Artificial Faces”. 2018.

[44] Perlin, K. (2002). Improving Noise. *Association for Computing Machinery*.

ANNEXES

Preliminary note:

The code was written in Jupyter Notebooks, as this format allows for easy execution through AWS. Notice that some results are printed to prove the correct functioning of the code, but they are not necessarily the runs that provided the best results.

ANNEX A: EmoModelPipeline

August 22, 2019

""" 1. build classifer with Transfer Learning on VGG16 2. Error Analysis, and observe Class Activation Maps on some of the examples 3. Since we are dealing with many objects, observe Genre and pass-through YOLO Object Detection 4. train a GAN for image generation 5. attempt NST for image emotion transfer """

In [26]:
```
import pandas as pd
from pandas import ExcelWriter
from pandas import ExcelFile
import base64
from IPython import display
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
import time
import os
from IPython import display
import PIL
from PIL import Image
from os import listdir
from os.path import isfile, join
from keras import applications
from keras.layers import Input
import keras
from keras.models import Model
```

In [27]:
```
data = pd.read_excel("emos.xlsx")
emotionImagesDir = "emotionData/"
emotionImagesDirResized = "emotionDataResized/"
dir_out_images_good = "images_good/"
dir_out_images_portraits = "images_portraits/"
length_max = 75
```

In [28]:
```
data["Style"] = data["Style"].str.upper()
data["Emotion"] = data["Emotion"].str.upper()
data["Genre"] = data["Genre"].str.upper()
```

Categorical Distributions

```
In [29]: # print(data.columns)
         print("Number of styles: {}".format(len(set(data.Style))))
         print("Number of artists: {}".format(len(set(data.Artist))))
         print("Number of genres: {}".format(len(set(data.Genre))))
         print("Number of tags: {}".format(len(set(data.Tags))))
         print("Number of emotions: {}".format(len(set(data.Emotion))))
```

```
Number of styles: 73
Number of artists: 398
Number of genres: 37
Number of tags: 1362
Number of emotions: 16
```

```
In [30]: test = (data[["Emotion", "Style"]].
                 groupby("Emotion").
                 count().
                 sort_values(by="Style", ascending=False))

         print("Emotion distrubution:")
         pd.DataFrame(test).head(4)
```

```
Emotion distrubution:
```

```
Out[30]:           Style
         Emotion
         NEUTRAL    1157
         OPTIMISM    542
         JOY         362
         SADNESS     237
```

```
In [31]: test = (data[["Emotion", "Genre"]].
                 groupby("Genre").
                 count().
                 sort_values(by="Emotion", ascending=False))

         print("Genre distrubution:")
         pd.DataFrame(test).head(5)
```

```
Genre distrubution:
```

```
Out[31]:                     Emotion
         Genre
         GENRE PAINTING          599
         LANDSCAPE               571
```

```
PORTRAIT                439
RELIGIOUS PAINTING      364
CITYSCAPE               193
```

In [32]:
```
s1 = ["SADMESS", "FEAR", "DISGUST", "ANGER"]
s2 = ["LUST", "ENVY", "SURPRISE"]
s3 = ["OPTIMISM", "JOY", "LOVE"]

# print(data[["Image URL", "Emotion"]].head())

e = data["Emotion"]
sum_1, sum_2, sum_3 = 0, 0, 0

def summer(sn, sum_n):
    for i in e:
        if i in sn:
            sum_n += 1
    return sum_n

sum_1 = summer(s1, sum_1)
sum_2 = summer(s2, sum_2)
sum_3 = summer(s3, sum_3)

print("state 1 emotions: {}".format(sum_1))
print("state 2 emotions: {}".format(sum_2))
print("state 3 emotions: {}".format(sum_3))
```

```
state 1 emotions: 245
state 2 emotions: 169
state 3 emotions: 1139
```

In [33]:
```
test = (data[["Emotion", "Style"]].
        groupby("Style").
        count().
        sort_values(by="Emotion", ascending=False))

print("Style distrubution:")
pd.DataFrame(test).head(5)
```

```
Style distrubution:
```

Out[33]:

Style	Emotion
REALISM	390
ROMANTICISM	335
IMPRESSIONISM	308
ART NOUVEAU (MODERN)	181
POST-IMPRESSIONISM	175

Resize Images

In [10]:
```
# read in an image from the image folder, resize and write to a new folder
basewidth, baseheight, channels = 224, 224, 3

def resize_things(things, dir_in, dir_out):
    for i in range(len(things)):

        # print("1: ", things[i])
        # temp = base64.urlsafe_b64decode(things[i])
        # tempImgName = temp.decode("utf-8") + str(".jpg")
        tempImgName = things[i]
        # print("2: ", tempImgName)

        # if(len(tempImgName) > length_max):
        #     continue

        img = Image.open(dir_in + tempImgName)
        wpercent = (basewidth / float(img.size[0]))
        hsize = int((float(img.size[1]) * float(wpercent)))
        hpercent = (baseheight / float(img.size[1]))
        wsize = int((float(img.size[0]) * float(hpercent)))
        img = img.resize((basewidth, baseheight), PIL.Image.ANTIALIAS)
        print("Image Loaded: {}".format(i+1))
        # plt.imshow(img)
        # plt.show()
        display.clear_output(wait=True)
        img = img.convert("RGB")
        img.save(dir_out + tempImgName)

emotionImages = [f for f in listdir(emotionImagesDir) if
                 isfile(join(emotionImagesDir, f))]
```

In [11]:
```
emotionImages.remove('.DS_Store')

import re
regex = re.compile(r'.*\(.*\).*')

emotionImages = [x for x in emotionImages if not regex.match(x)]
```

In [12]:
```
resize_things(emotionImages, emotionImagesDir, emotionImagesDirResized)
```

```
Image Loaded: 14592
```

In [13]:
```
print(emotionImages[0])
```

```
VGhlIEJlZXRob3ZlbiBGcmllemU6IFRoZSBMb25naW5nIGZvciBIYXBwaW5lc3MgRmluZHMgUmVwb3NlIGluIFBvZXRyeS4g
```

In [14]:
```
# decode the image name
imageTitleNames = []

for i in range(len(emotionImages)):

    temp = base64.urlsafe_b64decode(emotionImages[i][:-4])
    tempImgName = temp.decode("utf-8")
    imageTitleNames.append(tempImgName)

print(len(imageTitleNames))
```

```
14592
```

In [15]:
```
imageTitleNames[0]
```

Out[15]:
```
'The Beethoven Frieze: The Longing for Happiness Finds Repose in Poetry. Right wall'
```

In [16]:
```
imgTitles = pd.DataFrame(imageTitleNames)
imgTitles.columns = ["TitleNames"]

data_full = pd.merge(data, imgTitles, how='inner',
         left_on=['Title'],
         right_on=['TitleNames'])

good_data = data_full.loc[:, ["TitleNames", "Emotion", "Genre"]]
print("good_data shape:{}".format(good_data.shape))
```

```
good_data shape:(659, 3)
```

In [17]:
```
portrait_data = good_data.loc[good_data["Genre"] == "PORTRAIT"]
print("portrait_data shape:{}".format(portrait_data.shape))
portrait_data.head()
```

```
portrait_data shape:(103, 3)
```

Out[17]:

	TitleNames	Emotion	Genre
135	Vincenzo Anastagi	OPTIMISM	PORTRAIT
145	Pachtuwa-Chta, an Arrikkara Warrior, plate 27 ...	OPTIMISM	PORTRAIT
146	Portrait of a Knight	OPTIMISM	PORTRAIT
158	Young lady with a bird and dog	OPTIMISM	PORTRAIT
226	Untitled	JOY	PORTRAIT

In [18]:
```
good_data.to_csv("good_data.csv", sep="|", index=None, header=True)
# good_data = pd.read_csv("good_data.csv", sep="|", header=0, skiprows=0)
```

```
In [19]: def encode_data(df, length_max):

             encoded_data = []

             for i in range(df.shape[0]):
                 a = df.iloc[i, 0]
                 b = a.encode("utf-8")
                 temp = base64.urlsafe_b64encode(b)
                 c = df.iloc[i, 1]
                 # if(len(temp)<=length_max):
                 encoded_data.append((temp, c))

             return encoded_data

         encoded_data = encode_data(good_data, length_max)
         print("encoded_data: {}".format(len(encoded_data)))

         encoded_data_portrait = encode_data(portrait_data, length_max)
         print("encoded_data_portrait: {}".format(len(encoded_data_portrait)))
```

```
encoded_data: 659
encoded_data_portrait: 103
```

```
In [20]: i = 0

         name_temp = str(encoded_data[i][0])
         name_temp = name_temp[2:]
         name_temp = name_temp[:-1]
         name_temp = name_temp + ".jpg"

         print(name_temp)
```

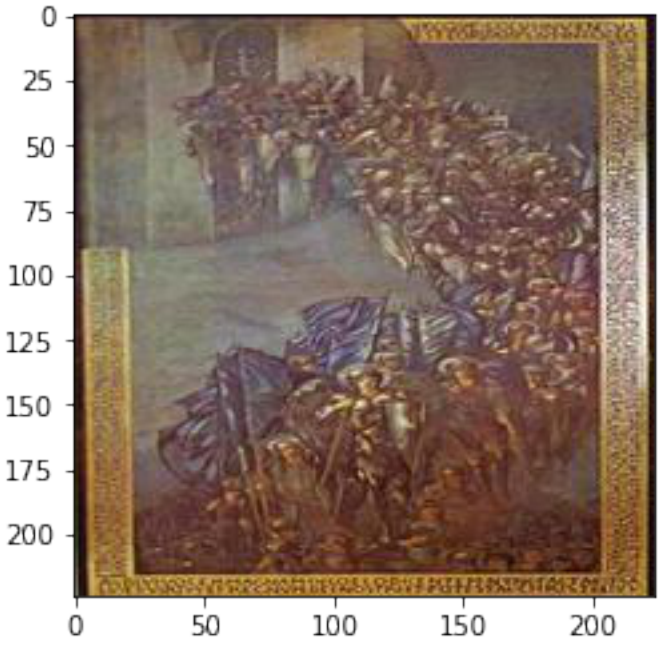
```
VGhlIEZhbGwgb2YgTHVjaWZlcg==.jpg
```

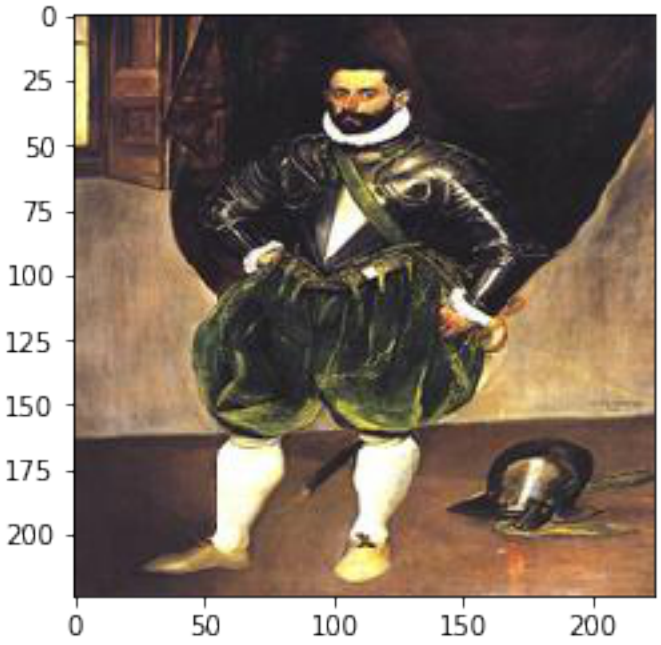
```
In [21]: def read_images(names, dir_in, label_dict):

             results = []

             for i in range(len(names)):

                 name_temp = str(names[i][0])
                 name_temp = name_temp[2:]
                 name_temp = name_temp[:-1]
                 name_temp = name_temp + ".jpg"

                 # if(len(name_temp) > length_max):
                 #     continue
```

```
                # print(name_temp)
                img = Image.open(dir_in + name_temp)
                print("Images Loaded: {}".format(i))
                # plt.imshow(img)
                # plt.show()
                display.clear_output(wait=True)
                label = names[i][1]

                if label not in label_dict:
                    label_final = 0
                else:
                    label_final = label_dict[label]

                # print("Label:{}".format(label_final))

                results.append((img, label_final))

            return results

        label_dict = {"SADNESS": 0, "FEAR":0, "DISGUST":0, "ANGER":0, "NEUTRAL":0, "nan":0,
                      "LUST": 0, "ENVY":0, "SURPRISE":0,
                      "OPTIMISM":2, "JOY":2, "LOVE":2}

        # label = 1
        images_good = read_images(encoded_data, emotionImagesDirResized, label_dict)
        images_portraits = read_images(encoded_data_portrait, emotionImagesDirResized,
                                       label_dict)

        print("images_good: {}".format(len(images_good)))
        print("images_portraits: {}".format(len(images_portraits)))
```

```
images_good: 659
images_portraits: 103
```

```
In [22]: plt.imshow(images_good[0][0])
         plt.show()

         plt.imshow(images_portraits[0][0])
         plt.show()
```






```
In [24]: temp_size = 608

         def resize_and_write(data_list, names_list, img_size, dir_out):
```

```
        for i in range(len(data_list)):

            img = images_good[i][0]
            img = img.resize((img_size, img_size), PIL.Image.ANTIALIAS)
            print("Images Written: {}".format(i+1))
            # plt.imshow(img)
            # plt.show()
            display.clear_output(wait=True)
            img = img.convert("RGB")

            name_temp = str(names_list[i][0])
            name_temp = name_temp[2:]
            name_temp = name_temp[:-1]
            name_temp = name_temp + ".jpg"
            # print(name_temp)
            img = img.convert("RGB")
            img.save(dir_out + name_temp)

    resize_and_write(images_good, encoded_data, temp_size, dir_out_images_good)
    resize_and_write(images_portraits, encoded_data_portrait, temp_size,
                     dir_out_images_portraits)
```

```
Images Written: 103
```

ANNEX B: BasicGAN

August 22, 2019

In [1]:
```
from IPython import display
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
import time
import os
from IPython import display
import PIL
from PIL import Image
from os import listdir
from os.path import isfile, join
dir_portraits_resized = "data/images_portraits_gan/"

portraits = [f for f in listdir(dir_portraits_resized) if
             isfile(join(dir_portraits_resized, f))]

print(len(portraits))
```

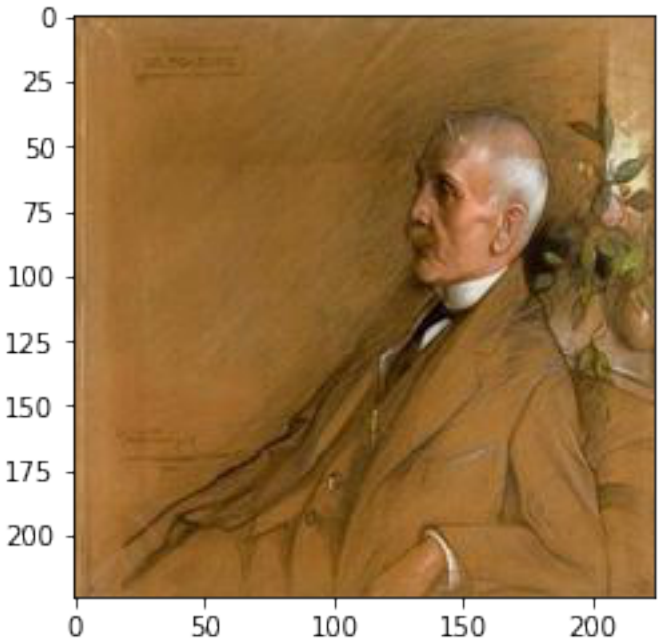
```
7136
```

In [2]:
```
basewidth, baseheight, channels = 224, 224, 3
length_max = 110

def read_images(names, dir_in, label):
    results = []

    for i in range(len(names)):
        if(len(names[i]) > length_max):
            continue
        img = Image.open(dir_in + names[i])
        print("Image: {}".format(i))
        # plt.imshow(img)
        # plt.show()
        display.clear_output(wait=True)
        results.append((img, label))
```

```
            return results

        portraits_images = read_images(portraits, dir_portraits_resized, 1) # label = 1
```

```
Image: 7135
```

In [3]:
```
plt.imshow(portraits_images[0][0])
plt.show()
```



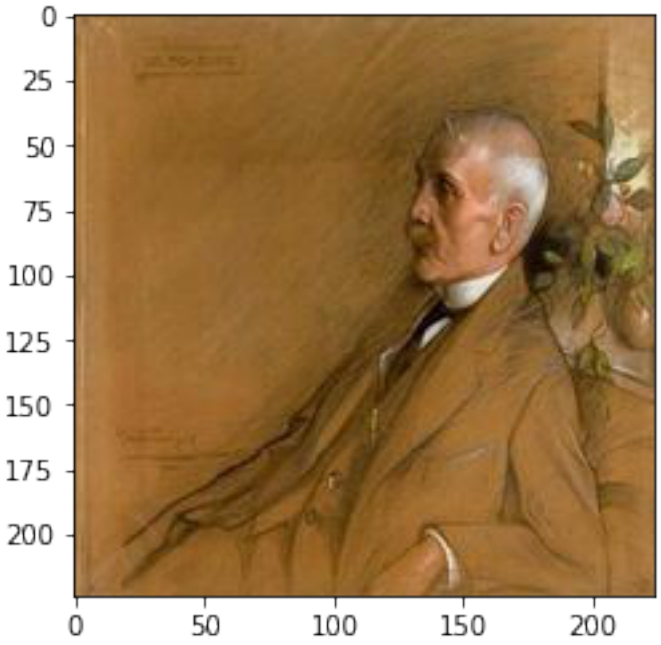
In [4]:
```
n_t = 7000 # training examples
n_c = 0 # channel number to use
n_std = 255.0

def segment_data(images, n_t):

    train_x, train_y = [], []

    for tup in images[:n_t]:

        if (np.array(tup[0]).shape[-1]) == channels:
            train_x.append(np.array(tup[0])[:,:,:]/n_std)
        else:
            continue

        train_y.append(tup[1])
```

```
            return train_x, train_y

        # select which data to use here
        train_x, train_y = segment_data(portraits_images, n_t)

        train_x_4d = np.stack(train_x, axis=0)
        print("train_x_4d shape:", train_x_4d.shape)

        train_y_2d = np.expand_dims(np.stack(train_y, axis=0), axis=1)
        print("train_y_2d shape:", train_y_2d.shape)
```

```
train_x_4d shape: (7000, 224, 224, 3)
train_y_2d shape: (7000, 1)
```

In [5]:
```
# print("Training exaples:{}, Test examples:{}".format(len(train_y), len(test_x)))
print("Label:{}".format(train_y[0]))
print("Shape:{}".format(train_x[0].shape))
plt.imshow(train_x[0])
plt.show()
```

```
Label:1
Shape:(224, 224, 3)
```



In [6]:
```
def gan_generator(z, reuse=None):
    with tf.variable_scope('gen', reuse=reuse):
        fc1 = tf.layers.dense(inputs=z, units=basewidth,
                              activation=tf.nn.leaky_relu)
        fc2 = tf.layers.dense(inputs=fc1, units=basewidth,
                              activation=tf.nn.leaky_relu)
        output = tf.layers.dense(inputs=fc2,
                                 units=basewidth * baseheight * channels,
                                 activation=tf.nn.tanh)

        return output

def gan_discriminator(X, reuse=None):
    with tf.variable_scope('dis', reuse=reuse):
        fc1 = tf.layers.dense(inputs=X, units=basewidth, activation=tf.nn.leaky_relu)
        fc2 = tf.layers.dense(inputs=fc1, units=basewidth, activation=tf.nn.leaky_relu)
        logits = tf.layers.dense(fc2, units=1)
        output = tf.sigmoid(logits)

        return output, logits

import math

def size_conv(n,p,f,s):
    return math.floor( (((n) + 2*p - f)/s) + 1 )

p = 5
f = 5
s = 1
print(size_conv(basewidth,p,f,s))

p = 0
f = 2
s = 2
print(size_conv(basewidth,p,f,s))
```

```
230
112
```

In [7]:
```
tf.reset_default_graph()
real_images = tf.placeholder(tf.float32,
                             shape=[None, basewidth * baseheight * channels])
z = tf.placeholder(tf.float32, shape=[None, basewidth * baseheight * channels])

G = gan_generator(z)
D_output_real, D_logits_real = gan_discriminator(real_images)
D_output_fake, D_logits_fake = gan_discriminator(G, reuse=True)
```

In [8]:
```
def loss_func(logits_in,labels_in):
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
        logits = logits_in, labels = labels_in))

 # Smoothing for generalization
D_real_loss = loss_func(D_logits_real, tf.ones_like(D_logits_real)*0.9)
D_fake_loss = loss_func(D_logits_fake, tf.zeros_like(D_logits_real))
D_loss = D_real_loss + D_fake_loss

G_loss = loss_func(D_logits_fake, tf.ones_like(D_logits_fake))
```

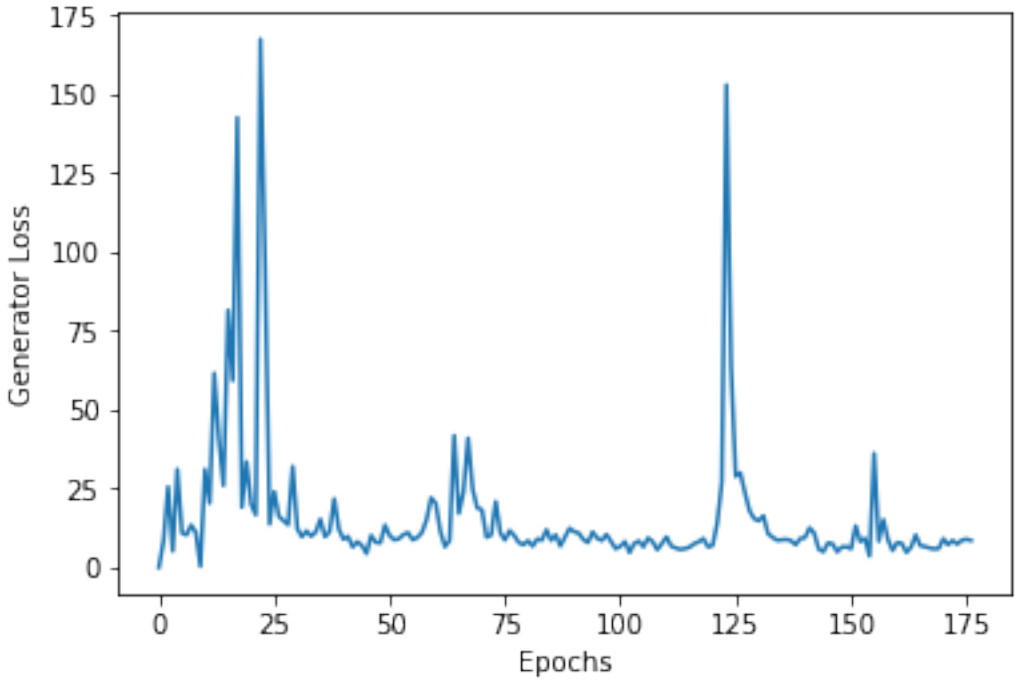
In [9]:
```
lr = 0.001
lr_gen = 0.001
batch_size = 200
epochs = 175
noise_low = -1.0
noise_high = 1.0

# Do this when multiple networks interact with each other
# returns all variables created (the two variable scopes) and makes trainable true
tvars = tf.trainable_variables()
d_vars = [var for var in tvars if 'dis' in var.name]
g_vars = [var for var in tvars if 'gen' in var.name]

gan_discriminator_trainer = tf.train.AdamOptimizer(lr).minimize(D_loss,
                                                                var_list=d_vars)
gan_generator_trainer = tf.train.AdamOptimizer(lr_gen).minimize(G_loss,
                                                                var_list=g_vars)

init = tf.global_variables_initializer()
```

In [ ]:
```
samples = [] # generator examples
dis_outputs = []
turns = 0
g_losses = []
d_losses = []

with tf.Session() as sess:

    sess.run(init)

    for epoch in range(epochs):

        # num_batches = mnist.train.num_examples//batch_size
        num_batches = len(train_x)//batch_size

        for i in range(num_batches): # 0, 1, 2 .. 10
```

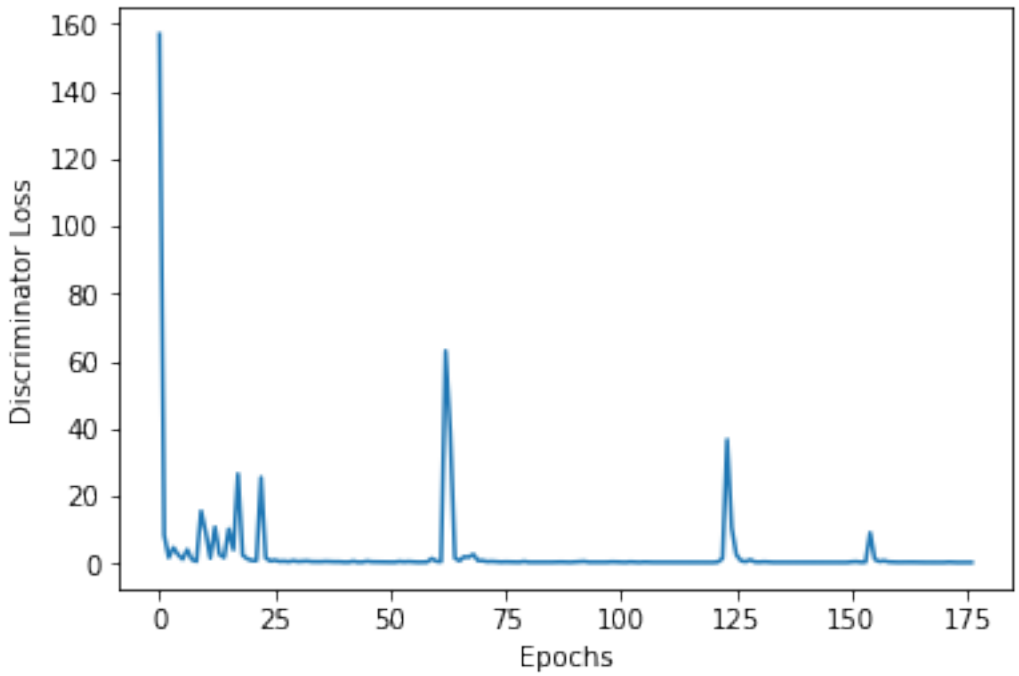
```
        batch = np.array(train_x[i*batch_size:(i+1)*batch_size])

        # print(batch.shape)
        batch_images = batch.reshape((batch_size,
                                      basewidth * baseheight * channels))

        # doubling since we are adding noise images
        batch_images = batch_images*(abs(noise_low) + abs(noise_high)) - 1
        batch_z = np.random.uniform(noise_low, noise_high,
                                    size=(batch_size,
                                   basewidth * baseheight * channels))

        _ = sess.run(gan_discriminator_trainer, feed_dict =
                     {real_images:batch_images, z:batch_z})

        if(turns%3 == 0):
            _ = sess.run(gan_generator_trainer, feed_dict = {z:batch_z})

        if i == 0:
            gl = sess.run(G_loss, feed_dict = {z:batch_z})
            g_losses.append(gl)
            dl = sess.run(D_loss, feed_dict =
                          {real_images:batch_images, z:batch_z})
            d_losses.append(dl)

        turns = turns + 1

    display.clear_output(wait=True)
    print("currently on epoch: {} ...".format(epoch))

    sample_z = np.random.uniform(-1, 1,
                         size=(1, basewidth * baseheight * channels))

    # store the generated image
    gen_sample = sess.run(gan_generator(z,
                                reuse = True), feed_dict={z:sample_z})

    samples.append(gen_sample)

    # FOr iteration 2:
    # store the output of discriminator
    sample_in = gen_sample.reshape((1, basewidth * baseheight * channels))
    dis_output, dis_logits = sess.run(gan_discriminator(real_images,
                        reuse = True), feed_dict={real_images:sample_in})

    dis_outputs.append(dis_output[0][0])

    plt.imshow(gen_sample.reshape(basewidth, baseheight, channels))
```

```
            plt.show()
            display.clear_output(wait = True)
```

In [12]:
```
import matplotlib.pyplot as plt
plt.plot(np.arange(177), g_losses)
plt.ylabel('Generator Loss')
plt.xlabel('Epochs')
plt.show()
```



In [14]:
```
import matplotlib.pyplot as plt
plt.plot(np.arange(177), d_losses)
plt.ylabel('Discriminator Loss')
plt.xlabel('Epochs')
plt.show()
```




```
In [ ]: freq = 1
        i = 0
        while i < epochs:
            print("Image at epoch:{}".format(i))
            plt.imshow(samples[i].reshape(basewidth, baseheight, channels))
            plt.show()
            display.clear_output(wait = True)
            time.sleep(0.2)
            i += freq
```

ANNEX C: GANDeconv-Conv6

August 22, 2019

In [1]:
```
from IPython import display
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
import time
import os
from IPython import display
import PIL
from PIL import Image
from os import listdir
from os.path import isfile, join
```

In [2]:
```
dir_portraits = "data/images_portraits_gan/"
dir_portraits_resized = "images_portraits_gan_resized/"

portraits = [f for f in listdir(dir_portraits) if isfile(join(dir_portraits, f))]

print(len(portraits))
```

```
7136
```

In [3]:
```
basewidth, baseheight, channels = 128, 128, 3
length_max = 110

def resize_things(things, dir_in, dir_out):
    for i in range(len(things)):
        if(len(things[i]) > length_max):
            continue
        img = Image.open(dir_in + things[i])
        wpercent = (basewidth / float(img.size[0]))
        hsize = int((float(img.size[1]) * float(wpercent)))
        hpercent = (baseheight / float(img.size[1]))
        wsize = int((float(img.size[0]) * float(hpercent)))
        img = img.resize((basewidth, baseheight), PIL.Image.ANTIALIAS)
        img.save(dir_out + things[i])

resize_things(portraits, dir_portraits, dir_portraits_resized)
```

In [4]:
```
basewidth, baseheight, channels = 128, 128, 3
length_max = 110

def read_images(names, dir_in, label):
    results = []

    for i in range(len(names)):
        if(len(names[i]) > length_max):
            continue
        img = Image.open(dir_in + names[i])
        print("Image: {}".format(i))
        display.clear_output(wait=True)
        results.append((img, label))

    return results

portraits_images = read_images(portraits, dir_portraits_resized, 1)
```

```
Image: 7135
```

In [5]:
```
plt.imshow(portraits_images[2][0])
plt.show()
```

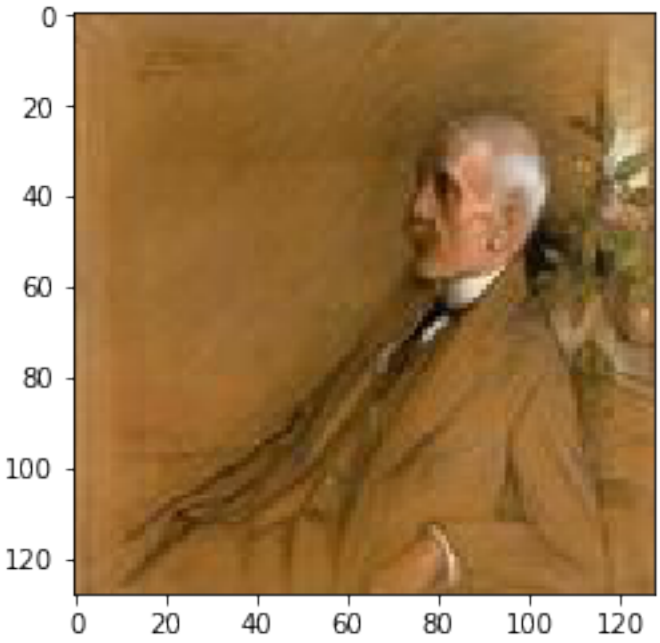
In [6]:
```
n_t = 7000 # training examples
n_c = 0 # channel number to use
```

```
        n_std = 255.0

        def segment_data(images, n_t):

            train_x, train_y = [], []

            for tup in images[:n_t]:

                if (np.array(tup[0]).shape[-1]) == channels:
                    train_x.append(np.array(tup[0])[:,:,:]/n_std)
                else:
                    continue

                train_y.append(tup[1])

            return train_x, train_y

        # select which data to use here
        train_x, train_y = segment_data(portraits_images, n_t)

        train_x_4d = np.stack(train_x, axis=0)
        print("train_x_4d shape:", train_x_4d.shape)

        train_y_2d = np.expand_dims(np.stack(train_y, axis=0), axis=1)
        print("train_y_2d shape:", train_y_2d.shape)
```

```
train_x_4d shape: (7000, 128, 128, 3)
train_y_2d shape: (7000, 1)
```

```
In [7]: # print("Training exaples:{}, Test examples:{}".format(len(train_y), len(test_x)))
        print("Label:{}".format(train_y[0]))
        print("Shape:{}".format(train_x[0].shape))
        plt.imshow(train_x[0])
        plt.show()
```

```
Label:1
Shape:(128, 128, 3)
```




```
In [8]: def gan_generator(z, reuse=None):
            # input_latent = Input(batch_shape=noise_dim, dtype=im_dtype)
            with tf.variable_scope('gen', reuse=reuse):
                weight_decay = 0.0001
                xx = tf.contrib.layers.fully_connected(z, num_outputs=4*4*128,
                                                       activation_fn=None)
                xx = tf.contrib.layers.batch_norm(xx)
                xx = tf.nn.relu(xx)
                xx = tf.reshape(xx, (tf.shape(z)[0], 4,4,128))
                xx = tf.contrib.layers.conv2d_transpose(xx, 64, kernel_size=(5,5),
                                                        stride=(2, 2), padding='SAME',
                                                        activation_fn=None)
                xx = tf.contrib.layers.batch_norm(xx)
                xx = tf.nn.relu(xx)
                xx = tf.contrib.layers.conv2d_transpose(xx,32, kernel_size=(5,5),
                                                        stride=(2, 2), padding='SAME',
                                                        activation_fn=None)
                xx = tf.contrib.layers.batch_norm(xx)
                xx = tf.nn.relu(xx)
                xx = tf.contrib.layers.conv2d_transpose(xx,16, kernel_size=(5,5),
                                                        stride=(2, 2), padding='SAME',
                                                        activation_fn=None)
                xx = tf.contrib.layers.batch_norm(xx)
                xx = tf.nn.relu(xx)
                xx = tf.contrib.layers.conv2d_transpose(xx, 8, kernel_size=(5,5),
                                                        stride=(2, 2),  padding='SAME',
```

```
                                                activation_fn=None)
            xx = tf.contrib.layers.batch_norm(xx)
            xx = tf.nn.relu(xx)
            xx = tf.contrib.layers.conv2d_transpose(xx,3, kernel_size=(5,5),
                                                stride=(2, 2), padding='SAME',
                                                activation_fn=None)
            xx = tf.contrib.layers.batch_norm(xx)
            gen_dat = tf.nn.tanh(xx) # 64x64x3
            #gen_dat_large = tf.image.resize_images(gen_dat, size = [224,224])

        return gen_dat

    def gan_discriminator(X, reuse=None):
        with tf.variable_scope('dis', reuse=reuse):
            Z1 = tf.layers.conv2d(X, filters = 8, kernel_size = [4, 4],
                                  strides = [1,1], padding = 'SAME')
            A1 = tf.nn.relu(Z1)
            P1 = tf.nn.max_pool(A1, ksize = [1,8,8,1], strides = [1,8,8,1],
                                padding = 'SAME')
            Z2 = tf.layers.conv2d(P1, filters = 16, kernel_size = [2, 2],
                                  strides = [1,1], padding = 'SAME')
            A2 = tf.nn.relu(Z2)
            P2 = tf.nn.max_pool(A2, ksize = [1,4,4,1], strides = [1,4,4,1],
                                padding = 'SAME')
            P2 = tf.contrib.layers.flatten(P2)
            Z3 = tf.contrib.layers.fully_connected(P2, 1, activation_fn=None)
            output = tf.sigmoid(Z3)
            return output, Z3


    import math

    def size_conv(n,p,f,s):
        return math.floor( (((n) + 2*p - f)/s) + 1 )

    p = 5
    f = 5
    s = 1
    print(size_conv(basewidth,p,f,s))

    p = 0
    f = 2
    s = 2
    print(size_conv(basewidth,p,f,s))
```

```
134
64
```

In [35]:
```
z_size = 100
tf.reset_default_graph()
real_images = tf.placeholder(tf.float32, shape= (None, basewidth,
                                                 baseheight, channels))
z = tf.placeholder(tf.float32, shape=[None, z_size])

G = gan_generator(z)
G1 = tf.reshape(G, [tf.shape(G)[0], basewidth, baseheight,
                    channels])
D_output_real, D_logits_real = gan_discriminator(real_images)
D_output_fake, D_logits_fake = gan_discriminator(G1, reuse=True)
```

In [36]:
```
def loss_func(logits_in,labels_in):
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
        logits = logits_in, labels = labels_in))

# Smoothing for generalization
D_real_loss = loss_func(D_logits_real, tf.ones_like(D_logits_real)*0.9)
D_fake_loss = loss_func(D_logits_fake, tf.zeros_like(D_logits_real))
D_loss = D_real_loss + D_fake_loss

G_loss = loss_func(D_logits_fake, tf.ones_like(D_logits_fake))
```

In [37]:
```
lr = 0.001
lr_gen = 0.001
batch_size = 200
epochs = 300
noise_low = -1.0
noise_high = 1.0

# Do this when multiple networks interact with each other
# returns all variables created (the two variable scopes) and makes trainable true
tvars = tf.trainable_variables()
d_vars = [var for var in tvars if 'dis' in var.name]
g_vars = [var for var in tvars if 'gen' in var.name]

gan_discriminator_trainer = tf.train.AdamOptimizer(lr).minimize(D_loss,
                                                                var_list=d_vars)
gan_generator_trainer = tf.train.AdamOptimizer(lr_gen).minimize(G_loss,
                                                                var_list=g_vars)

init = tf.global_variables_initializer()
```

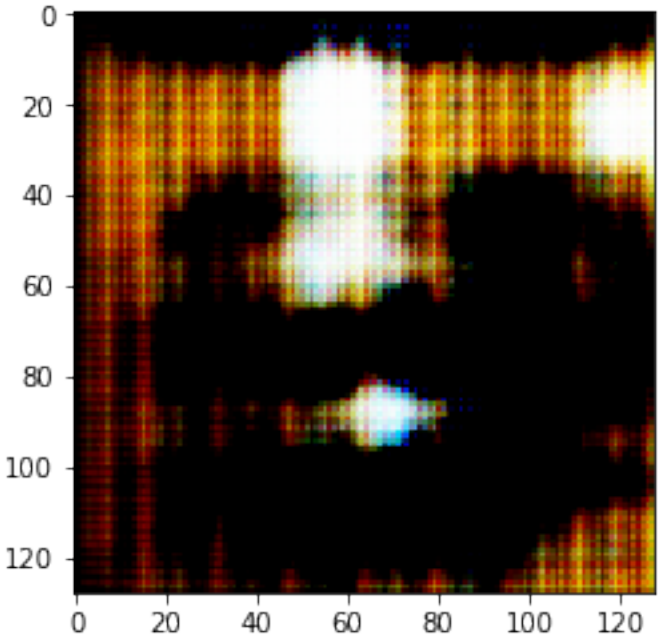
In [38]:
```
samples = [] # generator examples
dis_outputs = [] # discriminator outputs
turns = 0
g_losses = []
d_losses = []
```

```
with tf.Session() as sess:

    sess.run(init)

    for epoch in range(epochs):

        # num_batches = mnist.train.num_examples//batch_size
        num_batches = len(train_x)//batch_size

        for i in range(num_batches): # 0, 1, 2 .. 10

            batch = np.array(train_x[i*batch_size:(i+1)*batch_size])

            # print(batch.shape)
            batch_images = batch

            # doubling since we are adding noise images
            batch_images = batch_images*(abs(noise_low) + abs(noise_high)) - 1
            batch_z = np.random.uniform(noise_low, noise_high,
                                        size=(batch_size, z_size))

            _ = sess.run(gan_discriminator_trainer, feed_dict =
                         {real_images:batch_images, z:batch_z})

            # iter 1:
            #
            # iter 2:
            _ = sess.run(gan_generator_trainer, feed_dict = {z:batch_z})

            if i == 0:
                gl = sess.run(G_loss, feed_dict = {z:batch_z})
                g_losses.append(gl)
                dl = sess.run(D_loss, feed_dict =
                              {real_images:batch_images, z:batch_z})
                d_losses.append(dl)

            turns = turns + 1

        display.clear_output(wait=True)
        print("currently on epoch: {} ...".format(epoch))

        sample_z = np.random.uniform(-1, 1, size=(1, z_size))

        # store the generated image
        gen_sample = sess.run(gan_generator(z, reuse = True),
                              feed_dict={z:sample_z})
```

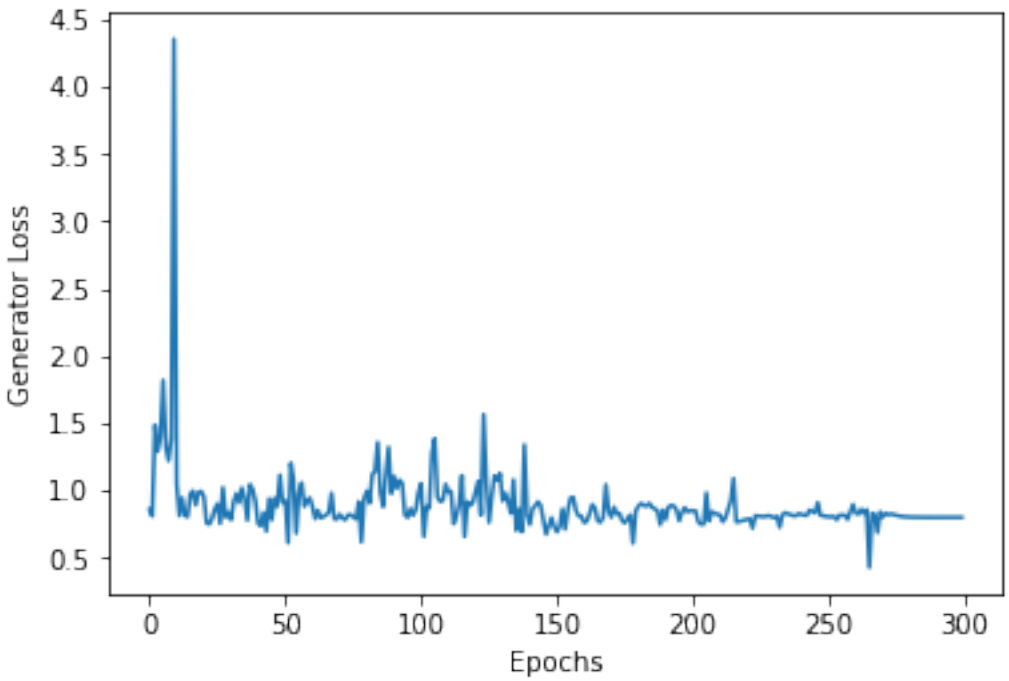
```
                    samples.append(gen_sample)

                    # FOr iteration 2:
                    # store the output of discriminator
                    sample_in = gen_sample.reshape((1, basewidth, baseheight, channels))
                    dis_output, dis_logits = sess.run(gan_discriminator(real_images,
                                                                        reuse = True),
                                                      feed_dict={real_images:sample_in})

                    dis_outputs.append(dis_output[0][0])

                    plt.imshow(gen_sample.reshape(basewidth, baseheight, channels))
                    plt.show()
                    display.clear_output(wait = True)
```

```
currently on epoch: 299 ...
```

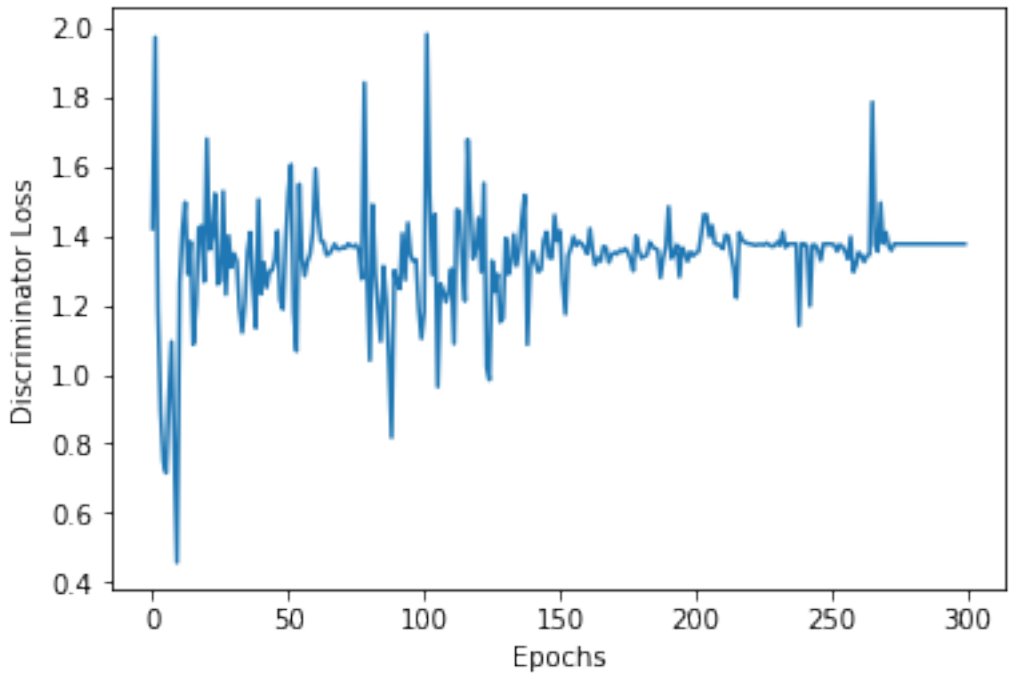
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] f
```




```
In [46]: import matplotlib.pyplot as plt
         plt.plot(np.arange(epochs), g_losses)
         plt.ylabel('Generator Loss')
         plt.xlabel('Epochs')
         plt.show()
```



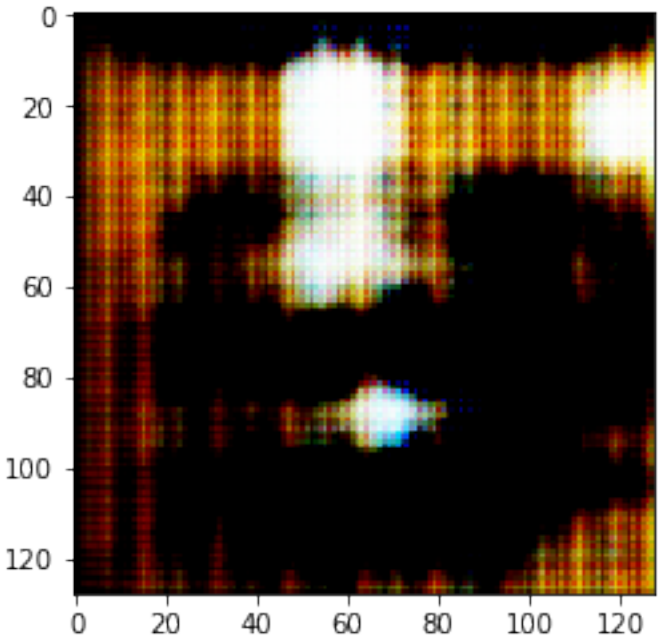

```
In [47]: import matplotlib.pyplot as plt
         plt.plot(np.arange(epochs), d_losses)
         plt.ylabel('Discriminator Loss')
         plt.xlabel('Epochs')
         plt.show()
```




```
In [48]: freq = 1
         i = 0
         while i < epochs:
             print("Image at epoch:{}".format(i))
             plt.imshow(samples[i].reshape(basewidth, baseheight, channels))
             plt.show()
             display.clear_output(wait = True)
             time.sleep(0.2)
             i += freq
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] f
```

```
Image at epoch:299
```


0
20
40
60
80
100
120
0
20
40
60
80
100
120

ANNEX D: Classification and Attribution

August 22, 2019

In [ ]:
```
# !pip install --user --upgrade tensorflow-model-optimization
!pip install --upgrade tensorflow
!pip install tensorflow-model-optimization
# help('modules')

# pip install tf-nightly-gpu
# pip install --upgrade tensorflow-model-optimization
```

In [ ]:
```
import tensorflow as tf
# import tf_nightly
import tensorflow_model_optimization as tfmot
# from tensorflow_model_optimization import *

model = tf.keras.Sequential([...])

pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(
                      initial_sparsity=0.0, final_sparsity=0.5,
                      begin_step=2000, end_step=4000)

# model_for_pruning = tfmot.sparsity.keras.prune_low_magnitude(
#     model, pruning_schedule=pruning_schedule)
# ...

# model_for_pruning.fit(...)
```

In [0]:
```
"""
1. build classifer with Transfer Learning on VGG16
2. Error Analysis, and observe Class Activation Maps on some of the examples
3. Since we are dealing with many objects, observe Genre
and pass-through YOLO Object Detection
4. train a GAN for image generation
5. attempt NST for image emotion transfer
"""
```

Out[0]:
```
'\n1. build classifer with Transfer Learning on VGG16\n2. Error Analysis, and observe Cl
```

In [0]:
```
!pip install noise
```

```
Collecting noise
  Downloading https://files.pythonhosted.org/packages/18/29/bb830ee6d934311e17a7a4fa1368faf3e73f
    100% |â[U+0096][U+0088]â[U+0096][U+0088]â[U+0096][U+0088]â[U+0096][U+0088]â[U+0096][U+0088]â[U+0096][U+0088]
Building wheels for collected packages: noise
  Running setup.py bdist_wheel for noise ... - \ | / done
  Stored in directory: /root/.cache/pip/wheels/fd/a3/c1/d36defe6e9f074b25dc0f018eb9f8fdd675a7ef8
Successfully built noise
Installing collected packages: noise
Successfully installed noise-1.2.2
```

```
In [0]: import pandas as pd
        import numpy as np
        from pandas import ExcelWriter
        from pandas import ExcelFile
        import base64
        from IPython import display
        import tensorflow as tf
        import numpy as np
        import matplotlib.pyplot as plt
        from tensorflow.examples.tutorials.mnist import input_data
        import time
        import os
        from IPython import display
        import PIL
        from PIL import Image
        from os import listdir
        from os.path import isfile, join
        from keras import applications
        from keras.layers import Input
        import keras
        from keras.models import Model
        import noise
        from PIL import ImageStat
        import math
        import scipy
        from sklearn.preprocessing import MinMaxScaler

        # import cv2
```

```
In [ ]: # data = pd.read_excel("emos.xlsx")
        data_full = pd.read_csv("good_data.csv", sep="|", header=0, skiprows=0)
        # emotionImagesDir = "emotionData/"
        # emotionImagesDirResized = "emotionDataResized/"
        dir_out_images_good =  "images_good/"
        dir_out_images_portraits = "images_portraits_gan/" # "images_all/"
        length_max = 150
```

```
In [0]: good_data = data_full.loc[:, ["TitleNames", "Emotion", "Genre"]]
```

```
        print("good_data shape:{}".format(good_data.shape))
        good_data.head()
```

```
good_data shape:(1818, 3)
```

Out[0]:

	TitleNames	Emotion	Genre
0	The Fall of Lucifer	SADNESS	RELIGIOUS PAINTING
1	The Four Horsemen of the Apocalypse, Death, Fa...	ANGER	RELIGIOUS PAINTING
2	The Four Horsemen of the Apocalypse, Death, Fa...	ANGER	RELIGIOUS PAINTING
3	Grazing horses	NEUTRAL	LANDSCAPE
4	Peasants and horse	JOY	GENRE PAINTING

```
In [0]: portrait_data = good_data.loc[good_data["Genre"] == "PORTRAIT"]
        print("portrait_data shape:{}".format(portrait_data.shape))
        portrait_data.head()
```

```
portrait_data shape:(264, 3)
```

Out[0]:

	TitleNames	Emotion	Genre
341	Vincenzo Anastagi	OPTIMISM	PORTRAIT
345	Portrait of Joachim II	OPTIMISM	PORTRAIT
362	Farinata degli Uberti	OPTIMISM	PORTRAIT
364	Noapeh Assiniboin Indian	OPTIMISM	PORTRAIT
365	Noapeh Assiniboin Indian	OPTIMISM	PORTRAIT

```
In [0]: def encode_data(df, length_max):

            encoded_data = []

            for i in range(df.shape[0]):
                a = df.iloc[i, 0]
                b = a.encode("utf-8")
                temp = base64.urlsafe_b64encode(b)
                c = df.iloc[i, 1]
                # if(len(temp)<=length_max):
                encoded_data.append((temp, c))

            return encoded_data

        encoded_data = encode_data(good_data, length_max)
```

```
        print("encoded_data: {}".format(len(encoded_data)))

        encoded_data_portrait = encode_data(portrait_data, length_max)
        print("encoded_data_portrait: {}".format(len(encoded_data_portrait)))
```

```
encoded_data: 1818
encoded_data_portrait: 264
```

```
In [0]: encoded_data[0]
```

```
Out[0]: (b'VGhlIEZhbGwgb2YgTHVjaWZlcg==', 'SADNESS')
```

```
In [0]: # encoded_data_portrait
```

```
In [0]: basewidth, baseheight, channels = 224, 224, 3
```

```
        def read_images(names, dir_in, label_dict):

            results = []

            for i in range(len(names)):

                name_temp = str(names[i][0])
                name_temp = name_temp[2:]
                name_temp = name_temp[:-1]
                name_temp = name_temp + ".jpg"

                # if(len(name_temp) > length_max):
                #     continue

                # print("Image Name: {}".format(name_temp))
                img = Image.open(dir_in + name_temp)
                # print("Images Loaded: {}".format(i))

                wpercent = (basewidth / float(img.size[0]))
                hsize = int((float(img.size[1]) * float(wpercent)))
                hpercent = (baseheight / float(img.size[1]))
                wsize = int((float(img.size[0]) * float(hpercent)))
                img = img.resize((basewidth, baseheight), PIL.Image.ANTIALIAS)

                # plt.imshow(img)
                # plt.show()

                display.clear_output(wait=True)
                label = names[i][1]

                if label not in label_dict:
                    label_final = 0
```

```
                else:
                    label_final = label_dict[label]

            #         if(label not in label_dict) or (label_dict[label]==0):
            #             label_final = (0,0,0,0)
            #         elif(label_dict[label]==1):
            #             label_final = (0,1,0,0)
            #         elif(label_dict[label]==2):
            #             label_final = (0,0,1,0)
            #         elif(label_dict[label]==3):
            #             label_final = (0,0,0,1)

                # print("Label:{}".format(label_final))

                results.append((img, label_final))

            return results

        label_dict = {"SADNESS": 0, "FEAR":0, "DISGUST":0, "ANGER":0,
                      "NEUTRAL":1, "nan":1,
                      "LUST":2, "ENVY":2, "SURPRISE":2,
                      "OPTIMISM":3, "JOY":3, "LOVE":3}
```

In [0]:
```
images_good = read_images(encoded_data,
                          dir_out_images_good,
                          label_dict) # label = 1

print("images_good: {}".format(len(images_good)))
```

```
images_good: 1818
```

In [0]:
```
images_portraits = read_images(encoded_data_portrait, dir_out_images_portraits,
                               label_dict) # label = 1
print("images_portraits: {}".format(len(images_portraits)))
```

```
images_portraits: 264
```

In [0]:
```
i = 0

print("Label:{}".format(images_portraits[i][1]))
images_portraits[i][0]
```

```
Label:3
```

Out[0]:

```
In [0]: n_t = 200 # training examples
        n_c = 0 # channel number to use
        n_std = 255.0

        def segment_data(images, n_t, channels=3):

            train_x, train_y, test_x, test_y = [], [], [], []

            for tup in images[:n_t]:

                if (np.array(tup[0]).shape[-1]) == channels:
                    train_x.append(np.array(tup[0])[:,:,:]/n_std)
                else:
                    continue

                train_y.append(tup[1])

            for tup in images[n_t:]:

                if (np.array(tup[0]).shape[-1]) == channels:
                    test_x.append(np.array(tup[0])[:,:,:]/n_std)
                else:
                    continue

                test_y.append(tup[1])

            return train_x, train_y, test_x, test_y
```

```
        # select which data to use here
        train_x, train_y, test_x, test_y = segment_data(images_portraits, n_t)

        train_x_4d = np.stack(train_x, axis=0)
        print("train_x_4d shape:", train_x_4d.shape)

        train_y_2d = np.expand_dims(np.stack(train_y, axis=0), axis=1)
        print("train_y_2d shape:", train_y_2d.shape)

        test_x_4d = np.stack(test_x, axis=0)
        print("test_x_4d shape:", test_x_4d.shape)

        test_y_2d = np.expand_dims(np.stack(test_y, axis=0), axis=1)
        print("test_y_2d shape:", test_y_2d.shape)
```

```
train_x_4d shape: (200, 224, 224, 3)
train_y_2d shape: (200, 1)
test_x_4d shape: (64, 224, 224, 3)
test_y_2d shape: (64, 1)
```

```
In [0]: # vgg_model = applications.VGG16(weights='imagenet', include_top=True)
        # vgg_model = applications.VGG16(weights='imagenet', include_top=False)

        # input_tensor = Input(shape=(160, 160, 3))
        # vgg_model = applications.VGG16(weights='imagenet', include_top=False,
        # input_tensor=input_tensor)
        vgg_model = applications.VGG16(weights='imagenet', include_top=True)
        vgg_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
```

```
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

```
In [0]: last_layer = 9

        # vgg_model = applications.VGG16(weights='imagenet', include_top=False,
        # input_shape=(224, 224, 3))
        layer_dict = dict([(layer.name, layer) for layer in vgg_model.layers])
        # print(layer_dict)
        # x = layer_dict['block2_pool'].output
        x = layer_dict['block5_conv3'].output
        # x = keras.layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu')(x)
        # x = keras.layers.MaxPooling2D(pool_size=(2, 2))(x)
        # x = keras.layers.Flatten()(x)
```

```
        # x = keras.layers.Dense(20, activation='relu')(x)
        # x = keras.layers.Dropout(0.5)(x)
        # x = GlobalAveragePooling2D()(x)
        # avg. each feat map into a value (512)
        x = keras.layers.GlobalAveragePooling2D(data_format=None)(x)
        # x = keras.layers.Dense(1, activation='sigmoid')(x)
        # x = keras.layers.Flatten()(x)
        x = keras.layers.Dense(512, activation='sigmoid', name="FC")(x)
        x = keras.layers.Dense(1, activation='softmax', name="SM")(x)

        custom_model = Model(input=vgg_model.input, output=x)

        for layer in custom_model.layers[:last_layer]: layer.trainable = False
        # custom_model.compile(loss='categorical_crossentropy',
        # optimizer='rmsprop', metrics=['accuracy'])
        custom_model.compile(loss='binary_crossentropy', optimizer='rmsprop',
                             metrics=['accuracy'])

        custom_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
```

```
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
global_average_pooling2d_1 ( (None, 512)               0
_________________________________________________________________
FC (Dense)                   (None, 512)               262656
_________________________________________________________________
SM (Dense)                   (None, 1)                 513
=================================================================
Total params: 14,977,857
Trainable params: 13,832,449
Non-trainable params: 1,145,408
_________________________________________________________________
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:20: UserWarning: Update your `Model
```

```
In [0]: # adam_1 = keras.optimizers.Adam(lr=0.1, beta_1=0.9, beta_2=0.999,
        # epsilon=None, decay=0.0, amsgrad=False)

        custom_model.compile(optimizer = "adam", loss = "binary_crossentropy",
                             metrics = ["accuracy"])
        # custom_model.compile(optimizer = adam_1, loss = "binary_crossentropy",
        # metrics = ["accuracy"])

        custom_model.fit(x = train_x_4d, y = train_y_2d, epochs = 1, batch_size = 10)
```

```
Epoch 1/1
200/200 [==============================] - 220s 1s/step - loss: -13.8699 - acc: 0.2900
```

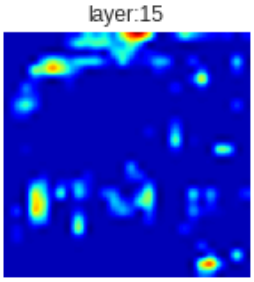
```
Out[0]: <keras.callbacks.History at 0x7fa05f9afa90>
```

Class Activation Maps

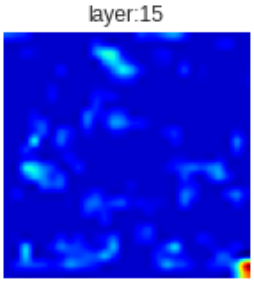
```
In [0]: import warnings
        warnings.filterwarnings('ignore')

        from vis.visualization import visualize_cam
        import matplotlib.pyplot as plt
```

```
import numpy as np
layer_start = 15
layer_end = 16

obs = 10
interesting = [1, 4, 6, 9, 12, 15]
# for i in range(len(test_y_2d)):
# for i in range(obs):
for i in interesting:

    print("label:" + str(test_y_2d[i,:][0]))
    fig = plt.figure(figsize=(15, 30)) #10*(layer_end-layer_start)
    fig.add_subplot(obs, layer_end-layer_start+1, 1)
    plt.axis("off")
    plt.title("input")
    plt.imshow(test_x_4d[i,:,:,:].reshape((224, 224, 3)))

    for j in range(layer_start, layer_end):


        heat_map = visualize_cam(custom_model,
                                 layer_idx = j,
                                 filter_indices = test_y_2d[i,:],
                                 seed_input = test_x_4d[i,:,:,:],
                                 backprop_modifier = 'guided',
                                 grad_modifier = None)

        fig.add_subplot(obs, layer_end-layer_start+1, j-layer_start+2)
        plt.axis("off")
        plt.title("layer:" + str(j))
        plt.imshow(heat_map)

    plt.show()
```

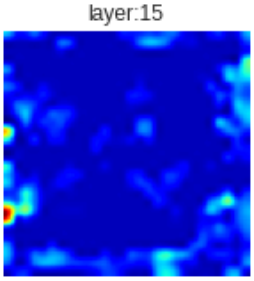
```
label:2
```



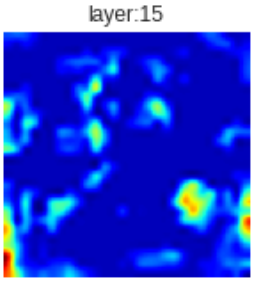
label:1



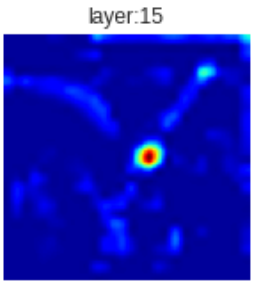
label:0



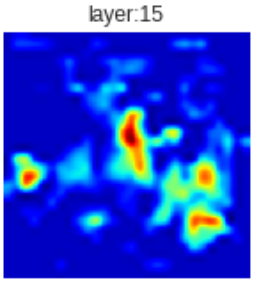
label:3



label:1



label:3



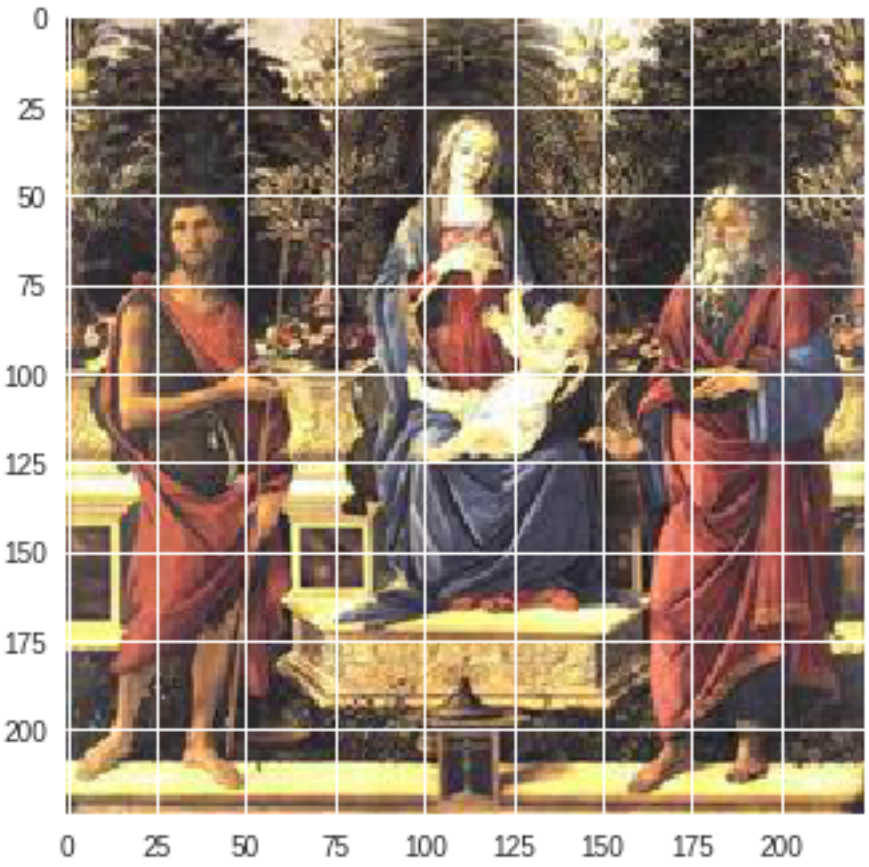

```
In [0]: preds = custom_model.evaluate(x = test_x_4d, y = test_y_2d)

        print()
        print ("Loss = " + str(preds[0]))
        print ("Test Accuracy = " + str(preds[1]))
```

```
64/64 [==============================] - 37s 571ms/step

Loss = -6.227494478225708
Test Accuracy = 0.5625
```

```
In [0]: from keras.preprocessing import image
        import matplotlib.pyplot as plt
        from matplotlib.pyplot import imshow
        from keras.applications.imagenet_utils import preprocess_input

        # for i in range(len(encoded_data)):
        for i in range(len(encoded_data_portrait)):
```

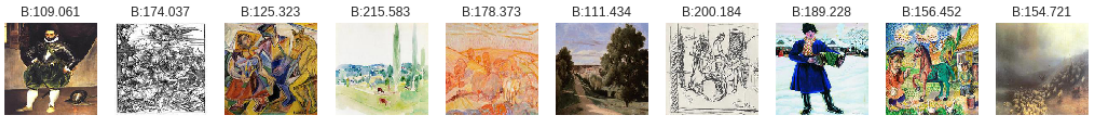
```
        #     name_temp = str(encoded_data[i][0])
            name_temp = str(encoded_data_portrait[i][0])

            name_temp = name_temp[2:]
            name_temp = name_temp[:-1]
            name_temp = name_temp + ".jpg"

            # if(len(name_temp) > length_max):
            #     continue

        #     print(name_temp)
            img_path = dir_out_images_portraits + name_temp
            img = image.load_img(img_path, target_size=(224, 224, 3))
            x = image.img_to_array(img)
            x = np.expand_dims(x, axis=0)
            x = preprocess_input(x)
            print("Example:{} Prediction:{}".format(i, custom_model.predict(x)))
            plt.axis("off")
            imshow(img)
            plt.show()
            display.clear_output(wait=True)
```

```
Example:263 Prediction:[[1.]]
```




```
In [0]: n = 10
        # i = 0

        fig = plt.figure(figsize=(18, 10))

        interesting = [0, 1, 3, 5, 11, 7, 8, 9, 10, 14]
        # for i in interesting:
        for j, i in enumerate(interesting):

            myarray = train_x_4d[i,:,:,:]
            im = Image.fromarray(np.uint8(myarray*255))
            stat = ImageStat.Stat(im)
            r,g,b = stat.rms
            b = math.sqrt(0.241*(r**2) + 0.691*(g**2) + 0.068*(b**2))

            fig.add_subplot(1, n, j+1)
            plt.axis("off")
            plt.title("B:"+str(round(b, 3)))
            plt.imshow(im)
```

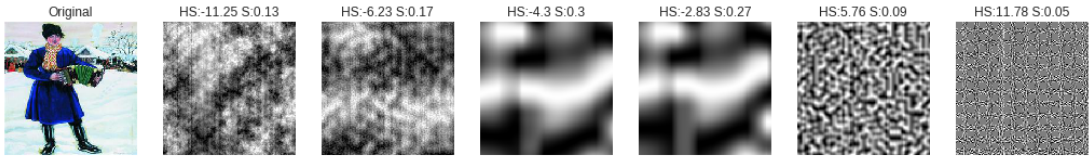
```
plt.show()
```




```
In [0]: import warnings
        warnings.filterwarnings('ignore')
        from skimage.measure import compare_ssim as ssim

        def gen_perlin(shape, scale, octaves, persistence, lacunarity):

            null_img = np.zeros(shape)

            for i in range(shape[0]):

                for j in range(shape[1]):

                    null_img[i][j] = noise.pnoise2(i/scale,
                                                   j/scale,
                                                   octaves = octaves,
                                                   persistence = persistence,
                                                   lacunarity = lacunarity,
                                                   repeatx = shape[0],
                                                   repeaty = shape[1],
                                                   base = 0)

            img = np.resize(null_img, (224, 224))
            img_3c = np.zeros((224, 224, 3))

            scaler = MinMaxScaler(feature_range=(0, 1))
            img = scaler.fit_transform(img)

            img_3c[:,:,0] = img
            img_3c[:,:,1] = img
            img_3c[:,:,2] = img

            return img_3c

        def calc_HS(scale, octaves, persistence, lacunarity):

            # hard-soft
            HS = (586.33 -
```

```
            178.78*octaves**(0.01) -
            84.20*scale**(0.01) -
            106.83*persistence**(0.02) -
            213.89*lacunarity**(0.01))

    return HS

textures = [[200.0, 100, 0.75, 2.0],
            [100.0, 4, 0.5, 6.0],
            [80.0, 75, 0.05, 2.0],
            [75.0, 1, 0.75, 3.0],
            [10.0, 1, 0.90, 0.1],
            [2.0, 1, 0.95, 0.01]]

shape = (224, 224)

fig = plt.figure(figsize=(18, 18))
columns = len(textures)+1
rows = 1

texture_landmarks = []

for i in range(columns):

    if(i==0):

        myarray = train_x_4d[i+9,:,:,:]
        orig = Image.fromarray(np.uint8(myarray*255))
        orig = np.array(orig)/255.0
        fig.add_subplot(rows, columns, i+1)
        plt.title("Original")
        plt.axis('off')
        plt.imshow((orig * 255).astype(np.uint8))

    else:

        scale = textures[i-1][0] # frequency
        octaves = textures[i-1][1]
        persistence = textures[i-1][2]
        lacunarity = textures[i-1][3]

        img = gen_perlin(shape, scale, octaves, persistence, lacunarity)

        texture_landmarks.append(img)

        hs = calc_HS(scale, octaves, persistence, lacunarity)

        fig.add_subplot(rows, columns, i+1)
```

```
            s = ssim(orig, img, multichannel=True)
            plt.title("HS:{} S:{}".format(round(hs, 2), round(s, 2)))
            plt.axis('off')
            plt.imshow((img * 255).astype(np.uint8))

    plt.show()
```



In [0]:
```
from sklearn.datasets import load_sample_image
import warnings; warnings.simplefilter('ignore') # Fix NumPy issues.
from sklearn.cluster import MiniBatchKMeans

def plot_pixels(data, title, colors=None, N=10000):

    if colors is None:
        colors = data

    # choose a random subset
    rng = np.random.RandomState(0)
    i = rng.permutation(data.shape[0])[:N]
    colors = colors[i]
    R, G, B = data[i].T

    fig, ax = plt.subplots(1, 2, figsize=(8, 3))
    ax[0].scatter(R, G, color=colors, marker='.')
    ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

    ax[1].scatter(R, B, color=colors, marker='.')
    ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

    fig.suptitle(title, size=20);


def extractTopKColors(img_in, k):

    kmeans = MiniBatchKMeans(k)
    kmeans.fit(img_in)
    kmc = kmeans.cluster_centers_/255

    return kmeans, kmc
```

```
        def filterTopKColors(kmeanb, img_in, kmc_in):

            img_out = kmc_in[kmeans.predict(img_in)]

            return img_out

        img_ = train_x_4d[i,:,:,:]
        data = img_ / 255.0
        data = scipy.misc.imresize(data, (224, 224, 3))
        data = data.reshape(224 * 224, 3)

        # run k-means to get primary colours, then transfer to the original image
        k = 5
        kmeans, kmc = extractTopKColors(data, k)
        new_colors = filterTopKColors(kmeans, data, kmc)

        img_ = scipy.misc.imresize(img_, (224, 224, 3))
        img_recolored = new_colors.reshape(img_.shape)

        print("The K-means cluster centriods (main colours) learned:")
        plt.figure(figsize=(3, 10))
        plt.imshow(kmeans.cluster_centers_.reshape(1,k,3)/255)
        plt.axis("off")
        plt.show()

        fig, ax = plt.subplots(1, 2, figsize=(10, 4), subplot_kw=dict(xticks=[], yticks=[]))
        fig.subplots_adjust(wspace=0.05)
        ax[0].imshow(img_)
        ax[0].set_title('Original Image', size=10)
        ax[1].imshow(img_recolored)
        ax[1].set_title('k-color Image', size=10);
```

```
The K-means cluster centriods (main colours) learned:
```


```
In [0]: cmap_names = [
            "joy.jpg.653x0_q80_crop-smart.jpg", # "the-happy-land-1882.jpg!Large.jpg" joy
            "savings_adobestock_111059256-1-min.jpeg",
            "love-tree.jpeg", # "talisman.jpg!Large.jpg" love
            "ingermanland-1876.jpg!Large.jpg", # fear
            "barbarossa.jpg!Large.jpg", # anger
            "lamentation-over-the-dead-christ-1617.jpg!Large.jpg"] # sadness

        k = 5
        k_map = [k, k, k, k, k, k]
        kmeans_all = {}
        kmc_all = {}

        for i in range(len(cmap_names)):
            k = k_map[i]
            img_ = np.array(Image.open(file_loc + cmap_names[i]))
            data = img_ / 255.0
            data = scipy.misc.imresize(data, (224, 224, 3))
            data = data.reshape(224 * 224, 3)
            kmeans, kmc = extractTopKColors(data, k)
            kmeans_all[cmap_names[i]] = kmeans
            kmc_all[cmap_names[i]] = kmc

        for i in range(len(kmeans_all)):
            k = k_map[i]
            plt.figure(figsize=(k, 5))
            plt.imshow(kmeans_all[cmap_names[i]].cluster_centers_.reshape(1, k, 3)/255)
            plt.axis("off")
            plt.show()
```

```
In [0]: # find nearest neighbour colour map for image
        def findNN(img, kmeans_all, k):

            data = img_ / 255.0
            data = scipy.misc.imresize(data, (224, 224, 3))
            data = data.reshape(224 * 224, 3)

            kmeans, kmc = extractTopKColors(data, k)
            k_centers = kmeans.cluster_centers_.reshape(1, k, 3)/255

            err_prev = float("inf")
            j = 0

            for i in range(len(k_map)):

                k_centers_ref = kmeans_all[cmap_names[i]].
                            cluster_centers_.reshape(1, k, 3)/255
                err_new = np.sum(np.abs(k_centers_ref - k_centers))

                if (err_new<err_prev):
                    j = i
                    err_prev = err_new

            return err_prev, j

        plt.title("Original")
        plt.imshow(img_)
        plt.axis("off")
        plt.show()

        img = train_x_4d[i,:,:,:]
        img = Image.fromarray(np.uint8(myarray*255))
        img = np.array(orig)/255.0

        # img = Image.open(file_loc + "Orig.jpg")
```

```
# img = scipy.misc.imresize(img, (224, 224, 3))
err, idx = findNN(img, kmeans_all, k)

plt.title("Err:{} Idx:{}".format(round(err, 2), idx))
plt.imshow(kmeans.cluster_centers_.reshape(1,k,3)/255)
plt.axis("off")
plt.show()

# feature vector
feat = np.zeros((6)) + 100
feat[idx] = err
```


```
In [0]: def brightness(arr):

            img = Image.fromarray(np.uint8(arr*255))
            stat = ImageStat.Stat(img)
            r,g,b = stat.rms
            b = math.sqrt(0.241*(r**2) + 0.691*(g**2) + 0.068*(b**2))

            return b


        def texture(img, texture_landmarks):

            l = len(texture_landmarks)
            texture_feat = np.zeros(l)

            for i in range(l):
                texture_feat[i] = ssim(img, texture_landmarks[i], multichannel=True)

            return texture_feat


        def colourmap(img, color_landmarks, k):

            err, idx = findNN(img, color_landmarks, k)
            colour_feat = np.zeros((6)) + 100
            colour_feat[idx] = err

            return colour_feat


        def feature_extractor(img, texture_landmarks, color_landmarks, k):

            feat_vec = np.zeros(6 + 1 + 6)

            b = brightness(np.array(img))
            t = texture(img, texture_landmarks)
```

```
    c = colourmap(img, color_landmarks, k)

    feat_vec[0] = b
    feat_vec[1:7] = t
    feat_vec[7:] = c

    return feat_vec

i = 0
img = train_x_4d[i,:,:,:]

plt.axis("off")
plt.imshow(img)
plt.show()

feature_extractor(img, texture_landmarks, color_landmarks=kmeans_all, k=5)
```

```
Out[0]: array([1.09061188e+02, 1.55347849e-01, 2.03315287e-01, 3.36788067e-01,
               3.16643073e-01, 1.14411861e-01, 5.23047857e-02, 1.00000000e+02,
```

```
        2.66448197e+00, 1.00000000e+02, 1.00000000e+02, 1.00000000e+02,
        1.00000000e+02])
```

```
In [0]: train_x_4d_cv = np.zeros((train_x_4d.shape[0], 13))
        test_x_4d_cv = np.zeros((test_x_4d.shape[0], 13))

        for i in range(train_x_4d.shape[0]):

            train_x_4d_cv[i,:] = feature_extractor(train_x_4d[i,:,:,:],
                                                   texture_landmarks,
                                                   color_landmarks=kmeans_all,
                                                   k=5)

        #     plt.axis("off")
        #     plt.imshow(train_x_4d[i,:,:,:])
        #     plt.show()
        #     print(train_x_4d_cv[i,:])
        #     display.clear_output(wait=True)


        for i in range(test_x_4d.shape[0]):

            test_x_4d_cv[i,:] = feature_extractor(test_x_4d[i,:,:,:],
                                                  texture_landmarks,
                                                  color_landmarks=kmeans_all,
                                                  k=5)

        #     plt.axis("off")
        #     plt.imshow(test_x_4d[i,:,:,:])
        #     plt.show()
        #     print(test_x_4d_cv[i,:])
        #     display.clear_output(wait=True)
```

```
In [0]: print(np.mean(train_x_4d_cv, axis=0))
        print(np.mean(test_x_4d_cv, axis=0))
```

```
[1.28710529e+02 1.59412353e-01 1.78887205e-01 3.06332227e-01
 3.01463205e-01 1.02198540e-01 5.61592977e-02 7.07038986e+01
 9.22198463e+01 7.76158883e+01 8.63650582e+01 8.97459696e+01
 8.57607365e+01]
[1.22944885e+02 1.43362760e-01 1.68456276e-01 2.76113890e-01
 2.68394161e-01 9.44704882e-02 5.33601748e-02 6.19249824e+01
 9.24218699e+01 8.62850596e+01 8.47932271e+01 9.54358470e+01
 8.16421685e+01]
```

```
In [0]: print("train_x_4d shape:", train_x_4d_cv.shape)
        print("train_y_2d shape:", train_y_2d.shape)
        print("test_x_4d shape:", test_x_4d_cv.shape)
```

```
        print("test_y_2d shape:", test_y_2d.shape)

        scaler = MinMaxScaler(feature_range=(0, 1))

        train_x_4d_cv = scaler.fit_transform(train_x_4d_cv)
        test_x_4d_cv = scaler.fit_transform(test_x_4d_cv)

        # standardize data
```

```
train_x_4d shape: (200, 13)
train_y_2d shape: (200, 1)
test_x_4d shape: (64, 13)
test_y_2d shape: (64, 1)
```

```
In [0]: print(np.mean(train_x_4d_cv, axis=0))
        print(np.mean(test_x_4d_cv, axis=0))
```

```
[0.5043914  0.58437253 0.52600935 0.4878846  0.50605    0.54467204
 0.63078749 0.70365555 0.92044718 0.77140157 0.86054839 0.89627322
 0.8571756 ]
[0.38216614 0.53954887 0.48266531 0.49559584 0.51179494 0.64072114
 0.59123862 0.61482814 0.92214438 0.85995637 0.84408748 0.95330682
 0.81427227]
```

Try a simple shallow network with only colour maps, brightness, texture

```
In [0]: def shallowNetwork(input_shape):

            X_input = Input(input_shape)
            X = keras.layers.Dense(4, activation='relu', name='FC')(X_input)
            X = keras.layers.Dense(1, activation='softmax', name='SM')(X)

            model = Model(inputs = X_input, outputs = X, name='ShallowModel')

            return model

        shallowNetwork = shallowNetwork(train_x_4d_cv[0,:].shape)

        opt = keras.optimizers.Adam(lr=0.0001, beta_1=0.9,
                                    beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)

        shallowNetwork.compile(optimizer = opt, loss = "binary_crossentropy",
                               metrics = ["accuracy"])
        shallowNetwork.summary()
        shallowNetwork.fit(x = train_x_4d_cv, y = train_y_2d, epochs = 10, batch_size = 10)
        preds = shallowNetwork.evaluate(x = test_x_4d_cv, y = test_y_2d)
```

```
        print ("Loss = " + str(preds[0]))
        print ("Test Accuracy = " + str(preds[1]))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 13)                0
_________________________________________________________________
FC (Dense)                   (None, 4)                 56
_________________________________________________________________
SM (Dense)                   (None, 1)                 5
=================================================================
Total params: 61
Trainable params: 61
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
200/200 [==============================] - 1s 3ms/step - loss: -13.8699 - acc: 0.2900
Epoch 2/10
200/200 [==============================] - 0s 157us/step - loss: -13.8699 - acc: 0.2900
Epoch 3/10
200/200 [==============================] - 0s 143us/step - loss: -13.8699 - acc: 0.2900
Epoch 4/10
200/200 [==============================] - 0s 138us/step - loss: -13.8699 - acc: 0.2900
Epoch 5/10
200/200 [==============================] - 0s 131us/step - loss: -13.8699 - acc: 0.2900
Epoch 6/10
200/200 [==============================] - 0s 189us/step - loss: -13.8699 - acc: 0.2900
Epoch 7/10
200/200 [==============================] - 0s 146us/step - loss: -13.8699 - acc: 0.2900
Epoch 8/10
200/200 [==============================] - 0s 181us/step - loss: -13.8699 - acc: 0.2900
Epoch 9/10
200/200 [==============================] - 0s 165us/step - loss: -13.8699 - acc: 0.2900
Epoch 10/10
200/200 [==============================] - 0s 200us/step - loss: -13.8699 - acc: 0.2900
64/64 [==============================] - 0s 3ms/step
Loss = -6.227494478225708
Test Accuracy = 0.5625
```

ANNEX E: 3 layer CNN classifier

August 23, 2019

In [1]:
```
import math
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
import PIL
from scipy import ndimage
import tensorflow as tf
from tensorflow.python.framework import ops
from cnn_utils import *
from PIL import Image
import glob
import pandas as pd
from pandas import ExcelWriter
from pandas import ExcelFile
import base64
from IPython import display
import time
import os
from os import listdir
from os.path import isfile, join
from keras import applications
from keras.layers import Input
import keras
from keras.models import Model


%matplotlib inline
np.random.seed(1)
```

```
C:\Users\R510J\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of the
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Load images

In [2]:
```
# Read the csv that contains the urls
data_full = pd.read_csv("good_data.csv", sep="|", header=0, skiprows=0)
# Define the directories
dir_out_images_good = "images_good/" # All images labeled by emotion
dir_out_images_portraits = "images_portraits/" # All portraits labeled by emotion
length_max = 75
```

We create a dataframe with the Title, Emotion and Genre of all the images

In [3]:
```
# Select the relevant columns
good_data = data_full.loc[:, ["TitleNames", "Emotion", "Genre"]]
print("good_data shape:{}".format(good_data.shape))
good_data.head()
```

```
good_data shape:(659, 3)
```

Out[3]:

	TitleNames	Emotion	Genre
0	The Fall of Lucifer	SADNESS	RELIGIOUS PAINTING
1	The Four Horsemen of the Apocalypse, Death, Fa...	ANGER	RELIGIOUS PAINTING
2	The Four Horsemen of the Apocalypse, Death, Fa...	ANGER	RELIGIOUS PAINTING
3	Peasants and horse	JOY	GENRE PAINTING
4	Pancake Tuesday in a Village. Maslenitsa	JOY	GENRE PAINTING

We now create a data frame with only the portraits

In [4]:
```
# Select the portraits
portrait_data = good_data.loc[good_data["Genre"] == "PORTRAIT"]
print("portrait_data shape:{}".format(portrait_data.shape))
portrait_data.head()
```

```
portrait_data shape:(103, 3)
```

Out[4]:

	TitleNames	Emotion	Genre
135	Vincenzo Anastagi	OPTIMISM	PORTRAIT
145	Pachtuwa-Chta, an Arrikkara Warrior, plate 27 ...	OPTIMISM	PORTRAIT
146	Portrait of a Knight	OPTIMISM	PORTRAIT
158	Young lady with a bird and dog	OPTIMISM	PORTRAIT
226	Untitled	JOY	PORTRAIT

The titles of the images in directories images_good and images_portraits are the base64 encoding of the title of the painting. Therefore, we encode the titles of our data frames using the same encoding, in order to select the desired images by their name

```
In [26]: def encode_data(df):

             encoded_data = []

             for i in range(df.shape[0]):
                 # Select the title of the painting
                 a = df.iloc[i, 0]
                 # Encode the title
                 b = a.encode("utf-8")
                 temp = base64.urlsafe_b64encode(b)
                 #Select the emotion
                 c = df.iloc[i, 1]
                 # Add both the encoded title and emotion to the list
                 encoded_data.append((temp, c))

             return encoded_data

         # Encode the data from good_data
         encoded_data = encode_data(good_data)
         print("encoded_data: {}".format(len(encoded_data)))

         # Encode the data from portrait_data
         encoded_data_portrait = encode_data(portrait_data)
         print("encoded_data_portrait: {}".format(len(encoded_data_portrait)))
```

```
encoded_data: 659
encoded_data_portrait: 103
```

```
In [27]: encoded_data[0]
```

```
Out[27]: (b'VGhlIEZhbGwgb2YgTHVjaWZlcg==', 'SADNESS')
```

```
In [28]: encoded_data_portrait[0]
```

```
Out[28]: (b'VmluY2Vuem8gQW5hc3RhZ2k=', 'OPTIMISM')
```

We now define the function to import the images from the corresponding directory

```
In [8]: basewidth, baseheight, channels = 224, 224, 3

        def read_images(names, dir_in, label_dict):

            results = []

            for i in range(len(names)):

                name_temp = str(names[i][0])
                name_temp = name_temp[2:]
```

```
                name_temp = name_temp[:-1]
                name_temp = name_temp + ".jpg"

                # We won't load the images with an invalid name
                if(len(name_temp) > length_max):
                     continue

                img = Image.open(dir_in + name_temp)
                print("Images Loaded: {}".format(i))

                # Resize the images
                wpercent = (basewidth / float(img.size[0]))
                hsize = int((float(img.size[1]) * float(wpercent)))
                hpercent = (baseheight / float(img.size[1]))
                wsize = int((float(img.size[0]) * float(hpercent)))
                img = img.resize((basewidth, baseheight), PIL.Image.ANTIALIAS)

                display.clear_output(wait=True)
                label = names[i][1]

                # Add the corresponding label
                if label not in label_dict:
                    label_final = 0
                else:
                    label_final = label_dict[label]

                # Add the image and the label to the list
                results.append((img, label_final))

            return results

        label_dict = {"SADNESS": 0, "FEAR":0, "DISGUST":0, "ANGER":0, "NEUTRAL":0, "nan":0,
                      "LUST":0, "ENVY":0, "SURPRISE":0,
                      "OPTIMISM":1, "JOY":1, "LOVE":1}
```

```
In [9]: images_good = read_images(encoded_data, dir_out_images_good, label_dict) # label = 1
        print("images_good: {}".format(len(images_good)))
```

```
images_good: 626
```

```
In [10]: images_portraits = read_images(encoded_data_portrait,
                                        dir_out_images_portraits, label_dict) # label = 1
         print("images_portraits: {}".format(len(images_portraits)))
```

```
images_portraits: 94
```

Segment and standardize data

```
In [11]: n_t = 70 # training examples
         n_c = 0 # channel number to use
         n_std = 255.0

         def segment_data(images, n_t, channels=3):

             train_x, train_y, test_x, test_y = [], [], [], []

             for tup in images[:n_t]:

                 if (np.array(tup[0]).shape[-1]) == channels:
                     train_x.append(np.array(tup[0])[:,:,:]/n_std)
                 else:
                     continue

                 train_y.append(tup[1])

             for tup in images[n_t:]:

                 if (np.array(tup[0]).shape[-1]) == channels:
                     test_x.append(np.array(tup[0])[:,:,:]/n_std)
                 else:
                     continue

                 test_y.append(tup[1])

             return train_x, train_y, test_x, test_y

         # select which data to use here
         train_x, train_y, test_x, test_y = segment_data(images_portraits, n_t)

         train_x_4d = np.stack(train_x, axis=0)
         print("train_x_4d shape:", train_x_4d.shape)

         train_y_2d = np.expand_dims(np.stack(train_y, axis=0), axis=1)
         print("train_y_2d shape:", train_y_2d.shape)

         test_x_4d = np.stack(test_x, axis=0)
         print("test_x_4d shape:", test_x_4d.shape)

         test_y_2d = np.expand_dims(np.stack(test_y, axis=0), axis=1)
         print("test_y_2d shape:", test_y_2d.shape)
```

```
train_x_4d shape: (70, 224, 224, 3)
train_y_2d shape: (70, 1)
test_x_4d shape: (24, 224, 224, 3)
```

```
test_y_2d shape: (24, 1)
```

We will follow a similar structure to the CNN programming assignments from the course

Create Placeholders

```
In [12]: def create_placeholders(n_H0, n_W0, n_C0, n_y):
             """
             Creates the placeholders for the tensorflow session.

             Arguments:
             n_H0 -- scalar, height of an input image
             n_W0 -- scalar, width of an input image
             n_C0 -- scalar, number of channels of the input
             n_y -- scalar, number of classes

             Returns:
             X -- placeholder for the data input, of shape [None, n_H0, n_W0, n_C0] and
             dtype "float"
             Y -- placeholder for the input labels, of shape [None, n_y] and dtype "float"
             """

             X = tf.placeholder(tf.float32, shape = (None, n_H0, n_W0, n_C0))
             Y = tf.placeholder(tf.float32, shape = (None, n_y))

             return X, Y
```

Initialize parameters

```
In [13]: def initialize_parameters():
             """
             Initializes weight parameters to build a neural network with tensorflow.
             The shapes are:
                                 W1 : [4, 4, 3, 8]
                                 W2 : [2, 2, 8, 16]
             Returns:
             parameters -- a dictionary of tensors containing W1, W2
             """

             tf.set_random_seed(1)

             W1 = tf.get_variable("W1", [4,4,3,8], dtype = tf.float32, initializer =
                                  tf.contrib.layers.xavier_initializer(seed = 0))
             W2 = tf.get_variable("W2", [2,2,8,16], dtype = tf.float32, initializer =
                                  tf.contrib.layers.xavier_initializer(seed = 0))

             parameters = {"W1": W1,
```

```
                  "W2": W2}

    return parameters
```

Forward propagation

```
In [14]: def forward_propagation(X, parameters):
    """
    Implements the forward propagation for the model:
    CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN ->
    -> FULLYCONNECTED

    Arguments:
    X -- input dataset placeholder, of shape (input size, number of examples)
    parameters -- python dictionary containing your parameters "W1", "W2"
                  the shapes are given in initialize_parameters

    Returns:
    Z3 -- the output of the last LINEAR unit
    """

    # Retrieve the parameters from the dictionary "parameters"
    W1 = parameters['W1']
    W2 = parameters['W2']

    # CONV2D: stride of 1, padding 'SAME'
    Z1 = tf.nn.conv2d(X, W1, strides = [1,1,1,1], padding = 'SAME')
    # RELU
    A1 = tf.nn.relu(Z1)
    # MAXPOOL: window 8x8, sride 8, padding 'SAME'
    P1 = tf.nn.max_pool(A1, ksize = [1,8,8,1], strides = [1,8,8,1],
                        padding = 'SAME')
    # CONV2D: filters W2, stride 1, padding 'SAME'
    Z2 = tf.nn.conv2d(P1, W2, strides = [1,1,1,1], padding = 'SAME')
    # RELU
    A2 = tf.nn.relu(Z2)
    # MAXPOOL: window 4x4, stride 4, padding 'SAME'
    P2 = tf.nn.max_pool(A2, ksize = [1,4,4,1], strides = [1,4,4,1],
                        padding = 'SAME')
    # FLATTEN
    P2 = tf.contrib.layers.flatten(P2)
    # FULLY-CONNECTED without non-linear activation function (not not call softmax).
    # 6 neurons in output layer. Hint: one of the arguments should
    # be "activation_fn=None"
    Z3 = tf.contrib.layers.fully_connected(P2, 3, activation_fn=None)

    return Z3
```

Compute cost

```
In [15]: def compute_cost(Z3, Y):

             cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = Z3,
                                                                           labels = Y))

             return cost
```

Model

```
In [24]: def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.009,
                   num_epochs = 100, minibatch_size = 70, print_cost = True):
             """
             Implements a three-layer ConvNet in Tensorflow:
             CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN ->
             -> FULLYCONNECTED

             Arguments:
             X_train -- training set, of shape (None, 64, 64, 3)
             Y_train -- test set, of shape (None, n_y = 3)
             X_test -- training set, of shape (None, 64, 64, 3)
             Y_test -- test set, of shape (None, n_y = 3)
             learning_rate -- learning rate of the optimization
             num_epochs -- number of epochs of the optimization loop
             minibatch_size -- size of a minibatch
             print_cost -- True to print the cost every 100 epochs

             Returns:
             train_accuracy -- real number, accuracy on the train set (X_train)
             test_accuracy -- real number, testing accuracy on the test set (X_test)
             parameters -- parameters learnt by the model. They can then be used to predict.
             """

             ops.reset_default_graph()
             tf.set_random_seed(1)
             seed = 3
             (m, n_H0, n_W0, n_C0) = X_train.shape
             n_y = Y_train.shape[1]
             costs = []

             # Create Placeholders
             X, Y = create_placeholders(n_H0, n_W0, n_C0, n_y)

             # Initialize parameters
             parameters = initialize_parameters()

             # Forward propagation:
```

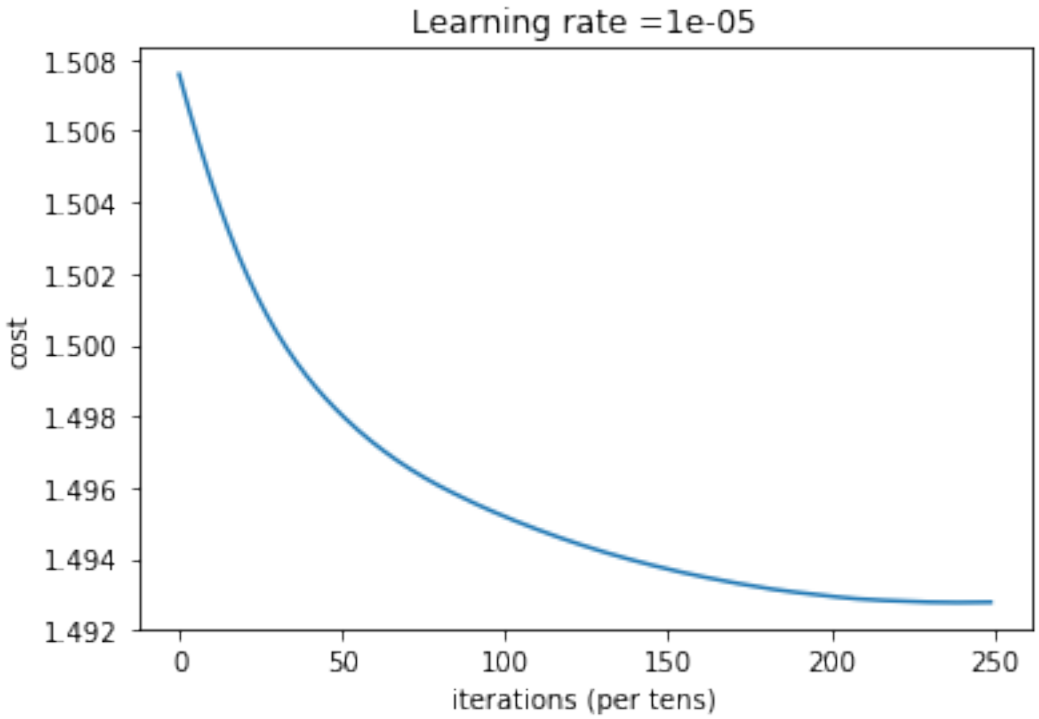
```
Z3 = forward_propagation(X, parameters)

# Cost function:
cost = compute_cost(Z3, Y)

# Backpropagation:
optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate).minimize(cost)

init = tf.global_variables_initializer()

# Start the session to compute the tensorflow graph
with tf.Session() as sess:

    sess.run(init)

    # Training loop
    for epoch in range(num_epochs):

        minibatch_cost = 0.
        # number of minibatches of size minibatch_size in the train set
        num_minibatches = int(m / minibatch_size)
        seed = seed + 1
        minibatches = random_mini_batches(X_train, Y_train, minibatch_size,
                                          seed)

        for minibatch in minibatches:

            # Select a minibatch
            (minibatch_X, minibatch_Y) = minibatch

            _ , temp_cost = sess.run([optimizer, cost], feed_dict=
                                     {X: minibatch_X, Y: minibatch_Y})

            minibatch_cost += temp_cost / num_minibatches


        # Print the cost every epoch
        if print_cost == True and epoch % 5 == 0:
            print ("Cost after epoch %i: %f" % (epoch, minibatch_cost))
        if print_cost == True and epoch % 1 == 0:
            costs.append(minibatch_cost)


    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
```

```
                plt.show()

                # Calculate the correct predictions
                predict_op = tf.argmax(Z3, 1)
                correct_prediction = tf.equal(predict_op, tf.argmax(Y, 1))

                # Calculate accuracy on the test set
                accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
                print(accuracy)
                train_accuracy = accuracy.eval({X: X_train, Y: Y_train})
                test_accuracy = accuracy.eval({X: X_test, Y: Y_test})
                print("Train Accuracy:", train_accuracy)
                print("Test Accuracy:", test_accuracy)

                return train_accuracy, test_accuracy, parameters
```

```
In [25]: _, _, parameters = model(train_x_4d, train_y_2d, test_x_4d, test_y_2d,
                                  learning_rate = 0.00001, num_epochs = 250)
```

```
Cost after epoch 0: 1.507588
Cost after epoch 5: 1.505980
Cost after epoch 10: 1.504525
Cost after epoch 15: 1.503241
Cost after epoch 20: 1.502127
Cost after epoch 25: 1.501168
Cost after epoch 30: 1.500344
Cost after epoch 35: 1.499638
Cost after epoch 40: 1.499027
Cost after epoch 45: 1.498494
Cost after epoch 50: 1.498022
Cost after epoch 55: 1.497600
Cost after epoch 60: 1.497220
Cost after epoch 65: 1.496877
Cost after epoch 70: 1.496568
Cost after epoch 75: 1.496289
Cost after epoch 80: 1.496036
Cost after epoch 85: 1.495803
Cost after epoch 90: 1.495582
Cost after epoch 95: 1.495375
Cost after epoch 100: 1.495180
Cost after epoch 105: 1.494994
Cost after epoch 110: 1.494818
Cost after epoch 115: 1.494650
Cost after epoch 120: 1.494491
Cost after epoch 125: 1.494342
Cost after epoch 130: 1.494200
Cost after epoch 135: 1.494066
Cost after epoch 140: 1.493941
```

```
Cost after epoch 145: 1.493823
Cost after epoch 150: 1.493710
Cost after epoch 155: 1.493605
Cost after epoch 160: 1.493508
Cost after epoch 165: 1.493418
Cost after epoch 170: 1.493332
Cost after epoch 175: 1.493251
Cost after epoch 180: 1.493175
Cost after epoch 185: 1.493110
Cost after epoch 190: 1.493052
Cost after epoch 195: 1.492999
Cost after epoch 200: 1.492949
Cost after epoch 205: 1.492901
Cost after epoch 210: 1.492864
Cost after epoch 215: 1.492839
Cost after epoch 220: 1.492818
Cost after epoch 225: 1.492798
Cost after epoch 230: 1.492785
Cost after epoch 235: 1.492776
Cost after epoch 240: 1.492775
Cost after epoch 245: 1.492779
```




```
Tensor("Mean_1:0", shape=(), dtype=float32)
Train Accuracy: 0.8142857
```

```
Test Accuracy: 0.8333333
```