Final Report CEP
Capstone Project 2020

European Art Classification as a Service

Project members: Gan Xin Xiang, Trevor Lee Jia Jin

Final presentation slides can be found here.

Project Introduction [≈ 0.5 pages]

This project is for art classification of European Art from the start of the 15th to the end of the 19th century. The front end application is a website for users to upload pictures of art for identification and classification. The current classes for classification can be grouped under three broad categories: Timeline, Art school, and art type. Timeline refers to the actual time period in which the bulk of the artwork was produced (1401-1450, 1451-1500 and so on all the way until 1851-1900). There are a total of 10 art classes for the timeline from 1401-1900, split into 50-year blocks. Art school refers to the location where the artists were schooled in/influenced by. There are 6 art classes for the art school, namely Dutch, Flemish, French, German, Italian and Spanish. Art genre refers to the general type of painting produced. There are 7 classes for the art genre, namely genre, historical, landscape, mythological, portrait, religious and still-life.

This project is restricted to classifying paintings from Europe from 1401-1900. Due to limitations in our dataset, there are insufficient paintings from before 1401. Also, we decided not to classify art pieces from 1901 onwards since there are many different classes that emerged that are vastly different from the art from previous eras, such as cubism and surrealism, which might throw the training models completely off tangent.

This project utilises Deep Neural Networks in the form of Convolutional Neural Networks for image classification. The machine learning architecture used is NasNetA-Large.

This project was proposed since there is currently no product for classification of European paintings. When group member Xin Xiang visited the Louvre, he saw many beautiful European paintings but had no idea of the history behind it. Without reading the label and searching up the painting, he would have no idea when it was painted, and from where. Thus this project was conceived to help folks like him know more about European art. Possible use cases would be to quickly find out information about paintings seen in galleries. Another possible use case would be to classify one's own paintings for learning and enrichment purposes, if an aspiring artist is trying to emulate a certain art period/school. This can also be used in art schools for educational purposes as a learning aid, as it is quicker and easier to use than manually searching Google for the artwork and sifting through the vast amounts of information trying to glean important data. Other educational uses also include identifying which artist/artwork had influences from various schools (for example, 60% French and 40% German predicted influence for an artwork is a strong indicator that these 2 countries/schools had considerable influence on an artist). This product can also be used to identify and learn more about artists that do not conform to the norm (for example, an artist can be from Flanders in the 1800s, but have a painting style of the 1650s and from Italy as predicted by the machine learning model), which can bring to light forgotten links and influences of artists on each other and offer important clues to researchers on artists' lives that

may be unknown before the usage of our web app.

Machine Learning Algorithm [≈ 0.5 pages]

Since this task was an image classification task, a convolutional neural network must be used. A typical convolutional neural network involves a convolution layer, which applies a filter to the pixels in an image (converted to a multidimensional array beforehand), for identification of edges and corners for example. A matrix of pixels (1x1, 3x3, 5x5 or 7x7 usually) have their RGB values computed and run through a filter (multiplied with a fixed matrix), and the output feature map is a smaller matrix. This filter is applied for every single possible 1x1, 3x3, 5x5 or 7x7 matrix (depending on the filter) in the image. Some types of convolutional filters include mean, median, Gaussian blur, sharpen and edge detection.

This convolutional filter is then passed through a ReLU (rectified linear unit) as an activation function which introduces nonlinearity into the model. ReLU returns either 0 or the value of the passed value, whichever is higher. Then the feature map is passed through a pooling layer to downsample the output feature map while retaining critical information. This works in a similar way to convolution but makes the resulting matrix even smaller to reduce compute times. The final layer of the image classification architecture is a Dense layer which connects to all nodes of the previous layer, which allows it to make a final prediction of the class as the output.

We used NASNet for our model architecture. This is because it has the highest ImageNet top 1 accuracy amongst all the available in-built image classification architectures in Tensorflow 2.2 (82.7%). In Tensorflow 2.3, there is the availability of EfficientNetB7 which returns a higher ImageNet top 1 accuracy (84.4%). However this was not utilised in the end since EfficientNetB7 requires massive RAM for computation (batch size of only 2 on a 11GB RAM GTX 1080 Ti, any increase would result in an Out Of Memory error). For NASNet, a passable batch size of 8 on the same GPU can be used before hitting an OOM error, and was therefore used.

NASNet presents advantages over hand-crafted architectures such as ResNet50 (top 1 accuracy of 77.15%) in image classification as the architecture for NASNet was determined by automated algorithms. The NASNet research paper suggests that an automated method for architecture building was instead used, where there is an automated search for the best architecture achieved by letting an algorithm build its own architecture and choose its own activation functions, then training on a set number of epochs on the same training data. Promising architectures with high validation accuracy were then chosen and refined further by this automated program, and the architecture with highest top 1 accuracy in the ImageNet dataset was chosen. Some parameters for the automated architecture building were set (search space): Skip connections were allowed, and the basic structure of CNN had to be used (convolutional filter, pooling layer, and dense layer at the end to connect all nodes and form a final prediction). Tens of thousands of architectures were considered, a few weeks of GPU compute using hundreds of powerful GPUs (such as Tesla

P100s) were used in the quest to obtain the best architecture. More information on how NASNet was developed and how it works can be found here: https://arxiv.org/pdf/1802.01548.pdf

Softmax was used as the activation function for the output of the final Dense layer. This was chosen since we are doing a multi-class classification task and not binary classification where a sigmoid activation function would be more appropriate.

The learning rate and optimiser function was finally chosen based on trial and error. We used the Stochastic Gradient Descent method with a learning rate of about 0.0001 to 0.00025, depending on which of the three models was used. Momentum of 0.88 was used, and Nesterov momentum was disabled. Learning rate determines the amount in which weights are updated, a lower learning rate means that weights are updated but to a smaller magnitude. Stochastic Gradient Descent optimiser is the algorithm which determines how much weights are to be updated, involving finding the partial derivative of the loss function with respect to each parameter. Since only one data point is used in the calculation (picked at random) instead of all the data points for the gradient calculation, SGD is much faster than Gradient Descent which requires computation using all data points. As a trade off, gradient calculated from SGD is not the true gradient value but an approximation, requiring more iterations to converge on the minima.

Loss function of Sparse Categorical Cross Entropy was used. This was because we are classifying more than 2 classes and our data is labelled with integers instead of one-hot encodings.

The input of this ML algorithm are 3 dimensional arrays (331x331x3), obtained by processing 331x331 RGB images. The output is a 1000 element array that returns the probability of the model from 0 to 1 (determined by softmax). Since we are using ImageNet weights for transfer learning, we cannot modify the output to return the number of classes for our tasks. Thus we took the first x elements of the array and computed our own probability (further explained in the Results section), where x is the number of classes in the classification problem. Number of classes for each task can be found in the introduction.

Dataset and Features [≈ 0.5 pages]

Images of paintings were obtained from www.wga.hu. The website provides a CSV file for offline viewing and search for particular artworks. However there is no direct download of all the images of the artworks. But a pattern was spotted in the naming scheme image links, and a script was quickly written to scrape the images with help of the labelling of the CSV file. Because artworks in the CSV file does not change order, we can safely label each image in order of appearance without ambiguity (for example, the first painting in the list is named as 0 and the second as 1, and so on).

The website includes other forms of art such as sculptures, architecture and graphics. These are filtered out and not downloaded as they are irrelevant to our project. In total the download took about ~3 days straight since the Python script is single-threaded and cannot download asynchronously. There are a total of 31742 paintings downloaded, which is a large enough dataset to achieve high accuracy.

The images are split into three categories: Type, school and timeline. Images are sorted into each category by class with reference to the CSV file. Since each category is separately trained and will have a separate model, the image dataset was not split into “type, school and timeline” arbitrarily. Instead all 31742 images were sorted in each category to its respective classes.

Training, validation and test sets were created. Since the CSV file sorts artworks by artist name, in order to prevent any bias that would be created by arbitrary cutting and pasting (perhaps a few artists with a distinct art style is removed from training set accidentally, decreasing accuracy significantly), images are randomly selected to be in the validation and test sets. This is achieved through using the random.randint function from 1 to 20000, and this randomly generated integer is passed through modulo 20. Images after modulo 20 that had 3 particular RNG values were sorted to the validation set, while images with another 3 values were sorted to the test set. The remaining images were sorted into the training set. This produces approximately 70% training set, 15% validation and test set split of images (minor variation due to RNG). View Table 1 for more information.

We found that paintings from before 1400 were extremely scarce. Thus all paintings from before 1400 were excluded from all categories. There are no paintings from the 1900s in the image dataset. Besides, paintings from the 20th century are too broad and deserve a separate model trained to separate 1900s art alone. More information can be found in Table 2.

In the art type category, we removed the “study” and the “interior” class. The “study” class only had about 50 samples in the training set, while the “interior” class consisted of images that were not paintings on canvas but were images of paintings on walls of churches for example. The “interior” class also had a low class count and was thus removed.

There are many art schools, over 20 after sorting. Any art school with less than 500 images in the training set were purged. More information can be found in Table 2.

	Train	Validation	Test	Total
Art Timeline	20714	4312	4311	29337
Art School	18546	3918	4116	26580
Art Type/Genre	19534	4266	4023	27823

Table 1: Number of images for train/validation/test sets in each category

Art timeline		Art school		Art type/genre	
Class	Image count	Class	Image count	Class	Image count
1401-1450	1180	Dutch	3032	Genre	1917
1451-1500	2598	Flemish	2531	Historical	630
1501-1550	3057	French	2999	Landscape	2866
1551-1600	1796	German	1264	Mythological	2179
1601-1650	3803	Italian	7864	Portrait	3293
1651-1700	2366	Spanish	856	Religious	7668
1701-1750	1554			Still-life	981
1751-1800	1162				
1801-1850	1135				
1851-1900	2063				

Table 2: Number of images in each class for the training set in each category

Sample images:

Art Timeline:

Painting from 1401-1450

Painting from 1601-1650

Painting from 1851-1900

Art school:

Flemish painting

German painting

Italian painting

Art genre/type

Genre painting

Historical painting

Mythological painting

Still-life painting

Machine Learning Development [≈ 0.5 - 1 pages]

Tensorflow was used. Version used in the final model and application is tf-nightly version 2.3.0.dev20200624. Prerequisites to be installed can be found in the README file.

Various development environments were used. In the initial experiments and models, they were trained on Google Colaboratory, using their free GPUs. Up to 4 Google Colab instances would be running at once in order to speed up training. Google Colab instances were hard reset until a preferable GPU was assigned (Tesla P100 with 16GB GPU RAM or Tesla T4 with 16GB GPU RAM). Towards the deadline, Cloud GPUs were utilised from genesiscloud.com, with both single GTX1080Ti (11GB GPU RAM) and dual GTX1080Ti configurations explored. The dual GPU configuration sped up computation times by almost 2x compared to a single GPU setup with GTX1080Ti's.

We had attempted to use Google Colab's Tensorflow Processing Units (TPU) and Google Cloud Compute's Tesla V100 GPU and TPUv3. However, after hours of research, we found out that Google Colab's TPU was unavailable to use in conjunction with Tensorflow's in-built function called flow_from_directory which we had used to process our images, and we thus had to abandon this venture. As for Google Cloud Compute, even after upgrading from the trial and into a paid account, Google Cloud Compute team did not approve of our repeated quota increases (from 0 to 1) for both GPU and TPU and we thus had to abandon this venture as well. This was extremely disheartening as research has shown that TPUs can speed up training in image classification projects by about 10x or more. In addition, the large RAM sizes per each TPU core (16GB) allows for even larger batch sizes, further decreasing training times. As for the Tesla V100, it is the most powerful GPU for machine learning available currently and we are extremely disappointed to be unable to utilise it.

A Jupyter notebook was used for training on Google Colab. Since Google Colab runs in Linux, Linux commands had to be used, such as sudo, ls, mv, and cp. Google Drive containing the training/validation sets were mounted, and unzipped into a newly created RAMdisk directory. Since the Colab instance has an abundance of RAM, we figured that putting our images in a RAMdisk would remove any potential I/O bottlenecks. Models were directly saved to our mounted Google Drive once training completes. A sample Google Colab file with commands that we had used can be found here.

As for Cloud computing, instances created from genesiscloud.com were in Ubuntu 18.04, and were headless. Therefore, we used Secure Shell (SSH) to connect to our instances via the program PuTTY, and uploaded/downloaded files to and from the server using SSH File Transfer Protocol (SFTP) through the program FileZilla. Linux commands were also used to create a RAMdisk and unzip files, and for general navigation. There were some issues with the CUDA driver on the server, therefore we had to run the command "conda install -c anaconda cudatoolkit=10.1 cudnn=7.6.4" in order to install the missing libraries and packages.

The evaluation of model accuracy was all through my own computer. There is a GTX1070 with 8GB GPU RAM available on my computer for evaluation. My computer runs Windows 10 Pro. Images and models were stored not on a RAMdisk but on a SATA SSD.

We had to use tf-nightly instead of the latest stable version Tensorflow 2.2 because TF2.2 was extremely buggy and generally incompatible with our code. Using Tensorflow 2.2, there were many errors. For example, there are various GPU errors, and the predict function was broken in TF2.2 (cannot import trained model for prediction, returned path incorrect error despite it being correct) and hence we cannot do any inference. Different bugs were also present in the TF2.3-rc0 release that rendered our code broken and hence cannot be used as well.

The download script and sorting code is saved as download.py. There are comments that explain the code found in the Python file itself, so please do open the file to read. There is also the first draft of our self-made architecture which was not used and thus not extensively commented on. The download.py file extracts paintings from the CSV list, and then downloads them from the Internet. It also sorts the images into various classes. Categories to sort into (timeline/type/school) can be changed in the code, as explained in the comments in the file.

Our final training code is located at nasnet.py. There are comments that explain the code found in the Python file itself, so please do open the file to read. Images are augmented through horizontal flipping, and are shifted, rotated and zoomed randomly to ensure that training images are not the same every time. This reduces the degree of overfitting. RGB values are normalised to fall within 0-1 to facilitate training. Class weights are used so that each class is weighted equally, and that overrepresentation (or underrepresentation) of a particular class in terms of images will not bias the training model. A custom function for measuring the top 3 accuracy of the predictions is also used. Some layers are frozen during training to reduce training times and ensure that only weights of higher-level layers are altered, so that transfer learning can be used better (explained in the next section). A learning rate scheduler is used to reduce the learning rate after a certain number of epochs for finer training and better convergence on minima. This code trains the NASNet-Large architecture, and saves every epoch when there is an improvement in the validation loss. When there is no improvement in the validation accuracy after 5 epochs, training is stopped early to save computational resources and time.

The evaluation of the model accuracy can be found at predict.py. There are comments that explain the code found in the Python file itself, so please do open the file to read. There is nothing new in the code that was not explained previously, except the evaluate function. In this file, the test set is loaded, the model is loaded, then evaluated using GPU with images fed through flow_from_directory. The test accuracy, loss and top-3 accuracy is printed when evaluation has finished. A sample predict function can also be found in the same file. It is used as a testing code for implementation in our application (application.py). Since we had used transfer learning with pretrained ImageNet weights, our last Dense layer had to be for 1000 classes. Thus we had to use the values of first x elements in the returned array (output of inference), where x is the number of classes. The other ~990 dummy classes were ignored as its predicted probability was in the order of 10^-6 to 10^-7 , and thus negligible and does not affect the accuracy and loss

during training/validation/evaluation stages in any significant manner. However, this was still accounted for in our code, where the proper accuracy without these dummy classes was calculated.

The code and images used in our programs can be found here.

Training Experiments/Results [≈ 1 - 2 pages]

Initially, we made a basic convolutional neural network consisting of 3 Convolution2D+Pooling layers slapped together, followed by Dense layers to predict the final class. Accuracy was extremely poor, at around 20-25% for the 10-class timeline prediction. Besides, we only used about 15 thousand images since not all were downloaded, further explaining the extremely poor accuracy achieved.

However, we soon discovered models and architectures for image classification that were already available for free online, and these models will be way better than whatever we can come up with. These architectures were produced by funded PhD teams or even employees at institutions such as Google, and it is in our best interests to use their research instead of trying to “outsmart” them with our own neural networks. Examples of such prebuilt neural networks for image classification are ResNet50, InceptionResNetV2, Xception, NASNet-Large and EfficientNetB7.

Initially, we had only heard of ResNet50 from Mr Akmal’s recommendation and decided to give it a go. Accuracy instantly went up to about 35%, but was still poor. It was at this stage where we did the bulk of our hyperparameters training such as batch size and optimiser. We have discovered that a larger batch size generally means better training accuracy, and cuts down on training times by approximately half. In addition, we implemented image augmentation from this stage onwards to prevent overfitting (flip/zoom/rotate/stretch image randomly). Two main optimisers were used in testing: Adam and Stochastic Gradient Descent. The best learning rates for each optimiser were determined in this stage to converge onto global minima accurately and quickly. We thus used the maximum batch size of 32 available on a 8GB RAM GPU (GTX1070), with compute times per epoch of about 10 minutes.

Then, after some bored and casual reading of Tensorflow and Keras documentation, we discovered transfer learning. Transfer learning is the utilisation of weights from pretrained models, and adapting these tuned weights for use in our own application. This reduces time taken to converge on global minima, and also increases accuracy as these weights are tuned on a much larger dataset and can generalise better. For example, a model trained on recognising dog breeds with an extremely large dataset can be applied and tuned to recognise cat breeds with a smaller dataset, since many features required for image classification are similar. These architectures available on Keras already had weights available (pretrained for ImageNet classification). Thus, we preloaded the ImageNet weights before training. Optimiser parameters were further tuned here.

It was at this stage that we learnt that low learning rates were required for transfer learning in order to not destroy the delicately tuned weights. Accuracy improved further: Testing accuracy improved to about 60% in the 6-class art school category compared to about 45% before using transfer learning. In all these and future experiments, techniques discovered in previous experiments such as image augmentation, transfer learning and batch size were utilised to push our models to the highest accuracy possible. Here, the top-3 accuracy of our model was included in all subsequent experiments. A custom function had to be written in order to get this metric. The full dataset was used from this point onwards as all the images had finally downloaded. This meant that accuracy would go up even further as our dataset was even larger, since each class now had a bare minimum of about 1000 images compared to an average of about 400 in the first experiment (with the self-made neural networks).

When reading Keras documentation, we stumbled upon other models also available for image classification that were already provided. After doing some research, we found out that some of these image classification architectures had higher accuracy than ResNet50 and thus used them in experimentation. For example, the InceptionResNetV2 architecture achieved a slightly better 67% test accuracy for the 6-class art school category. Overall, accuracy was slowly creeping into the “acceptable” range, but still low.

In this stage, models such as Xception, InceptionResNetV2, NASNet-Large and EfficientNetB7 were all experimented with as they offered a higher accuracy on ImageNet. These models all turned out to be more accurate than ResNet50 when used for art classification as well, which was a pleasant surprise. However, as these better models had more parameters and layers, it was more memory and compute intensive. Compute times per epoch had shot up to 30-40min (depending on architecture used) using a top of the line P100 from Google Colab. For NASNet-Large, a maximum batch size of 8 could be used on a 8GB RAM GPU before encountering an OOM error. For EfficientNetB7, this was even more abysmal at a puny batch size of **2** even on a 11GB RAM GTX1080Ti.

Optimiser parameters were further optimised and experimented with, building on previous works from the ResNet50 experiments. It was here that the learning rate scheduler function was introduced, where the learning rate was reduced as a function of epoch number. This reduces overfitting and ensures that the model converges on loss minima better. In conclusion, with all of these techniques and new architectures experimented, the top model was EfficientNetB7 with a top-1 test accuracy of 77.97% and top-3 test accuracy of 94.31%, and the second best model was NASNet-Large with a top-1 test accuracy of 74.88% top-3 test accuracy of 94.30%. These accuracy metrics are measured against the 6-class art school category.

Still not satisfied, we searched for other methods to increase the accuracy further. We found out that in unbalanced datasets such as ours (for example, 7864 Italian images vs 856 Spanish images), class weights could be used. Thus we implemented class weights in our training model. To our surprise, top-1 test accuracy for the 6-class art school category was 89.55% and top-3

accuracy was 98.62% using the NASNet-Large architecture. This was the final model for art school used in our application as we did not believe that we would make any more significant improvements to accuracy.

Despite EfficientNetB7 showing higher accuracy in the previous experiment than NASNet-Large, NASNet-Large was used as our final architecture. This was because NASNet-Large had an acceptable batch size of 8 on 8GB of GPU RAM and was still decently accurate. In contrast, EfficientNetB7 could only have a maximum batch size of 2 on a 11GB RAM GPU which was unacceptably small, and this small batch size also compromised on model accuracy. Since Google Colab restricts the usage of their 16GB RAM Tesla P100 GPUs to avoid abuse and hogging of resources, and the fact that they have idle timeouts, we have determined that for our sanity and for the more efficient use of our time, that we would stop training on EfficientNetB7 and stick exclusively to NASNet-Large. If we could utilise Google Cloud Compute's TPUs or Tesla V100s which contain at least 16GB of GPU RAM, we would definitely have exclusively used EfficientNetB7 and not NASNet-Large in order to achieve a higher test accuracy for all our final models. NASNet-Large has around 88 million trainable parameters, and approximately 300 layers which takes approximately 40-50 minutes to train per epoch.

After the video presentation, we had discovered that we could freeze low-level layers in order to not update its weights. This is advantageous since low-level layers recognise more general features from the pretrained weights. Only higher-level features need to be tweaked and trained to adapt the model from ImageNet to our art classification. With consultation from a StackOverflow thread, we found that the ideal layers to not freeze were from the layer named "activation_166" onwards. The thread can be found here. This had the advantage of reducing compute times on a GTX1080ti GPU to about 13 minutes per epoch. However, it did not improve on the accuracy of our models, either remaining the same or instead dropped about 0.5-1% (73.75% top-1 accuracy in 7-class art type/genre compared to 74.57% top-1 accuracy without freezing layers). This is presumably because our previous experiments did not modify the weights of these low-level layers much anyway (as we had used an extremely low learning rate to account for transfer learning). Its advantages in our case is merely to reduce training times by more than 60% for a slight dip in eventual test accuracy. However, this technique could be useful in training other models to save time. After some epochs, all layers could be unfreezed and then trained again at an extremely low learning rate for fine tuning of weights, thus saving some time. This was not done in our case due to a lack of time, and we believe that this will not increase our model accuracy much, perhaps by a maximum of 0.5-1% only in the most optimistic scenario.

In conclusion, our final training models were trained from variants of the same file, nasnet.py. Only the learning rate (slight tweak), model filename, and the directories of training/validation sets were changed between each training run. We utilised NASNet-Large using Stochastic Gradient Descent optimiser with a learning rate of about 0.0001-0.00025 (depending on model trained). Momentum of 0.88 was used, Nesterov momentum not used. Learning rate scheduler was used, and early callback was used if validation accuracy did not improve to save time and computational resources. Only epochs that contained models with improved validation loss were saved, thus ensuring that only the best, non-overfitted models were saved. Image augmentation was utilised

to reduce overfitting. Transfer learning was utilised (using pre trained and tuned ImageNet weights) to improve accuracy and training times further. The largest possible batch size was used (largest batch size that is a power of 2, before OOM errors appear) which is 8 in the case of NASNet-Large. Layers were not frozen for transfer learning in our final models since the results were worse.

Evaluation is done on Xin Xiang's home desktop, utilising his GTX1070. Test accuracy results are shown in Table 3 and 4.

	Art timeline accuracy/%	Art school accuracy/%	Art type/genre accuracy /%
Top-1 test accuracy	85.27	89.55	74.57
Top-3 test accuracy	97.15	98.62	95.45

Table 3: Best Top-1 and Top-3 test accuracy for each category

Art timeline			Art school			Art type/genre		
Class	Accuracy/%		Class	Accuracy/%		Class	Accuracy/%	
	Top-1	Top-3		Top-1	Top-3		Top-1	Top-3
1401-1450	87.05	97.77	Dutch	94.91	99.27	Genre	64.13	92.66
1451-1500	91.19	98.56	Flemish	91.58	98.53	Historical	45.58	76.19
1501-1550	84.01	98.53	French	89.48	98.59	Landscape	88.64	97.03
1551-1600	85.25	95.44	German	89.04	94.18	Mythological	59.04	92.59
1601-1650	80.03	98.98	Italian	86.49	99.26	Portrait	83.48	96.93
1651-1700	82.31	97.42	Spanish	93.71	97.38	Religious	72.92	97.42
1701-1750	85.04	93.35				Still-life	92.68	96.10
1751-1800	83.33	95.00						
1801-1850	85.28	92.21						
1851-1900	92.77	97.90						

Table 4: Test accuracy breakdown by each class in each classification category

Evidently, most classes for each category had a high test accuracy and we can thus conclude that the class weights worked as intended. The features from Italian paintings for example did not overwhelm the features from Spanish paintings, as seen by the higher accuracy percentage for Spanish paintings over Italian ones even though there were almost 10x more Italian paintings than Spanish paintings. Instead, each class had an equal total weight during training.

However, one category to note is the art type/genre. We suspect that some classes either had features that were shared by other classes, or that some labelling was wrong. It could also because labelling was subjective to the curator of the website: For example, the curator could classify one painting as historical, while another might classify the same painting as a religious painting. This is supported by the fact that paintings that have unambiguous classes such as Still-life, Portrait and Landscape had relatively high accuracy of over 80%. However, other classes such as historical and genre might contain paintings that look extremely similar to paintings from Religious or Mythological classes. Thus the model was unable to differentiate these paintings well. In addition, the relatively low image count of the Historical class in the training set (630) provided few data points for the model to learn from, reducing its accuracy further. To improve the generalisations of art type/genre, more samples could be obtained, or use a dataset that is better labelled. This will probably increase accuracy of the affected classes, but improving these similar categories' accuracies to be comparable to that of other unambiguous classes such as Still-life is probably impossible since these affected classes have paintings that look extremely similar even under human inspection.

Overall, the results are commendable especially for art classification. Art is extremely subjective, and the fact that the Machine Learning model can generalise so well for most classes in most categories warms the cockles of my heart. Thus, we have confidence in claiming that our machine learning model can identify and classify art more accurately than most humans, including experts, by the painting alone without any other extra background information.

Future Work and Recommendations (ML)

In the future, freezing some layers, training for a few epochs, then unfreezing all layers and retraining at an extremely low learning rate can be explored. However, this technique will only be employed to gain the last 0.5-1% of accuracy, after optimising for all other variables such as learning rate, and when we are extremely confident that we have found the global minima and are merely trying to reach a better convergence.

With the suitable processing resources such as the Tesla V100 GPU or Google's TPUv3s, we would definitely explore other better architectures for image classification such as EfficientNetB7 that we had to abandon due to insufficient computational resources and time constraints. Currently, the top architecture in the ImageNet accuracy leaderboards is the FixEfficientNet-L2 architecture which has an extremely high top-1 accuracy of 88.5%, compared to EfficientNetB7's 85%, NASNet-Large's 82.7% and ResNet50's 77.15%. We have confidence that using FixEfficientNet-L2 will increase our art classification accuracy by a further 5-10% or more,

depending on the category and class.

We recommend Mrs Neo to appeal to the school to purchase a few fast GPUs with sufficient RAM to facilitate students' training of models and learning of Machine Learning. For example, RTX2080/RTX2080Tis can be purchased for cheap but powerful computational power, while RTX Quadro 6000s to 8000s which are more expensive can be bought in lower numbers for machine learning architectures requiring more RAM. Tesla V100 and Tesla A100 GPUs can also be purchased, as well as the DGX A100. If we had these resources available, we would spend much less time training and could obtain better, more optimised models. TPUs which are harder to obtain are suitable too, as they are even faster than these GPUs in training certain models.

If these purchases are not possible due to budget constraints, the school could register with Google, Amazon, or Microsoft as a research institution for students to apply for research credits to use. This is free since no hardware is required to be purchased and maintained. Or, GPU computational credits on Google Cloud Platform (GCP), Amazon Elastic Compute Cloud (EC2) and Microsoft Azure can be given to students in CEP that require them for machine learning. This means that students who require GPU speedup in CEP projects do not need to spend real money (like us) on cloud compute services. As previously mentioned, we could not use the provided free credits on GCP even after upgrading to a paid account, as repeated quota increases to use GPUs and TPUs were rejected time after time.

Application Deep Dive [≈ 1 - 2 pages]

A video explaining and detailing how to use the application as well as the code used in the application is found here. Please watch the video for a better understanding of how the application works. Instructions on how to operate the application can be found in the README file as well. The annotated and explained Python code for the application is found at application.py.

For our application, we used Flask to create an interactive website that users can use to identify certain characteristics of artworks or images that they uploaded to the site.

The main target audience or intended users that we had in mind were primarily art enthusiasts, researchers, artists or simply ordinary people curious about an image they found somewhere (e.g. school, museum, Internet etc). To this end, the average user might be an artist looking to see what influences might be present in his own art piece, students doing research on artworks for their school project, or researchers who want to qualitatively glean data using our machine learning algorithm.

In fact, we started on this project simply because such a service was not available or readily accessible in the status quo-- Users of Google Image Reverse Search, for instance, would perhaps be able to find similar images/artworks on the internet, but they wouldn't have access to the qualitative information as provided by our database and machine learning algorithm. They would have to slowly navigate their way through pages and pages on the internet simply to retrieve the relevant information that they require. As the famous maxim goes, "modern problems

require modern solutions.” This project was really about bridging the gap between the vast database of historical art pieces and information with modern needs of the 21st century. Users needed a fast and easily accessible way to glean specific information about whatever images/artworks they were interested in based on historical trends and artworks.

To this end, our front-end application needed to be intuitive, simple to use, and extremely accessible to the average individual who may not have as comprehensive an understanding of machine learning as programmers. This was why we chose to use a website as our primary tool for users to interface with our machine learning algorithms. Websites are ubiquitous on the Internet and are the most popular way through which people get access to information. One need only look at sites like Wikipedia, news sources like BBC or CNN, search engines like Google or Firefox, and even forums like StackOverflow or Reddit as evidence of our reliance on websites for information. Hence, we felt like users would be most comfortable and familiar with using a website given that it is both an informative and educational tool.

Team Effort [≈ 0.5 pages]

Xin Xiang worked mostly on the back-end, machine learning side of things. He was responsible for implementing and optimising the training model, adding weights and making changes as necessary after extensive research and training to maximise both efficiency and accuracy of the final machine learning model. He also carried out the actual training of the model based on the images by splitting them into training, validation and test sets and conducting training of the models using his Graphics Processing Unit and cloud servers as well, which took up a significant amount of time and computational resources. Finally, he also assisted in the integration of machine learning with the website application, making sure the Flask app was able to receive and make use of the model and successfully predict user inputs.

Trevor worked mostly on the front-end, Flask application side of things. He was responsible for creating the Flask infrastructure that would allow users to access, create and manipulate websites using Python. He also was responsible for website design, ensuring that the website user interface looked presentable, intuitive and easy to use. To this end, he oversaw the implementation of features that would enhance the user experience (e.g. informative pages, samples, simple explanations of our back-end processes) and ensured that users would be able to successfully make use of our machine learning algorithm without having esoteric and in-depth knowledge about programming or machine learning. Finally, he assisted in the integration of machine learning with the website application, ensuring that the display of information and data generated by the machine learning algorithm would be easily accessible and displayed in a readable format for users.

Conclusion [≈ 0.2 pages]

Overall, our capstone project has been a fulfilling and enlightening experience that allowed us to gain many insights into both machine learning and website design, giving us hands-on experience

on how both are able to function and how they might be integrated together for a successful and practical user experience. Our project is unique as it creates something that is currently unavailable in the market, and serves an important educational purpose as well. If we had more time, manpower or computational resources, we would explore implementing more features for users of the Flask website and try to train our model to process other categories or classes of information, and to achieve even higher accuracy, so that we can create a more fulfilling and rounded user experience. We could try to use different loss optimisers which might be able to increase the accuracy further (like RMSProp or AdaGrad) as well as different model architectures for example.