Convolutional Neural Networks for Artist and Celebrity Detection

Pouya Pourakbarian Niaz
Department of Mechanical Engineering
Robotics and Mechatronics Laboratory
KUİŞ AI Center
Koç University
pniaz20@ku.edu.tr

Abstract

This project aims to help us understand and use convolutional neural networks (CNN) in computer vision tasks such as image classification. The first part of the assignment deals with an artist classification task where a CNN will be trained to recognize the artist by which the work of art was created. The second part of the assignment delves into transfer learning (specifically, model adaptation), which is frequently used in computer vision for adapting a pre-trained model to new data with a limited size to improve accuracy over a model trained from scratch on the limited target data.

1 Convolutional Neural Networks

In this section, a dataset of 451 works of art made by 11 different artists is available. We will train a supervised multi-class classification task in which, given an image (a painting), the model will classify which artist created the painting.

1.1 Convolutional Filter Receptive Field

Considering that a filter is slid through an image, every node in the output image is the application of one single position of the filter to a specific position of the image. This means that regardless of the size of the image if the filter applied to the image is $f \times f$, then every node in the output image will have a receptive field for this image of $f \times f$.

Considering the first convolutional layer of a CNN network, using a filter size of $f_1 \times f_1$, the output image Z_1 will have a receptive field of $f_1 \times f_1$. However, since every pixel of Z_1 depends on more than one input pixel, this will propagate through the following convolutional layers. For instance, every node of the second convolutional layer (every pixel of the second layer Z_2) sees $f_2 \times f_2$ pixels of Z_1, with $f_1 \times f_1$ being the filter size in the second convolutional layer, and every pixel in Z_1 sees $f_1 \times f_1$ pixels of the input. That being said, there is a lot of overlap between the pixels that are seen by the different nodes of the first convolutional layer, meaning the receptive fields cannot be summed up together.

Receptive field of every node in Z_1 $\quad R(Z_1) = f_1 \times f_1$

Every two adjacent nodes in Z_1 overlap $f_1 - 1$ pixels in width and height of the input image, meaning each node (pixel) in Z_1 has only one pixel in the input image that the adjacent pixel in Z_1 cannot see (assuming stride of 1 and ignoring padding). Bearing this in mind, once a filter of width f is slid across a dimension n times, the corresponding n nodes in Z_1 see the n pixels in the input image, as

Preprint. Under review.

well as the following $f-1$ pixels belonging the final position of the filter. This means that every window of n consecutive pixels in Z_1 collectively has a receptive field of $n+f_1-1$.

Continuing with the 1D image analogies explained above, let us move on to Z_2. Every node in this layer can see f_2 pixels in Z_1 because that is the filter size of Z_2, and these f_2 pixels collectively can see f_2+f_1-1 pixels of the input image.

Receptive field of every node in Z_2 $\quad R(Z_2) = (f_1+f_2-1) \times (f_1+f_2-1)$

As we can see, by adding convolutional layers on top of each other, the receptive field of the nodes in deeper layers keeps increasing, meaning pixels can hold latent information from larger areas of the input image. This is one of the reasons why deeper CNNs have usually achieved better image classification results than shallow CNNs.

1.2 Running the Pytorch ConvNet

By studying the starter code provided, the following can be concluded:

- The provided code consists of two convolutional layers, each followed by a pooling layer. A flattening layer follows the final pooling layer and a fully connected layer carries out the classification.
- The well-known ReLU activation is used on the hidden layers.
- Since it is a multi-class classification problem, the cross-entropy loss is used. This is the equivalent of a SoftMax activation followed by a negative log-likelihood loss.
- The Adam optimizer is used as a gradient descent method for minimizing the loss function.

After running the provided code with the provided model, we get 62.77% training accuracy and 49.45% validation accuracy. This tells us that we are suffering from both high bias and high variance; that is, the model is not sophisticated enough to capture the information from the training set properly and is not sufficiently generalizable to fit the validation set. The remedy to this situation is typically to solve high bias first, i.e., make the model sophisticated enough to perform well on the training set. If the high variance problem resists, we typically perform regularization to increase model performance on the validation set. In a nutshell, first, we solve the high-bias problem (underfitting) and then the high-variance problem (overfitting).

1.3 Adding Pooling Layers

By turning the max pooling layer flag on in the provided code, without any further modifications, using max-pooling layers with a kernel size of 2 and stride of 2, the training accuracy increased to 67.22% and validation accuracy improved to 51.65%. This improvement can perhaps be compounded further by tuning the kernel size and stride of the max-pooling layer.

After removing the stride parameter of the max-pooling layers, training accuracy improved to 71.94%, and the validation accuracy improved to 54.95%. This is a rather large improvement over the previous case. This is typical since having a stride of greater than 1 is not typical for max=pooling layers, especially when images are already relatively small, and having a stride for the max-pooling layers could leave out important information or features from the image.

1.4 Regularization

Regularization is a general term referring to methods that are designed to reduce variance/overfitting. There are multiple regularization techniques used frequently, the results of some of which are reported below.

1.4.1 Dropout

Using Dropout, in simple terms, the network learns not to rely too much on any specific feature or path through the network. It sets the weights such that multiple latent features and information collectively help learn the intended function without giving large weights to any node.

Dropout is frequently deployed immediately after the activation function in ANNs, or after the pooling layer in CNNs. By adding a Dropout layer to the network that has max-pooling layers in it, and training it for 50 epochs with the default training procedure given in the code, the results are reported as a function of the Dropout parameter in Table 1.

Table 1: Model performance after using Dropout. Max-pooling layers are used with a kernel size of 2 and a stride of 1 after each convolutional layer. The convolutional and fully connected layers are kept as given by the assignment.

Dropout	Training Accuracy	Validation Accuracy
0.0	0.7194	0.5495
0.1	0.6666	0.5385
0.2	0.5889	0.4835
0.3	0.5778	0.4066
0.4	0.5639	0.4505
0.5	0.5111	0.4286

One observation noted when trying Dropout is that there would often be relatively large deviations in performance when training multiple times consecutively due to the random initialization of weights. This variation, however, typically falls into a small range. In this particular problem, with the given architecture and training procedure, only with max-pooling enabled and Dropout added after all pooling layers and the fully connected layer, a large variation of 10-20% in both the training and validation accuracies was noticed. This may perhaps have something to do with the fact that we are not making typical batch gradient descent; we are making mini-batch gradient descent, and therefore the learning curve of the model through the iterations and epochs would be much more jagged and much noisier than the case without Dropout because some neurons from the previous layer are randomly dropped when using Dropout.

One of the reasons Dropout did not improve accuracy drastically in this particular scenario, according to Table 1 is that the current architecture and training scenario is not overfitting, to begin with. It is, in fact, underfitting. Dropout only helps when there is no underfitting (training error is small), but there is overfitting (validation error is high). To see the effects of Dropout much better, we would need to build a much more complex model that can perform acceptably on the training set and then apply Dropout on that model to improve the validation performance.

1.4.2 Different optimizers

Different optimizers can also affect the performance of the model. We tried Adam, SGD and RMSProp, with their performances reported on Table 2. As it can be seen on this table, Adam and RMSProp generally have the best performances. SGD is basically an equivalent of Adam with a minibatch size of 1, which makes the training process very noisy, and can sometimes have detrimental effects on the final training and validation accuracies.

Table 2: Model performance with max-pooling and with the given architecture and training procedure. Learning rates are the same for all optimizers.

Optimizer	Training Accuracy	Validation Accuracy
Adam	0.7083	0.5724
SGD	0.2305	0.1978
RMSProp	0.6639	0.5055

1.4.3 Early stopping

Early stopping can occur when validation accuracy does not improve for a specific number of epochs or iterations. This number is a potential hyperparameter known typically as “patience".

When using the Adam optimizer and early stopping for 50 epochs, it was observed that a patience value of 5 or 10 is typically sufficient for this task. Model performance with all default settings, max-pooling, and 50 epochs, while also enabling early stopping, is reported on Table 3. According to this table, a relatively small patience of 2 to 5 is usually sufficient for this model and this dataset.

Table 3: Model performance while enabling early stopping. Default settings are used, with max-pooling, the given architecture, and the training procedure.

Patience	Stopping Epoch	Training Accuracy	Validation Accuracy
2	11	0.5083	0.5275
5	20	0.5222	0.4615
10	39	0.5916	0.3846
20	49	0.6027	0.4505

1.4.4 Learning Rate Scheduling

Here we will investigate the effects of the learning rate schedule. By reducing the learning rate as we get close to the minimum in the loss-parameter space, there is less likelihood of overshooting the minimum or diverging.

For instance, by trying exponential learning-rate decay scheduling, the learning rate is attenuated by a factor γ at every epoch. Using the `StepLR` scheduling, this attenuation happens once every n epochs. Multiple learning rate scheduling techniques are available to use in PyTorch.

After using `StepLR` and `ExponentialLR` learning-rate schedulers, no significant improvement in the model's performance was noted. That being said, when using the learning rate schedule, since the learning rate kept decreasing, the progression rate also kept decreasing; therefore, the early stopping method would stop the training slightly later because the plateau was reached later than normal, with smaller steps having been taken due to progressively smaller learning rates.

1.5 Experimenting with Model Architecture

In this section, we will be free to change the model's architecture, including the number of convolutional, pooling, and dense layers, layer width, convolution/pooling kernel sizes, convolutional filter counts, and so on.

While experimenting with model architecture, training was done on 10 epochs, so different architectures could only be compared against one another. Once the optimum architecture and hyperparameters are chosen, the final model can be trained with a larger number of epochs.

With the default architecture given in the assignment and the default training procedure, major hyperparameters of the architecture were changed. They are reported in this section.

- **Convolution kernel size:** kernel sizes of 3, 5, and 7 were tried while keeping the max-pooling kernel size constant at 2. Stride was kept at 1 for both convolution and max-pooling layers. *The best kernel size was selected as 3*. By increasing the kernel size, the image keeps getting smaller along the depth of the network, and a large kernel size does not allow the model to extract meaningful information from smaller regions of the image.
- **Convolution stride:** Stride is the equivalent of a down-sampling rate for convolution or pooling layers. The idea is that if the image is too large, there may be no need to slide the filter across all the pixels of a dimension. Using the filter on one out of every s with s being the stride size, consecutive pixels might be enough to extract vital information while keeping the architecture light. Greater strides than 1 are more useful in large high-resolution images. Strides of 1, 2, 3, and 4 were tried for the convolution layers with the default architecture. *A convolution stride of 1 was selected as optimum for this scenario*, which is logical since the input image already has very small dimensions. When increasing the convolution stride, training and validation accuracies consistently decreased when using the given architecture.
- **Convolution depth:** Depth in convolution refers to the number of convolution blocks (Convolution, [optionally] Batch-Normalization, Activation, Pooling, and [optionally] Dropout layer) in the encoder part of the network, i.e., the part before the fully connected layers. Depths 1,2,3 and 4 were tried to see the effects. No Dropout or BatchNormalization was used in these tests. *A depth of 3 convolution blocks seemed to be optimal for this architecture and this dataset, with a training accuracy of about 0.61 to 0.73 and a validation accuracy of about 0.57 to 0.63.* Increasing the depth of a CNN does increase its capacity to extract complex features and latent information, which often improves the training set accuracy,

though not necessarily validation accuracy, but depending on size and type of dataset, class distribution in the dataset, image size, and many other factors, beyond a certain point, increasing depth may not be beneficiary even for the training set performance.

- **Padding:** Using the so-called `valid` padding for convolutional layers, after each convolution, the image becomes smaller but thicker (its number of channels increases as it passes through convolutional layers because convolutional layers in CNNs often have an increasing number of filters along the depth). Another padding option is the so-called `same` padding, where the image is padded before applying the convolution, so the output image of the convolution has the same size as its input image. *Choosing `same` padding did not significantly affect the model performance when tried multiple times consecutively to account for the random initialization of weights. The overall performance was generally slightly better with `valid` padding*.
- **Dense Depth**: The depth of the Dense (fully connected) section that comes after the convolutional section (a.k.a. the decoder) can also sometimes affect the model performance. Sometimes, the information that can be extracted from the image is not very meaningful and must be processed through multiple Dense layers to be successfully mapped to the target outputs. After trying depths of 1,2,3, and 4 hidden layers after the convolutional section, *we achieved the best results with only one Dense layer (not including the final output layer)*. Performances are reported on Table 4.

Table 4: Model performance against the depth of the Dense section of the CNN. Three convolutional blocks with doubling filter sizes and max-pooling layers are used, and the width of the Dense layers is halved as we approach the final output layer. There is no regularization whatsoever. Training is done with 10 epochs to save time.

Dense Depth	Training Accuracy	Validation Accuracy
1	0.61	0.58
2	0.57	0.51
3	0.52	0.49
4	0.50	0.50

1.6 Hyperparameter Tuning and Model Optimization

Another way of optimizing a deep learning model, apart from architecture, is hyperparameter tuning. Hyperparameters can include parameters of the optimizer, mini-batch size, number of epochs, regularization parameters, parameters of specific layers like Batch-Normalization layers, and so forth.

We repeat the hyperparameter optimization process done in the previous sections after choosing the optimal architecture for the CNN.

- **Mini-Batch Size:** Table 5 reports the effect of mini-batch size on training. The best architectures from the previous section were used for training the model for 10 epochs. A default learning rate of 0.0001 was used, without any regularization. A mini-batch size of 32 appears to be optimal for this scenario.
- **Learning Rate:** Since Adam was seen to be the best optimization algorithm in the previous sections, it was used with its default parameters for training the model. The model was trained with the previous best hyperparameters without any learning rate schedule, regularization, or normalization layers. Its results are reported in Table 6. According to this table, a learning rate of 0.0001 seems to be optimal for this scenario.
- **Local Response Normalization:** Using local response normalization, the excited neurons conveying important information can suppress the other ones, making a better impact on the model's output. A local response normalization layer with a size of 4 neighboring channels happened to bear the best results in this study.

After coming up with the best model according to our hyperparameters, we train the model for 50 epochs, five consecutive times. **We get a mean training accuracy of 0.6354 and a mean validation accuracy of 0.5879**.

Table 5: Model performance against mini-batch size. The best architecture is chosen, trained with 10 epochs.

Dense Depth	Training Accuracy	Validation Accuracy
8	0.5233	0.6000
16	0.5933	0.5767
32	0.6133	0.6233
64	0.5000	0.5833
128	0.4900	0.5400

Table 6: Model performance against learning rate. The best architecture is chosen and trained with 10 epochs.

LR	Validation Accuracy	Training Accuracy
10^{-2}	0.1318	0.1416
10^{-3}	0.0659	0.0944
10^{-4}	0.6813	0.6667
10^{-5}	0.4550	0.4500

1.7 Data augmentation

Using transforms available in PyTorch, we can dynamically and randomly augment the images before feeding them to the network. This way, the effective size and variety of the training data will be multiplied. Alternatively, we can test a model trained on the original image on a test set of augmented images to see how generalizable the trained model can be. A brief report of the model performance under different data augmentation techniques tried individually can be found in Table 7. These augmentations were done on the validation set alone, meaning the purpose is to estimate the generalization error of the trained model.

Table 7: Model performance against data augmentation technique deployed to the validation data alone. Training is done with 10 epochs on the optimized architecture and training procedure.

Transform	Amount	Validation Accuracy	Training Accuracy
Color Jitter	0.5	0.5714	0.5777
Affine	40 deg	0.4995	0.5833
Horizontal Flip	0.5	0.5385	0.5792
Vertical Flip	0.5	0.4236	0.5056
Invert	0.5	0.3407	0.5361
Adjust Sharpness	0.5	0.5000	0.5653

According to Table 7, the trained model is slightly less vulnerable to slight color distortions and sharpness but more vulnerable to inversion and vertical flips. This means that the model pays a lot of attention to the geometric positions of the features in the pictures and the general color mapping throughout the image.

2 Transfer Learning with Deep Networks

Here we will train a neural network capable of detecting popular figures and celebrities. The FaceScrub dataset will be used. Input will be a 28×28 RGB image, and the output of the model will be one of many classes, each corresponding to one person. There are a total of 40 celebrities, male and female, provided in the dataset. The downloaded images will be cropped using the boundary box coordinates provided by the dataset, then resized to the appropriate square size, before being fed to the model.

2.1 Multi-layer Perceptron

In this section, a simple one-hidden-layer multi-layer perceptron will be used for performing the classification. The input layer will be the flattened image containing $28 \times 28 \times 3 = 2352$ features, the hidden layer will contain 300 neurons, and the output layer will contain 40 neurons, because there are 40 celebrities in the dataset, thereby meaning we need to perform 40-class classification.

Using default settings, optimizers and other hyperparameters, we achieve using this architecture a training accuracy of about 90-95% and a testing accuracy of about 65-70%, Fig. 1. The high training performance compared to the weak validation set or testset performance indicates overfitting, a problem that can be alleviated using regularization technmiques such as Dropout, and L2 regularization.

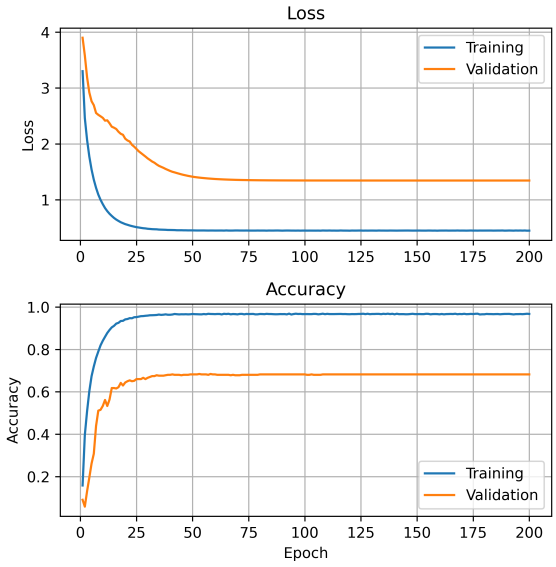

Figure 1: Initial learning curve, without any regularization.

After some hyperparameter tuning, we find that a Dropout rate of 0.2-0.3 after the ReLU activation, a BatchNormalization layer before the ReLU layer, and an L2 regularization parameter of 0.01-0.03 appears to have the best results, Fig. 2. We will use these hyperparameters for the following sections as well, when using AlexNet. With this setting, we can get 0.9-0.93 training accuracies and 0.69-0.71 validation/testing accuracies. In this particular setting, ReLU is used as activation function for the hidden layer for its popularity, SoftMax is used as the activation function for the output layer (because we are doing multi-class classification), Xavier-Uniform initialization is maintained (which is default in TensorFlow but not PyTorch), all layers have biases eanbled, and the input image is normalized and converted to float32 to be between 0 and 1. A categorical crossentropy loss function was used, and the Adam optimizer with default parameters (learning rate of 0.001) was used, again due to its popularity. The hyperparameters and training pocedures explained above were also maintained throughout the rest of this report.

2.2 AlexNet as Feature Extractor

In this part, we will use transfer learning to use a pre-trained AlexNet, frozen (untrainable), and only train its final classification layer(s). In this type of transfer learning (model adaptation), a pre-trained model is basically used as a feature extractor. The idea is that the pre-trained model has already been trained on a very large dataset using premium computational resources, and is therefore capable of extracting very nice meaningful features from a plethora of different kinds of images. It can therefore be effectively used as a feature extractor. The final one or few classification layers (the fully

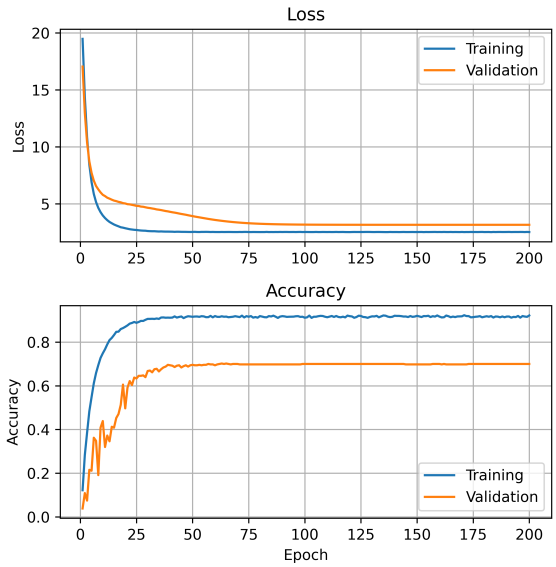

Figure 2: Learning curve of the multi-layer perceptron model regularized with Dropout and L2 regularization.

connected layers that use the extracted features and process them to reach the proper prediction) can be reinitialized and retrained on the limited data that is available for the current task at hand.

Using the pre-trained feature extraction section of AlexNet, a model was built using the output of the AlexNet features as inputs to the dully-connected network. After a flattening layer, there is 300-unit Dense hidden layer, followed by a BatchNormalization layer, a ReLU activation layer and a Dropout after that. Then, there is a 40-unit output layer with SoftMax activation. In other words, the classification section of the model is identical to what was used in the previous section, only instead of flattened raw images, flattened extracted feature images are used as input to the model. Other hyperparameters and the training procedure were identical to the previous multi-layer perceptron case. The learning curve of this training procedure can be seen on Fig. 3. A training accuracy of 0.9842 and a testset accuracy of 0.7420 were achieved after 200 epochs in this setting.

2.3 Weights Visualization

By visualizing the individual weights or activations for different neurons in the network for a specific input image, we can find out how that neuron sees the image, and which features the neuron is most sensitive to, Figs. 4 and 5. Since the neural network is very shallow, and the input image is of very low resolution, it is hard to make out what features are extracted from the input faces. However, we can see by looking at the outputs that not only the faces are getting segmented out of the images, but also features such as eyes, nose and mouth are being highlighted with similar colors. This indicates that the color and position and form of these features are what determines the output prediction of the network. For deeper networks, it would be easier to visualize the weights, because every layer would deal with a certain level of abstraction and feature extraction. With a one-hidden-layer network, however, the layer is being forced to extract all the features and information at the same time.

There is no noticeable difference between a 300-neuron hidden layer and an 800-neuron hidden layer in terms of the interpretability and quality of the visualized weights or outputs. This is because as mentioned before, it is the depth of a network that allows it to tackle different levels of information retrieval and abstraction, not width. A wider network would extract more number of diverse features from the input layer, but it would not be able to connect the features together and extract deeper more meaningful features from the features that are extracted. In our approach we used the first neuron and the middle neuron rather than the end neuron because there is always some symmetry across the

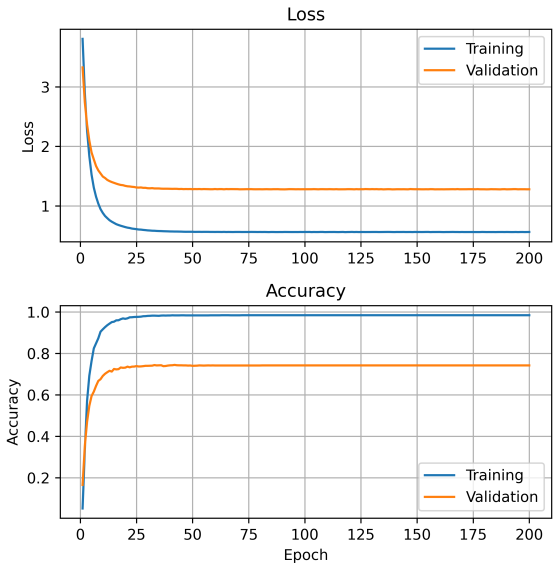

Figure 3: Learning curve of the refit model, transferred from a pre-trained AlexNet model, with a single classification layer of 300 units after the feature extraction section of AlexNet.

widths of hidden layers even after using batch-normalization and Dropout, therefore middle neurons are better to be comapred with the first and last neurons.

2.4 Tuning AlexNet Model

In this section, instead of extracting the feature extraction section of the AlexNet network, and using it as a fixed preprocessor for the multi-layer perceptron model we had built priorly, we will extract the entire pre-trained AlexNet model, with all of its sections. We will only replace the final output layer to have the appropriate shape needed for our case. We will then freeze the entire AlexNet network and only train the final output layer (a.k.a. the classifier layer) to see how it performs. Fig. 6 shows the learning curve for this model. In this model, the architecture of the entire model is identical with the original AlexNet, except the final output layer, which is retrained. We received a training accuracy of 0.8083 and testset/validation accuracy of 0.6987 with the same training procedure as the previous one. Even though the testset accuracies are not too far from the previous attempt, training accuracy is much lower, meaning there is less variance but much more bias than the previous case. In fact, the performance of this particular model is closer to the simple MLP we trained in previous sections, than the one in which we used AlexNet as a feature extractor.

According to the results we have achieved so far, the best model+procedure we have seen is the one in which AlexNet is used as a feature extractor, and a simple one-hidden-layer MLP processes the AlexNet features (without the adaptive average pooling layer after the feature extractor in AlexNet), and sends it to the output layer with SoftMax activation.

3 Conclusion

In this assignment, we have tried to take a brief look at convolutional neural networks and how they can be used for effectively classifying images, as well as many other computer vision and image detection tasks. We have only looked at celebrity detection and artwork classification. We have tried a few CNN typical CNN architectures for the artwork classification, to see how we can effectively classify the artworks. We have also used a pretrained AlexNet architecture to perform simple transfer learning in order to leverage the meaningful information it has acquired from its large dataset, so it can be adapted for our case. We have found that a simple one-hidden-layer MLP attached to the

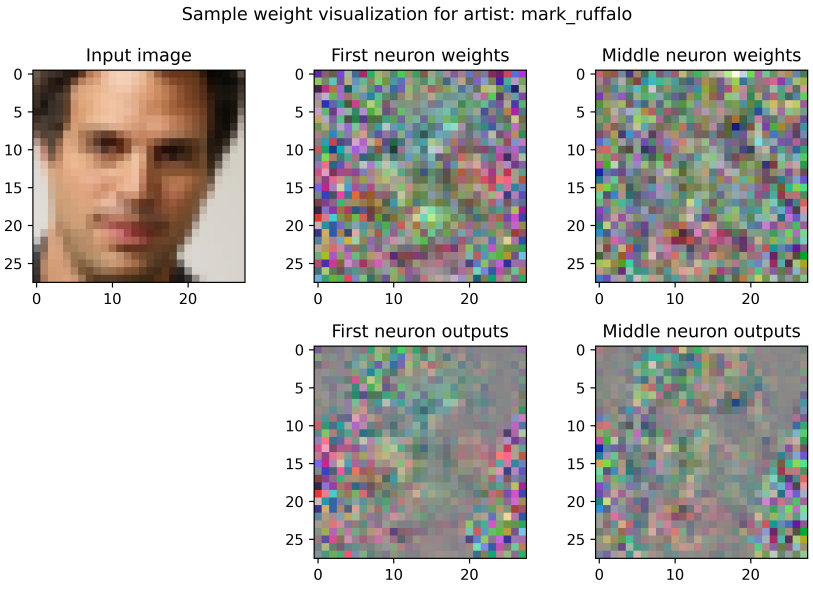

Figure 4: Visualization of weights and outputs for the first and middle neurons in the hidden layer of the multi-layer perceptron for a sample example of Mark Ruffalo.

feature extractor CNN of the pre-trained AlexNet has the best performance in celebrity detection. In addition, we have found that using Dropout, L2 regularization and BatchNormalization layers can improve model performance by helping reduce both bias and variance.

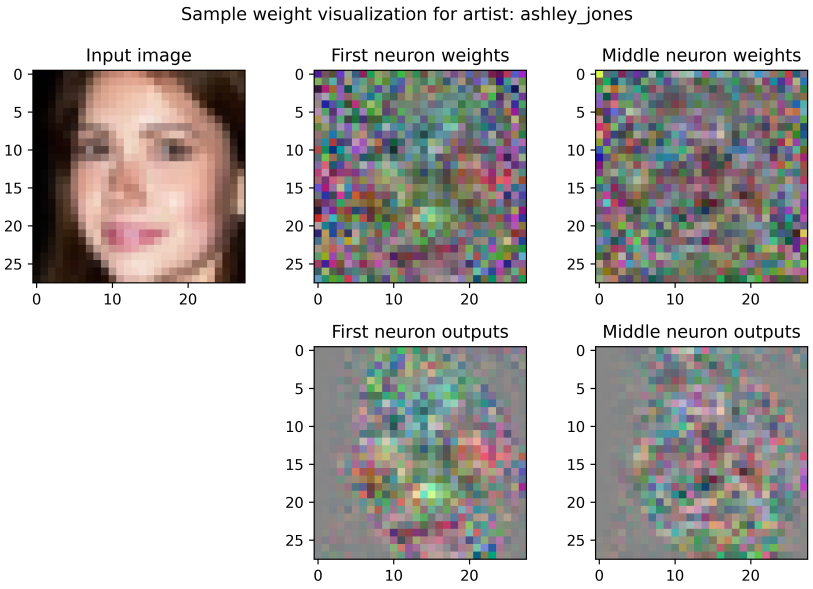

Figure 5: Visualization of weights and outputs for the first and middle neurons in the hidden layer of the multi-layer perceptron for a sample example of Ashley Jones.

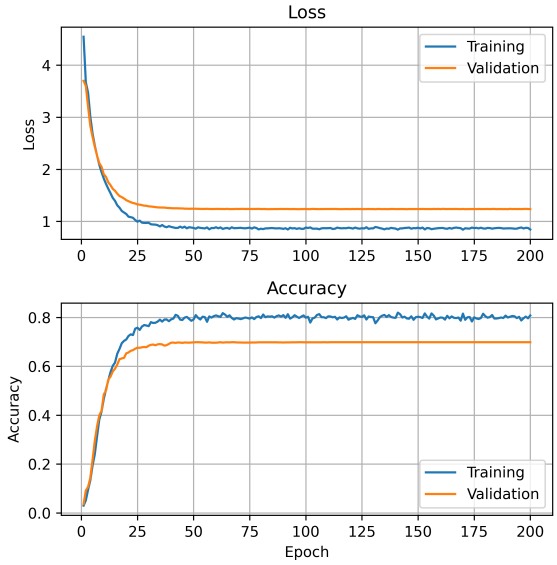

Figure 6: Learning curve of the refit model, transferred from a pre-trained AlexNet model, with the classification part identical to the AlexNet, where only the final output layer has a different shape.