Classifying Sculptures by Art Style Using Convolutional Neural Networks

by Harry Shomer

May 17, 2019

Course Instructor: Professor Justin Steinberg

Advisor: Professor Miriam Deutch

Abstract: My research attempts to see how well we can classify sculptures into their corresponding art styles using a convolutional neural network. Twelve different art styles are used with the data, being an image of each sculpture and their art style, drawn from various online art databases. The model was built by fine-tuning the pre-trained network Xception for our task. The final evaluation showed that the model determined the correct style for 54% of sculptures in the test set.

Contents:

Acknowledgements

I would like to thank Kyle for taking the time to listen to me.

Introduction

During my junior year of college I took an art history course. We did a brief overview of the history of art and discussed the distinct features and motivations for a number of different periods and types of art. During class, the professor would often display a piece of art on the board and ask us to think about what style or period it may belong to and why. This forced us to closely examine the art object and think about what distinguishes one period from another. As a Computer Science major, this class exercise led me to think about how well artificial intelligence can be used to determine what style or period an art object belonged to. Can a computer recognize the small nuances that distinguish various art styles?

This motivating question led me to do research on the topic to see what others have done. Upon sifting through the previous work I noticed a slight gap. A vast majority of the work focused on analyzing just paintings. These works attempted to see how well we could classify paintings by style, artist, or genre. But there was very little work done concerning sculptures and of that work none used state-of-the-art visual recognition techniques like convolutional neural networks (CNNs). This realization led me to formulate the following question. How accurately can we classify sculptures by art style, given an image of the sculpture, using a convolutional neural network?

To start off I will give a brief history of artificial intelligence by discussing the different ways computer scientists define intelligence for a computer. To do this I will use the four different approaches mentioned in *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig (2010). This will serve as a way for the reader to have a sense of what we mean when we say we are “training” a model to do something.

I will then discuss the basic properties of artificial neural networks. CNNs are a type of feed-forward artificial neural network that I’ll be using to develop the proposed model. This discussion includes noting what a general artificial neural network consists of and how it attempts to learn a specific pattern. I then move onto CNNs and discuss the different layers that constitute a CNN, the reasoning behind each, and different strategies when using them. I then discuss related work done in art visual recognition. This includes an overview of work done classifying paintings by artist, style, or genre with an emphasis on those studies that use CNNs. I also mention similar work done with sculptures of which much less literature has been written.

The next chapter focuses on the methodology used for building my proposed model. This includes a discussion of where the data was obtained from and how it was processed. I then detail the type of CNN architecture used and how it was trained for the specific classification task. The following chapter focuses on the evaluation of the model created in this previous chapter. To do so we compute the accuracy and loss of the model on the test set. A confusion matrix is also produced to see where the strengths and weaknesses of the model lay.

Lastly, I will conclude the paper by providing a brief overview of the work and conclusions produced by the research. I will then detail the possible applications of the research. I will end off with a discussion on potential improvements to my work and future research avenues that can be explored.

Chapter 1

In this chapter, I am introducing the topic of Artificial Intelligence (AI) by focusing on Stuart Russell and Peter Norvig's (2010) analysis of the different methods taken to model intelligence in machines. I will do this by examining the text in which Russell and Norvig discuss this. They list four different views on AI intelligence, which I will summarize and review. This includes modeling human behavior, how humans think, to think rationally, and lastly to act rationally. This analysis will also be used to concurrently explore the history and influences on AI. After examining each of the four methods we will discuss where our proposed research question falls. My contention is that a convolutional neural network is an example of a combination of modeling how humans think and acting rationally. This understanding helps us better comprehend how the model created in this research is able to learn to classify sculptures.

AI is the study of how we can create intelligent beings or "agents" (Russell et al., 2010, 1). AI attempts to create machines that are able to think and make decisions to problems on their own. The field of AI is considered to have "officially" started in the summer of 1956 with a workshop at Dartmouth College put together by John McCarthy (Russell et al., 2010, 17). Since then it has seen tremendous growth and has received contributions from other fields such as: computer science, philosophy, neuroscience, and economics. The interdisciplinary nature of AI has led to multiple approaches being taken by researchers in attempt to solve the problem of teaching something to think intelligently. The adoption of a particular method hinges on how one understands intelligence. How does one define intelligence for a computer? What do we expect an intelligent computer to act like? In their book, *Artificial Intelligence: A Modern Approach* (2010), Stuart Russell and Peter Norvig describe four ways in which this issue of intelligence has been tackled.

The first approach they mention is "acting humanly" which is inspired by work done by Alan Turing (Russell et al., 2010, 2). In 1950 Alan Turing came up with a test to determine if a computer is intelligent. This test is now known as the Turing Test. The test is designed as followed: A human is placed in one room and has a means of communicating with a being in a separate room (so they are unable to see each other). The being in the other room may be a computer or another human. The human has no physical way of determining if the being in the other room is a computer or not. It's the human's job to interrogate the being in the other room to determine if they are in fact human. The human questioner passes questions to the computer in the other room and evaluates the answers given back. If the interrogator is unable to tell if a computer or a human provided the answers then we consider the computer to be intelligent (Turing, 1950, 435). The test is designed to see if we can design a machine that can fool humans into thinking that they are human. Turing's basis of intelligence is therefore acting like a human. The Turing test is influential and important as it's one of the first tests designed to test the intelligence of a computer. The test also requires that a computer be able to: better use human language, be able to store information it receives, be able to reason and draw conclusions from the information it has, and learn how generalize and act in new situations (Russell et al., 2010, 3). These requirements have all become major research topics in AI. With that said, while the Turing test is considered influential it was never generally regarded as a goal or benchmark used in evaluating intelligent agents. As Russell and Norvig mention, "it is more important to study the underlying principles of intelligence than to duplicate an exemplar"(Russell et al., 2010, 3). Most researchers feel that it is more important to focus on the general problem of intelligence and not to attempt to copy beings that are intelligent.

The second way of approaching the idea of AI is to model computers based on "how humans think" (Russell et al., 2010, 3). An early example is the General Problem Solver (GPS) created by Newell and Simon. GPS was a computer program that was supposed to be able to solve any kind of problem you gave it (only a limited number could be solved in a manageable amount of time) by mimicking the thought processes humans use when solving a problem. By "problem" we mean any question that can be stated formally in a series of clauses. Once the problem is properly stated we can search all the possible answers (know as the "state space") until we find the answer we are looking for. Their main concern was "with comparing the trace of its reasoning steps to traces of human subjects solving the same problems" (Russell et al., 2010, 3). Newell and Simon's goal was to get their program to solve those problems like humans do. In order to do so we need to have a sufficient understanding of how humans think. An attempt to do this is the field of neuroscience, which studies how our brain works. In 1943, inspired by neuroscience, McCulloch and Pitts created artificial neurons. Artificial neurons attempt to computationally model how neurons work in our brain. Biological neurons make up the neural networks in our brain, which is how information is processed and transferred in our brain. This work laid the foundation for artificial neural networks, which we will cover more in depth later, that are a very successful tool used in many areas of AI (Russell et al., 2010, 16).

The third approach to designing AI, as noted by Russell and Norvig, is to create machines that think rationally. We focus more on how machines determine a course of action rather than on the actual outcome. The significance here is placed on making “correct inferences” in order to reach some conclusion (Russell et al., 2010, 4). The idea of making only rational decisions can be traced back to Aristotle. Aristotle developed a set of syllogisms, which are propositions that can derive a conclusion “for proper reasoning” that “in principle allowed one to generate conclusions mechanically” (Russell et al., 2010, 5). Using these syllogisms should then lead to a correct result “when given correct premises” (Russell et al., 2010, 4). Others such as Leonardo Da Vinci, Blaise Pascal, and Gottfried Leibniz later utilized this idea in the creation of early calculators (Russell et al., 2010, 6). Given a set of operands and operators (the initial premises) it can correctly give the solution mechanically. Logicians also later used Aristotles’ ideas in the 19th Century to create “notation for statements about all kinds of objects in the world and the relations among them” (Russell et al., 2010, 4). Once we have this notation we can set up our initial conditions and deduce some conclusion. We can now adapt this into a computer program and let the computer determine the final result given the premises and procedures. We set up our initial facts, we create the mechanism for logical relations, and we see what inferences the program concludes. There are a couple of drawbacks to this approach most notably that it can be a computationally expensive procedure. Any given problem can be made up of thousands of initial "facts" so deriving an answer may be infeasible (Russell et al., 2010, 4). With that said, an advantage is we know exactly how we arrived at any conclusion.

Lastly, the fourth approach outlined by Norvig and Russell to determine what qualifies as intelligence for a computer is if it acts rationally (Russell et al., 2010, 4). This is not to be confused with thinking rationally. In this approach we don't care how it gets to its conclusion as long as it's the rational decision given all the information. It prioritizes the end result over the path to it, as the main goal is maximize the expected value of the outcome given all the available information. With the "thinking rationally" approach we are trying to focus on using correct reasoning to reach some conclusion. This is "sometimes part of" acting rationally but not always. There are times when it isn't possible to infer any rational answer or a non-rational approach may work better (Russell et al., 2010, 4). This type of intelligence is regarded as the "rational agent" model and can also be found in economic theory. In economics, decision theory describes trying to maximize one's decision based on the probability and utility of the outcome. The goal is to choose the decision that maximized one's expected utility (Russell et al., 2010, 9). In order to judge what actions are considered rational we must define some function to determine that. This function is called the performance measure. We therefore want our machine to choose the action "that is expected to maximize its performance measure" (Russell et al., 2010, 37).

In this paper our goal is to teach a model to discern the style of a sculpture from an image. For that reason we can think of it as a combination of Russell and

Norvig's second and fourth principles. The second is to model artificial intelligence based on how humans think. For our model we will utilize a convolutional neural network, which while not analogous to how humans process visual information, is inspired by it. The fourth principle is to have computers act rationally given the data presented to them. The main goal of an algorithm like a neural network is to maximize how well it does. Interpretability and how it achieves its success isn't important. In this context the fourth principle describes the algorithm better than the second. While it is biologically inspired, the comparisons are considered superficial.

Chapter 2

In this chapter, we will discuss the topic of feed-forward artificial neural networks (ANNs). This will be done in two sections. In the first we will talk about the general theory behind ANNs. This includes the biological inspiration, their structure, how they work, and how they are able to “learn”. In the second section, we will discuss a specific type of feed-forward ANN known as a convolutional neural network (CNN) that is the type of framework we will be using later in our methodology. These two sections will allow us to understand how CNNs work before we use them to create our model.

Feed-Forward Artificial Neural Networks (ANNs)

In this section, I will introduce the topic of feed-forward artificial neural networks and detail how they work (for brevity’s sake I will not use the term “feed-forward” when referencing them from here on out). Artificial neural networks are based on the biological neural networks found in our brain and are used as a framework to model phenomena. As opposed to other types of networks, feed-forward networks only move information in one direction without any cycles. In this section I will first detail the basic operations of a biological neuron. I will then talk about perceptrons, which attempts artificially model neurons. Lastly, I will discuss the idea behind how feed-forward artificial neural networks work and how they are trained for a specific task.

Biological Inspiration

Artificial neural networks attempt to computationally model biological neural networks found in the brain. A biological neural network is a collection of interconnected neurons. Each neuron receives an input, processes it in some way, and then transfers it onwards to a new collection of neurons (Rojas, 2013, 10). In the brain, the signals are electrical and are a way of sending or receiving information from other areas. Neurons are made up of a few different components (see Figure 2.1). The dendrites receive incoming signals from the neurons in the previous level of the neural network (Rojas, 2013, 10). The axons are the part of the neuron that output signals to other neurons (Rojas, 2013, 11). The gap between layers of neurons is called the synapse. That is the area where the dendrites of a neuron meet the axons of the previous layer. So a given neuron receives a number of inputs from other neurons in the dendrites. These inputs are received from those neuron's axons, which meet the dendrites. After a neuron receives all its inputs it adds them all together into one value.

We now know how a neuron receives and sends out the various signals but we don't know how it chooses to either send or not send out signals. What happens is the neurons "compares" the summed value to a threshold value. The threshold value is an arbitrary number that decides whether or not the input to a neuron will cause a response. If the summed value of the inputs is greater than the threshold then the neuron will output some constant (which is known as an action potential) otherwise nothing happens (Rojas, 2013, 18). If the inputs to a certain neuron reach a certain value, then that neuron will "fire" some signal out to the next layers of neurons (Rojas, 2013, 20). This is known as the "all-or-nothing" principle as it either fires some value or none at all (Rojas, 2013, 25). It's also found that synapses matter more than other. A neuron receives inputs from many other neurons but it's seen that some of those inputs have a larger effect on whether that neuron will fire (Rojas, 2013, 20).

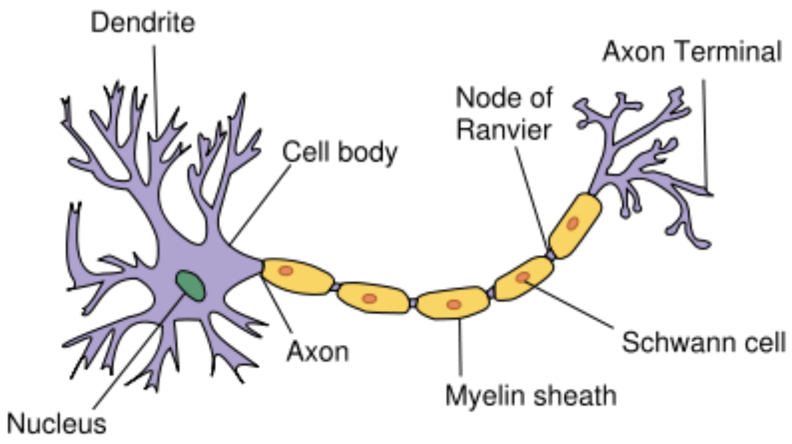


Figure 2.1 – Biological Neuron (Neuron)

Perceptron

Given this information, in 1958 Frank Rosenblatt building off of the work done by Walter Pitts and Frank McCulloch created what is known as the perceptron (Rojas, 2013, 55). Figure 2.2 shows an illustration of how it looks. Like an actual neuron it receives a given number of inputs. Each input is either a 0 or 1. There is also a weight associated with each input because as mentioned earlier some synapses matter more than other. The perceptron then sums the product of all the inputs and their corresponding weights. The sum is then passed through a function that determines whether or not the perceptron sends any signal. This is done using threshold logic. It equals one when the function input is greater than a given value and zero when it is not (Rojas, 2013, 57). Figure 2.3 shows an example when the threshold is 1. When the combined value of the inputs (the Sum-value) is less than 1 then the output of the function (the y-value) is 0. Otherwise the y-value is 1. This function is known as the activation function. While perceptrons are useful they also have their limitations. This is seen from the inability of a perceptron to describe the logical XOR function (Rojas, 2013, 63).

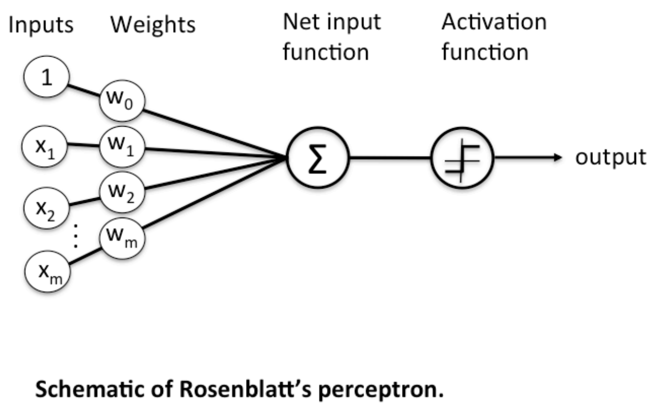


Figure 2.2 – Rosenblatt Perceptron (Raschka, "Rosenblatt Perceptron", 2015)

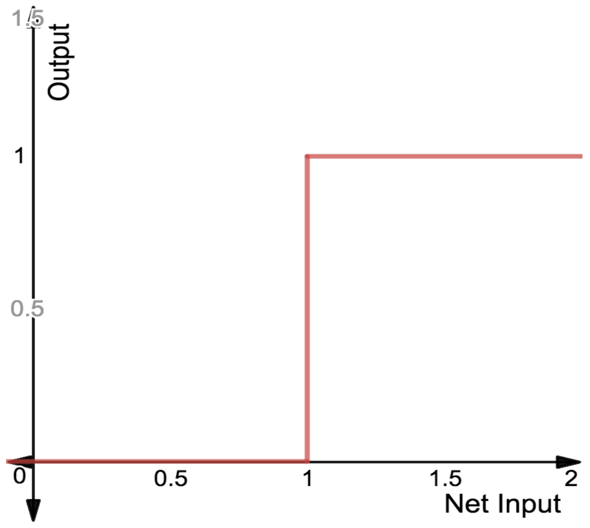


Figure 2.3 – Activation Function where the threshold is 1

Multilayer Perceptron

Using the idea of perceptrons we can now create a network of them which will produce a much more powerful structure. This is known as a multilayer perceptron (MLP) and is an interconnected layer of perceptrons. The goal of an MLP is to find the ideal set of weights between artificial neurons for our given problem (our set of inputs and what we are trying to predict). MLPs have three types of layers: input, hidden, and output layers. Each layer contains a certain number of perceptrons. The input layer receives the initial input parameters. The number of perceptrons in the first layer equals the number of initial inputs. The output layer is the last layer in our network. The number of perceptrons corresponds to the number of classes we are classifying. The hidden layers are the layers in between the input and output. While there can only be one input and output layer, there can be however many hidden layers you want (Rojas, 2013, 126). The neurons for each layer only receive inputs from perceptrons in the previous layer. So let's say we have an MLP with one input layer, three hidden layers, and one output layer. The second hidden layer only receives input from the first hidden layer and the output layer from the third hidden layer. As before with perceptrons the inputs to any perceptron is weighted and the summed together and compared to a threshold to see if the neuron should "fire".

For example, imagine for a given artificial neuron our threshold value is one and it receives four inputs (for the sake of making this clearer let's assume that these inputs have already been weighted). Those four inputs are: .25, .34, .32, and .4. If we add those together we get the value 1.31 which is greater than our threshold of one. Than means this neuron will output a signal. But, let's say our four inputs were: .1, .34, .32, and .2. If we add these values together we get .96, which is slightly less than 1. That means this neuron won't fire.

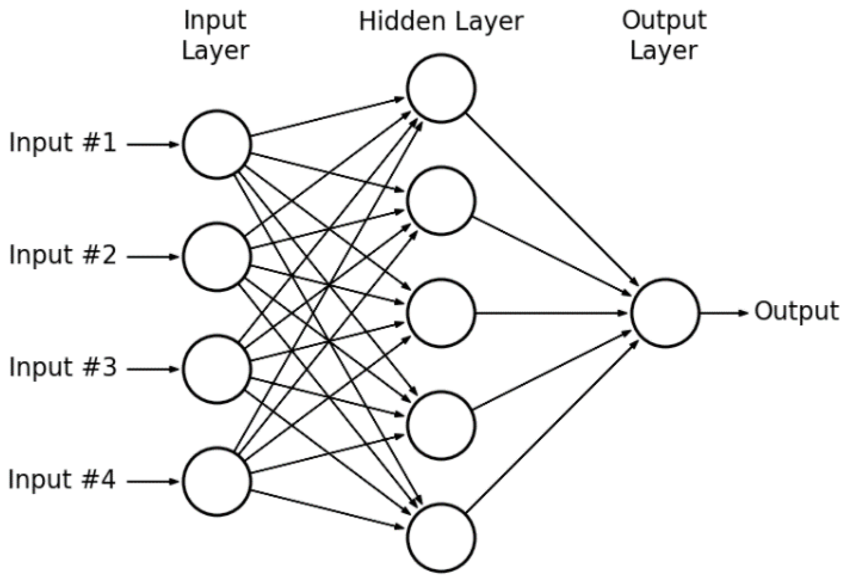


Figure 2.4 – An example of a Multilayer Perceptron Network (Zahran, Multilayer Perceptron Network, 2015)

Backpropagation

The goal is to train our network to output the correct results as often as possible. To do so, we need to have a good set of weights for every edge in our network. The procedure developed to handle this is called backpropagation. The goal of backpropagation is to find the optimal set of weights between nodes in our network. We want to teach our network to be as accurate as possible. So given any inputs it should output as close to the correct answer as possible. To achieve this we need to feed the network our data, see how well it does, and then modify the weights in a way that will make it perform better. We then repeat this process (feed the network data, evaluate the results, and change the weights) a fixed number of times or until we want to stop.

The first two steps of backpropagation involve feeding the network our data and calculating the error of our network. We start off with a dataset that contains both the initial inputs and the expected output. We use this train our network to correctly map the inputs to the output for unseen or newer data. We then randomly initialize the weights. We do this as it doesn't matter what we start off with, because as will be mentioned the weights will soon be adjusted anyways. To calculate the error we need a way to compute how much our model is off. To do so we use the sum of squared errors function, which is defined as:

$$E = \frac{1}{2}\sum_{i=1}^{n}(target_i - output_i)^2$$

That is, for each of our "n" outputs on each training example we output a number and compare it to what we expect (the target). The higher the error the worse our model performed. For example, let's say our network outputs: .75 and .25 while we expect .6 and .4 respectively. In that case our error would equal:

$$E = \frac{1}{2}((.6 - .75)^2 + (.25 - .4)^2) = .0225$$

Our next step involves getting the error to be as low as possible. We want to minimize the error function. The only way we can do that is by modifying the weights in the network. We do this by attempting to minimize the error function in regards to every weight in our network. The process we use to do this is called gradient descent. Gradient descent is an algorithm for searching for the minimum of a function.

The concept behind gradient descent can be thought of a person trying to find the lowest point in a valley from a random starting point when blindfolded. To find the lowest point we need a way of determining which way to step and how big of a step to take. In order to determine a direction to step in, we calculate the slope of the point we are standing at. This can be thought of as the steepness. The steeper we are, the more likely we are at a higher point in the valley and the more we need to travel down. When we take a step it's hard to know how big of a step to take. The basic rule of thumb is the steeper we are, the more we move, but we don't want to move too much (lest we pass over the lowest point). This concern is solved though the concept of a step-size. This is a constant that we multiply by our slope to determine how big our step is. If the slope if 12 feet and our step-size is .25, we then take a 3 foot (12 x .25) step downwards. So we keep: judging how steep we are, sensing the direction, and moving down a little in proportion to the steepness. The issue is that it can be very hard to find the lowest point in a valley. For this reason the backpropagation algorithm only runs a set number of times. Finding the actual lowest point can take a very long time so we are willing to settle for a point that's close enough for speed purposes.

But, before we can use gradient descent, we must first change the activation function. As a reminder, the activation function is how we convert the summed inputs in a neuron to an output signal. As shown in Figure 2.3, the neuron outputs a 1 if the sum is greater than or equal to some threshold θ (there 1), otherwise it outputs a 0. In order to do gradient descent the activation function needs to be differentiable, as we need to be able to take the derivative. The previous activation function isn't differentiable because the limit at the threshold θ isn't defined. The limit is the value some function approaches at a particular point. For example in Figure 2.3, as we approach the value of the "Net Input" equaling 1.5 we see that the output is 1. When we approach the "Net Input" equaling zero, the output is then 0. But, we run into an issue at our threshold. This is because the one-sided limits aren't equal. Using Figure 2.3 again, if we approach the value of "Net Input" equals one (our threshold) from the left side we see the output is 0. But approaching the threshold from the right side we get a value of 1. So the left hand limit at "Net Input" equals one isn't equal to the right hand limit. This means the limit at that point is undefined and therefore not differentiable. Another way to think about this is the fact that the slope of a function at a point is defined as rise over run (change in y-axis over change in x-axis). At our threshold the change in the y-axis is 1 but the change in the x-axis is 0. That means the slope at the threshold is 1 divided by 0, which is undefined. So instead we use sigmoid function, which is similar to our previous function and commonly used instead of a threshold function (Rojas, 2016, 125). As a consequence the neurons in the network now output values between zero and one. Figure 2.6 shows the sigmoid function.

$$Threshold \rightarrow out_{ij} = \begin{cases} 1 \text{ if } x \geq \theta \\ 0 \text{ if } x < \theta \end{cases}$$

$$Sigmoid \rightarrow out_{ij} = \frac{1}{1 + e^{-net_{ij}}}$$

Figure 2.5 – Definitions of Threshold and Sigmoid functions

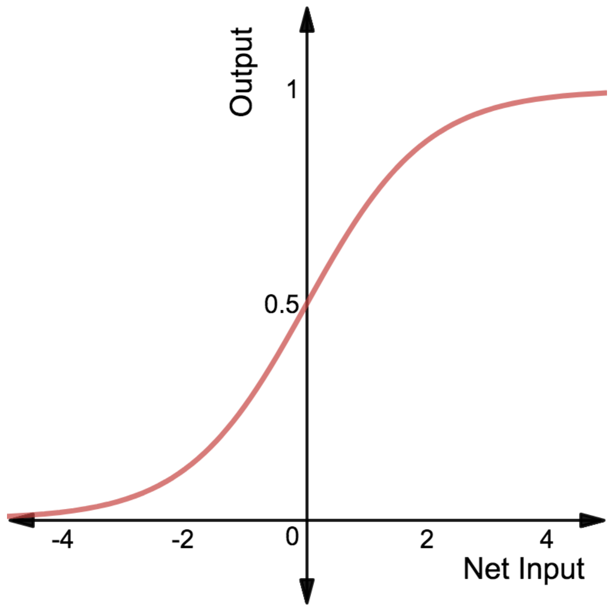


Figure 2.6 – Graph of sigmoid function

Now we can start using gradient descent to train our network. As mentioned previously we first feed the network the data and compute the error. We then use gradient descent to potentially adjust every weight in the network to make it more accurate. So for each weight w_{ij} we want to update it by some value Δw_{ij}. We are looking to find the value of Δw_{ij} for each w_{ij}. To do this we attempt to determine how shifting each weight in our network would affect the total error.

To do this we calculate the slope (also known as the gradient) of the error function, defined earlier, with respect to each weight w_{ij}. Using the gradient with respect to a given weight we then update that weight by moving in the direction of the negative slope. This is because we are searching for the lowest spot of our function, which would be where the slope is zero. Using the previous analogy, at the lowest point in a valley it is completely flat so there is no slope. We are searching for this point in our function. The amount we choose to move is the slope multiplied by a constant η (known as the step size). We then subtract this from our previous value of the weight.

The general equations for doing so are shown below where: Δw_{ij} is the update amount, η is the step size, i is the layer, j is the perceptron in the layer, $net_{i\,j}$ is the summed input for a neuron, $out_{i\,j}$ is the signal the neuron outputs, and the last equation is the partial derivative of the error function.

$$\omega_{ij} = \omega_{ij} + \Delta\omega_{ij}$$

$$\text{Slope of Error Function} = \frac{\partial E}{\partial w_{ij}}$$

$$\Delta\omega_{ij} = \eta \frac{\partial E}{\partial \omega_{ij}}$$

$$\frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial out_{ij}} \times \frac{\partial out_{ij}}{\partial net_{ij}} \times \frac{\partial net_{ij}}{\partial \omega_{ij}}$$

In order to evaluate the slope of the error function in regards to a given weight we need to use the chain rule, which the last equation above shows. The chain rule is a formula for calculating the derivative of a function that it is a composite function (a function that depends on the output of another function). We are trying to measure how a change in each of the weights will affect the total error of the system. This is how we determine how to change each weight to potentially get a lower total error. The issue is that a change in a given weight doesn't directly affect the error. It does so through intermediaries. As you will recall from earlier discussion, for the input to a neuron we multiply the weights ($w_{i\ j}$) by the outputs of the previous layer of neurons and sum up the products. Then using that sum ($net_{i\ j}$) we pass it through our activation function that produces the output for the neuron ($out_{i\ j}$). When the output of the neuron happens to be the output for the outer layer we can then calculate the error given that output and what we expect. This can be thought of as the error depending on the output, the output depending on the input, and the input depending on the weights. That is what the partial derivative above displays. This is the general schematic and it works well with dealing with weights in the outer layer. When dealing with those in the hidden layers, it takes longer for their effect to propagate to the output. This is because what the neuron outputs in the hidden layers will go on to feed the next layer (whether it be another hidden or output layer). This means we need to differentiate between the two weights in the output layer and the hidden layers (Rojas, 2013, 168-169). The two final equations for updating the weights for the outer and inner layers are shown below:

$$\delta_j = out_j \times (1 - out_j) \times (out_j - target_j)$$
$$Outer\,Layer \rightarrow \frac{\partial E}{\partial w_{ij}} = -out_i \delta_j$$
$$InnerLayer \rightarrow \frac{\partial E}{\partial w_{ij}} = out_i \times out_j \times (1 - out_j) \sum_{q=1}^{m} w_{jq} \delta_q$$

We now have all we need to train a feed-forward artificial network. We first feed the data forward and compute the total error. We then propagate backwards in the network and update the weights given the two formulas above. We keep repeating this process (feeding the data and adjusting the weights) for a specified number of times.

Convolutional Neural Networks (CNNs)

In this section, I will introduce the topic of convolutional neural networks (CNNs) and detail how they work. Convolutional neural networks are a special type of feed-forward artificial neural networks and are shown to work very well for visual classification tasks. I will do this by first going over the various layers that a CNN is comprised of. I will then discuss how a CNN is trained for a specific classification task.

Convolution Layer

The convolution layer is the most important of all the layers and lends itself to the name of the type of network. There are often many convolution layers found through the network, with one convolution layer always being the first layer. The main goal of this type of layer is to extract features from the current feature representation using the mathematical operation of convolution (Yamashita et al., 2018, 613). To perform the operation of convolution we define a number of kernels. A kernel is a matrix, usually 3x3 or 5x5, where each cell contains a weight. We ourselves don't define the weights in the cells but they are randomly initialized. As we will later discuss the goal is to find the ideal set of weights for the kernels (among other things). Figure 2.7 shows an example of a 3x3 kernel.

$$\begin{bmatrix} 12 & 30 & 27 \\ 6 & 76 & 92 \\ 11 & 101 & 43 \end{bmatrix}$$

Figure 2.7 – Example of a 3x3 kernel with randomly initialized weights

Now that we have a convolution kernel we can apply the operation of convolution. This begins by superimposing the kernel over the top left region of the input. We then multiply the kernel by the area of the input that it is superimposing (which by definition will have the same dimension). We then sum the multiplied weights together to get a single value that will be placed in the output (Yamashita et al., 2018, 613). Once we do that we then shift the kernel over to the right and repeat the same operation. We keep doing so until the kernel can't move to the right anymore (it doesn't fully superimpose a region of the input). We then shift the kernel down and repeat the same operation moving from left to right. We are done when we can't shift it down anymore. This type of operation will yield fewer features as nearby cells are combined. Figure 2.8 details an example of the operation.

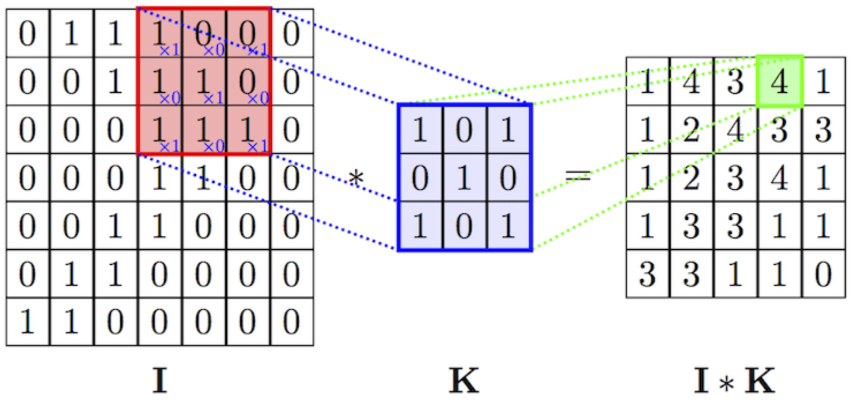


Figure 2.8 – An example of a convolution operation (Mohamed, "An example of pooling with a 2 × 2 filter and a stride of 2", 2015)

But how many cells do we shift when moving to the right and when moving down between each multiplication? This value is defined as the stride. Upon inspection we can determine that Figure 2.8 has a stride of one. This is because in the example the kernel can only move to the right and down 5 times when moving one cell at a time, and the resulting matrix is 5x5. So it must have a stride of one. The stride is often normally equal to one but can be found to be higher (Yamashita et al., 2018, 614).

Before applying convolution we often also may pad the input, which is done by adding cells with the value 0 around the input tensor before doing any operation (Yamashita et al., 2018, 614). For example, a padding of two will add a border of two zeroes around the input. This is done to preserve the current dimension of the matrix. An issue that arises when performing a number of convolution operations is that the input becomes smaller and smaller making it harder to create large networks (Deshpande). As seen in Figure 2.8 the resulting matrix I * K is smaller than the original matrix K. By applying padding we can perform more convolutions, as it takes longer to shrink down to a smaller size. Figure 2.9 shows an example of an input being padded using a padding of two.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 0 & 6 & 7 & 8 & 9 & 10 & 0 & 0 \\ 0 & 0 & 11 & 12 & 13 & 14 & 15 & 0 & 0 \\ 0 & 0 & 16 & 17 & 18 & 19 & 20 & 0 & 0 \\ 0 & 0 & 21 & 22 & 23 & 24 & 25 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.9 - An example of padding an input with a padding of two.

It is very common after a convolution layer to apply a nonlinear activation unit (Yamashita et al., 2018, 614). This is sometimes referred to as a layer in and of itself. What this does is apply a function to each cell of an input tensor. The most common function used is the RelU function. This is defined as the maximum of the value and zero. This eliminates all negative values and replaces them with zeroes. Other common functions include the hyperbolic tangent (Tanh) or the sigmoid function (Yamashita et al., 2018, 615). The purpose of this operation is to make the inputs more nonlinear so as to be able to find more elaborate patterns (Wu, 2017, 10).

Pooling Layer

A pooling layer reduces the dimensionality of the input by pooling together the value of nearby cells. Applying a kernel (often 2x2) over the input like done previously with the convolution layer achieves this. This provides "downsampling" by reducing the number of parameters in the network (Yamashita et al., 2018, 615). There are currently two popular ways to apply the operation of pooling. The first takes just the max of the group and the second the average (Yamashita et al., 2018, 616). For example, using max pooling and a 2x2 kernel, we would superimpose the kernel over the top left corner of the input and extract the maximum cell value in that input area. We again shift to the right and down as much as we can. This is similar to the same operation we did with convolution. These max values are then placed in a new matrix that is the output of the layer. Figure 2.10 shows an example with a 2x2 filter and a stride of 2.

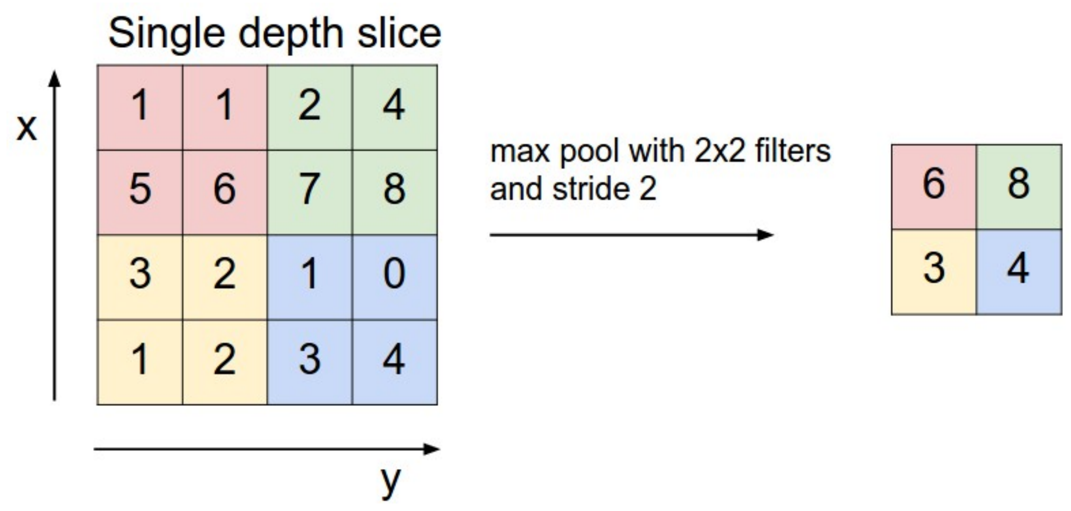


Figure 2.10 – An example of max pooling (cs231n, "The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square)").

Fully-Connected (FC) Layer

The fully-connected layer is always the last layer in the network. The job of it is to facilitate the final classification of the input into the set of discrete classes. A main component of achieving this is to flatten the current feature representation into a vector (Yamashita et al., 2018, 616). The layer includes an activation function at the end (usually softmax or sigmoid) to classify and generate the probabilities of the input belonging to each of the classes. Because of this, the fully-connected layer is basically a regular neural network with the current features created from the previous layers as the input. The previous section on general neural networks details how this layer would work.

Model Training

The creation of the model is generally referred to as training. That is, we attempt to train (or teach) the model to be able to match the inputs to outputs. In order to do so we must set aside a certain percentage of our total images to be used to create the model. This is known as the training set. We also set aside two other portions known as the validation and testing sets. The validation set is used to tune the model's hyperparameters and the test set is used to evaluate the final model. While there is no exact science behind how to split the data, it is done randomly and usually around 60% of the total data is placed in the training with the validation and testing sets each containing 20%.

In order to train the model we must be able to make use of the images. We do so by converting them to a numerical representation. Every image has a corresponding height and width representing the number of pixels in that image. Each pixel has a certain color combination, which is represented by a mixture of the three additive colors red, green, and blue (RGB). Each of the three colors takes on a value of 0 to 255 (which represents one byte of information). For example, a given pixel can take on the RGB value of 100, 200, 50 (which comes out to a kind of lime green). This means that we can represent an image as three matrices with one for each of the three colors. This is the structure that can be used as an input to a convolutional neural network and is referred to as a third-rank tensor. Also while not necessarily needed, it is very common to normalize each of the RGB values. This means we divide each value by 255 to get a number between 0 and 1.

As mentioned in the previous section the weights for a neural network are learnt through backpropagation. But what are the weights in a CNN? As mentioned for every convolution layer there is a number of kernels. What we are trying to do is modify the weights in those kernels. This is the objective when training a CNN. To do this we feed the network our inputs and see how well it does. We then propagate backwards through the network and update the weights in a way that we think would improve the model. The number of times we do this operation (feed forward all of the inputs) is referred to as an epoch. The most common algorithm used to update the weights is known as stochastic gradient descent. This is similar to gradient descent discussed in the previous section but differs in that we don't go though all the training examples before deciding how to update the weights. Instead we update them "stochastically". The common approach is to do so using a portion of the training set known as a batch size (Masters & Luschi, 2018, 2). This normally takes on a number between 1 and 64. For example, given a batch size of 16, we update the weights of the network after every 16 training examples.

A common concern when engaging in any machine or deep learning task is the risk of overfitting. As Salman and Liu note, overfitting is "when the model does well with training data and fails to perform well on test data" (Salman & Liu, 2019, 3). This occurs when the model learns the patterns in the training data that aren't generalizable to new data. Basically the model matches patterns found only in the training data and in no other set. There are many ways of attempting to combat this. One is known as regularization. Regularization introduces a penalty to the loss function for a given network (Keras Regularizers). This penalty helps the network avoid learning complex weights, which is a symptom of overfitting. Two popular variants of regularization are known as L1 and L2 regularization. Another popular technique is known as dropout. Dropout works by ignoring certain units (or neurons) during a single pass through the network. The units ignored are chosen randomly, as each one has a certain probability 'p' of being dropped during that epoch (Srivastava et al., 2014, 1930). This generally leads to less complex weights and also has the property of making the model more robust (Srivastava et al., 2014, 1937).

Transfer Learning and Fine-Tuning

Transfer learning and fine-tuning are two popular strategies used to train CNNs. Both of these strategies utilize a previously trained CNN to train the new network. The previously trained CNN is often trained on the ImageNet dataset. The ImageNet dataset currently contains over 14 million images for around 20 thousand different classes (Russakovsky et al., 3, 2014). It is used as a base to train networks on new datasets as it contains many classes that are considered very general.

For transfer learning we first create what is referred to as a "base network" often built on some dataset (e.g. ImageNet). We then take that network and use the features derived to train the network on a new classification task (Yosinski et al., 2014, 2). This means we just replace the fully-connected layer of the network. That is, we modify the classes the network will classify into (e.g. moving from almost 20 thousand with ImageNet to 12 for this experiment). Besides for this replaced layer we don't modify any of the weights trained from the base network. This technique is used when we think the classification tasks are somewhat similar.

Fine-tuning also uses a base network to construct the new CNN. The difference between this and transfer learning is that instead of not modifying the weights of the previous network we do (Yosinski et al., 2014, 2). When fine-tuning the network we "freeze" the first X layers of the network. This means we don't allow modification of those weights. For the Y remaining layers we allow the network to modify the weights of the previously trained network. Fine-tuning networks has been shown by research to produce strong results (Yosinski et al., 2014, 5).

Biological Inspiration

As noted last section, artificial neural networks have a biological inspiration. Mainly neural networks found in the brain. Convolutional neural networks are inspired specifically by neurons in the visual cortex. David Hubel and Torsten Wiesel showed that for most neurons in the visual cortex of cats, the neurons were activated or fired only when light was shone in a specific region of the eye. The area of the eye that causes a specific neuron to fire is referred to as the receptive field (576, 1959). In CNNs, the term receptive field is also used to refer to the specific regions covered by the convolution kernel that are mapped to the output (Le & Borj, 1, 2017). As mentioned previously, in the convolution layer we move the kernel across the input and derive a single weight from the area it covers. The area covered is analogous to the specific region of the eye in Hubel and Wiesel's experiment and the output produced to the specific neuron it can activate.

Chapter 3

In this chapter we will examine some of the literature in the field of art visual recognition. This will give us a starting point and a basis of comparison in regards to our methodology and results. Most of the previous work in visual art recognition has been done in regards to paintings. Specifically classifying paintings by artist, style, and genre. Recent advances in deep learning, specifically, CNNs, have spurred much work on the topic. In contrast to paintings, much less research has been done with sculptures. I specifically wasn't able to find any work done classifying sculptures by style using deep learning, which is what spurred the research presented in this work.

Related work with Paintings

There are a number of different public datasets that are utilized in these studies. The most prevalent one is the WikiArt dataset. The dataset contains over 250,000 pieces of art ("About Visual Art Encyclopedia"). The WikiArt dataset contains for every piece of art: the name of the object, the artist, the style, and the genre. A second dataset used is courtesy of the Web Gallery of Art ("Web Gallery of Art"). It mostly contains paintings and sculptures for pre-modern European art. Lastly is the TICC Printmaking dataset (Van Noord, & Postma, 2017). The dataset contains over 50,000 objects of art from the Rijksmuseum in the Netherlands. Some studies also utilize private datasets from an independent source such as Adrian Lecoutre et al. (2017).

The work done has focused on classifying paintings by artist, style, and genre. Various techniques have been used to achieve this task with the most prominent and successful involving convolutional neural networks (CNN). Eva Cetinic and Sonja Grgic created six different models using CNN-derived features, SIFT, GIST, HOG, GLCM and HSV color histograms to classify a painting into its genre by training a SVM over the features (2016). Utilizing the WikiArt dataset, they randomly sampled 1000 images from each genre for training. The model using the CNN-derived features performed over 20% better than the other models. Eva Cetini et al. later expanded on this study by attempting to classify paintings by style, artist, and genre. In the study they used the WikiArt, Web Gallery of Art, and TICC Printmaking datasets, creating separate models for each set (2018). To train each model they fine-tuned the CNN CaffeNet using different initial weights and by tuning a variable number of layers. They were able to achieve 57% testing accuracy predicting the style of the painting using the WikiArt dataset. Wei Ren Tan et al. also attempted to classify by style, artist, and genre using the multiple strategies on the WikiArt dataset (2016). They fine-tuned a custom CNN by first training it on the ImageNet dataset. They also attempted to derive features and use a SVM to classify the paintings. They were able to achieve a 54.5% accuracy rate for style using the fine-tuned CNN. Christian Hentschel et al. attempted to see how well they could classify paintings by epoch (which is analogous to style) using as little training data as possible (2016). To do so, they selected 22 styles from the WikiArt dataset and randomly chose 1000 images for training and 50 for testing. They trained different models on various ratios of the training data ranging from 5% to 100% using a fine-tuned CNN, a standard CNN, and a model built using improved fisher encodings. The best results were achieved using the fine-tuned CNN, which was able to correctly classify 55.9% of the test sculptures. Lastly, Adrian Lecoutre et al. attempted to also classify by style using the WikiArt dataset and a dataset obtained from an independent source (2017). They did so by fine-tuning the previously trained AlexNet and ResNet a variable number of layers. The ResNet trained model, classified slightly over 60% accurate, achieving the best results.

Related work with Sculptures

As opposed to paintings, much less work has been done in regards to sculpture classification. Relja Arandjelović and Andrew Zisserman attempted to identify the sculptor of a given sculpture and the name of it (2012). Sculptor identification was done by extracting features from the images using RootSIFT and boundary descriptors and converting them to bag of word representations. Using TF-IDF and K-means allowed them to query uniquely similar sculptures. Winner takes all and weighted voting schemes were used to select the sculptor for the given object. Sculpture identification was achieved by using the metadata of the image to perform a Google search and by checking if the sculpture was in the top 15 results. The data for this study contained 50128 images collected from Flickr. An earlier work by Gansel et al. focused on determining the region of specific Levantine ivory sculptures (2008). Using categorical features they trained a SVM to discern which region the sculptures belonged to, and were able to achieve 98% test accuracy.

Chapter 4

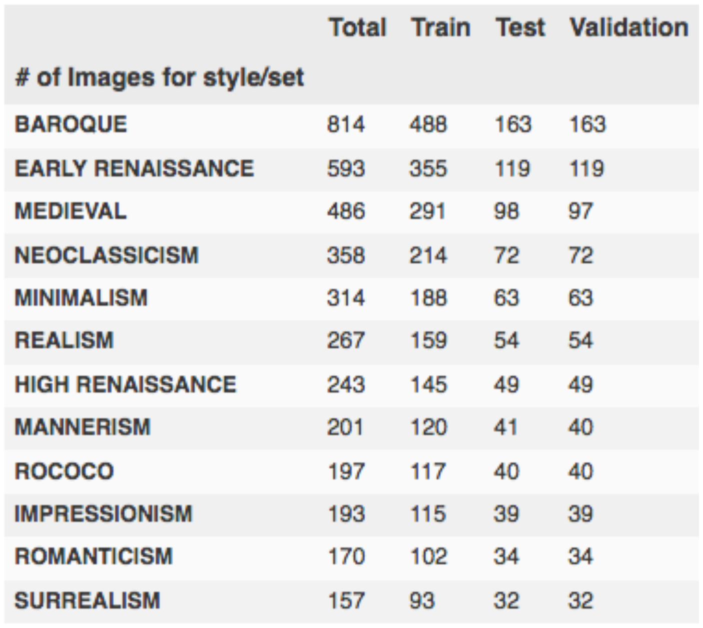
In this chapter, I will explain the methodology used to create and choose my final model. I will do this by noting where I obtained the data and how it was processed for the experiment. I will then detail the CNN architecture used. Next, I will go over how the model was trained. Lastly, I will show an example of how the model works on a single sculpture. This chapter outlines how I attempted to answer my initial research question.

Data Sources

Three separate data sources were used to construct the dataset for this analysis. The first source used was the WikiArt database (“Visual Art Encyclopedia”). WikiArt is one of the largest and well-documented public art databases available. It contains a large quantity of paintings and sculptures with their associated artist, object name, and style along with an image of the artwork. WikiArt mostly contains artworks from 19th century onwards and contains roughly 3000 sculptures. The second source used was the Web Gallery of Art. As per the website they contain an extensive collection of “European fine arts from the 3rd to 19th centuries” (“Web Gallery of Art”). The database contains around 5000 sculptures. The third source used is the National Gallery of Art (“National Gallery of Art”). The National Gallery of Art is a government run art museum located in Washington D.C. The entire collection is available online. This collection was used to supplement the artworks for a couple of styles found in the other two data sources.

A large number of styles were available from the combined datasets. In total 12 styles were used. That is, I trained the model to classify sculptures into 12 different styles. Before I go any further it's important to qualify what I mean by 'style'. In this piece style is defined as a period or movement of art. This means that both Baroque and Minimalism are considered styles (even though frequently the former is considered a period and the latter a movement). All art produced during the medieval period are also assumed to be a single style. Analysis of the dates of the medieval objects in my dataset indicates a vast majority of them belong to the Gothic period. The styles used and the number of images for each type is listed in Figure 4.1.

Before we start the training process, we must partition our data into three separate sets: A training, validation, and testing set. The training set will be used to create the model. The validation set will be used to check the progress of each individual model and help us choose a final model. And lastly, the testing set will allow us to evaluate the performance of the final model. I chose to keep 60% of the pictures for the training set, 20% for the validation set, and 20% for the testing set. The decision of which images to place in each set was done randomly. Due to class imbalance each style was itself split randomly by the above percentages and placed into the three groups. This ensures the same percentage of sculpture styles is in each set. All images were then rescaled and resized to 299x299. The number of images in each group is 2387 in the training set, 802 in the validation set, and 800 in the testing set.



# of Images for style/set	Total	Train	Test	Validation
BAROQUE	814	488	163	163
EARLY RENAISSANCE	593	355	119	119
MEDIEVAL	486	291	98	97
NEOCLASSICISM	358	214	72	72
MINIMALISM	314	188	63	63
REALISM	267	159	54	54
HIGH RENAISSANCE	243	145	49	49
MANNERISM	201	120	41	40
ROCOCO	197	117	40	40
IMPRESSIONISM	193	115	39	39
ROMANTICISM	170	102	34	34
SURREALISM	157	93	32	32

Figure 4.1 – The number of sculptures in each set for each style

CNN Architecture

The architecture used to construct the model was the Xception architecture (Chollet, 2016) that is based on the Inception architecture (Szegedy, 2014). The Xception architecture uses only depthwise separable convolutional layers and is made up of 36 separate convolutional layers that are arranged into 14 blocks. The model takes the input as a 299x299 RGB image with each value being normalized to ensure a value between 0 and 1. The model was also trained separately using both the ImageNet and JFT datasets (Chollet, 2016, 4). In this experiment the ImageNet weights are used and fine-tuned for our specific task. The architecture for the Xception model and the ImageNet weights were obtained via the deep learning library Keras (Chollet, 2015).

Model Training

As mentioned, fine-tuning is a popular strategy used when constructing a CNN for a specific task. Using this method we use the architecture and initial weights (usually from being trained on the ImageNet dataset) and retrain the network as to modify them slightly. The model is also adjusted as to classify images into the correct number of classes we are using in our experiment. We will be using the CNN architecture defined in the previous sub-section.

The first step in this procedure is to replace the top of the model that is used to classify the input into a finite set of classes. In our model we want it to be classified into one of the 12 styles. A diagram of that new fully-connected classifier is shown below:

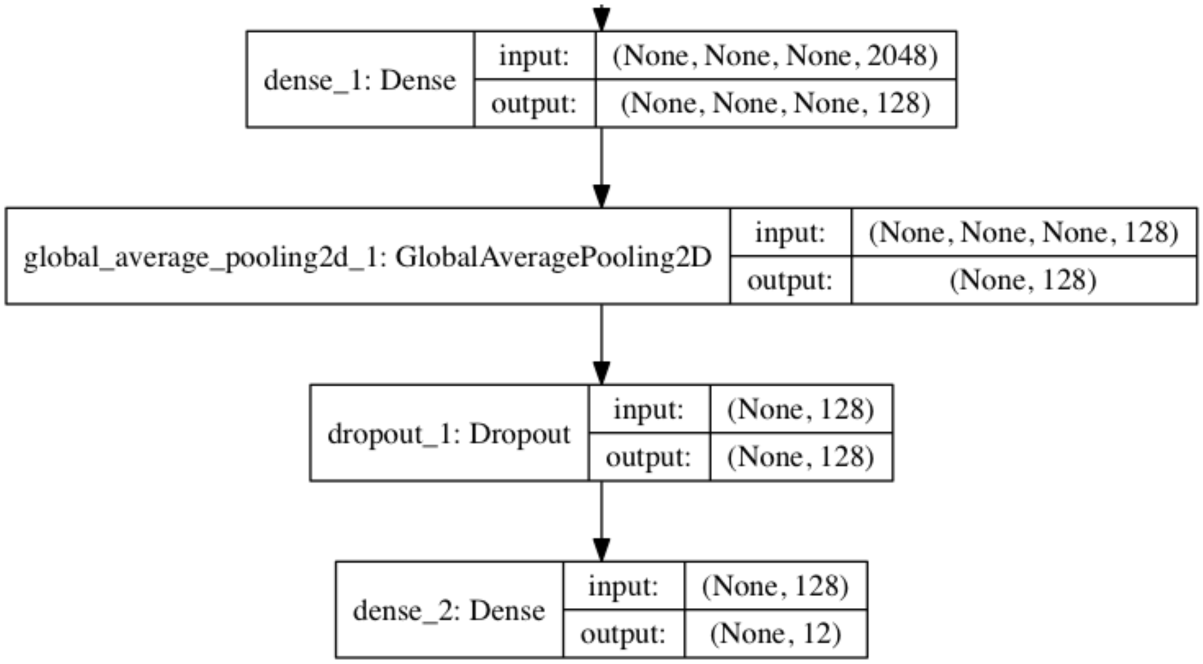


Figure 4.2 – Diagram of the new fully connected classifier for our model

The dense_1 layer is defined as a Rectified Linear Unit (ReLU) that includes l2 regularization with a penalty of .0075. The next unit is a pooling layer (global_average_pooling2d_1). This helps us reduce the dimensionality of the input (here to a 2D structure). Next is a dropout layer (dropout_1). This is used to combat overfitting by randomly ignoring a percentage of neurons. This layer specifies that 50% of the neurons in the model are to be ignored or “dropped”. Lastly is the prediction layer (dense_2). This layer uses a softmax function to classify the inputs into one of the 12 classes. The output of the CNN is now mapped to our classes.

We have now defined the structure of our CNN. It is now time to train the weights for our task. Given the small amount of data at our disposal, we will use data augmentation techniques to increase the number of training images. This will allow us to have a bigger dataset and help us prevent overfitting. The strategy to augment the training data was done by producing images by randomly adjusting the shear range, randomly zooming in, and randomly flipping an image on its horizontal axis.

The first step is to only train the classifier we just added. This has to be done before we can fine-tune our model. To do so we freeze every layer besides those in the new top classifier. By freezing the layers we don’t allow the weights in them to be modified, so only the weights of the layers we just added will be modified. During this training procedure the model was trained for 25 epochs with a batch size of 16. The optimizer used is RMSprop with a learning rate of .001. After each epoch the current model was evaluated on the validation set. The results of the accuracy on both the training and validation sets are shown in Figure 4.3.

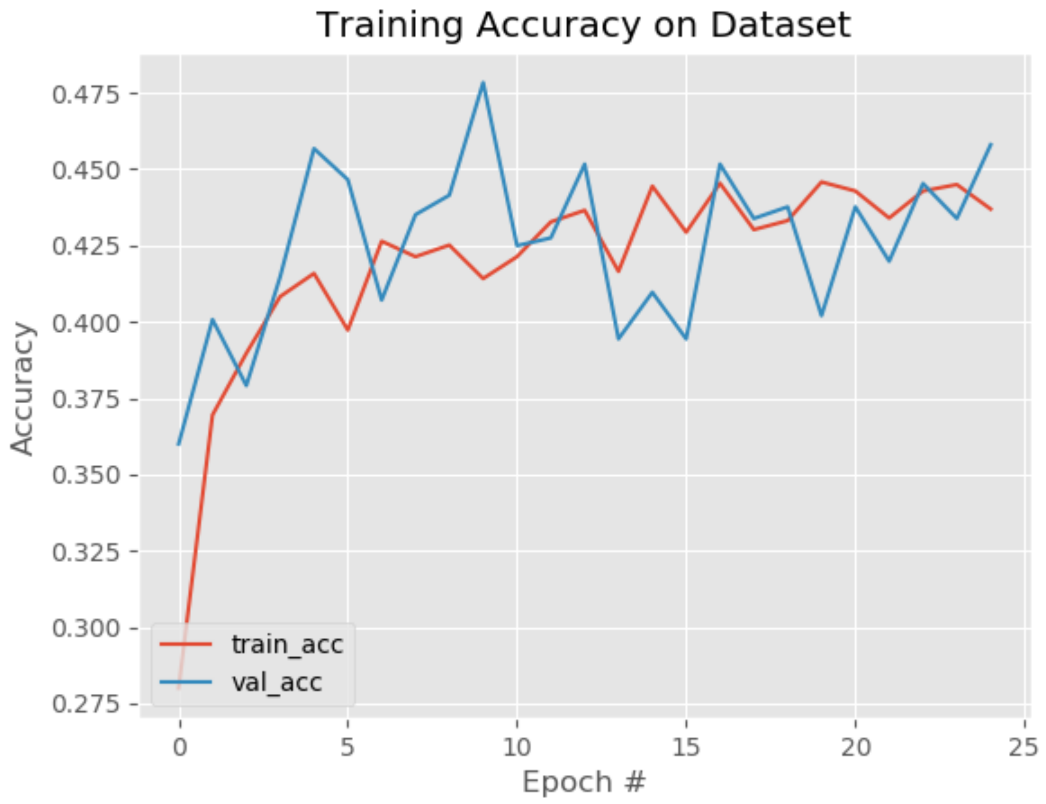


Figure 4.3 – Plot of the training and validation accuracy during pre-training

Now that the newly added top classifier's weights are initialized we can now begin fine-tuning the model. The obvious question that comes to mind when doing so is how many layer do we fine-tune? Do we allow all of the weights in the network to be modified or only a portion? In order to determine this, a number of separate models are built each fine-tuning a different number of layers in the network. We will then evaluate each model and see which one performs best. Each model begins with the same weights (which was just previously trained) and uses the same procedure. Which layers are allowed to be modified or not is determined by the block it is contained in. Of important note is the fact that the Xception module contains 14 blocks. The following models were built:

- The weights of the first 13 blocks were frozen
- The weights of the first 11 blocks were frozen
- The weights of the first 9 blocks were frozen
- The weights of the first 7 blocks were frozen
- The weights of the first 5 blocks were frozen
- The weights of the first 3 blocks were frozen

Each sub-model was trained for 50 epochs with a batch size of 16. The optimizer used was Stochastic Gradient Descent with a learning rate of .0001 and a momentum of .9. After each epoch the current model was evaluated on the validation set. The results of both the accuracy and the loss on the validation set for each model are shown in Figures 4.4 and 4.5 (Note: Each model is referred to by the number of blocks frozen).

We see a clear overall trend between how well the model did and the number of blocks frozen. Those with fewer blocks frozen did better. With that said, while that progression exists the top few models each performed very similar to each other making it difficult to distinguish between them. Upon closer inspection I've determined that the 'block3' model performed best, this being the model with only the first three blocks frozen. For this reason this was chosen as my final model.

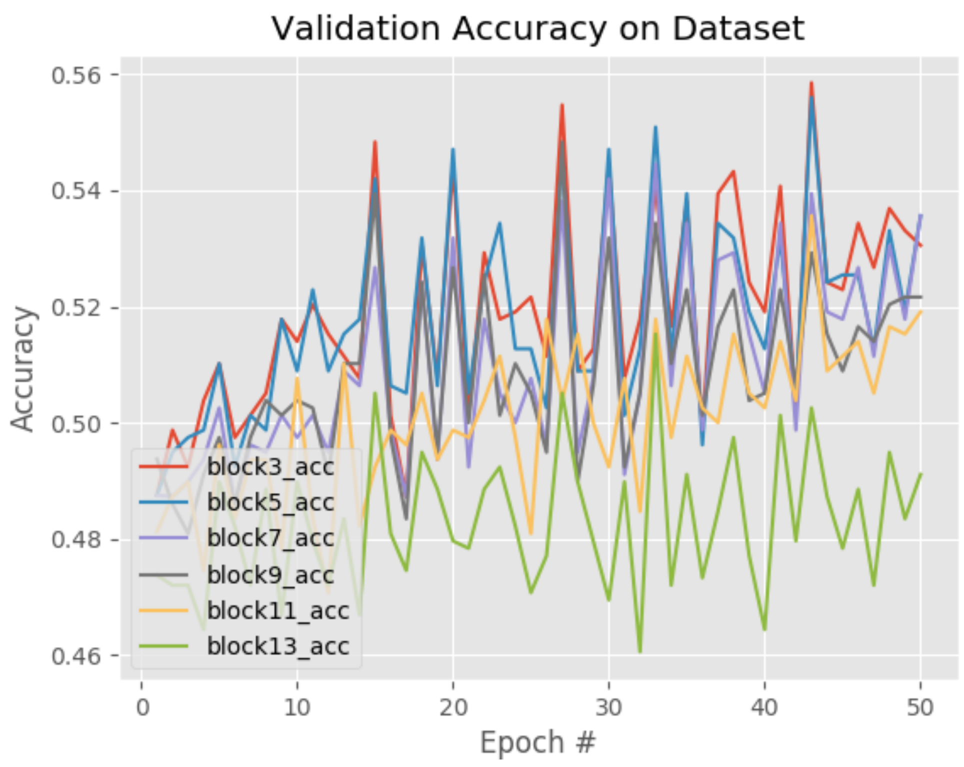


Figure 4.4 – Plot of the accuracy for the various fine-tuned models for each epoch

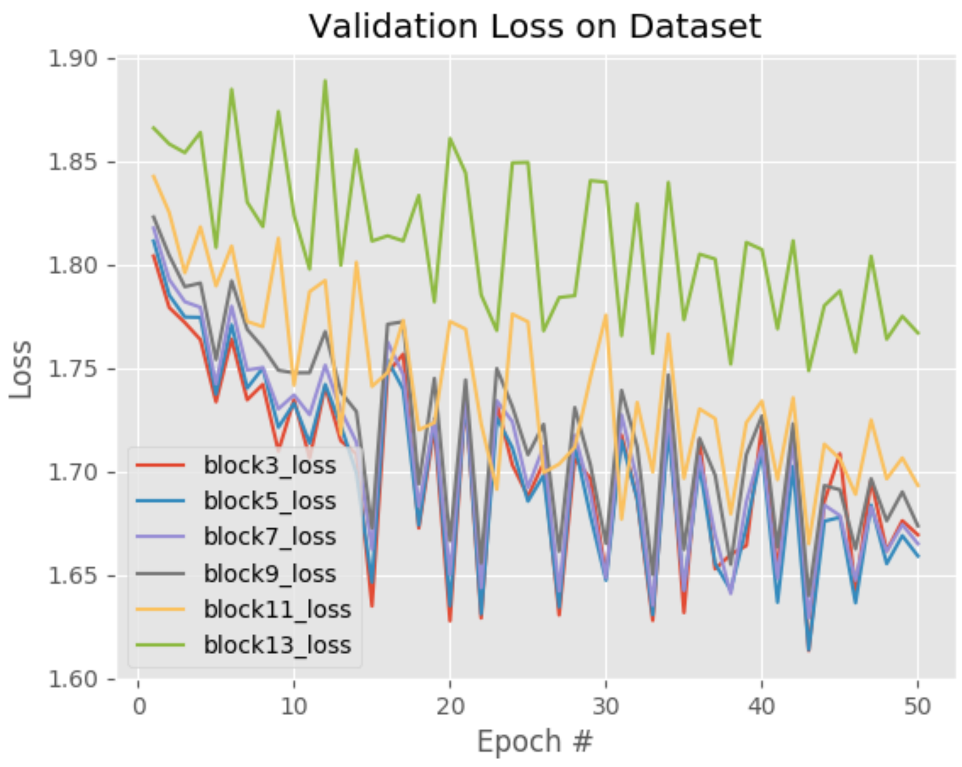


Figure 4.5 – Plot of the loss for the various fine-tuned models for each epoch

Example

Before evaluating the model, it is worthwhile to see how the model works in action. To this point, we will see how it does on a single example. We will use the sculpture *David* by Michelangelo for this exercise. This sculpture is considered to be belonging to the High Renaissance style. Figure 4.6 is an image of the sculpture.

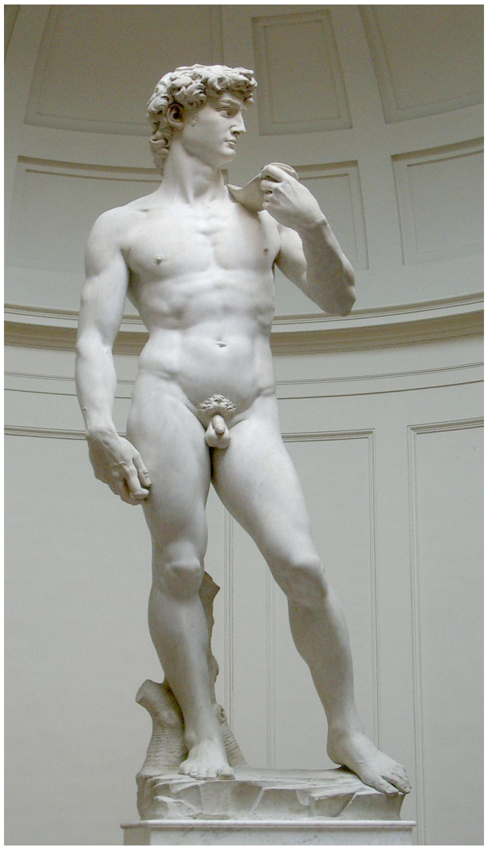


Figure 4.6 – The sculpture *David* by Michelangelo

The model outputs the probability that the sculpture belongs to every style. Figure 4.7 displays a chart with those probabilities. The model correctly thinks that the most probable style of this sculpture is High Renaissance with an almost 50%

probability. We also see that the two other most likely styles, Baroque and Early Renaissance, are two that are considered to be similar to High Renaissance.

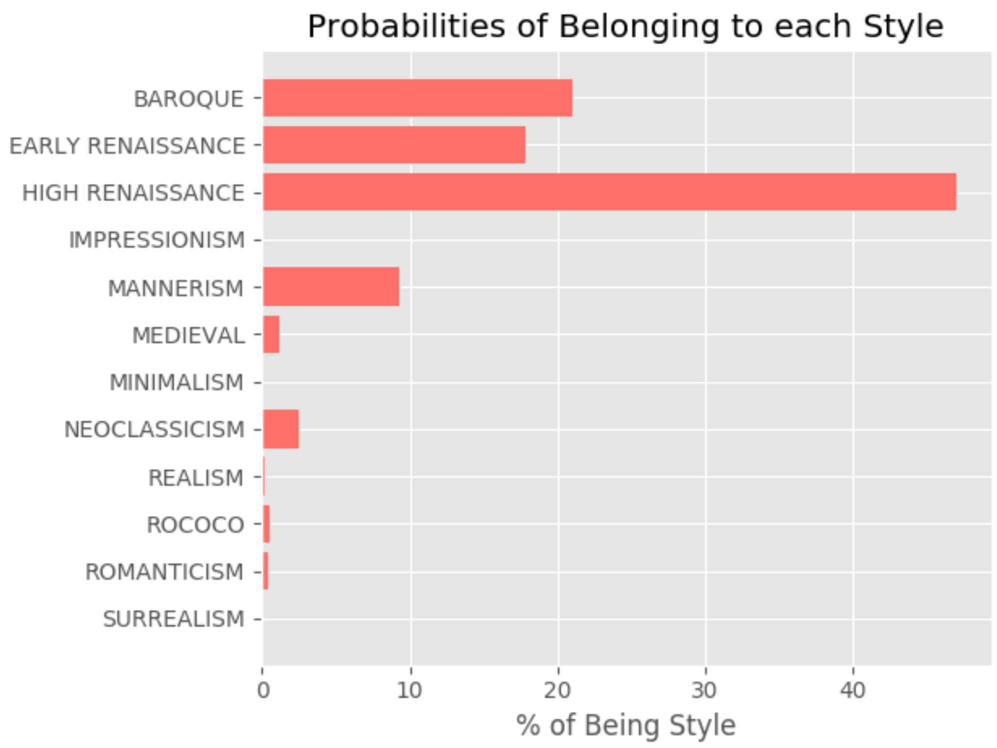


Figure 4.7 – Probability of *David* by Michelangelo belonging to each style.

Chapter 5

In this chapter, I will conduct an evaluation of the model created in the last section. I will do this by calculating the overall accuracy and loss on the test set. I will then construct a confusion matrix to determine the styles in which the model excels and struggles with. Lastly, I will show an example of the model incorrectly classifying a sculpture to highlight the limitations of the model. This allows us to determine how well I was able to achieve my objective.

Evaluation

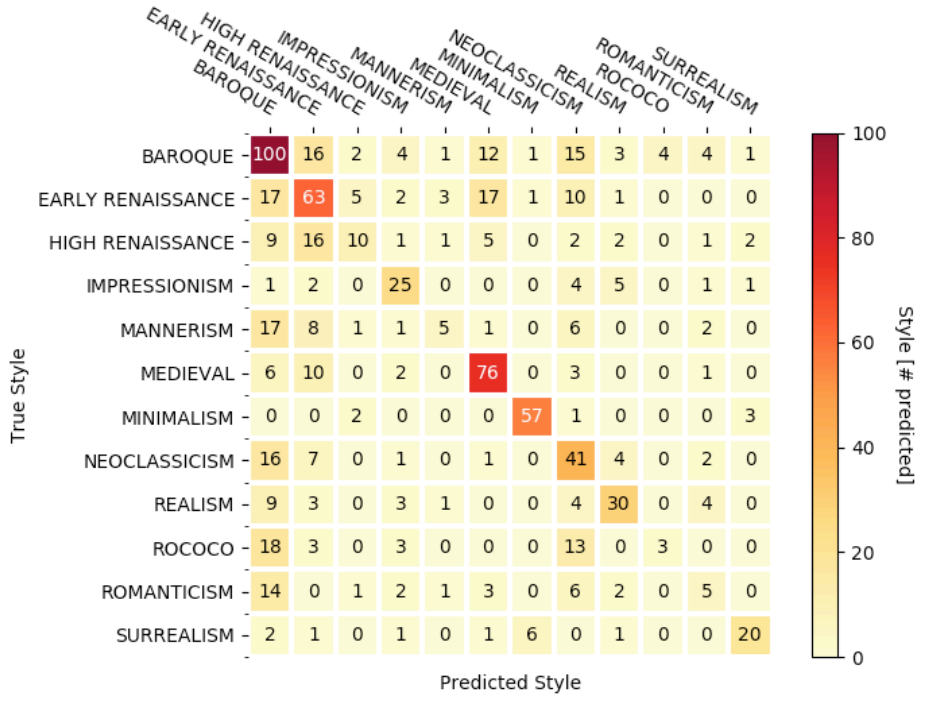
In the last section we went over the methodology used in constructing the model. The final model was chosen used the Xception architecture and fine-tuned the network to our classification task. We found that the model that gave the best performance was the one in which the weights for the last 11 layers were fine-tuned (or the first three were frozen). Given this model we can now evaluate it on the previously mentioned test set.

We first evaluate it on the test set as a whole. Using our model and the test sculptures we look to see how well it does it predicting their correct classes. We find that following results:

- **Test Accuracy: 54.1%**
- **Test Loss: 1.62**

The accuracy is slightly less than similar work done with paintings. In terms of classifying paintings by style, as discussed earlier in Chapter 3, most work ranges in accuracy from 55% and 61%.

It is also interesting to how the model performs on a style-by-style basis. This will allow us to see where the strengths and weaknesses of the model lay. To this point, we will look at a confusion matrix for the predictions on the test set (of which the number of images for each can be found in Figure 4.1). The confusion matrix is show in Figure 5.1:

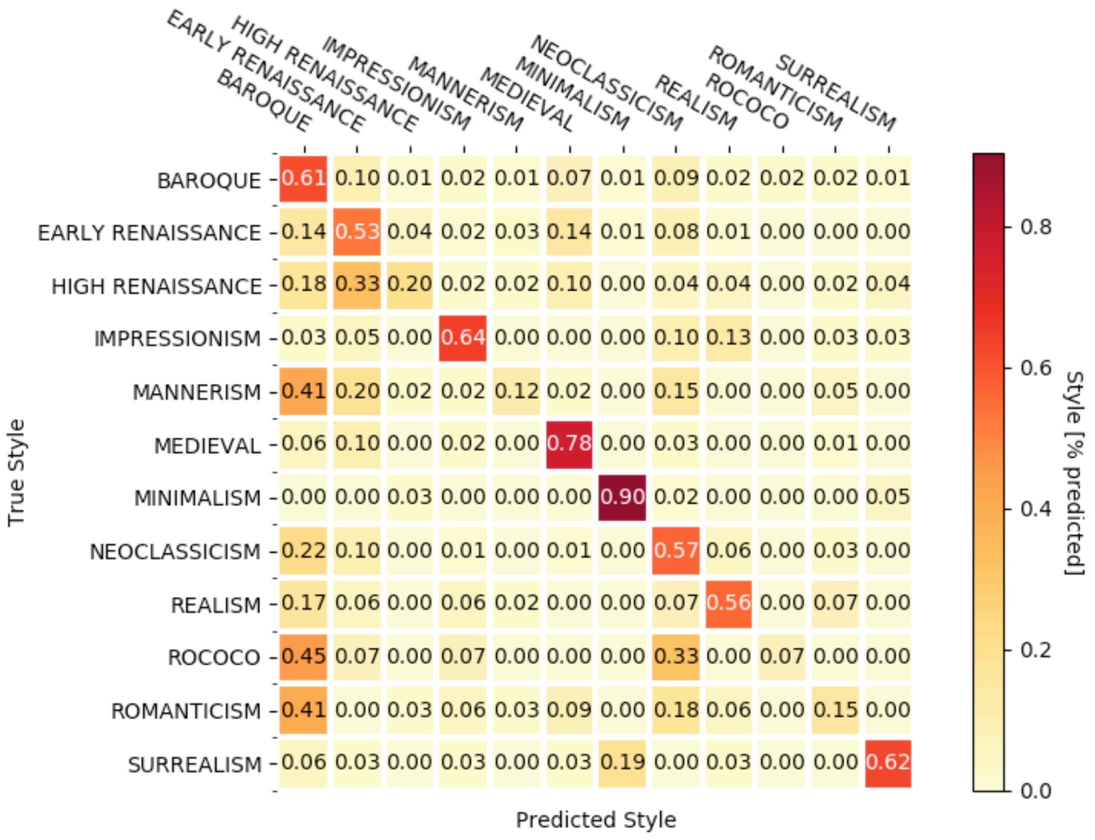


	BAROQUE	EARLY RENAISSANCE	HIGH RENAISSANCE	IMPRESSIONISM	MANNERISM	MEDIEVAL	MINIMALISM	NEOCLASSICISM	REALISM	ROCOCO	ROMANTICISM	SURREALISM
BAROQUE	100	16	2	4	1	12	1	15	3	4	4	1
EARLY RENAISSANCE	17	63	5	2	3	17	1	10	1	0	0	0
HIGH RENAISSANCE	9	16	10	1	1	5	0	2	2	0	1	2
IMPRESSIONISM	1	2	0	25	0	0	0	4	5	0	1	1
MANNERISM	17	8	1	1	5	1	0	6	0	0	2	0
MEDIEVAL	6	10	0	2	0	76	0	3	0	0	1	0
MINIMALISM	0	0	2	0	0	0	57	1	0	0	0	3
NEOCLASSICISM	16	7	0	1	0	1	0	41	4	0	2	0
REALISM	9	3	0	3	1	0	0	4	30	0	4	0
ROCOCO	18	3	0	3	0	0	0	13	0	3	0	0
ROMANTICISM	14	0	1	2	1	3	0	6	2	0	5	0
SURREALISM	2	1	0	1	0	1	6	0	1	0	0	20

Figure 5.1 – Confusion matrix for the test predictions

A confusion matrix tells us for each class – what classes our model classified the sculptures of that class into. For example, in Figure 5.1 the first box in the top left corner contains the number 100. This means that for sculptures in which the true style (the style on the y-axis) is Baroque, 100 images were classified as Baroque (style on the x-axis). What the box in the top right corner tells us is for sculptures that truly belong to Baroque, 1 of them was classified as Surrealism. Using this table we can compute important measures like accuracy, true positive rate, and others.

A slightly more convenient way to look at a confusion matrix is by normalizing the numbers. By this we mean dividing every number by the true count of sculptures for that style. This will then give us the percentage of each sculpture style that was classified into each corresponding style, which is more convenient than just the count. This can be found in Figure 5.2.



	BAROQUE	EARLY RENAISSANCE	HIGH RENAISSANCE	IMPRESSIONISM	MANNERISM	MEDIEVAL	MINIMALISM	NEOCLASSICISM	REALISM	ROCOCO	ROMANTICISM	SURREALISM
BAROQUE	0.61	0.10	0.01	0.02	0.01	0.07	0.01	0.09	0.02	0.02	0.02	0.01
EARLY RENAISSANCE	0.14	0.53	0.04	0.02	0.03	0.14	0.01	0.08	0.01	0.00	0.00	0.00
HIGH RENAISSANCE	0.18	0.33	0.20	0.02	0.02	0.10	0.00	0.04	0.04	0.00	0.02	0.04
IMPRESSIONISM	0.03	0.05	0.00	0.64	0.00	0.00	0.00	0.10	0.13	0.00	0.03	0.03
MANNERISM	0.41	0.20	0.02	0.02	0.12	0.02	0.00	0.15	0.00	0.00	0.05	0.00
MEDIEVAL	0.06	0.10	0.00	0.02	0.00	0.78	0.00	0.03	0.00	0.00	0.01	0.00
MINIMALISM	0.00	0.00	0.03	0.00	0.00	0.00	0.90	0.02	0.00	0.00	0.00	0.05
NEOCLASSICISM	0.22	0.10	0.00	0.01	0.00	0.01	0.00	0.57	0.06	0.00	0.03	0.00
REALISM	0.17	0.06	0.00	0.06	0.02	0.00	0.00	0.07	0.56	0.00	0.07	0.00
ROCOCO	0.45	0.07	0.00	0.07	0.00	0.00	0.00	0.33	0.00	0.07	0.00	0.00
ROMANTICISM	0.41	0.00	0.03	0.06	0.03	0.09	0.00	0.18	0.06	0.00	0.15	0.00
SURREALISM	0.06	0.03	0.00	0.03	0.00	0.03	0.19	0.00	0.03	0.00	0.00	0.62

Figure 5.2 – Normalized confusion matrix for the predictions on the test set. The numbers can be interpreted as percentage of the true style that was predicted another style (the 'Predicted Style'). For example 61% of sculptures that were actually Baroque were predicted to be Baroque and 10% of Baroque images were predicted as Early Renaissance.

Using Figure 5.2 we see some fairly interesting results. We can see that the highest sculpture accuracy is 90% for minimalism and the lowest is 7% for Rococo. A vast majority of the styles seems to be in the 50-65% range. Another interesting find can be in the false positive rate for the Baroque style. A high number of other sculpture styles were classified as Baroque with Romanticism at 41%, Rococo at 45%, and Mannerism at 41%. This is likely due to both the similarity of these styles to Baroque and the fact the Baroque style contains the most training data by a margin. In the absence of more data to distinguish similar styles, the model likely defaulted to Baroque.

Example

I think it would be worthwhile to see an example of where the model mistakenly classifies a sculpture to show the areas where the model struggles. We will use the sculpture *The Intoxication of Wine* by Claude Michel for this exercise. This sculpture is considered to be belonging to the Rococo style. Figure 5.3 is an image of the sculpture.

The model outputs the probability that the sculpture belongs to every style. Figure 5.4 displays a chart with those probabilities. The model incorrectly thinks that the most probable style of this sculpture is Baroque with about an 80% probability. The true style, Rococo, comes in second with only a 15% probability.

Due to the lack of data available for Rococo sculptures the model likely isn't able to accurately generalize the Rococo style. In the dataset we have multiple styles that are similar to each other. For example, Early and High Renaissance, Baroque, Neoclassicism, Rococo, and Romanticism are all similar to an extant. What this means is that the differences between them are more subtle and harder for the model to pick up. In order to accurately model them, we need a lot of data. This is likely the issue here with Rococo. Not enough data exists for the model to find the appropriate patterns to properly distinguish it. And in the absence of being able to determine what constitutes Rococo, the model will default to a style like Baroque, which contains the most data and is therefore best understood.

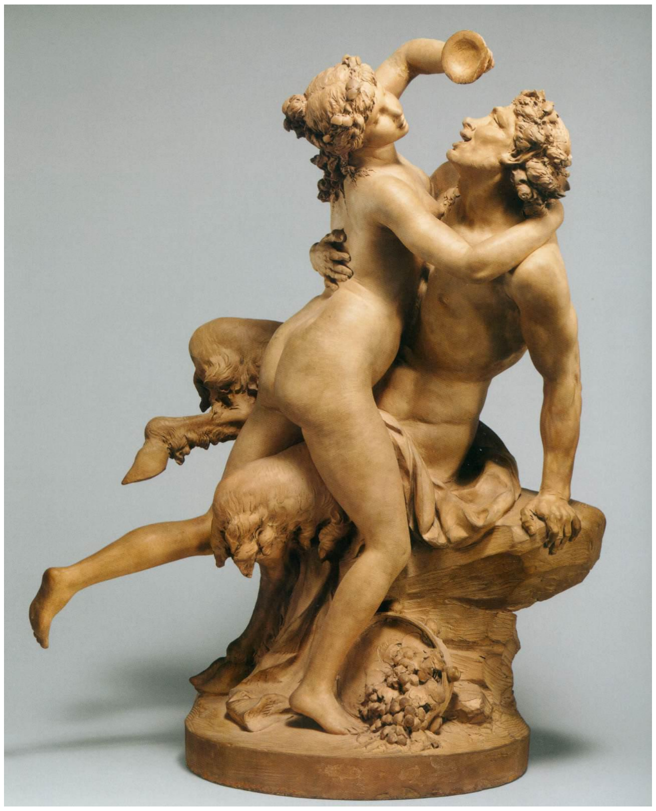


Figure 5.3 - *The Intoxication of Wine* by Claude Michel

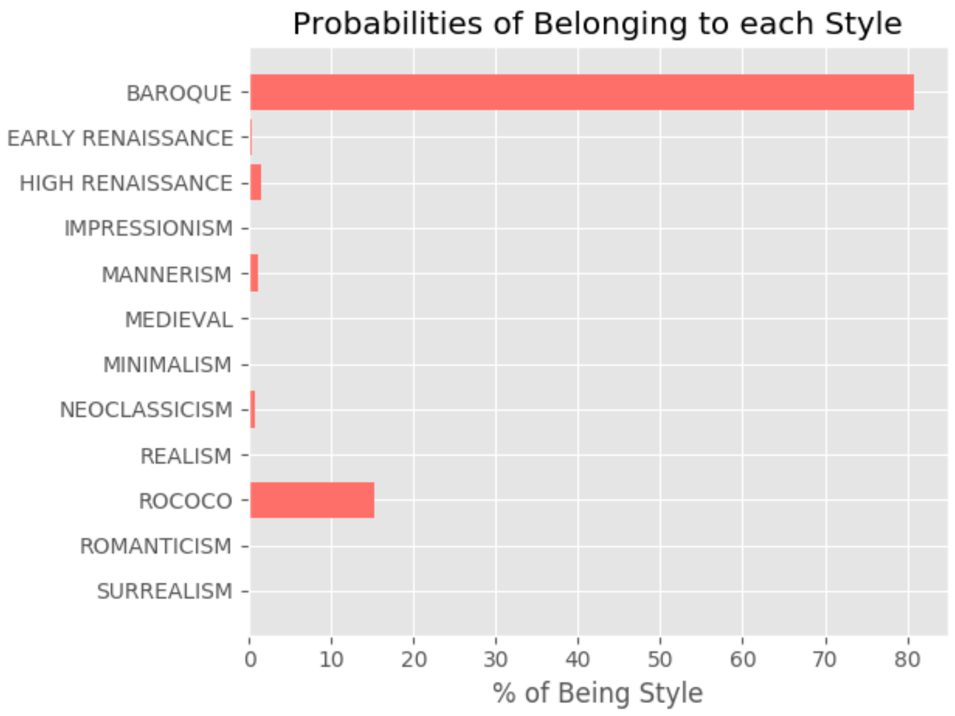


Figure 5.4 – Probability of *The Intoxication of Wine* by Claude Michel to each style

Conclusion

In this paper we attempted to see how well can classify sculptures by art style using convolutional neural networks (CNNs). In order to achieve this, sculpture images from 12 different styles were collected. Using this data we fine-tuned the Xception architecture for out classification task. Evaluating the resulting model on the pre-portioned test set showed positive results. Specifically 54.1% of sculptures in the test set were classified correctly. These results, while not as good, are close behind similar research done with paintings.

The motivation behind this research was to extend the previous art visual recognition research from paintings to sculptures. I'd argue it is noteworthy to see how well this type of work can be done for 3D art objects like sculptures as opposed to 2D objects like paintings. The applicability of this research mostly lay with image retrieval tasks, specifically content-based image retrieval (CBIR). CBIR attempts to retrieve images matching a certain criteria based on visual features extracted from those images, as opposed to using metadata. The benefits of such a system is by not having to engage in the task of writing metadata for every image in a database. This can come in handy for large art collections where there can be tens of thousands of images.

There are multiple avenues in which the work presented here can be expanded. A more robust approach could be taken in building the model. Hyperparameter tuning could be utilized to determine better values for hyperparameters like the learning rate, the dropout rate, and the amount of L2 or possibly regularization. Through this technique we would try out a number of different combination of different hyperparameter values and test to see which model performs best. Different CNN architectures can also be tested to see which one performs best. In this paper only the Xception architecture was fine-tuned. Other architectures such as ResNet (He et al., 2016) or DenseNet (Huang et al., 2016) can be tested to see if they are able to perform better. More data can also be collected. In this paper the number of different styles used was constrained by the number of available images present for the styles. The dearth of images for certain styles used in the model also made it difficult for the model to classify them. More images would allow the model to better distinguish among the different styles and allow for more styles to be included. Lastly, as opposed to using one image per sculpture, an alternate approach can be used that utilizes multiple images from different views of the sculpture. Since a sculpture is not a 2D object like a painting a single image isn't able to capture all of the details present in a sculpture. Therefore an approach that uses images from multiple views of the sculpture and utilizes a multi-input CNN architecture could plausibly perform better. An issue with this approach is collecting a suitable amount of data to train the model since multiple images per sculpture are required.

Works Cited

About Visual Art Encyclopedia. (n.d.). Retrieved March 10, 2019, from https://www.wikiart.org/en/about

Arandjelović, Relja, & Andrew Zisserman. (2012). Name that sculpture. *Proceedings of the 2nd ACM International Conference on Multimedia Retrieval*. ACM, 2012.

Cetinic, Lipic, & Grgic. (2018). Fine-tuning Convolutional Neural Networks for fine art classification. Expert Systems With Applications, 114, 107-118.

Cetinic, E., & Grgic, S. (2016). Genre classification of paintings. 2016 International Symposium ELMAR, 2016, 201-204.

Chollet. F. (2015) Keras. https://github.com/fchollet/keras.

Chollet, F. (2016). Xception: Deep Learning with Depthwise Separable Convolutions.

cs231n. (n.d.). The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square). Retrieved March 23, 2019, from http://cs231n.github.io/convolutional-networks/#pool

Deshpande, A. (n.d.). A Beginners Guide To Understanding Convolutional Neural Networks Part 2. Retrieved March 19, 2019, from https://adeshpande3.github.io/A-Beginners-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

Gansell, Amy Rebecca, et al. (2008) Predicting Regional Classification of Levantine Ivory Sculptures: A Machine Learning Approach. 2008.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2016, 770-778.

Hentschel, C., Wiradarma, T., & Sack, H. (2016). Fine tuning CNNS with scarce training data - Adapting imagenet to art epoch classification. 2016 IEEE International Conference on Image Processing (ICIP), 2016, 3693-3697.

Huang, G., Liu, Z., Van der Maaten, L., & Weinberger, K. (2016). Densely Connected Convolutional Networks.

Hubel, D., & Wiesel, T. (1959). Receptive fields of single neurones in the cat's striate cortex. The Journal of Physiology, 148, 574-91.

Keras Regularizers. (n.d.). Retrieved March 19, 2019, from https://keras.io/regularizers/

Le, H., & Borji, A. (2017). What are the Receptive, Effective Receptive, and Projective Fields of Neurons in Convolutional Neural Networks?

Lecoutre, A., Negrevergne, B., & Yger, F. (2017). Recognizing Art Style Automatically in painting with deep learning. In *Asian conference on machine learning* (pp. 327-342).

Masters, D., & Luschi, C. (2018). Revisiting Small Batch Training for Deep Neural Networks.

Mohamed, I. (2017, September). An example of pooling with a 2 × 2 filter and a stride of 2. Retrieved March 17, 2019, https://www.researchgate.net/figure/An-example-of-pooling-with-a-2-2-filter-and-a-stride-of-2_fig4_324165524

National Gallery of Art. (n.d.). Retrieved February 3, 2019, from https://www.nga.gov/collection/collection-search.html

Neuron [Digital image]. (n.d.). Retrieved November 28, 2018, from https://en.wikipedia.org/wiki/File:Neuron.svg

Raschka, S. (2015, May 24). Rosenblatt Perceptron [Digital image]. Retrieved November 28, 2018, from https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html

Rojas, R. (2013). *Neural networks: a systematic introduction*. Springer Science & Business Media.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., . . . Fei-Fei, C. (2015). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision, 115(3), 211-252.

Russell, S. J., Norvig, P., & Davis, E. (2010). *Artificial intelligence: a modern approach.* 3rd ed. Upper Saddle River, NJ: Prentice Hall.

Salman, S., & Liu, X. (2019). Overfitting Mechanism and Avoidance in Deep Neural Networks.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15, 1929-1958.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Rabinovich, A. (2014). Going Deeper with Convolutions.

Turing, A. M. "Computing Machinery and Intelligence." Mind, vol. 59, no. 236, 1950, pp. 433–460.Van Noord, & Postma. (2017). Learning scale-variant and scale-invariant features for deep image classification. *Pattern Recognition, 61*(C), 583-592.

Visual Art Encyclopedia. (n.d.). Retrieved February 3, 2019, from https://www.wikiart.org/en/

Web Gallery of Art. (n.d.). Retrieved February 3, 2019, from https://www.wga.hu/

Wei Ren Tan, H., Chee Seng Chan, Aguirre, & Tanaka. (2016). Ceci n'est pas une pipe: A deep convolutional network for fine-art paintings classification. 2016 IEEE International Conference on Image Processing (ICIP), 2016, 3703-3707.

Wu, J. (2017). Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, 5-23.

Yamashita, R., Nishio, M., & Togashi, K. (2018). Convolutional neural networks: An overview and application in radiology. Insights into Imaging, 9(4), 611-629.

Yosinski, J., Clune, J., Bengio, Y., & Lipson, H. (2014). How transferable are features in deep neural networks? Advances in Neural Information Processing Systems 27, pages 3320-3328. Dec. 2014.

Zahran, M. (2015, December). Multilayer Perceptron Network [Digital image]. Retrieved November 28, 2018, from https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065