Senior Thesis in Mathematics

Style Transfer with Neural Networks

Author:
Nolan McCafferty

Advisor:
Dr. Jo Hardin

Submitted to Pomona College in Partial Fulfillment
of the Degree of Bachelor of Arts

April 30, 2020

Abstract

Neural network models are powerful tools in the world of machine learning. In this paper I attempt to peel back the layers of complexity and examine the underlying math of this class of models. I extend this examination to convolutional neural networks and how they achieve state-of-the-art performance on image classification problems. Finally, I apply convolutional neural networks to a problem called neural style transfer, which creates image combinations from different content and style images.

Contents

Chapter 1

Introduction

Machine learning is a field that combines statistics and computer science to generate predictions for all kinds of applications and data sources. One of the most popular machine learning models used today is the neural network. Neural networks are machine learning models that are loosely based on the structure of neurons in a human brain. In recent years, neural networks have gained popularity because the computational power of hardware has caught up to their complexity. They have revolutionized several areas including computer vision and natural language processing. For my thesis, I will focus on neural network models.

The majority of my thesis will be analyzing and understanding the mathematics behind neural networks. Typically, they are thought of as black boxes, meaning it is generally difficult to understand how the model reaches its final prediction. I will walk through the math behind forward propagation, backpropagation, and various model optimizers.

Convolutional neural networks (CNNs) are a subclass of neural networks that apply preprocessing techniques to extract features from patterns in data. CNNs excel at automated feature engineering, creating feature maps that can then be fed into traditional fully-connected neural network layers. This makes CNNs attractive for dealing with image data because they are able to capture the high-level characteristics of an image and make accurate predictions [17].

Finally, I will utilize these machine learning techniques in an image processing application in order to combine the style and content of two images. This technique is called neural style transfer.

Chapter 2

Mathematics of Neural Networks

2.1 Overview

Neural networks are a class of machine learning models that are able to learn incredibly complex functions. The way that these models learn is by *training* the model to make accurate predictions on a specific dataset. For supervised learning, which we will focus on, this training is done by feeding in inputs, calculating outputs, and then comparing the model output to the actual label for that given input. Once the prediction error is calculated, the error can be propagated back to adjust the parameters of the network in a process called backpropagation [1]. The end goal of a neural network model is to be able to make accurate predictions on data that the network has not seen before, generally referred to as *test* data.

Figure 2.1 shows an example of a fully-connected (dense) neural network with inputs $x_1, x_2, \ldots x_n$, two intermediate layers of sizes k_1 and k_2, and finally a single output node. The intermediate layers of a network are also referred to as *hidden* layers because they are "hidden" from the outside world inside the network. Thus, all of the layers in a network are hidden layers except for the input and output layers. The example network below has two hidden layers of sizes k_1 and k_2, but in practice neural networks can have any number of hidden layers, and typically have many, each having various non-uniform sizes.

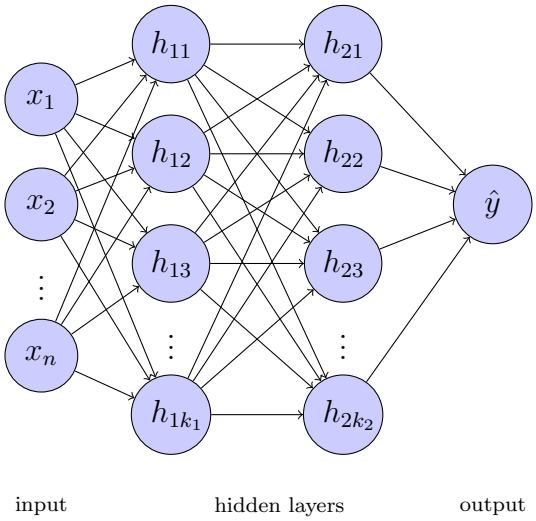


Figure 2.1: Network with 2 Hidden Layers

2.2 Perceptron/Neuron

Many consider Frank Rosenblatt's paper on what he called the "Perceptron" [11] to be the beginning of neural networks as we know them today. Rosenblatt's Perceptron models a single neuron in the human brain, and is the building block of neural networks. Figure 2.2 shows the layout of a neuron, with $x_1, x_2, \ldots x_n$ as the inputs, $w_0, w_1, \ldots w_n$ as their corresponding weights, an activation function, and an output.

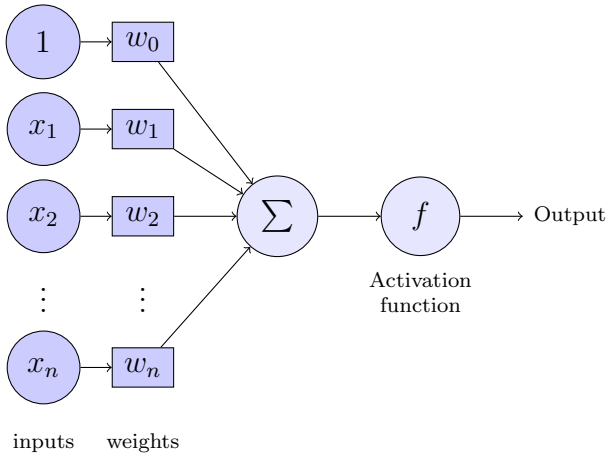


Figure 2.2: Neuron

To calculate the output of the neuron, we first find z, the linear combination of the inputs:

$$z = w_0 + \sum_{i=1}^{n} x_i w_i \tag{2.1}$$

The weight w_0 does not have a corresponding input value and is sometimes referred to as the *bias* [11]. This is useful because once we start training our network it will be nice to have an adjustable weight that does not directly coincide with a specific input. Once z is calculated, it is run through an activation function f to get a, the output of the neuron:

$$a = f(z) \tag{2.2}$$

The output of a neuron will be a single value that is then passed on as an input into the next neuron in the sequence. The single value will have

varying characteristics depending on the activation function f. For example, the activation function could be just the identity function $f(z) = z$, although this is not normally used in practice. There are many different activation functions and the next section will explore some of them further.

2.3 Activation Functions

Activation functions are used in neural networks to induce a non-linearity in order for the network to represent more complex functions. Without non-linear intermediate activation functions, it would be possible to represent the entire network as a composition of matrix multiplications [1]. For example, for a network with one hidden layer, the output of the hidden layer would be:

$$z^{[i]} = xw^{[i]} + b^{[i]} \tag{2.3}$$

where x is the input vector, $w^{[i]}$ is the weight matrix for layer i, $b^{[i]}$ is the bias for layer i, and $z^{[i]}$ is the output for layer i. Below would be the final output for the network:

$$z^{[i+1]} = z^{[i]}w^{[i+1]} + b^{[i+1]} \tag{2.4}$$

These two equations can be composed into the following equation for the entire network:

$$\begin{aligned} z^{[i+1]} &= (xw^{[i]} + b^{[i]})w^{[i+1]} + b^{[i+1]} \\ z^{[i+1]} &= xw^{[i]}w^{[i+1]} + b^{[i]}w^{[i+1]} + b^{[i+1]} \end{aligned} \tag{2.5}$$

As you can see, the network has been reduced to a simple linear combination of the inputs. Activation functions are the key to increasing the function space that neural networks can approximate.

Consider an activation function f that is applied to the output of each layer of our network. The equations for our simple network are now shown below.

$$
\begin{aligned}
z^{[i]} &= x^{[i]}w^{[i]} + b^{[i]} \\
a^{[i]} &= f(z^{[i]}) \\
z^{[i+1]} &= a^{[i]}w^{[i+1]} + b^{[i+1]} \\
a^{[i+1]} &= f(z^{[i+1]})
\end{aligned}
\tag{2.6}
$$

where $a^{[i]}$ is the output for layer i after the activation function has been applied.

The function f adds complexity to our model and allows us to predict more comprehensive functions. The next sections will analyze some of the activation functions currently used in practice.

2.3.1 ReLU

ReLU (Rectified Linear Unit) is currently the most popular activation function for the hidden layers of neural networks [1]. For a given input z, $ReLU(z)$ is defined below.

$$
ReLU(z) = \max(0, z) \tag{2.7}
$$

Thus, ReLU essentially sets all negative inputs to 0 and acts as the identity for positive inputs. As we will discuss later, this reduces the time it takes for the network to converge when adjusting the weights during backpropagation, because the loss that gets propagated back will get canceled when multiplied by 0.

2.3.2 Sigmoid

The sigmoid function is another popular activation function used to map an input to a value in the interval $(0, 1)$. The function σ is defined as follows:

$$
\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.8}
$$

One popular application of the sigmoid activation function is to map the penultimate output of a network to a probability for a binary classification problem [1].

2.3.3 Softmax

The softmax function is another popular activation function used for the output layer of a neural network. The difference between softmax and sigmoid is that softmax is used for multi-class classification problems, where the neural network must predict the class of an input from many ($K > 2$) classes [1]. For the softmax function on a single class output k_i representing class i, we first calculate e^{k_i} and then divide by $\sum_{j=1}^{K} e^{o_j}$, the sum of the rest of the classes. The softmax function ensures that we get a prediction in the interval $[0, 1]$. The complete formula is shown below:

$$softmax(k_i) = \frac{e^{k_i}}{\sum_{j=1}^{K} e^{k_j}} \tag{2.9}$$

The softmax would be evaluated similarly for the other $K-1$ classes. This concludes our discussion of activation functions. The next section will go into detail about evaluating the predictions of our network via loss functions.

2.4 Loss Functions

Loss functions are used in models throughout machine learning to determine the error in their predictions, which can be used to tune the model parameters and improve prediction accuracy. The type of loss function used depends on the specifications of the learning problem. For regression problems, where the prediction is continuous, sum of squared error (SSE) is a popular loss function given by equation 2.10 [1].

$$SSE = \sum_{i} (y_i - \hat{y}_i)^2 \tag{2.10}$$

where y_i is the ground-truth response value and $\hat{y}_i$ is the prediction from our model for observation i. From this equation we can see that the loss is minimized when the prediction is equal to the true response for a given input. A variant of SSE is typically used to evaluate the overall performance of a model, namely mean-squared error (MSE) shown below:

$$MSE = \frac{1}{n} \sum_{i}^{n} (y_i - \hat{y}_i)^2 \tag{2.11}$$

where n is the number of predictions being evaluated.

For classification problems, the loss function is slightly more complicated because the output is generally a prediction of the probability of a given label among a number of discrete possible labels. The loss function used is called *cross-entropy loss* [1]. For binary classification, where the ground truth label for observation i (y_i) has only two possibilities ($y_i = 0$ or $y_i = 1$), the cross-entropy loss is shown in equation 2.12.

$$CrossEntropyLoss = \sum_i -(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)) \tag{2.12}$$

where $\hat{y}_i$ is the model prediction for observation i.

We can see that one of the two components in the loss function will be set to zero for each observation, depending on the ground truth label, and the other component will be small if the prediction is close to the label. This formula generalizes to multi-label classification for a problem with L possible labels as follows:

$$CrossEntropyLoss = \sum_i - \sum_l^L y_i^{(l)} \log \hat{y_i}^{(l)} \tag{2.13}$$

where $y_i^{(l)}$ is the ground-truth for observation i for label l and $\hat{y_i}^{(l)}$ is the model prediction for observation i for label l.

The next section will combine the knowledge of loss functions with the neural networks discussed above in the process of learning the network parameters: backpropagation.

2.5 Backpropagation

Backpropagation in a neural network is the process of adjusting the parameters of the network (weights) in order to decrease the loss (prediction error). This procedure is the key to the success of neural networks in learning complex functions [1]. Backpropagation starts at the end of the network, the output, and works all the way back to the beginning of the network, updating the respective weights as they are encountered through an optimization technique called *gradient descent*.

Using gradient descent to update weight w_i involves the derivative of the loss function with respect to w_i: $\frac{\partial L}{\partial w_i}$. This derivative describes the change in the loss function with respect to the change in weight w_i. Thus, if the derivative is positive, the loss will increase as w_i increases and if the derivative is negative the loss will decrease as w_i increases. Since the goal of training our network is to minimize the loss, w_i must be updated to ensure the loss will decrease [12]. Therefore, by subtracting this derivative from w_i, we can be certain that the loss will decrease:

$$w_i' = w_i - \frac{\partial L}{\partial w_i} \tag{2.14}$$

However, equation 2.14 is only partially complete. To complete the update step for w_i, we introduce a *hyperparameter* λ. A hyperparameter is a meta-parameter for our network that is determined by the programmer before training the model. In practice, the programmer will use some sort of optimization technique to choose the hyperparameters that give the best model performance [12]. λ represents the *learning rate* of the model, which determines how fast or how slow the model converges to the optimal parameters. High values of λ correspond to larger jumps in the value of w_i, which causes the network to learn faster. Equation 2.15 shows the updated version of the learning step.

$$w_i' = w_i - \lambda \frac{\partial L}{\partial w_i} \tag{2.15}$$

At the point, the major question left to be answered is: how do we find $\frac{\partial L}{\partial w_i}$? This is where the *chain rule* from calculus comes into play. As a quick refresher, given $F = f \circ g = f(g(x))$, the chain rule is shown in equation 2.16.

$$F'(x) = f'(g(x)) \cdot g'(x) \tag{2.16}$$

The chain rule lets us propagate the value given by the loss function back through each node and weight in the network, from the final output all the way to the original layer. For each weight w_i, the chained derivatives must represent all the possible paths from w_i to the output of the model [1].

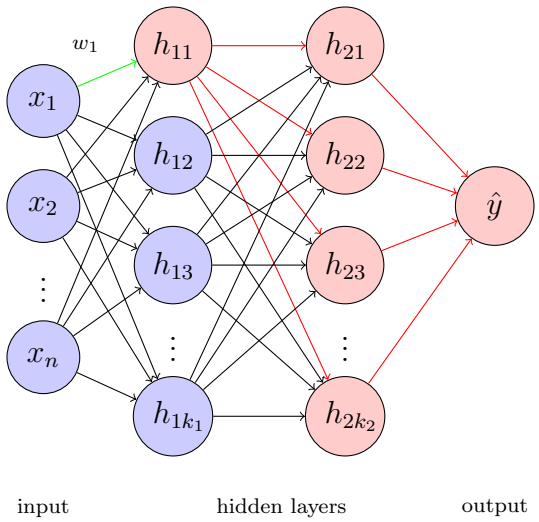


Figure 2.3: Paths from w_1 to the model output.

Figure 2.3 shows an example of the paths from w_1, the weight connecting x_1 and h_{11} in the network. w_1 is highlighted in green and the possible paths from w_1 to the output of the network are highlighted in red. To calculate $\frac{\partial L}{\partial w_1}$ for the above example, we trace each path using the chain rule:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} \tag{2.17}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \left(\frac{\partial \hat{y}}{\partial h_{21}} \cdot \frac{\partial h_{21}}{\partial w_1} + \cdots + \frac{\partial \hat{y}}{\partial h_{2k_2}} \cdot \frac{\partial h_{2k_2}}{\partial w_1} \right) \tag{2.18}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \left(\frac{\partial \hat{y}}{\partial h_{21}} \cdot \frac{\partial h_{21}}{\partial h_{11}} \cdot \frac{\partial h_{11}}{\partial w_1} + \cdots + \frac{\partial \hat{y}}{\partial h_{2k_2}} \cdot \frac{\partial h_{2k_2}}{\partial h_{11}} \cdot \frac{\partial h_{11}}{\partial w_1} \right) \tag{2.19}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \left(\frac{\partial \hat{y}}{\partial h_{21}} \cdot \frac{\partial h_{21}}{\partial h_{11}} + \cdots + \frac{\partial \hat{y}}{\partial h_{2k_2}} \cdot \frac{\partial h_{2k_2}}{\partial h_{11}} \right) \frac{\partial h_{11}}{\partial w_1} \tag{2.20}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \left(\frac{\partial \hat{y}}{\partial h_{21}} \cdot \frac{\partial h_{21}}{\partial h_{11}} + \cdots + \frac{\partial \hat{y}}{\partial h_{2k_2}} \cdot \frac{\partial h_{2k_2}}{\partial h_{11}} \right) x_1 \tag{2.21}$$

In the above equations, the $\frac{\partial L}{\partial \hat{y}}$ represents the derivative of the loss function with respect to $\hat{y}$. The terms inside the parenthesis in equation 2.21 represent the weights from the second hidden layer to the output and between node h_{11} and the second hidden layer. These terms also have to take

the activation functions at each node into account. As expected, the above equations get much more complex as the given w_i gets farther from the output of the network and the number of paths grows.

In practice, backpropagation is computed using a *dynamic programming* approach, which means that the partial derivative of the loss function (with respect to the given node) is saved at each node on the way back through the network to save computation time [1].

The process of backpropagation, similar to forward propagation, is also done using matrices to speed up computation. Thus, instead of specifically updating w_1 as shown above, the algorithm would update the entire first weight matrix $w^{[1]}$ in one step. This process is shown in equation 2.22, keeping in mind that we are starting at the end of the network and working our way back, and we are now taking into account activation function $g^{[i]}$ for layer i.

$$\frac{\partial L}{\partial a^{[2]}} = \frac{\partial L}{\partial \hat{y}} \tag{2.22}$$

$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial a^{[2]}} \odot (g^{[2]})'(z^{[2]}) \tag{2.23}$$

$$\frac{\partial L}{\partial a^{[1]}} = \frac{\partial L}{\partial z^{[2]}} (w^{[2]})^T \tag{2.24}$$

$$\frac{\partial L}{\partial z^{[1]}} = \frac{\partial L}{\partial a^{[1]}} \odot (g^{[1]})'(z^{[1]}) \tag{2.25}$$

$$\frac{\partial L}{\partial w^{[1]}} = x^T \frac{\partial L}{\partial z^{[1]}} \tag{2.26}$$

2.6 Optimizers

The backpropagation section discussed using a technique called gradient descent to update the parameters of a network. In this section, we will review additional algorithms to optimize gradient descent.

2.6.1 Momentum

Momentum solves the problem of large variation in the gradient of the loss function from one observation to another [12]. Figure 2.4 shows a loss contour, with the arrows representing the paths taken by two optimization

algorithms. The red path in Figure 2.4 depicts an example of one such inefficient gradient path, while the blue path shows a much more efficient path.

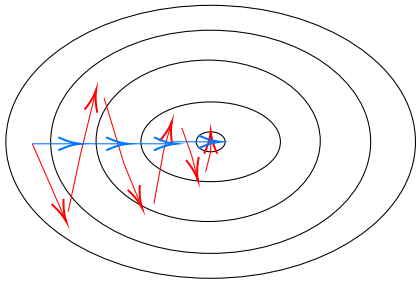


Figure 2.4: Example of oscillating gradient (red) and path we would prefer (blue).

By preventing wild oscillations in the gradient, the parameters will find a smoother, faster path to their optima (shown in blue in Figure 2.4). A more efficient path is achieved by incorporating the previous gradient, from the observations already seen, into the calculation for the new gradient.

If we let t denote the number of times this optimization algorithm has been run, then momentum incorporates the gradient at time $t-1$ into the calculation for the gradient at time t.

Thus, equation 2.15 becomes:

$$w_{i,t} = w_{i,t-1} - V(t) \tag{2.27}$$

where $V(t)$ includes the current gradient and the previous gradient multiplied by a decay factor γ. $V(t)$ is defined as follows.

$$V(t) = \gamma V(t-1) + \lambda \frac{\partial L}{\partial w_{i,t-1}} \tag{2.28}$$

$t = 1$ represents the first time the optimization is run, which is after the first training observation (or batch of observations) has gone through forward propagation. There is no previous gradient at $t = 1$ so we must have $V(0) = 0$.

Equation 2.28 now incorporates the current gradient ($\frac{\partial L}{\partial w_{i,t-1}}$), the previous gradient (included in $V(t-1)$), and a hyperparameter γ that determines how much weight the previous gradient deserves (normally 0.9) [12].

2.6.2 Adagrad

Adagrad is a gradient descent extension that incorporates an adjustable learning rate for each parameter. Thus, the algorithm is able to perform smaller updates (low learning rate) for parameters associated with frequently occurring features and larger updates (high learning rate) for parameters associated with infrequent features [12]. Thus, equation 2.15 is extended to equation 2.29.

$$w_{i,t} = w_{i,t-1} - \frac{\lambda}{\sqrt{G_{ii,t-1} + \epsilon}} \cdot \frac{\partial L}{\partial w_{i,t-1}} \tag{2.29}$$

where G_{t-1} is a diagonal matrix where each element i, i is the sum of the squares of the gradients with respect to parameter i up to time $(t-1)$ and ϵ is a smoothing term that ensures no division by zero.

Since G_{t-1} contains the sum of the squares of the past gradients with respect to all parameters along its diagonal, we can vectorize our optimization using element-wise matrix multiplication, called a Hadamard product, denoted with $\odot$.

$$\Theta_t = \Theta_{t-1} - \frac{\lambda}{\sqrt{G_{t-1} + \epsilon}} \odot \frac{\partial L}{\partial \Theta_{t-1}} \tag{2.30}$$

where Θ denotes all the weights and biases of our network.

2.6.3 Adam

Adaptive moment estimation (Adam) combines the concept of momentum and the adaptive learning rates for each parameter from the previous two methods. Whereas momentum can be thought of as a ball running down a slope, Adam has been described as behaving like a heavy ball with friction [6].

First, compute the decaying averages of the past and past squared gradients m_t and v_t as follows:

$$m_t = \beta_1 m_{t-1} - (1 - \beta_1) \frac{\partial L}{\partial \Theta_{i,t}} \tag{2.31}$$

$$v_t = \beta_2 v_{t-1} - (1 - \beta_2) \frac{\partial L^2}{\partial \Theta_{i,t}} \tag{2.32}$$

m_t is the first moment (mean) and v_t is the second moment (variance) of the gradient, with hyperparameters β_1 and β_2 as rates of decay. Default values of 0.9 and 0.999 for β_1 and β_2 respectively perform very well in practice and are rarely ever changed [2].

Then, since m_t and v_t are initialized as zeros, they will be biased towards zero, therefore they must be bias-corrected to $\hat{m}_t$ and $\hat{v}_t$ in equations 2.33 and 2.34.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} \tag{2.33}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{2.34}$$

Finally, the update step is similar to what we have seen with the optimizers above:

$$\Theta_{i,t+1} = \Theta_{i,t} - \frac{\lambda}{\sqrt{\hat{v}_t + \epsilon}} \cdot \hat{m}_t \tag{2.35}$$

Adam combines the previous two optimization methods and is a popular optimization method used in practice to update the weights in a neural network.

Chapter 3

Convolutional Neural Networks

3.1 Motivation

Convolutional neural networks (CNN) are a class of neural networks that are most commonly applied to visual image data. The mathematical fundamentals of CNNs have been around since the middle of the 20^{th} century, similar to the models discussed above, however their effectiveness with visual images has become much more pronounced with the increased capabilities of hardware in the last decade [1].

The structural difference between convolutional neural networks and the networks previously discussed is that CNNs have a built in preprocessing section before the fully-connected layers. An example of this preprocessing section is shown in Figure 3.1, followed by two dense layers at the end of the network [15]. The beginning section of the network is made up of *convolutional* layers and *pooling* layers.

The role of a convolutional layer is to extract features from the data by training filters. These filters, once learned, allow higher level features of the input to be captured and ultimately used by the dense layers to make a prediction.

The pooling layers are much simpler, as they just aggregate the data from the previous layer in a specified way. Pooling is used to reduce variance and computational complexity in a CNN.

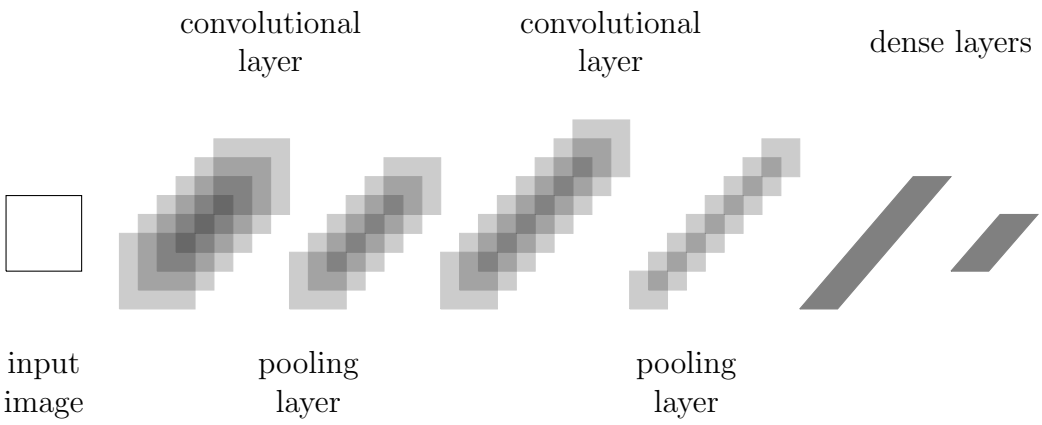


Figure 3.1: Example CNN Architecture

CNNs were inspired by the biological processes that make up the visual cortex in many animals. The connections between neurons in a convolutional network resemble the patterns found in the process that humans use to recognize images. Thus, it is not surprising that CNNs have shown the most promise in the field of visual imagery. In humans, individual cortical neurons only respond to stimuli in a limited region of the visual field called the *receptive field* [17]. The receptive fields of various neurons partly overlap so that the entire visual field is covered by the complete set of neurons.

Convolutional neural networks also have the concept of a receptive field. Once the image has been processed by the convolutional and pooling layers, each element of the output has a different receptive field of pixels that it represents in the original image. By increasing the receptive field of an element in the output, the network can represent higher level features by relating pixels that were not necessarily near each other in the original image.

3.2 Mathematics

3.2.1 Convolution

Suppose the input to the convolution is a n $\times$ m image I, which can be represented by an array of size n $\times$ m. Thus, $I_{i,j}$ represents the element at the i^{th} row and j^{th} column of I. Given a filter $K \in \mathbb{R}^{2k_1+1 \times 2k_2+1}$, where

$$K = \begin{pmatrix} K_{-k_1,-k_2} & \dots & K_{-k_1,k_2} \\ \vdots & K_{0,0} & \vdots \\ K_{k_1,-k_2} & \dots & K_{k_1,k_2} \end{pmatrix} \tag{3.1}$$

the convolution of the image I with the filter K is given by

$$(I * K)_{i,j} = \sum_{u=-k_1}^{k_1} \sum_{v=-k_2}^{k_2} K_{u,v} I_{i+u,j+v} \tag{3.2}$$

It is important to consider that the convolutional behavior shown in the above equation needs to be specifically defined for the borders of the image [1].

3.2.2 Layers

Convolutional Layer

Consider a convolutional layer l. The input of layer l consists of $m^{[l-1]}$ feature maps from the previous layer, each of size $n_1^{[l-1]} \times n_2^{[l-1]}$. When $l = 1$, the input is a single image I which is the original input to the network. The output of layer l includes $m^{[l]}$ feature maps of size $n_1^{[l]} \times n_2^{[l]}$. The equation below shows how to compute the i^{th} feature map in layer l, denoted Z_i

$$Z_i^{[l]} = B_i^{[l]} + \sum_{j=1}^{m^{[l-1]}} K_{i,j}^{[l]} * Z_j^{[l-1]} \tag{3.3}$$

where $B_i^{[l]}$ is the bias matrix for layer l, similar to the bias in a dense neural network layer. $K_{i,j}^{[l]}$ is the filter connecting the j^{th} feature map in layer $(l-1)$ with the i^{th} feature map in layer l [1].

The center of the filter can either start outside the border of the image, on the border of the image, or inside the border of the image. The term for this distinction is *padding* (as in padding the outside of the image with zeros in order to extend the filter). Full padding refers to starting the filter outside the border of the image, half padding to start the center of the filter on the image border, and no padding for keeping the filter entirely inside the input image.

Another parameter used in convolution is *stride* [1]. Stride refers to the number of pixels the filter is shifted over each time. When the stride is 1 the

filter moves 1 pixel at a time. When the stride is 2 the filter moves 2 pixels at a time and so on. Thus, if the stride is greater than 1, the dimensions of the output image will be smaller than the dimensions of the input image. Changing the padding can also affect the size of the output image.

Assuming that we are using half padding, a stride of s, and the filter $K_{i,j}^{[l]}$ has size $2k_1 + 1 \times 2k_2 + 1$, the size of the output feature maps of layer l is given by

$$n_1^{[l]} = \frac{n_1^{[l-1]} - 2k_1^{[1]}}{s^{[1]} + 1} \quad \text{and} \quad n_2^{[l]} = \frac{n_2^{[l-1]} - 2k_2^{[1]}}{s^{[1]} + 1} \tag{3.4}$$

The tunable parameters in the convolutional layer l are the weights of the filters $K_{i,j}^{[l]}$ and bias matrix $B_i^{[l]}$ [1].

Pooling Layer

Let l be a pooling layer. The output of l contains the same number of feature maps as the input, $m^{[l]} = m^{[l-1]}$, but they are of reduced size. Pooling, also called subsampling, typically involves placing non-overlapping windows on the feature maps and keeping one value per window.

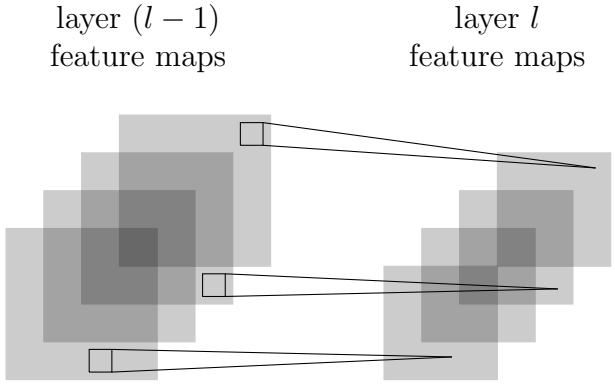


Figure 3.2: Illustration of a pooling layer.

There are two main types of pooling: *average pooling* and *max pooling*.

In average pooling, the mean of the values in each window is used in the output. This method generally smooths out the image and is less effective for identifying sharp features.

For max pooling, the maximum value in each window is taken. In practice, max pooling has been shown to produce faster convergence during training [1].

Chapter 4

Neural Style Transfer

4.1 Overview

Neural style transfer is a technique that creates new images by combining the content of a reference image and the style of a separate reference image. The algorithm was first introduced by Gatys et al. in 2015 [3]. They showed that the content and style of an image can be represented by using different intermediate feature maps from CNN layers. Then, a combination image can be optimized to incorporate the style of the style reference image and the content of the content image.

The inputs to our algorithm are the content image, the reference style image, and a few hyperparameters to control the weighting of content vs. style in the output image. The algorithm output will be the combination image with the content and style of the input images respectively.

In order to arrive at a combination image with these desired properties, the algorithm starts with a copy of the content image, called the *input* image, and optimizes this new image based on a loss function [3]. The loss function incorporates both the content loss and the style loss to ensure that both properties are present in the final optimized image.

A high level outline of this algorithm is as follows:

1. Identify the content and style reference images.
2. Construct the loss function.
3. Run the reference images through the CNN to get values for the loss function.

4. Optimize the input image to minimize loss.

Since the first step is trivial, the next section will focus on constructing the loss function.

4.2 Loss Function

As stated above, the loss function is a combination of the content loss and the style loss of the input image. Each of these elements has an associated weight which can be adjusted based on the ratio of content vs. style desired in the final combination image [3]. Thus, our loss function can be written as:

$$L = w_{content} L_{content} + w_{style} L_{style} \tag{4.1}$$

where $w_{content}$ and w_{style} are the content and style weights respectively.

4.2.1 The CNN

In order to calculate the loss of our input image, we are going to run both it and the reference images through a trained convolutional neural network. A "trained" CNN means that the network has already learned how to categorize images, meaning that it is very good at answering questions like: "which of these five categories does this image belong to: dog, cat, horse, mouse, or bear?". A network learns to make these kinds of predictions by being trained on thousands of labeled images and adjusting its weights through backpropagation.

Luckily for us, there exist pre-trained CNNs that are publicly available and very good at predictions. One of the most popular of these networks is called `VGG-19` (for its 19 layers) [14]. This is the model we will use in our algorithm. Below is the summary of the "head" of the network, which refers to the convolutional and pooling layers that come before the fully-connected section of the network. This figure will be helpful in our discussion of content and style loss.

It is important to note that for our application we will not use the final output of the CNN. Instead we will utilize the intermediate feature maps that are output from various convolutional layers in the head of the network because our goal is to change the *input* image, not the weights of the network (which is the typical goal in training a CNN) [3].

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 300, 300, 3)	0
block1_conv1 (Conv2D)	(None, 300, 300, 64)	1792
block1_conv2 (Conv2D)	(None, 300, 300, 64)	36928
block1_pool (MaxPooling2D)	(None, 150, 150, 64)	0
block2_conv1 (Conv2D)	(None, 150, 150, 128)	73856
block2_conv2 (Conv2D)	(None, 150, 150, 128)	147584
block2_pool (MaxPooling2D)	(None, 75, 75, 128)	0
block3_conv1 (Conv2D)	(None, 75, 75, 256)	295168
block3_conv2 (Conv2D)	(None, 75, 75, 256)	590080
block3_conv3 (Conv2D)	(None, 75, 75, 256)	590080
block3_pool (MaxPooling2D)	(None, 37, 37, 256)	0
block4_conv1 (Conv2D)	(None, 37, 37, 512)	1180160
block4_conv2 (Conv2D)	(None, 37, 37, 512)	2359808
block4_conv3 (Conv2D)	(None, 37, 37, 512)	2359808
block4_pool (MaxPooling2D)	(None, 18, 18, 512)	0
block5_conv1 (Conv2D)	(None, 18, 18, 512)	2359808
block5_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block5_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block5_pool (MaxPooling2D)	(None, 9, 9, 512)	0

Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0

Figure 4.1: Summary of VGG-19 model head [9]

4.2.2 Content Loss

The content of an image consists of its high level features. These could be people, buildings, or any objects prominent in the image. For example, in a grayscale image of a house, the house is the content of the image. If color

was added to the image, the house would remain the content.

In a convolutional neural network, as you propagate forwards, farther from the starting layer, the feature maps of the input image become higher and higher level. Since image content is composed of high level features, we want to use the feature map of the final convolutional layer to represent the content of an image [17].

In the VGG-19 model, shown in Figure 4.1, the last convolutional layer is `block5_conv3`. The output will be a matrix $F \in \mathbb{R}^{n \times m}$, where n is the number of feature maps each of size m (which is the height times the width of the feature map). F_{ij} corresponds to the i^{th} feature at position j. Thus, for content loss, we will take the squared difference between the output of this layer for the input image and the content reference image:

$$L_{content} = \sum_{i,j} (F_{ij,ref} - F_{ij,input})^2 \tag{4.2}$$

where F_{ref} and F_{input} are the outputs of layer `block5_conv3` for the content reference image and input image respectively [3].

4.2.3 Style Loss

On the other hand, the style of an image is made up of both low and high level features. There are many aspects of image style including the color palette and brush stroke size. Thus, instead of using the feature map of the final convolutional layer of our CNN, we use several layer outputs evenly spaced throughout the CNN to include both low and high level style features [3]. The layers used for style loss are `block1_conv1`, `block2_conv1`, `block3_conv1`, `block4_conv1`, and `block5_conv1` (refer to Figure 4.1).

Additionally, it is important to examine the correlation between different filter responses in a given feature map in order to capture the texture information of the input image but not the global arrangement. This can be done by computing the Gram matrix between all pairs of features in each feature map. The Gram matrix G^l_{ij} is the inner product between the vectorized feature maps i and j in layer l, shown in Equation 4.3. G_{ij} represents the correlation between feature maps i and j [3].

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk} \tag{4.3}$$

The style loss from layer l is given by the difference between the Gram matrices for the style reference image G_{ref} and the input image G_{input} for every pair of feature maps:

$$S^l = \sum_{i,j} (G^l_{ij,ref} - G^l_{ij,input})^2 \tag{4.4}$$

To calculate the overall style loss we take the average style loss from all the style layers used.

$$L_{style} = \frac{1}{L} \sum_{l}^{L} S^l \tag{4.5}$$

where L is the number of style layers used, which is five in our case [3].

4.3 Optimization

Now that the loss function has been defined, there needs to be a way to optimize the input image based on the given loss. The key to this optimization is backpropagation. Backpropagation in convolutional neural networks is very similar to the backpropagation through fully-connected neural networks described in chapter 2. The concept of propagating error back through the weights of the network via the chain rule remains unchanged.

However, for this application the actual pixel values of the input image will be adjusted instead of the weights of the network. The parameters of the network will be "frozen" so that they will not change during the backpropagation. Instead, each iteration of backpropagation will adjust the pixel values of the image based on an optimizer [3].

I chose to use the Adam optimizer, described in chapter 2, in order to adjust the input image. Adam works very well in practice and produces satisfying style content combinations in under a minute of optimization.

Psuedocode for the algorithm is given below:

Algorithm 1 Neural Style Transfer

Start with a copy of the content reference image, call this *input* image
Run content and style reference images through VGG-19 model
for num_epochs **do**
  Run input image through VGG-19
  Calculate L using intermediate feature maps
  Backpropagate error through VGG-19 and adjust pixel values of input image
end for

4.4 Results

Below are examples of the neural style transfer algorithm output with various content and style reference images. These example were created from my neural style transfer implementation in the Python programming language using the `TensorFlow` framework. The code can be found in a repository on my personal GitHub page [9]. The images used in these examples are all available in the Wikimedia Commons.

Content reference image [13]

Style reference image [16]

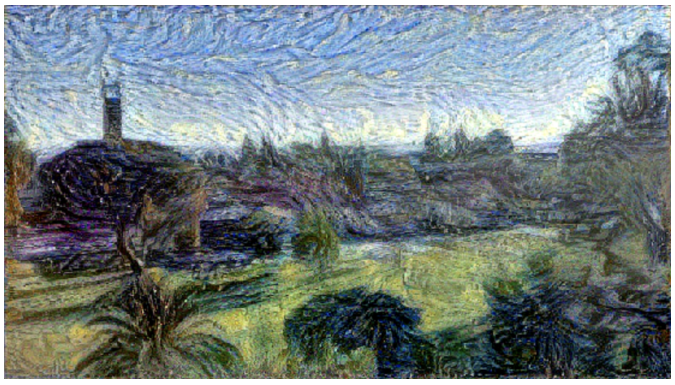


Figure 4.2: Neural Style Transfer Walker Dorm Example

Content reference image [10]

Style reference image [7]

Figure 4.3: Neural Style Transfer Beagles Example

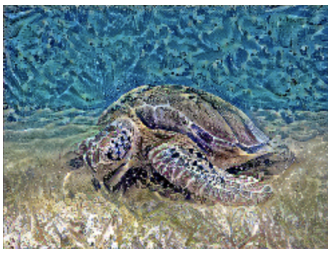
Below is an example of how varying the style weight in the loss function can change the output of the algorithm to incorporate different amounts of the style representation. As the figure shows, as the style weight increases so does the amount of "abstractness" from the style reference image. The content weight is kept constant at $w_{content} = 100$ for this analysis.

Content reference image [8]

Style reference image [5]

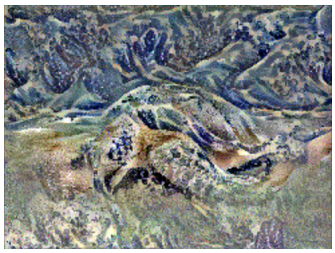
$w_{style} = 1 \times 10^{-7}$

$w_{style} = 1 \times 10^{-6}$

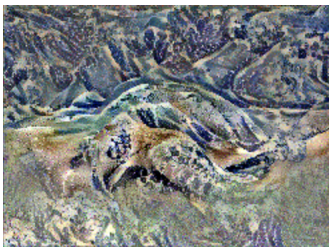


$w_{style} = 1 \times 10^{-5}$

$w_{style} = 1 \times 10^{-4}$



$w_{style} = 1 \times 10^{-3}$



$w_{style} = 1 \times 10^{-2}$

Figure 4.4: Neural Style Transfer Turtle Example

Chapter 5

Conclusion

In this analysis, we were able to utilize the inner structure of convolutional neural networks in an algorithm to synthesize the content and style of two reference images. This process began with opening up the black box of neural networks and examining how these models are able to learn patterns in data. Chapter 2 dove into the details of neural networks and explained their many complexities including activation functions, backpropagation, and optimizers. Then our focus shifted to convolutional neural networks, a specialized class of model that performs well with image data. CNNs incorporate additional data preprocessing in the form of convolutional and pooling layers at the front of the network. Finally, chapter 4 detailed our application: neural style transfer. NST takes advantage of the intermediate layers of a convolutional neural network to capture style and content representations of two images and then uses backpropagation to optimize a combined image. Using this technique we were able to create abstract works of art using the mathematics of neural networks as our paintbrush.

Bibliography

[1] Charu Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, 2018.

[2] Vitaly Bushaev. Adam — latest trends in deep learning optimization. *Towards Data Science*, 2018.

[3] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015.

[4] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Günter Klambauer, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a nash equilibrium. *CoRR*, abs/1706.08500, 2017.

[5] Katsushika Hokusai. The great wave off kanagawa, circa 1930.

[6] Yongcheng Jing, Yezhou Yang, Zunlei Feng, Jingwen Ye, and Mingli Song. Neural style transfer: A review. *CoRR*, abs/1705.04058, 2017.

[7] Wassily Kandinsky. Wassily kandinsky: Composition vii, 1913.

[8] P. Lindgren. Green sea turtle grazing seagrass, 2013.

[9] Nolan McCafferty. Nueral style transfer, 2020.

[10] Robertmugabe1. Beagle dogs, 2019.

[11] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *The Phycological Review*, 65(6):386 – 408, 1958.

[12] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[13] Sdkb. Pomona college walker beach, 2015.

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[15] David Stutz. latex-resources, 2016.

[16] Vincent van Gogh. The starry night, 1889.

[17] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *Computing Research Repository*, abs/1311.2901, 2013.

[18] Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. Recent advances in convolutional neural network acceleration. *Elsevier*, 2018.