Senior Thesis in Mathematics

Artistic Style Transfer using Deep Learning

Author:
Chris Barnes

Advisor:
Dr. Jo Hardin

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

April 7, 2018

Abstract

This paper details the process of creating pastiches using deep neural networks. Pastiches are works of art that imitate the style of another artist of work of art. This process, known as style transfer, requires the mathematical separation of content and style. I will begin by giving an overview of neural networks and convolution, two of the building blocks used in artistic style transfer. Next, I will give an overview of current methods, such as those used in the Johnson et al. (2016) and Gatys et al. (2015a) papers. Finally, I will describe several modifications that can be made to create subjectively higher quality pastiches.

Contents

Chapter 1

Background

Deep neural networks are the basis of style transfer algorithms. In this chapter, we will explore the theoretical background of deep learning as well as the training process of simple neural networks. Next, we will discuss convolutional neural networks, an adaptation used for image classification and processing. Finally, we will introduce the idea of transfer learning, or how knowledge from one problem can be applied to a different problem.

1.1 Neural Network Basics

Deep learning can be characterized as a function approximation process. Given a function $y = f^*(x)$, our goal is to find an approximation $f(x;\theta)$ parameterized by θ such that $f^*(x) \approx f(x;\theta)$ for all values of x in the domain.

Approximations are used in cases where the original function $f^*(x)$ is either computationally expensive to compute or impossible to write down. For example, consider the mapping between online images and a binary variable identifying if the image contains a cat. This function cannot be written down. Instead, we must create a function approximator in order to use the function.

Function approximators can be created by composing several functions. For example, given $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$, we can create a 3-layer neural network $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ where $f^{(1)}$ is the input layer, $f^{(2)}$ is the hidden layer, and $f^{(3)}$ is the output layer. This is known as a multilayer feedforward network.

The functions $f^{(i)}$ take the form $f^{(i)} = \phi(x^T w_i + b_i)$, where ϕ is a non-linear

activation function, w_i is a weight matrix, and b_i is a bias term. The choice of $f^{(i)}(x)$ was loosely guided by neuroscience to represent the activation of a neuron (Goodfellow et al., 2016).

This architecture has a range of benefits. Firstly, multilayer feedforward networks are universal function approximators. That is, given a continuous, bounded, and nonconstant activation function ϕ and an arbitrary compact set $X \in \mathbb{R}^k$, a multilayer feedforward network can approximate any continuous function on X arbitrarily well with respect to some metric ρ (Hornik, 1991).

Secondly, this architecture allows for iterative, gradient-based optimization. The property of universal function approximation is only useful if accurate approximations can be found in a timely and reasonable manner. Gradient based learning allows us to find a function approximator given an optimization procedure and a cost function.

Unfortunately, the nonlinearity of the activation function ϕ means that most cost functions are nonconvex (Goodfellow et al., 2016). Thus, there is no guarantee of convergence to a global optimum. While this makes training deep neural networks more difficult than training linear models, it is not an intractable problem. Kawaguchi (2016) showed that every local minimum of the squared loss function of a deep neural network is a global minimum. This means there are no "bad" local minima on the loss surface. However, there are "bad" saddle points where the Hessian has no negative eigenvalues. This can result in slow training times, where the model gets stuck on a high error plateau. While recent papers have addressed these training plateaus using more advanced optimization techniques, gradient descent remains the most common method to train neural networks.

1.2 Training Neural Networks

Before we begin training a neural network, we must create a metric that determines how well (or poorly) the network is performing at the approximation task. One simple example is the mean squared error function which compares the output of the neural network to the desired value.

Once we select the error function, also known as the loss or cost function, we will change the weights of the neural network in order to minimize the cost function. This is known as backpropagation. For each example or set of examples, we will first input them into the neural network. Next, we will calculate the total loss across all samples. Thirdly, we will calculate

the derivative of the total loss with respect to each weight in the network. Finally, we will update the weights using the derivatives.

The architecture of the network requires using the chain rule to calculate derivatives with respect to the weights and biases. To gain intuition into how the backpropagation algorithm works and how the chain rule is employed, I will begin with a simple numerical example given by Mazur (2015).

Consider the simple neural network with two input features, a hidden layer with two neurons, and an output layer of two outputs depicted in Figure 1.1. Each layer computes a linear combination of inputs followed by a non-linear activation function applied elementwise. We use b_i to refer to the bias term in layer i and w_n to refer to the nth weight term. In this case, w_n is a scalar.

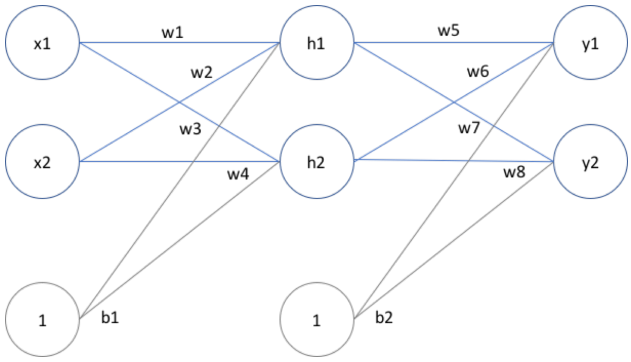

Figure 1.1: Neural Network

1.2.1 Initialization

First, we must initialize the network. For this simple example, we will initialize the weights and biases simply by incrementing by 0.05. In practice, the weights are usually randomly initialized.

$$w_1 = 0.15 \quad w_2 = 0.20 \quad w_3 = 0.25 \quad w_4 = 0.30$$
$$w_5 = 0.40 \quad w_6 = 0.45 \quad w_7 = 0.50 \quad w_8 = 0.55$$

$$b_1 = 0.35 \quad b_2 = 0.60$$

Suppose we are given the following data. We want to train a neural network to approximate some function $f^*(x_1, x_2)$ that returns two numbers y_1 and y_2.

$$x_1 = 0.05 \quad x_2 = 0.10$$
$$y_1 = 0.01 \quad y_2 = 0.99$$

1.2.2 Forward Pass

Now that our network is initialized, we will examine its accuracy. By computing a forward pass, we can determine how much error the model has given the current weights. First, let's calculate the value of each hidden neuron. We define z_{h_1} to be the linear combination of the inputs of h_1:

$$\begin{aligned} z_{h_1} &= w_1 x_1 + w_2 x_2 + b_1 \\ &= 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 \\ &= 0.3775 \end{aligned}$$

Next, we apply an activation function in order to generate non-linearities. In this case, we will use the sigmoid function. Thus, we define

$$\begin{aligned} a_{h_1} = \sigma(z_{h_1}) &= \frac{1}{1 + e^{-z_{h_1}}} \\ &= 0.593269992 \end{aligned}$$

Similarly, we get $a_{h_2} = 0.596884378$ using the same process. Now that we have computed the activations in the hidden layer, we continue (propagate) this process forward to the output layer.

$$\begin{aligned} z_{y_1} &= w_5 a_{h_1} + w_6 a_{h_2} + b_2 \\ &= 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 \\ &= 1.105905967 \end{aligned}$$

Next, we apply the activation function.

$$\hat{y}_1 = a_{h_1} = \sigma(z_{y_1}) \qquad = 0.75136507$$

We apply the same process to get $\hat{y}_2 = 0.772928465$.

Now we will calculate the total error. For this example, we will use the sum of the squared error (multiplied by a constant of $\frac{1}{2}$ in order to simplify derivatives later on).

$$\begin{aligned} E_{total} &= \sum_i \frac{1}{2}(y_i - \hat{y}_i)^2 \\ &= \frac{1}{2}(y_1 - \hat{y}_1)^2 + \frac{1}{2}(y_2 - \hat{y}_2)^2 \\ &= \frac{1}{2}(0.01 - 0.75136507)^2 + \frac{1}{2}(0.99 - 0.772928465)^2 \\ &= 0.298371109 \end{aligned}$$

1.2.3 Backwards Pass

We have now computed the total error of the model based on our initialized weights. To improve this error, we will calculate the derivative of the loss function with respect to each weight. We can use this derivative to improve the error later on.

Consider w_5. Using the chain rule, we find

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial \hat{y}_1} * \frac{\partial \hat{y}_1}{\partial z_{y_1}} * \frac{\partial z_{y_1}}{\partial w_5}$$

Now we have an equation for the derivative in much more manageable terms. We know that

$$E_{total} = \frac{1}{2}(y_1 - \hat{y}_1)^2 + \frac{1}{2}(y_2 - \hat{y}_2)^2$$

Taking the derivative, we find

$$\begin{aligned} \frac{\partial E_{total}}{\partial \hat{y}_1} &= -(y_1 - \hat{y}_1) && (1.1) \\ &= -(0.01 - 0.75136507) && (1.2) \\ &= 0.74136507 && (1.3) \end{aligned}$$

Next, we calculate the derivative of $\hat{y}_1$ with respect to z_{y_1}, the linear combination. In other words, we calculate the derivative of the activation function. In this case, $\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$. Using this, we get

$$\frac{\partial \hat{y_1}}{\partial z_{y_1}} = \hat{y_1}(1 - \hat{y_1}) = 0.186815602 \tag{1.4}$$

Lastly, we calculate the final derivative. Since $z_{y_1} = w_5 a_{h_1} + w_6 a_{h_2} + b_2$,

$$\frac{\partial z_{h_1}}{w_5} = a_{h_1} = 0.593269992$$

Combining these three derivatives, we get

$$\begin{aligned}
\frac{\partial E_{total}}{\partial w_5} &= \frac{\partial E_{total}}{\partial \hat{y_1}} * \frac{\partial \hat{y_1}}{\partial z_{y_1}} * \frac{\partial z_{y_1}}{\partial w_5} \\
&= 0.74136507 * 0.186815602 * 0.593269992 \\
&= 0.082167041
\end{aligned}$$

We can then apply the same method to w_6, w_7, and w_8. Next, we continue to propagate the errors back through the hidden layer using the chain rule. Using the same approach, we will calculate

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial a_{h_1}} \frac{\partial a_{h_1}}{\partial z_{h_1}} \frac{\partial z_{h_1}}{\partial w_1} \tag{1.5}$$

We will break equation 1.5 into three parts. First, we will focus on the derivative of the total error with respect to the activation at the hidden layer. We can expand this term as follows:

$$\begin{aligned}
\frac{\partial E_{total}}{\partial a_{h_1}} &= \frac{\partial E_{y_1}}{\partial a_{h_1}} + \frac{\partial E_{y_2}}{\partial a_{h_1}} \\
&= \frac{\partial E_{y_1}}{\partial \hat{y_1}} \frac{\partial \hat{y_1}}{\partial z_{y_1}} \frac{\partial z_{y_1}}{\partial a_{h_1}} + \frac{\partial E_{y_2}}{\partial \hat{y_2}} \frac{\partial \hat{y_2}}{\partial z_{y_2}} \frac{\partial z_{y_2}}{\partial a_{h_1}}
\end{aligned}$$

Since $z_{y_1} = w_5 a_{h_1} + w_6 a_{h_2} + b_2$ and $z_{y_2} = w_7 a_{h_1} + w_8 a_{h_2} + b_2$,

$$\begin{aligned}
\frac{\partial z_{y_1}}{\partial a_{h_1}} &= w_5 = 0.40 \\
\frac{\partial z_{y_2}}{\partial a_{h_1}} &= w_7 = 0.50
\end{aligned}$$

Recall that we calculated the derivative of the error term with respect to $\hat{y}$ in equation 1.1 and the derivative of $\hat{y}$ with respect to z_{y_1} in equation 1.4. So for $\frac{\partial E_{y_1}}{\partial a_{h_1}}$,

$$\begin{aligned}\frac{\partial E_{y_1}}{\partial a_{h_1}} &= \frac{\partial E_{y_1}}{\partial \hat{y_1}} \frac{\partial \hat{y_1}}{\partial z_{y_1}} \frac{\partial z_{y_1}}{\partial a_{h_1}} \\ &= 0.74136507 * 0.186815602 * 0.40 \\ &= 0.055399425\end{aligned}$$

We follow the same steps for $\frac{\partial E_{y_2}}{\partial a_{h_1}}$ to get -0.019049119. Thus,

$$\begin{aligned}\frac{\partial E_{total}}{\partial a_{h_1}} &= \frac{\partial E_{y_1}}{\partial a_{h_1}} + \frac{\partial E_{y_2}}{\partial a_{h_1}} \\ &= 0.055399425 + (-0.019049119) \\ &= 0.036350306\end{aligned}$$

Next, we focus on the second portion of derivative, $\frac{\partial a_{h_1}}{\partial z_{h_1}}$. Since $a_{h_1} = \sigma(z_{h_1})$,

$$\begin{aligned}\frac{\partial a_{h_1}}{\partial z_{h_1}} &= a_{h_1}(1 - a_{h_1}) \\ &= 0.59326999(1 - 0.59326999) \\ &= 0.241300709\end{aligned}$$

Finally we will focus on the last portion of the derivative, $\frac{\partial z_{h_1}}{\partial w_1}$. Since $z_{h_1} = w_1 x_1 + w_2 x_2 + b_1$,

$$\frac{\partial z_{h_1}}{\partial w_1} = x_1 = 0.05$$

Finally, combining the three derivatives we just calculated, we get

$$\begin{aligned}\frac{\partial E_{total}}{\partial w_1} &= \frac{\partial E_{total}}{\partial a_{h_1}} \frac{\partial a_{h_1}}{\partial z_{h_1}} \frac{\partial z_{h_1}}{\partial w_1} \\ &= 0.036350306 + 0.241300709 + 0.05 \\ &= 0.000438568\end{aligned}$$

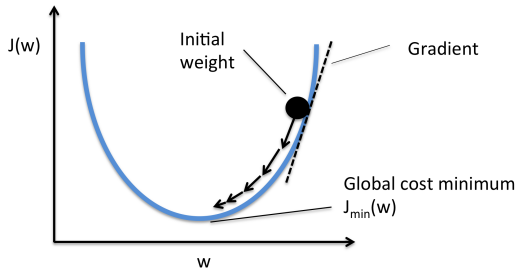

Figure 1.2: Visualization of gradient descent optimization. Following the negative of the gradient multiplied by some learning rate γ results in minimizing the objective function. Figure from Raschka (2018)

Now that we have calculated the gradients, we can update the weights of the model. We will do this through a process known as gradient descent. We will select a learning rate, γ, that determines how much each weight is updated by the gradient. Figure 1.2 provides intuition for this process. The curve $J(w)$ can be thought of as the total error or loss function. Since changing the weights changes the total loss, we can think of the loss function as a function of the weights. As we change the weights of the neural network, we can move to a lower point on the loss function. By subtracting the gradient from the weights, we arrive at weights with a lower total error. The parameter γ is represented by the relative proportion of the arrows. As γ grows larger, all of the tangent lines grow longer. If γ is too large, the weights could move away from the cost minimizing values. This can lead to "bouncing" where weights shift from side to side of the error valley rather than converging to the minimum. This can be seen from the tangent line. If γ is too small, convergence could take too long to be practical. This value can be selected through cross validation. For simplicity's sake, we select $\gamma = 0.5$ Thus,

$$\begin{aligned} w_1^* &= w_1 - \gamma \frac{\partial E_{total}}{\partial w_1} \\ &= 0.15 - 0.5 * 0.000438568 \\ &= 0.149780716 \end{aligned}$$

We can repeat this process for $w_2 \dots w_8$. Once gradients for the entire network are found, the weights are updated simultaneously. This yields a total error of 0.291027924, down from 0.298371109. Running a forward pass

followed by backpropagation $10,000$ more times yields weights which produce a total error of 0.000035102 on our training set.

1.2.4 Vectorized Backpropagation

The prior example demonstrates backpropagation on a single sample with 2 input values. In the real world, our dataset may have hundreds of millions of examples. Furthermore, each example may have hundreds or thousands of input values (e.g., an image). Dealing with this set up efficiently requires vectorization. We can think of each sample $x^{(i)}$ as a vector of $x_1, x_2, \ldots, x_m$ input values, and the set of all inputs X as an $m \times n$ matrix of samples.

Additionally, we can use other functions to calculate the total error. In the prior example, we utilized the mean squared error function. Mean squared error corresponds to the L2-norm of the difference between the observed response and the output of the neural network. Generalizing the idea of error functions allows us to use other functions to optimize a neural network, such L1 loss.

According to Nielsen (2015), a valid cost function must meet the following criteria :

1. Total cost $C(X)$ over all training examples X must equal the average of the cost function evaluated at each training example $x^{(i)}$:

$$C(X) = \frac{1}{n} \sum_{i=1}^{n} C(x^{(i)})$$

2. Cost must be a nonconstant differentiable function of $A^{(L)}$, the activation (output) of the final layer L in the neural network:

$$C = f(A^{(L)})$$

The first criterion allows us to recover the derivative of the cost function with respect to our weights by averaging over training examples such that

$$\frac{dC(X)}{dw_{ij}^{l}} = \frac{1}{n} \sum_{i=1}^{n} \frac{dC(x^{(i)})}{dw_{ij}^{l}} \tag{1.6}$$

where w^l is the matrix of weights in layer l. This is required since each step of backpropagation only gives us $\frac{dC(x^{(i)})}{dw_{ij}^{l}}$.

The second assumption ensures that changing the weights of the network (which affect the output) will change the value of the cost function, allowing the network to learn.

Table 1.1: Backpropagation Notation

$x^{(i)}$	$\triangleq$	Single training example of shape $(m, 1)$. Sometimes referred to as $a^{(0)}$
n_l	$\triangleq$	Number of neurons in layer l
w^l	$\triangleq$	Weight matrix in layer l of shape (n_l, n_{l-1})
b^l	$\triangleq$	Bias term in layer l of shape $(n_l, 1)$
z^l	$\triangleq$	$w^l a^{(l-1)} + b^l$
$g(\cdot)$	$\triangleq$	Activation function in a given layer
a^l	$\triangleq$	$g(z^l)$

Next, we will consider vectorized backpropagation. In addition to the variables defined in Table 1.1, we will use an intermediate quantity δ_j^l such that

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

where z_j^l is the linear combination of neuron j in layer l. Following Hallstrom (2016), we begin at the output layer L. Using the chain rule, we find that

$$\delta^L = \nabla_{a^L} C \odot g'(z^L) \tag{1.7}$$

where $\odot$ denotes the elementwise product. We use $g'(z^L)$ to denote the derivative of the activation function applied elementwise to the vector z^L. For example, if $g(x) = \sigma(x)$, then $g'(x) = \frac{d}{dx}\sigma(x)$.

Next, we find an equation for δ at any given layer l based on the following layer $l + 1$. This follow from basic matrix calculus and is left as an exercise for the reader:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot g'(z^l) \tag{1.8}$$

Since we know δ^{l+1}, we can propagate the error backwards through the network using the transposed weight matrix $(w^{l+1})^T$. We then apply the

chain rule to move through the activation function $g(z)$ in order to calculate δ^l.

Using the recursive definition to calculate δ^l for each layer l in the network, we can now calculate the derivative of the cost function C with respect to the weights and biases at each layer of the network.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \tag{1.9}$$

since $z_j^l = w_k^l j a_j^{l-1} + b_j^l \implies \frac{\partial z_j^l}{\partial b_j^l} = 1$.

Additionally, we find that

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \tag{1.10}$$

$$= a_k^l \delta_j^l \tag{1.11}$$

With these derivatives, we can now update our weights:

$$w^{l*} = w^l - \gamma \nabla_{w^l} C(X)$$
$$b^{l*} = b^l - \gamma \nabla_{b^l} C(X)$$

where γ is the learning rate and $\nabla_w C(X)$ is an (n_l, n_{l-1}) matrix containing the gradients of the cost function C with respect to the current weights w evaluated on the training set X for a single layer of the neural network.

1.3 Convolutional Neural Networks

Convolutional neural networks ("CNNs") are neural networks that use the convolution operation in place of general matrix multiplication (Goodfellow et al., 2016). CNNs take advantage of the grid-like structure of images and provide four main advantages: sparsity, parameter sharing, equivariant representations, and variable input size.

First, I will detail the convolution operation. Convolution is an operation applied to two functions. We use $(I * F)(i, j)$ to denote the convolution of image I and filter F at position (i, j) in the image. The filter is generally much smaller than the image. In practice, a neural network might use

256x256 images and 3x3 or 5x5 filters. We can write the convolved feature map $S(i, j)$:

$$\begin{aligned} S(i,j) = (I * F)(i,j) &= \sum_m \sum_n I(m,n)\ F(i-m, j-n) \\ &= \sum_m \sum_n I(i-m, j-n)\ F(m,n) \end{aligned}$$

Since convolution is commutative, we can flip the arguments and preserve equality (Goodfellow et al., 2016).

Many machine learning libraries implement cross-correlation rather than convolution. In cross-correlation, the filter F is simply "flipped", meaning that we change subtraction to addition. This means that as the index m increases, we move down both the image and the kernel. Since the parameters inside the filters are randomly initialized and tuned using backpropagation, it does not matter whether convolution or cross correlation is used. We write the cross-correlation of I and F:

$$S(i,j) = (I * F)(i,j) = \sum_m \sum_n I(i+m, j+n)\ F(m,n) \tag{1.12}$$

The smaller size of the filter F relative to the image I creates sparse interactions, meaning fewer connections between neurons. Let n_{in} be the number of inputs and n_{out} be the number of outputs in one layer of a fully connected neural network. Thus, the matrix multiplication step will have $O(n_{in} \times n_{out})$ runtime. If each output only looks at k input values (corresponding to looking at a small patch of the image), the step has complexity $O(k \times n_{out})$ where $k << n_{in}$.

Convolutional networks reuse the same weight parameters across different positions of an image. In a fully connected network, each parameter is used once (multiplied by the input at a certain position) to generate part of an output. In a convolutional network, each element of the filter is used at every element of the input. This means we only need to learn k parameters, or perhaps $k \times n_{filters}$ where k is the number of elements in each filter.

Another important characteristic of CNNs is equivariance. Mathematically, $f(x)$ is equivariant to function $g(x)$ if $f(g(x)) = g(f(x))$. Convolution is equivariant to changes in object location in an image. Consider an image

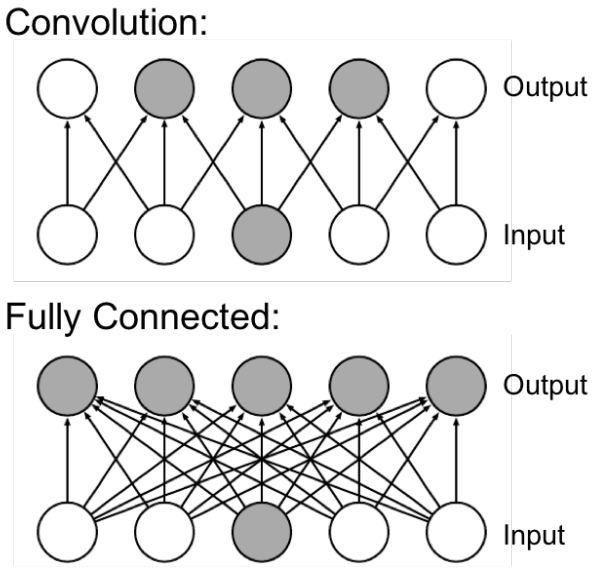

Figure 1.3: Visualization of convolutional neural network (top) compared to a fully connected neural network (bottom). Each input only affects 3 outputs in the convolutional network, while each input affects every output in the fully connected network. (Goodfellow et al., 2016)

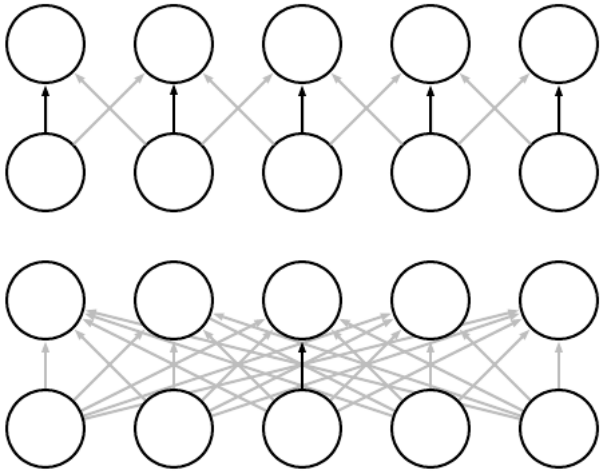

Figure 1.4: Visualization of convolutional neural network (top) compared to a fully connected neural network (bottom). Each parameter is used across each input in the convolutional case, compared to only being used once in the fully connected case. (Goodfellow et al., 2016)

I and filter F. If we apply some function $g(I)$ that shifts each pixel down and apply the same convolution $(g(I) * F)$, we get the same result as applying the function after the convolution $g(I * F)$. In practice, this means that convolutional neural networks do not care about the location of an object in an image. If a CNN recognizes a bird in the center of an image, it will also be able to recognize the same bird in the corner of an image. Note that this only applies to direct movements of an image. Convolution is not equivariant to changes in scale or rotation (Goodfellow et al., 2016).

Lastly, convolution allows for variable input sizing. Since the number of parameters does not depend on the number of inputs but rather on the filter size and number of filters, a neural network that works on 256x256 images will work on 512x512 images. Note however, this may require adaption to deal with changes in scale since objects in a 256x256 image appear relatively smaller in 512x512 images.

It can be useful to think of convolution in terms of sliding windows. In the case of a 4×4 image convolved with a 3×3 filter, the filter "scans" along the image, moving with a stride of 1 across the image until it hits the end before moving down one pixel and repeating the process again. This results

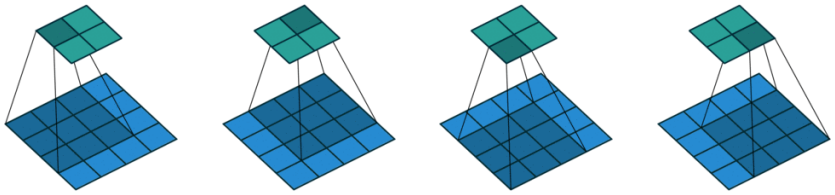

Figure 1.5: Convolution as a sliding window problem. Figure from deeplearning.net

in a 2×2 output image.

This demonstrates one important consideration in convolutional networks: convolution results in a smaller output. We can adjust the size of the output image by changing 3 hyperparameters: filter size, padding, and stride. Filter size refers to the number of parameters used in each filter. Larger filters will "see" more of the image, but also result in a smaller output. Padding refers to the amount of zeros we add around the sides of an image or matrix. Finally, stride refers to the amount the sliding window moves after each step. In the example above, the stride length was 1. We will normally use $f \times f$ to refer to the filter size, s to refer to the stride, and p to refer to the amount of padding added around the image.

Consider an image of size $n \times n$. The convolution operation will result in an $m \times m$ output where

$$m = \left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1 \tag{1.13}$$

Based on this equation, we can see that a 3×3 filter with stride 1 result in an image 2 pixels smaller in each dimension. This is known as a valid convolution. If we want to preserve the input size, we can simply add 1 row of zeros around all sides of the input image. This is known as a same convolution.

If we want to downsample, or reduce the image size, we can use a stride length of 2 or more.

1.4 Transfer Learning

Taking an image and convolving it with a filter creates an output known as a feature map. Feature maps created by deep neural networks are sur-

prisingly useful and transferable. Sharif Razavian et al. (2014) used feature maps from a pretrained network to train a linear SVM classifier on various new recognition tasks such as image classification, scene recognition, and attribute detection, achieving near state-of-the-art results. Yosinski et al. (2014) quantified the generality versus specificity of feature representations in each layer of a deep neural network, finding that using low level feature maps significantly increases performance even for distant tasks. They further noted that many deep neural networks learn low level feature maps similar to Gabor filters, a type of image filter that resembles those found in the human eye. In a way, training a CNN on a large image dataset teaches it to "see" the world. We can describe this learning process as follows:

Definition 1.1. *(Transfer Learning)* Given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, transfer learning aims to help improve the learning of the target predictive function $f_T(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$ or $\mathcal{T}_S \neq \mathcal{T}_T$ (Pan and Yang, 2010)

The approach to transfer learning used in neural style transfer is known as feature representation transfer. This approach attempts to learn "good" feature representations (also known as feature maps) from the source task $\mathcal{T}_S$ and source domain $\mathcal{D}_S$ which can be useful for our target task $\mathcal{T}_T$ and target domain $\mathcal{D}_T$.

Concretely, we can train a large neural network on a supervised learning task such as image classification and then use the feature representations created internally for a new task. In this case, the source learning task is image classification and the source domain is a set of 10 million images. The target task is neural style transfer, and the target domain is a set of 120k images. We will utilize feature maps created from the filters of the image classification network to stylize new images.

Chapter 2

Content and Style Loss Functions

Style transfer can be thought of as the creation of pastiches, or works of art that imitate the style of another artist or piece (Dumoulin et al., 2016). For example, one could apply the style of Starry Night by Van Gogh to a photograph of the Golden Gate bridge, creating an image of the golden gate bridge that looks like it was painted by Van Gogh.

This task requires the separation of style from content. The pastiche preserves the colors and textures of a certain artist or work while displaying different objects or scenery.

This approach requires a mathematical definition of content and style. One naive approach might be to average the pixel values of the content and style images. However, averaging pixels does not result in images of high perceptual quality. Changing the color of an image could result in a large deviation in pixel values while not changing the content. Likewise, fine details that define the image could be lost in favor of large blocks of color. True content and style representations require the algorithm to recognize the objects that the image contains.

Gatys et al. (2015a) introduced the idea of using deep convolutional neural networks to create style and content representations of an image. Specifically, the authors used feature representations from a convolutional neural network known as VGG-16 to create a definition of content and style. VGG-16 is an image classification network trained on 10 million images to classify objects into 1,000 categories. Along its architecture, it develops feature representations for images that make semantic information more and more explicit

across layers (Gatys et al., 2015b). In earlier layers, representations capture information corresponding to colors and textures. In deeper layers, representations capture information about the objects in the image and their arrangement (Gatys et al., 2015a). Eventually, VGG-16 returns a 1x1000 vector that indicates which object is most likely contained in the image.

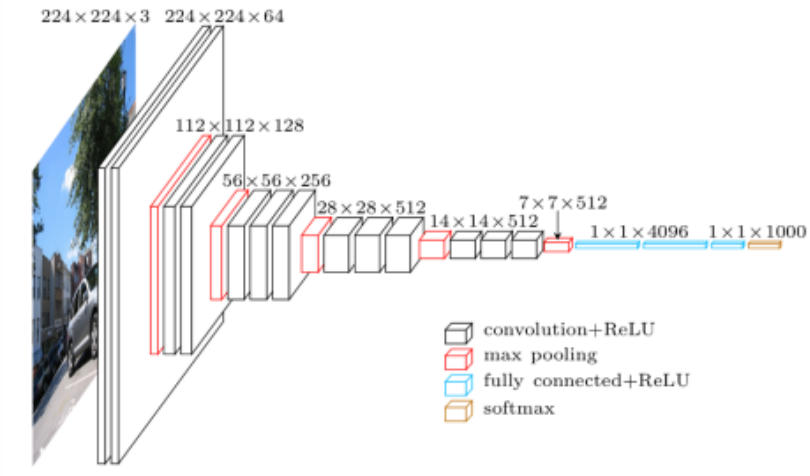

Figure 2.1: VGG network architecture (Cord, 2016).

We can use feature responses in deeper layers as the content representation. Concretely, the content representation is a matrix F^l of shape $(n_{filters}, n_{pixels})$ where F^l_{ij} is the activation of the ith filter at position j in layer l of the convolutional neural network. Filters are analogous to color channels in an image. Neural networks may develop feature representation that have 128, 256, or 512 filters, compared to the 3-filter RGB color representation used in most image formats.

Gatys et al. (2015a) obtained the style representation by calculating the correlations between filter responses across multiple layers of the neural network. This feature space was originally created to capture texture information while discarding (x, y) positional information. The style representation is calculated by taking the Gram matrix of the vectorized feature maps:

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk} \tag{2.1}$$

G^l represents an $(n_{filters}, n_{filters})$ sized feature space where each entry is

a measure of correlation between filter response i and filter response j across the image.

Content representations F^l and style representations G^l can be used to create a loss function.

We define content loss as the Frobenius norm of the difference between two feature representations

$$\mathcal{L}_{content}(\boldsymbol{p}, \boldsymbol{x}) = \|F_{ij}^l - P_{ij}^l\|_F^2 \tag{2.2}$$

where F_{ij}^l is the content representation of $\boldsymbol{x}$, the input image, and P_{ij}^l is the content representation of $\boldsymbol{p}$, the content photograph, in layer l. Gatys et al. (2015b) used only one layer of VGG16, 'conv4_2', to calculate the content representation.

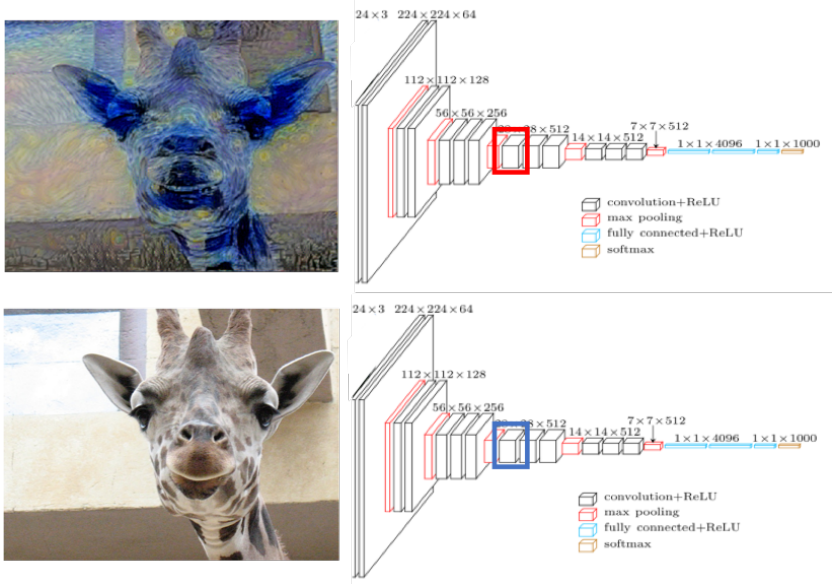

Figure 2.2: Visualization of the content loss function

Figure 2.2 gives an illustration of how the content loss function is calculated. Two images are plugged into the VGG-16 neural network. The activations of each image inside the neural network are compared, giving a metric of the similarity of the two images in terms of content.

Figure 2.3 demonstrates what happens if we start with white noise and minimize only the content loss function. The resulting image maintains the same content as the original image despite minor aberrations.

Figure 2.3: Reconstructing an image from the content representation of the image in layer 'conv3_1' of the VGG16 network (Mahendran and Vedaldi, 2015). (Photo: sftravel.com)

Similarly, for style loss, we calculate the norm of the difference between the style representations. Since we use multiple layers (usually 'conv1_1' through 'conv5_1' in VGG16) to create the style representation, we average across those layers to get the total style loss. Let G^l_{ij} be the style representation of $\boldsymbol{x}$ and A^l_{ij} be the style representation of $\boldsymbol{a}$, the style artwork:

$$\mathcal{L}_{style}(\boldsymbol{a}, \boldsymbol{x}) = \sum_{l=1}^{L} w_l \cdot \|G^l_{ij} - A^l_{ij}\|^2_F \tag{2.3}$$

where w^l is the weight of each layer such that w sums to 1.

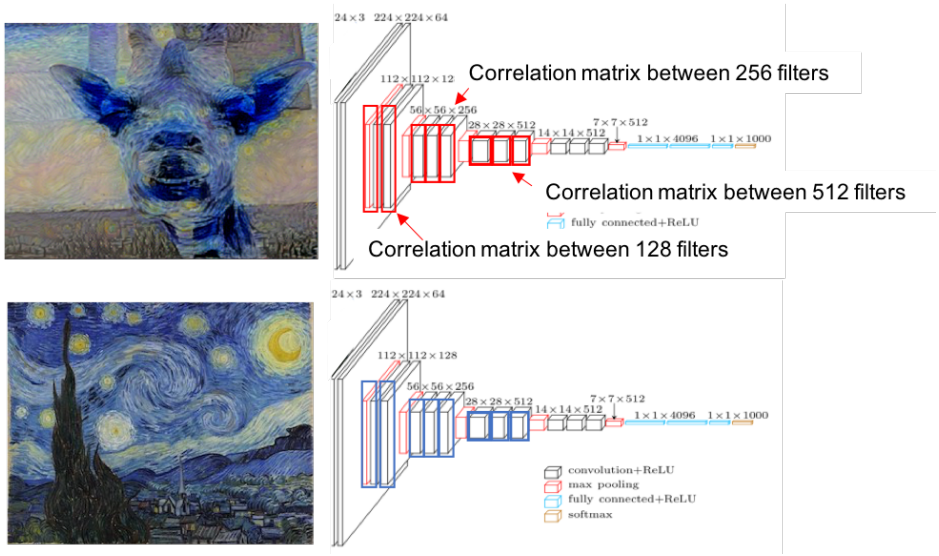

Figure 2.4: Visualization of the style loss function

Figure 2.4 gives an illustration of the style loss function. Two images are plugged into the VGG-16 network. Next, we calculate the correlation between different features at each layer. Finally, we compare the difference between the correlation matrix of the input image to the correlation matrix of the style image to arrive at the style loss.

Figure 2.5: Reconstructing an image from the average style representation of the image across layers 'conv1_1' to 'conv5_1' of the VGG16 network (Mahendran and Vedaldi, 2015). (Photo: Starry Night by Van Gogh)

Figure 2.5 demonstrates what happens if we start with white noise and minimize only the style loss function. The resulting image discards positional information due to the usage of correlation matrices and seems to do a good job of maintaining color and texture information.

If we select an $\boldsymbol{x}$ that minimizes the weighted sum of the loss functions such that

$$\boldsymbol{x} = \arg\min_{\boldsymbol{x}} \ \alpha \mathcal{L}_{content}(\boldsymbol{p}, \boldsymbol{x}) + \beta \mathcal{L}_{style}(\boldsymbol{a}, \boldsymbol{x}) \tag{2.4}$$

we achieve a new image with the content of $\boldsymbol{p}$ in the style of $\boldsymbol{a}$. The minimization problem is intractable analytically since the loss function is a function of a neural network, which is not invertible. However, optimization methods such as gradient descent can be used. We will simply calculate the derivative of the loss function with respect to each pixel in $\boldsymbol{x}$ and update each pixel to minimize the loss function.

Algorithm 1 Gatys et al. (2015a) Neural Style Transfer

$\boldsymbol{x} \leftarrow$ white noise of shape (height, width, channels)
$\boldsymbol{p} \leftarrow$ content photograph of size x.shape
$P \leftarrow$ content representation of $\boldsymbol{p}$ in layer 'conv4_2'
$\boldsymbol{a} \leftarrow$ style artwork of size x.shape
$A \leftarrow$ style representation of style artwork
$\gamma \leftarrow$ learning rate
for $i \leftarrow 1$ to n_iterations **do**
    $F \leftarrow$ content representation of $\boldsymbol{x}$
    $G \leftarrow$ style representation of $\boldsymbol{x}$
    $\mathcal{L}_{total} \leftarrow \mathcal{L}_{style}(G, A) + \mathcal{L}_{content}(F, P)$
    $\boldsymbol{x}_{ij} \leftarrow \boldsymbol{x}_{ij} - \gamma \frac{\partial \mathcal{L}_{total}}{\partial \boldsymbol{x}_{ij}}$
return $\boldsymbol{x}$

Figure 2.6: Neural style transfer using the algorithm proposed by Gatys et al. (2015a). (Photo: sftravel.com; Artwork: Starry Night by Van Gogh)

Figure 2.6 demonstrates the result of Algorithm 1 using a picture of the golden gate bridge as the content image and Starry Night as the style image. The resulting image seems to maintain the colors and textures of Starry Night while remaining recognizable as the golden gate bridge.

While this process results in an image of high perceptual quality, the reason why the Gram matrices represent artistic style is not entirely clear. Li et al. (2017) offers an interpretation of Gram matrix matching in terms of a special domain adaptation problem known as Maximum Mean Discrepancy minimization.

Maximum Mean Discrepancy is a test statistic introduced by Gretton (2012) to determine if two samples are drawn from the same distribution. Consider two sets of samples $X = \{x_i\}_{i=1}^n$ and $Y = \{y_j\}_{j=1}^m$ defined in a topological space $\mathcal{X}$, where $x_i \sim p$ and $y_j \sim q$, *i.i.d.*. Then squared MMD is defined as the difference in expected value between X and Y when mapped into the reproducing kernel Hilbert space H by the function $\phi : \mathcal{X} \to \mathcal{H}$:

$$\begin{aligned}\mathrm{MMD}^2[X,Y] &= \|E_{x\sim p}[\phi(X)] - E_{y\sim q}[\phi(Y)]\|^2 \\ &= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\phi(x_i)^T\phi(x_j) + \frac{1}{m^2}\sum_{k=1}^{m}\sum_{l=1}^{m}\phi(y_k)^T\phi(y_l) - \frac{2}{nm}\sum_{i=1}^{n}\sum_{k=1}^{m}\phi(x_i)^T\phi(y_k)\end{aligned}$$

Letting kernel function $k(x, y) = \langle\phi(x), \phi(y)\rangle$,

$$\mathrm{MMD}^2[X,Y] = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}k(x_i,x_j) + \frac{1}{m^2}\sum_{k=1}^{m}\sum_{l=1}^{m}k(y_k,y_l) - \frac{2}{nm}\sum_{i=1}^{n}\sum_{k=1}^{m}k(x_i,y_k) \tag{2.5}$$

Following the steps of Li et al. (2017), we can show that our style loss function $\mathcal{L}_{style}^l$ on a single layer l is equivalent to the squared MMD metric on F^l and S^l with the kernel function $k(x, y) = (x^T y)^2$, where F^l and S^l are the feature representations of $\boldsymbol{x}$ and $\boldsymbol{a}$ in the neural network, respectively.

$$\mathcal{L}^l_{style} = \sum_{i=1}^{N}\sum_{j=1}^{N}(G^l_{ij} - A^l_{ij})^2 \tag{2.6}$$

$$= \sum_{i=1}^{N}\sum_{j=1}^{N}(\sum_{k}^{M} F^l_{ik}F^l_{jk} - \sum_{k}^{M} S^l_{ik}S^l_{jk})^2 \tag{2.7}$$

$$= \sum_{i=1}^{N}\sum_{j=1}^{N}((\sum_{k}^{M} F^l_{ik}F^l_{jk})^2 + (\sum_{k}^{M} S^l_{ik}S^l_{jk})^2 - 2(\sum_{k}^{M} F^l_{ik}F^l_{jk})(\sum_{k}^{M} S^l_{ik}S^l_{jk})) \tag{2.8}$$

$$= \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k_1=1}^{M}\sum_{k_2=1}^{M}(F^l_{ik_1}F^l_{jk_1}F^l_{ik_2}F^l_{jk_2} + S^l_{ik_1}S^l_{jk_1}S^l_{ik_2}S^l_{jk_2} - 2F^l_{ik_1}F^l_{jk_1}S^l_{ik_2}S^l_{jk_2}) \tag{2.9}$$

$$= \sum_{k_1}^{M}\sum_{k_2}^{M}\sum_{i=1}^{N}\sum_{j=1}^{N}(F^l_{ik_1}F^l_{jk_1}F^l_{ik_2}F^l_{jk_2} + S^l_{ik_1}S^l_{jk_1}S^l_{ik_2}S^l_{jk_2} - 2F^l_{ik_1}F^l_{jk_1}S^l_{ik_2}S^l_{jk_2}) \tag{2.10}$$

$$= \sum_{k_1=1}^{M}\sum_{k_2=1}^{M}((\sum_{i=1}^{N} F^l_{ik_1}F^l_{ik_2})^2 + (\sum_{i=1}^{N} S^l_{ik_1}S^l_{ik_2})^2 - 2(\sum_{i=1}^{N} F^l_{ik_1}S^l_{ik_2})^2) \tag{2.11}$$

$$= \sum_{k_1=1}^{M}\sum_{k_2=1}^{M}(k(f^l_{k_1}, f^l_{k_2}) + k(s^l_{k_1}, f^l_{s_2}) - 2k(f^l_{k_1}, s^l_{k_2})) \tag{2.12}$$

$$= M^2 \ \mathrm{MMD}^2[F^l, S^l] \tag{2.13}$$

with f^l_k and s^l_k as the kth column of F^l and S^l respectively. Therefore, style transfer can be seen as aligning the distribution of the input image $\boldsymbol{x}$ with the style image $\boldsymbol{a}$ in the reproducing kernel Hilbert space associated with the second order polynomial kernel (Li et al., 2017). Since the activations at each position of the filter responses is taken as an individual sample, the style loss function ignores positional information within the image.

Chapter 3

Training Neural Networks with Perceptual Loss Functions

It may be infeasible to run Algorithm 1 for each content image. It takes a graphics card thousands of iterations to arrive at an acceptable image. Given that neural networks are universal function approximators, we could instead train a neural network to transfer the style of a particular image onto any image that is inputted. We can write

$$\boldsymbol{y} = f^*(\boldsymbol{a}, \boldsymbol{p}) = \arg\min_{\boldsymbol{x}} \ \alpha \mathcal{L}_{content}(\boldsymbol{x}, \boldsymbol{p}) + \beta \mathcal{L}_{style}(\boldsymbol{x}, \boldsymbol{a}) \tag{3.1}$$

as the function to approximate, where $\boldsymbol{a}$ is the artwork, $\boldsymbol{p}$ is the photograph or content image, and $\boldsymbol{y}$ is the stylized image that the network outputs. A single forward pass through a neural network requires far less computation than the optimization scheme employed by Gatys et al. (2015b). This allows new images to be stylized nearly instantaneously or on lower power machines such as smartphones. This speed in stylizing new images comes at the cost of training the image transformation network in the first place. Training the image transformation network requires hundreds of thousands of content images. The training process takes around 1.5 hours on an Nvidia 1080TI.

In this chapter, we will detail the type of neural network that can be used to transform one image into another. Finally, we will describe the loss function used to train this specialized type of neural network.

3.1 Network Architecture

Image transformation requires a new type of neural network architecture. Unlike other CNNs, image classification networks do not output a classification. Instead, they output a new or transformed image. As a result, image transformation networks are entirely convolutional, meaning they do not use any fully connected layers. This significantly reduces the number of parameters needed to train our network and allows us to stylize any sized image once the network is trained.

Image transformation network architectures tend to have several similar characteristics in common. In the first few layers, image size is reduced while the number of filters increases. Next, a series of residual blocks are applied to the image. Residual blocks are a specific type of convolution which will be explored later in this section. Finally, the image is enlarged back to its original size as the number of filters decreases back to the r-g-b color range.

This architecture has two main benefits. Firstly, downsampling, or reducing the image size, allows us to use a much larger network with the same computational cost as a smaller network on a non-downsampled image. Secondly, downsampling by a factor of D increases the receptive field by size $2D$, allowing for a larger field with the same number of layers (Johnson et al., 2016). Figure 3.1 demonstrates the idea of receptive field. If the image was one pixel smaller on each side, every neuron could see the entire image starting in the second layer instead of the third layer.

Residual blocks are another significant component of the image transformation network architecture. Residual blocks perform a series of convolutions, then add the input $\boldsymbol{x}$ back before returning the output. This is commonly referred to as a shortcut mapping. He et al. (2015) argues that shortcut mappings allow deep neural networks to more easily learn the identity function. Non-residual networks can experience performance degradation while training if the network becomes too deep. One hypothesis for this is that identity mappings can be difficult for multiple non-linear functions to learn. However, in residual networks, the identity function can be learned by pushing the weights towards zero. Hypothetically, this is a useful feature for image transformation networks since our desired output looks similar to the input image. In practice, equivalent non-residual convolutional blocks result in similar looking images, but take longer to train (Johnson et al., 2016).

Figure 3.2 shows the design of the convolutional blocks used in JohnsonNet. First, a 3x3 same convolution is performed, maintaining image size.

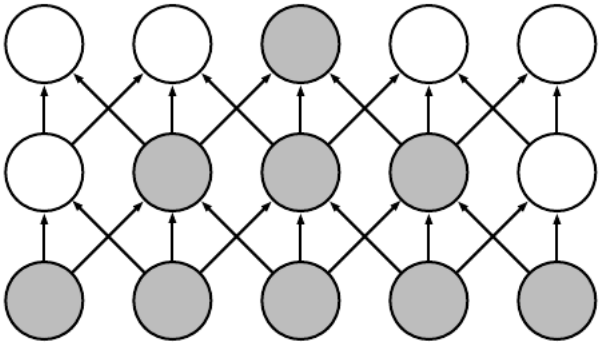

Figure 3.1: Visualization of the receptive field in a convolutional neural network. Neurons in deeper layers are able to "see" a larger portion of the image. If the input is downsampled, each neuron can see a much larger portion of the image in earlier layers. (Goodfellow et al., 2016)

Next, a batch normalization layer normalizes the distribution of the images across each filter. Since only a handful of images are put through the network at once due to memory limitations, this normalization only takes place across the batch of images currently in the network. Batch normalization helps avoid a problem known as internal covariate shift. As the weights in one layer change, so does the distribution of inputs in the next layer. Ioffe and Szegedy (2015) noted that adding layers which normalize images across each batch allows for faster training time and increased accuracy. After the batch normalization layer, a ReLU or rectified linear activation function is applied elementwise. The ReLU function is simply defined as 0 for $x < 0$ and x otherwise. This serves the same function as the sigmoid function in Chapter 1, but is computationally simpler. Next, a 3x3 convolution is applied again followed by more batch normalization. Again, since same convolution is used, the input size remains the same. Finally, the original input is added back to the output before it is returned.

Table 3.1 shows the architecture of the image transformation network used by Johnson et al. (2016). The network takes the content image as an input and returns a stylized image. The content and style loss functions from Chapter 2 are used to train JohnsonNet.

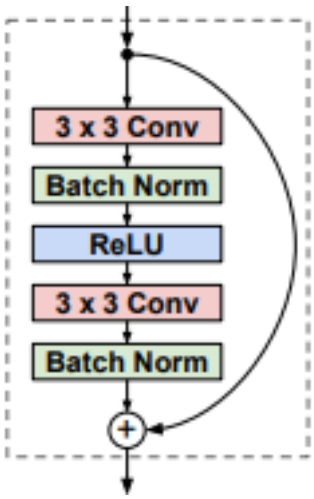

Figure 3.2: Residual block used in Johnson et al. (2016)

Layer	Activation Size
Input	$3 \times 256 \times 256$
$32 \times 9 \times 9$ conv, stride 1	$32 \times 256 \times 256$
$64 \times 3 \times 3$ conv, stride 2	$64 \times 128 \times 128$
$128 \times 3 \times 3$ conv, stride 2	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
$64 \times 3 \times 3$ conv, stride 1/2	$64 \times 128 \times 128$
$32 \times 3 \times 3$ conv, stride 1/2	$32 \times 256 \times 256$
$3 \times 3 \times 3$ conv, stride 1	$3 \times 256 \times 256$

Table 3.1: Network architecture for style transfer used by Johnson et al. (2016)

3.2 Loss Function

There are two potential ways to train a neural network to approximate the content and style loss function. One way would be to use Algorithm 1 to

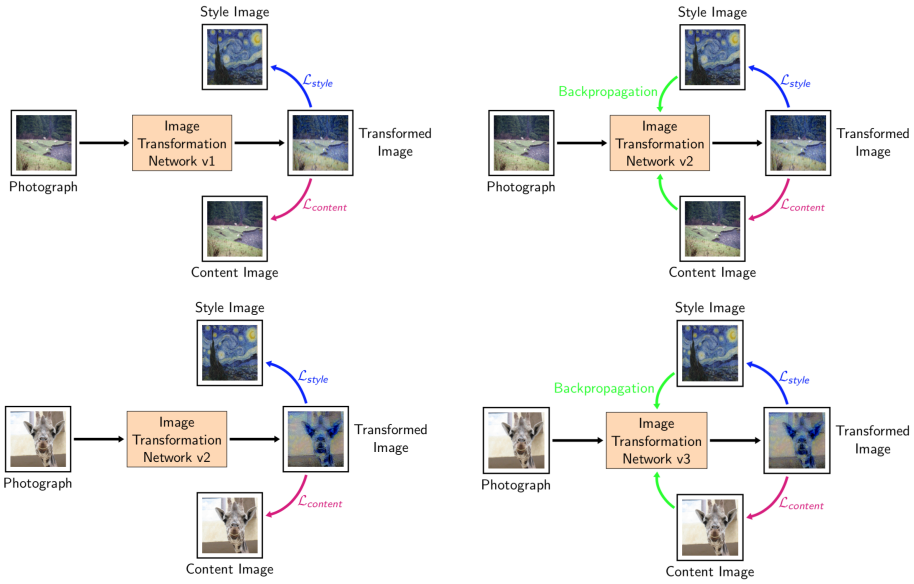

Figure 3.3: Training using perceptual loss functions

generate millions of pair images which could be used to train a neural network using a simple MSE loss function. This approach has a large downside – it is infeasible to generate that many stylized images. Instead, a more direct approach can be used. We can simply use the loss function from Gatys et al. (2015b) as a loss function for our image transformation network. This means we will not have to generate any stylized images before training.

Figure 3.3 illustrates the process of training a transformation network using the style and content loss functions. First, a new photograph is inputted into the image transformation network. Next, the transformed image is compared to the style image using the loss function $\mathcal{L}_{style}$ and compared to the content image using the loss function $\mathcal{L}_{content}$. Next, the image transformation network is updated using backpropagation in order to minimize the weighted average of these functions. This process repeats again with a new photograph over and over. Due to the large number of parameters in the image transformation network, it can take a large number of training images to achieve a suitable result. In practice, a dataset of 120,000 images is usually sufficient.

Although the loss function itself remains largely the same, Johnson et al. (2016) makes one modification to the loss function from Chapter 2. In order

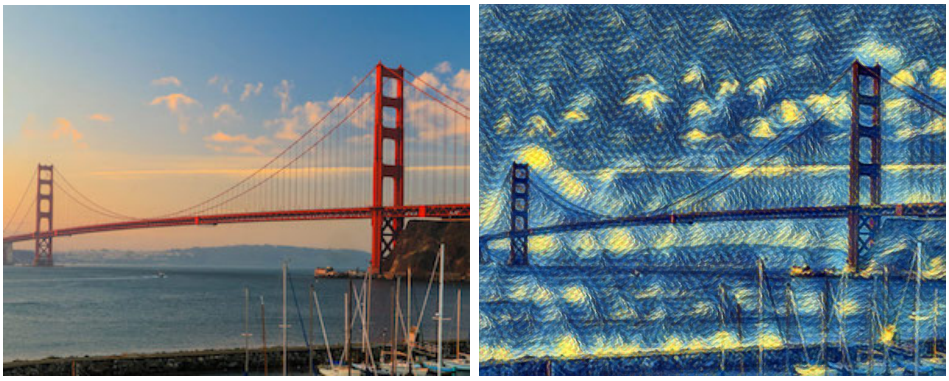

Figure 3.4: Results from applying the Johnson et al. (2016) Starry Night model to the Golden Gate Bridge content image used previously. (Johnson et al., 2016)

to encourage spatial smoothness, total variation regularization is added. This simply adds a loss $\mathcal{L}_{TV}(\boldsymbol{x})$ multiplied by weight parameter λ to the content and style loss function from Chapter 2, where

$$\mathcal{L}_{TV}(\boldsymbol{x}) = \sum_{i,j} |x_{i+1,j} - x_{i,j}| + |x_{i,j+1} - x_{i,j}| \tag{3.2}$$

This effectively sets a limit on the total difference in the horizontal and vertical dimensions of an image which helps reduce noise.

Figure 3.4 applies the Starry Night model from Johnson et al. (2016) to our content image of the Golden Gate Bridge. This was created by downloading the weights made available from Johnson et al. (2016) and loading them into the JohnsonNet architecture.

Chapter 4

Improving on neural style transfer

While the result from Chapter 3 is much faster to generate than the result from Chapter 2, it does not subjectively resemble the original Starry Night painting. We can make several alterations to the image transformation network and training procedure in order to boost the perceptual quality.

4.1 Deconvolution

As stated in Chapter 3, image transformation networks generally shrink the output image to a smaller size before applying the residual blocks and then increase the image size for output. A same convolution with stride 2 will result in an image half as large as the original. As a result, Johnson et al. (2016) uses strided convolutions to accomplish downsampling. Another technique, known as fractionally strided convolution or deconvolution, can be used to upsample an image at the end of the network.

Fractionally strided convolution can be thought of as adding rows and columns of zeros between each pixel in the input image. Figure 4.1 demonstrates this idea with a stride of $\frac{1}{2}$. This undoes the effect of a stride 2 convolution earlier in the network, so a stride $\frac{1}{2}$ same convolution will double the size of an input image.

Deconvolution unfortunately results in strange checkerboard artifacts. In the JohnsonNet architecture, a filter size of 3 and partial stride of $\frac{1}{2}$ are used. Figure 4.2 demonstrates how this can be problematic. The uneven

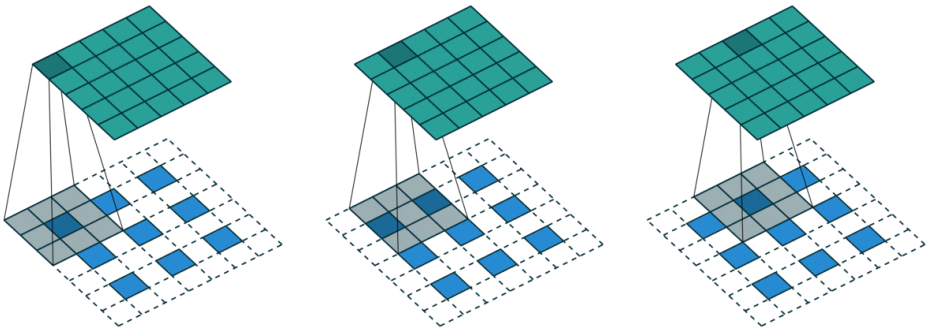

Figure 4.1: Fractionally strided convolution. The bottom matrix represents the input, which has been expanded by placing rows and columns of zeros in between each pixel. The top matrix represents the output. Figure from deeplearning.net

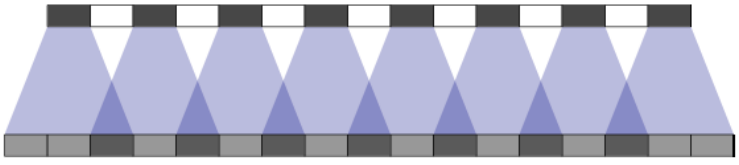

Figure 4.2: Visualization of checkerboard effect resulting from deconvolution with stride length $\frac{1}{2}$ and filter size 3. Figure from Odena et al. (2016)

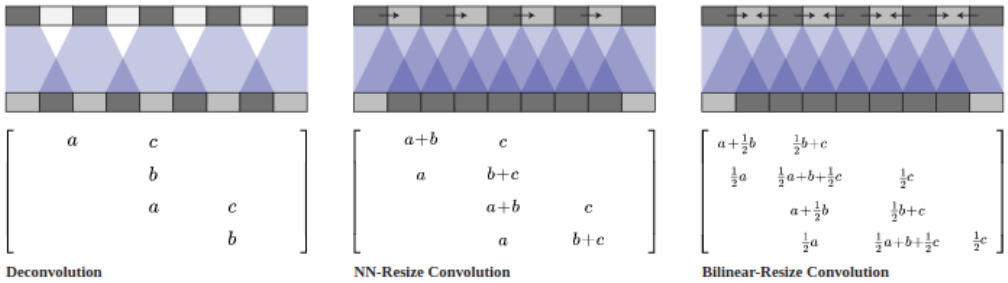

Figure 4.3: Figure from Odena et al. (2016)

division of the reciprocal stride (2) and the filter size (3) results in uneven overlap. When the smaller image is upsampled, some spots in the output image get "seen" twice, resulting in those spots getting more "paint" than other spots. While neural networks could theoretically choose weights that avoid this problem, in practice it is difficult for them to do so (Odena et al., 2016).

Instead of using deconvolution, we can use nearest neighbor or bilinear upsampling followed by a same convolution. This allows us to use the same filter size as the previous network while eliminating the existence of checkerboard artifacts.

Nearest neighbor upsampling simply repeats each row and column in the smaller image over again to get a larger image. For example, to get an image twice as large, simply repeat each row twice and each column twice. This, however, results in blocky, pixelated looking images.

Bilinear upsampling uses bilinear interpolation to add additional pixels. Bilinear interpolation takes an average of the pixels to either side of it, then takes an average of the pixels on top and below. Because the interpolation is done sequentially, the new pixel is a quadratic function of the pixels around it.

In our updated network, we will use bilinear upsampling.

4.2 Instance Normalization

In the JohnsonNet architecture, increasing or decreasing the contrast of the content image can result in a radically different stylized image. Intuitively,

this should not be true. As contrast is a substantial marker of style, the contrast of the output image should ideally align closely to the contrast of the style image. One approach introduced by Ulyanov et al. (2016) to solve problems relating to contrast is to replace batch normalization layers with instance normalization layers. Batch normalization, as stated earlier, limits the internal covariate shift inside a neural network. Let x_{tijk} be the (j,k)th pixel in the ith filter of the tth image in the batch. Then batch normalization is given by

$$y_{tijk} = \frac{x_{tijk} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}},\ \mu_i = \frac{1}{HWT}\sum_{t=1}^{T}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm},\ \sigma_i^2 = \frac{1}{HWT}\sum_{t=1}^{T}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm}-\mu_i)^2 \tag{4.1}$$

This normalizes the contrast across the batch of images. In order to normalize the contrast of each instance (in this case, each image), we can modify the definition slightly:

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}},\ \mu_{ti} = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm},\ \sigma_{ti}^2 = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm} - \mu_{ti})^2 \tag{4.2}$$

This modification significantly increases the performance of style transfer networks. Figure 4.4 demonstrates the effect of instance normalization. This change results in images that seem to better preserve style and content.

Figure 4.4: Images generated from JohnsonNet architecture using batch normalization (right) compared to instance normalization (left). Images from Ulyanov et al. (2016)

4.3 L1 Loss

While L2 loss functions are commonly used in image processing, they do not correlate well with human perception of image quality. The human visual system is sensitive to luminance and color variations in texture-less regions. However, L2 loss functions penalize larger errors and are more tolerant to splotchy regions (Zhao et al., 2017).

Figure 4.5: Figure from Zhao et al. (2017)

Replacing the L2 norm with the L1 norm in the style and content loss function of our neural network results in fewer splotchy regions. Although more advanced loss functions such as Structural Similarity Index (SSIM) can be used, L1 loss functions are available in most neural network software packages and provide similar performance.

Neural networks trained with L2 loss functions are less resilient to outliers. A network trained with an L2 loss function will avoid larger deviations at the cost of smaller ones. On the other hand, L1 loss will adjust to deviations more smoothly. This results in fewer splotchy regions and lines up closer with human perception of image quality.

Figure 4.5 demonstrates this idea applied to an image transformation network trained to denoise images. L2 loss results in splotchy regions, while L1 loss results in smoother images. Although the task of image denoising is different than reconstructing an image from feature representations, the same principles apply.

Chapter 5

Results

I implemented neural style transfer in PyTorch using open source code from PyTorch's github repository. The code was modified to incorporate the changes detailed in Chapter 4. Each model was trained for one epoch, or one pass, through the Microsoft Common Objects in Context ("COCO") dataset which contains 118,000 images. Training took place on an Nvidia 1080TI, taking around 1.5 hours per epoch.

Four main hyperparameters were tuned: ratio of style to content, use of instance normalization, loss function, and total variation loss. Each model was trained using Starry Night as the style image. The image of the Golden Gate Bridge was used for comparison purposes, although each model can stylize any photograph.

Figure 5.1: Starry Night by Van Gogh (style image, left), and the Golden Gate Bridge (content image for comparison, right)

Figure 5.2: Batch normalization (left) compared to instance normalization (right)

Figure 5.2 shows the effect of batch normalization compared to instance normalization. Both models are trained with the same hyperparameters, but the batch normalization layers in each residual block are swapped for instance normalization layers. Instance normalization results in a subjectively better image. This may be due to the large difference in contrast between the content and style images.

Figure 5.3: L2 loss (left) compared to L1 loss (right)

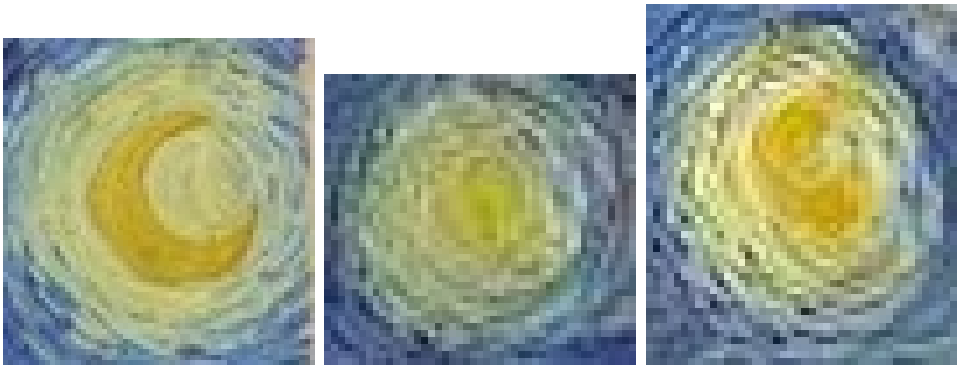

Figure 5.4: Starry Night (left), L2 Loss (middle), L1 Loss (right)

Figures 5.3 and 5.4 demonstrate the effect of training the model using an L1 loss function compared to an L2 loss function. The L2 norm in the content and style loss function is replaced with an L1 loss function. Next, each model is trained using the same dataset and hyperparameters. The effect of using L1 loss is especially prominent in smaller areas of the image. L1 loss seems to preserve the moon shapes within the spirals better than L2 loss. This can be explained by the fact that L1 loss penalizes smaller 'deviations' more uniformly while L2 loss tends to penalize larger deviations much more (quadratic).

Figure 5.5: Weight of total variation loss from left to right: 1, 1/10, 1/100, 1/1000

Figure 5.5 demonstrates the effect of the total variation loss. Total variation increases the smoothness of the stylized image. When total variation loss is high, the image is roughly the average color of the style image. As total variation loss decreases, more variation is allowed in the image. A value of 1/1000 was selected.

Figure 5.6: Ratio of style to content from left to right: 100, 500, 1000, 10000

Figure 5.6 shows the effect of increasing the ratio of content to style. With a ratio of 100, the image looks largely similar to the original content image but with shifted colors and an artifact in the upper right corner of the image. This could be due to the algorithm attempting to fit all of the "style" in one part of the image. As the ratio increases, these artifacts go away. As the ratio further increases, the image starts to resemble the style image. A ratio of 500 was selected. This number will vary depending on the style image chosen. Certain style images will need higher or lower ratios to create subjectively high quality pastiches.

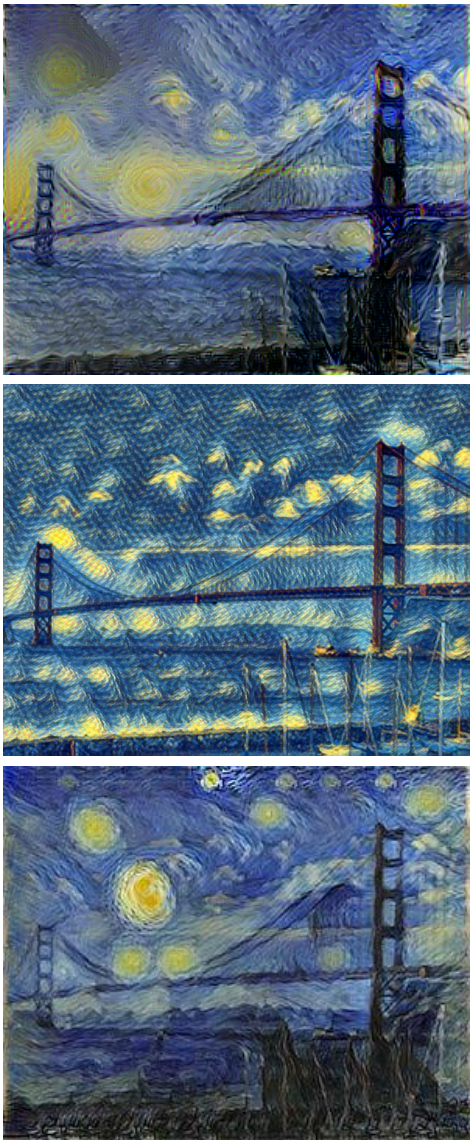

Figure 5.7: Method from Gatys et al. (2015b) (top), Johnson et al. (2016) (middle), my implementation (bottom)

Figure 5.7 shows a comparison of three different methods for neural style transfer: Algorithm 1 from Gatys et al. (2015a), the image transformation network from Johnson et al. (2016), and my modified image transformation network.

Bibliography

Cord, M. (2016). Deep cnn and weak supervision learning for visual recognition.

Dumoulin, V., Shlens, J., and Kudlur, M. (2016). Supercharging style transfer.

Gatys, L. A., Ecker, A. S., and Bethge, M. (2015a). A neural algorithm of artistic style. *CoRR*, abs/1508.06576.

Gatys, L. A., Ecker, A. S., and Bethge, M. (2015b). Texture Synthesis Using Convolutional Neural Networks. pages 1–10.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Gretton, A. (2012). A Kernel Two-Sample Test. *Journal of Machine Learning Research*, 13:723–773.

Hallstrom, E. (2016). Backpropagation from the beginning.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep Residual Learning for Image Recognition. *ArXiv e-prints*.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257.

Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ArXiv e-prints*.

Johnson, J., Alahi, A., and Li, F. (2016). Perceptual losses for real-time style transfer and super-resolution. *CoRR*, abs/1603.08155.

Kawaguchi, K. (2016). Deep Learning without Poor Local Minima. *ArXiv e-prints*.

Li, Y., Wang, N., Liu, J., and Hou, X. (2017). Demystifying neural style transfer. *IJCAI International Joint Conference on Artificial Intelligence*, pages 2230–2236.

Mahendran, A. and Vedaldi, A. (2015). Understanding deep image representations by inverting them. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June-2015:5188–5196.

Mazur, M. (2015). A step by step backpropagation example.

Nielsen, M. A. (2015). Neural networks and deep learning.

Odena, A., Dumoulin, V., and Olah, C. (2016). Deconvolution and checkerboard artifacts. *Distill*.

Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

Raschka, S. (2018). Gradient descent and stochastic gradient descent.

Sharif Razavian, A., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). CNN Features off-the-shelf: an Astounding Baseline for Recognition. *ArXiv e-prints*.

Ulyanov, D., Vedaldi, A., and Lempitsky, V. (2016). Instance Normalization: The Missing Ingredient for Fast Stylization. *ArXiv e-prints*.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? *ArXiv e-prints*.

Zhao, H., Gallo, O., Frosio, I., and Kautz, J. (2017). Loss functions for image restoration with neural networks. *IEEE Transactions on Computational Imaging*, 3(1):47–57.