University of Prince Edward Island

Honours Thesis

Fashion Forward-Propagation: Machine Learning Tackles Fashion Curation

Hailey LeClair

supervised by
Dr. Andrew Godbout

November 12, 2017

Contents

List of Figures

Chapter 1

Introduction

Imagine an up and coming clothing designer with a small boutique. They have a small inventory of about 100 different items, and only carry a few different types of items. Say dresses, shirts, shoes, and jeans. The designer wants to provide their customers with an easy way to find the clothing they make. To do this, they want to use an app in which the user can take a photo of any clothing item, and if the boutique has an item that looks similar, that item will be shown to the customer. Is this feasible for a small boutique with minimal technical resources? Can a computer that has five images of different dresses take another image of one of these dresses and find its match? This paper will attempt at answering these questions by exploring how an app like this might be possible using convolutional neural networks, and how a designer with minimal technical knowledge could use this sort of app.

To look at this problem on a very high level, let us assume we have already created an app for these purposes. The designer would need to provide the app with several images of each of their items, so that this app, can learn things about these items within the image. As humans, we can easily distinguish the defining features of an object by looking at the object itself, or by looking at an image of the object. Without turning this into a biology paper, the neurons in our brain that are connected to our eyes are very intelligent and complex, and can recognize certain features of objects and objects themselves[11]. For example, in two different images, we can easily recognize a similar red dress on two different women with different body shapes, in different positions, with different scenery around them in the images. For a computer, it is not so easy.

1.1 Images

To understand how it might be possible for a computer to recognize a dress in an image we must first understand how an image is represented on a computer. Computers see images as two dimensional data structures of pixels, as in the image on the left of Figure 1. Each pixel has a numeric value. In this simple image, each pixel is either 0 or 1, representing either white or black pixels, respectively. A greyscale image can similarly be represented, where the value is a number between 0 and 255, but larger, and images in colour are obviously a

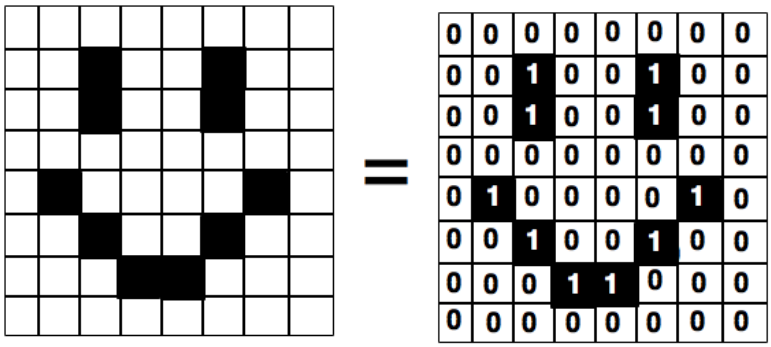

Figure 1.1: A simple black and white image represented by pixel values 0 and 1

lot more complicated than this trivial example. They often have hundreds or thousands of pixels and each pixel is represented by an RGB (red, green, and blue) value. RGB values are a set of 3 values each of which is a number from 0 to 255. Because an RGB pixel has 3 values, and an image is already a two dimensional structure, RGB images are seen as 3 dimensional by a computer. For this reason, images take more processing power than many other types of data. [15].

The field of computer vision emerged in the early 1970's as an attempt to get a computer to process and view digital images in the same way our brain and eyes do. In 1966, a professor at MIT asked a student to spend the summer with a camera connected to a computer, and to get the computer to explain what it was seeing through the camera. This kind of experiment was bound to fail with the state of technology in 1966, but led to much more research into how images can be represented digitally, how scenes are perceived visually, and how objects are recognized in a scene or image. Much of this research was done using 3D mathematical modelling, algorithms for digital image processing, and image and scene analysis. [15]

1.1.1 Objects in Images

Since the idea here is to take an image, classify it, and find its match amongst a set of similar photos, this means that we need to be able to recognize the object in another context in another image, which is exactly what is done in the field of computer vision. Rigid, articulated, and deformable objects all need to be considered when doing object detection, recognition, or classification in images. In terms of physics, a rigid object is a body in which "the distance between any two given points on [the] rigid body remains constant in time regardless of external forces exerted on it "[13]. In terms of images, this means that a rigid object will always have the same dimensions, or some translation of the same dimensions, in any given photo taken of this object. There exist many methods discovered within the field of computer vision to recognize rigid objects in an image, one of which is Scale Invariant Feature Transform (SIFT)[4].

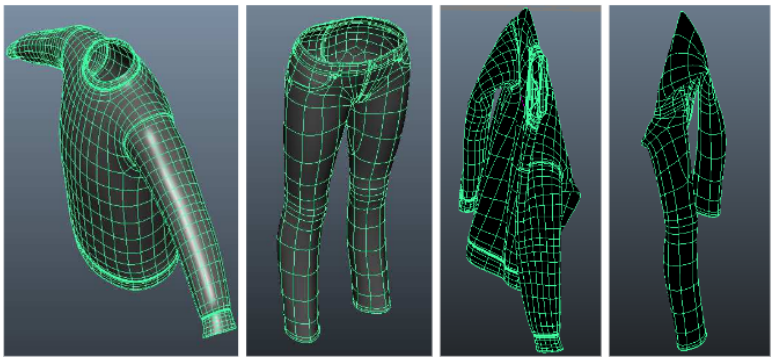

Figure 1.2: Clothing shown as a Deformable Object

SIFT can recognize any objects which do not vary, based on image rotation, scaling, or translation (rigid objects). It also does well with different kinds of lighting and illumination, as well as 3D projection[4]. The "key" to SIFT, is something called "key localization". Key localization chooses "key locations at maxima and minima of a difference of Gaussian function applied in scale space."[4]. The term "scale space" is used here, because any scaling of the image will not effect the locations selected during this process. Without delving too deeply into SIFT, this process is possible by building image pyramids at different levels of scaling. These pyramids make the locations very stable, which aids greatly in recognizing objects in an image.[4].

This is a step in the right direction in the attempt to recognizing clothing in images. Theoretically, if we already have something in place that can take an image and tell us what category of clothing an object belongs to, we can then use these methods like SIFT to match rigid objects like shoes. It is tremendously useful that we have access to methods, but is the same type of object detection possible for clothing that changes shape because of the articulated nature of the human body? Can we recognize a deformable dress that looks completely different on the hanger than it does while being worn? To attempt to answer these questions, we need to look at the possibilities surrounding these difficult types of objects.

Articulated objects may have rigid parts, but like human body parts, are connected at a basic joint which can move[15]. Multiple images of the same articulated object may show the object in different formations or positions. Although parts of these objects are rigid, possible movements from the joints mean that many different positions are possible for one articulated object. Think of all the possibilities of position for a human arm. An arm can be up in the air, straight out to the side, hanging down by the side of the body, bent at the elbow, behind the back, just to list a few. Even though articulated objects are rigid to a point, or, a joint, the degrees of freedom involved with these objects mean they are much more difficult to detect in images.

A deformable object, in contrast with the others, "changes its shape and/or volume while being acted upon by any kind of external force"[12]. Depending on the degree to which an object deforms, or the amount of force applied, many images of an object could

show it as having completely different dimensions and shape. As with articulated objects, the uncertainty of the general shape or dimensions makes deformable objects especially challenging. This problem, and the problem with articulated objects is particularly relevant with clothing.

With some exceptions (like shoes, jewelry, some accessories, for example), many everyday clothing items are deformable to some degree. For instance, a pair of skinny jeans may have a general shape when lifeless on a hanger, but take on very different dimensions and possibly even a totally different shape when being worn by different people. Like these jeans, of course, many other clothing items will take on the shape of the person wearing them. An example of this is shown in figure 2.[**?**] In terms of shape and dimension of these two clothing items being worn and not being worn, there are barely any similarities. We can also consider changes in an environment that could change the shape of an object. If it is a particularly windy day, a long flowing dress may take on a completely different shape in the wind.

As mentioned earlier, clothing items are typically being worn by someone in images. This simple fact adds one extra dimension to the recognition of any given clothing item, because the human body is an articulated body. A loose denim jacket, which may have a similar shape on anyone who wears it, can be easily deformed by placing the arms in a different position. Different articulations of the joints in the human body (especially our knees, shoulders, and elbows) can dramatically affect the shape and dimensions of a piece of clothing, and this carries over to the digital image. These obstacles add a layer to object recognition in clothing items, which would not be present in recognizing various other types of objects in images. Unfortunately, we cannot rely on something like SIFT[4] to recognize these sorts of items in an image, as it struggles to detect these nuances.

Along with the problem of clothing being deformable and worn on an articulated body, we also have to consider the other contextual aspects of an image. As intelligent humans, we can normally determine if a dress in an image is the same as another dress in another image, regardless of what other objects are in the background. When it comes to digital recognition, though, an image containing an explicit clothing item could be convoluted with other objects or people in the same image, or could have an explicit, plain background which becomes a defining feature of an image. When introduced to another image of the same object with a similarly noisy background, a computer may then think this is a different object altogether, because of the dramatic difference in backgrounds between this photo and the previous one(s).

1.2 Other Difficulties in Object Recognition in images

Figure 3 shows the same nike shoe in two different images. We can even speculate here that the photo has been taken from almost the same angle in both images. With a quick glance at both of these images, we can confidently say that the shoes in these images are in fact the same, but as learned by many in a first year computer science course: a computer does exactly what you tell it to do. This is relevant because unless a computer knows to look for a background in an image, it will assume these two images are of completely different objects.

Figure 1.3: Two photos of Nike Rosch One sneaker. One being worn by a person on cobblestone, and one with a white background

The angle that a photo was taken and the location of a particular item in the image are also important considerations. All known images of an object could be front-facing, and if a side view of this object is introduced, a computer will have nothing to compare with this side view. In this event, the computer will not be able to correctly classify or recognize the object, since it has no previous knowledge of its side view.

When worn, clothing items like shoes and hats are almost always near the bottom and top of an image, respectively (for obvious reasons). If a computer has only seen images of shoes in which they are located near the bottom edge of an image, the computer may think that this location is a common, fundamental characteristic of shoes themselves! This must be considered when choosing images for a computer to classify or match. A shoe, of course, can be anywhere in an image, and recognizing this when choosing images lets the focus be on features of the shoes themselves, rather than the object location.

Even if articulation, deformability, angle, and rotation have all been accounted for when attempting to classify and recognize objects, colour and illumination must also be added to this long list of considerations. Depending on the lighting conditions and the camera itself, the same red shirt may look pink in one image and orange in another (in fact, there was a viral internet sensation over a blue and black, or white and gold dress, a few years ago, due to this very perceptual issue). An image must be recognizable under different lighting conditions and in light of minor variations in the object's colour.

Yet, what if an image of this shirt is in greyscale? This is an important question. If the goal is to recognize a red shirt in a given image, greyscale images cannot be considered. If they are considered, they must be accepted as either grey or colourless, even if the shirt in the image is actually red. In this case, the greyscale image is irrelevant. On the other hand, If the goal is to recognize a generic T-shirt, colour becomes unimportant, and a greyscale image can be as relevant as an RGB image. When it comes to recognizing clothing in images, it is naturally almost impossible to account for every possible variant from amongst these images. However, if we keep all of these degrees in freedom in mind when attempting to classify or match images, we can avoid many of the problems that come with such variations.

Chapter 2

Filters and Feature Maps

Where do we start in terms of finding similarities in different images? One approach, and a good approach, is to find and identify specific features of an object which are common to almost every image of the object. Viola and Jones did this in 2004, with Real-Time Robust Face Detection. They classified faces in images, based on the computed values of simple features specific to the faces.[6]. They found that simple rectangles (figure of Viola and Jones rectangles) could locate areas of brightness in a photo. If these specific areas of brightness are present, then the image contains a face.

For our purposes, Viola and Jones' specific facial recognition, by design, only works with images of faces. Naturally, though, one would imagine that a similar process to this facial recognition could be used in attempting to recognize other objects in images, like clothing. In theory, we should be able to find prominent features that belong to any given type of clothing item, and use this feature to consistently classify the image.

As stated above, in images, we count on an item to have one or more identifying features, like a human face. One of the ways that we can find the parts in an image that are likely to be important, and make them more prominent, is by applying a filter to the image.

A common example of a filter is the use of a neighbourhood operator, or local operator on pixels in an image. A neighbourhood operator uses an area of pixels around a specific pixel in an image to compute an output value for that pixel. When applied to all pixels across the image, the operator "filters" the image. A neighbourhood operator takes a weighted sum of all pixels in a neighbourhood. It is also known as a correlation, where h(k, l) are labelled the filter coefficients.[15]

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l) \tag{2.1}$$

More relevant for the purposes of this paper is the convolutional operator. Convolution is a variation on correlation(filtering) and is both commutative and associative. It is a reversal of the sign of the offsets of the correlation function g = f * h where h is now an impulse response function.[15]. To explain this not just in formulas, but with respect to pixels in images, a convolution on a single pixel is done by taking a weighted sum of the pixels in the neighbourhood of this pixel. The weights, and size of the neighbourhood depend

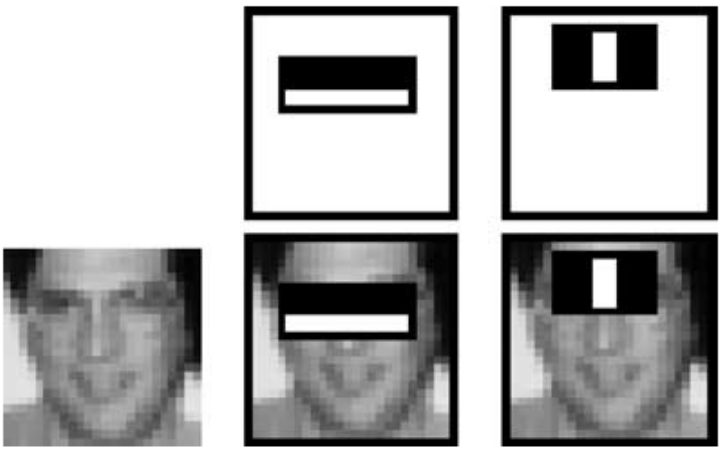

Figure 2.1: Image from Viola and Jones well known paper, Robust Real-Time Face Detection[6]

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 2.2: Sobel Operator: Convolution Kernels G_x and G_y [5]

on the convolution kernel[1].

$$g(i,j) = \sum_{k,l} f(k,l)h(i+k, j+l) \tag{2.2}$$

A convolution kernel can be any size, but is usually quite small. Since we only want to use pixels within a certain neighbourhood around a pixel, this makes sense. A convolution kernel too small will not consider any pixels around it, and too large will end up loosing the importance of the pixel itself. A common convolutional operator is the sobel operator, whose 2 convolution kernels are shown in figure 5. This filter is used for edge detection and is very helpful in finding shapes, objects, and defining features in images. Two 3x3 convolution kernels are used to perform this edge detection[5]. On their own, $G_x$highlights vertical lines in an image and G_y highlights horizontal lines in an image. To apply one convolution kernel across an image, for instance G_x, we need to inversely apply the weights in the kernel around

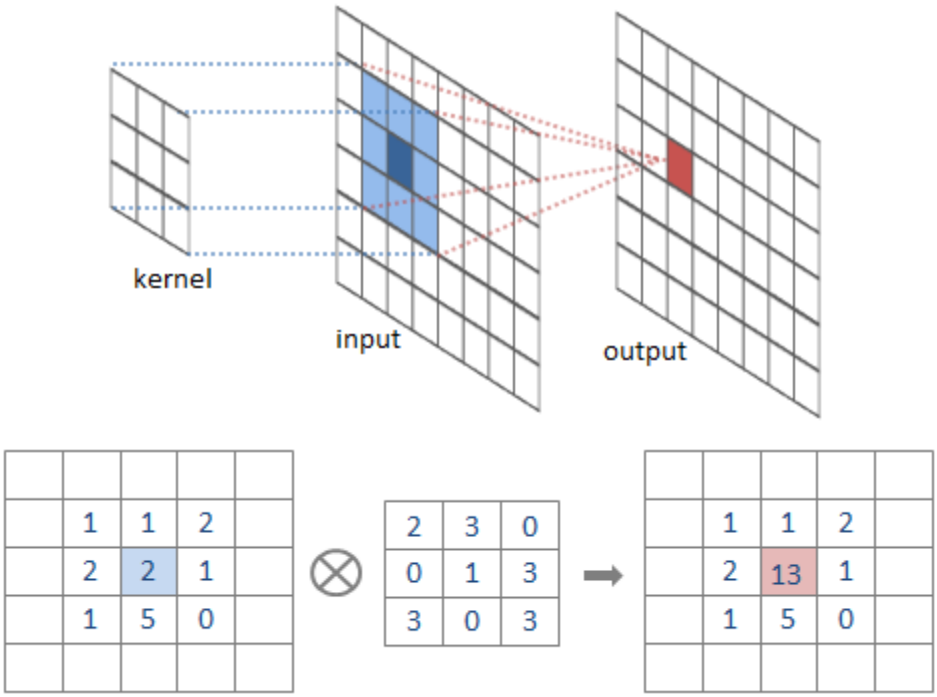

Figure 2.3: A Convolution kernel being applied to an image [1]

the neighbourhood of a given pixel (a convolution) and take the sum of these values. In the case of Sobel's G_x kernel, the pixel in question will be at (2, 2) or in the middle of the kernel. This weighted sum then becomes the value for that pixel. This is done across all pixels in an image[1]. An example of this can be seen in figure 6. The image on the top shows a high level view of a convolution being applied, and underneath shows a convolution kernel being applied to a set of pixels in the neighbourhood of a number 2 (in the middle of the 3x3 section of pixels).

What is important to note is that this operation cannot be done on the pixels around the edges or corners of an image, because their neighbourhood is limited. To deal with this problem, often zero padding is added, which adds a border of pixels all with the value 0 around the image. In this way, the original edge pixels in an image can then be part of the convolution. [**?**, **?**, **?**] Another approach is to avoid applying the convolution kernel to the outer pixels, and in this case these pixels will essentially be thrown away after the convolution. This should only be done if one is assuming that the pixels around the edge of the image are unimportant, which is often the case.

After applying a convolution operator, correlation operator, or any type of filter to an image, we are left with a feature map. A feature map is essentially a filtered image. Since each filter highlights a certain feature, or set of features in the image, a feature map can provide us with a lot of information about what is in an image, and what is important in an image[11]. Having multiple feature maps, which can show us if images have a number of different features, these can then be used to help us learn things about an image.

As outlined above, filters can alter an image in dramatic ways. They can be used for edge detection, to blur photos, to adjust colour, or to sharpen images.[15]. Filters can play a crucial part in detecting any object in an image, as a feature map helps a computer to recognize certain features in an image. Even so, to manually find which features are important and which filters or convolution kernels work for each object is not always a feasible task. Filters are helpful in discovering features in an image and making them more prominent, but when dealing with 1000 or more images, how do we know which filters to use and what features are important?

Chapter 3

Machine Learning

Machine Learning can help us "learn" things about these images, namely which filters or convolution kernels work best, without having to manually go through and picking filters for each clothing type. Simply put, Machine learning is programming a computer to learn something based on the data itself. People use machine learning to analyze and monitor data much more efficiently than executing it manually. It recognizes patterns, and can be used to learn what people are buying, which emails in your inbox are spam, and in this case, if a pair of jeans is in an image, and indeed which pair of jeans it is. [11].

3.1 Training Sets and Classifiers

Using machine learning algorithms, we can program a computer to learn about almost anything as long as we provide it with enough data, and the right data. For instance, if we want a computer to use machine learning to "learn" if a pair of shoes are present in an image, we have to make sure that we provide it with many images of many different kinds and brands of shoes, in different positions and orientations. These images are called our training set.[11]. The images in the training set are input into a machine learning algorithm called a "classifier"[14]. The algorithm then goes through the entire set of images and looks for similar features among all or most of the images.

Choosing a training set can be difficult, to properly train a machine learning classifier we need (as stated earlier) a lot of the right data. That might be impossible for a small boutique with only 100 items in total. Because most people do not have a database with thousands of photos, there must be some resource providing photos to take advantage. And there is. Imagenet [**?**]Imagenet) is a website that gives access to millions of pre-classified images, 14,197,122 at this moment, to be exact. An entire database of images of a specific classification can be downloaded, or all images can be accessed by urls. Something like this will be useful for the clothing designer in question. Instead of having to personally take thousands of photographs to have a decent sized training set for the clothing items that this designer has, Imagenet can be used to easily get whatever images are needed, whenever they are needed. Figure 7 shows a screen shot of imagenet in which 1227 images are available

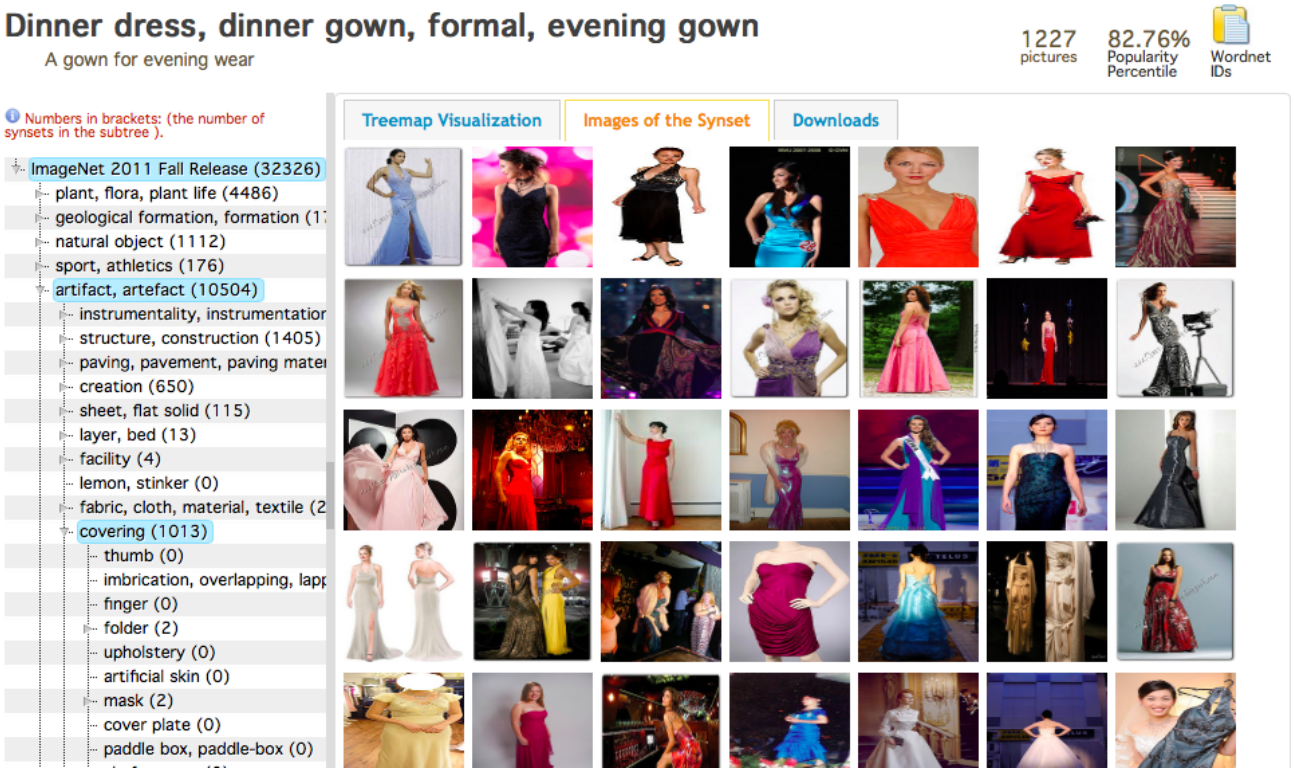

Figure 3.1: What is returned from Imagenet after searching for dress: http://image-net.org/synset?wnid=n03201638

and were classified as "Dinner dress, dinner gown, formal, evening gown". If this clothing designer specializes in something like fancy dresses, they can take these images as part of their training set that represents fancy dresses, or any of the labels previously mentioned. Even with the tremendous help of Imagenet[**?**]Imagenet) providing a good training set, we need a classifier to recognize (or correctly classify) a clothing item in an image. Once a classifier is trained, it gives us a probability that the image belongs to each of the classes it has been trained on. Whichever probability is the highest, is the predicted class of an item[14]. If the designer with a boutique has photos of their clothing ready on hand, the theoretical app and classifier should be able to take images of all of the designer's clothes, and give each image a label, or class. Before being able to do this though, the classifier needs to learn the way in which it will get the best results with the data (in this case, images) available. This data is how a machine learning algorithm will be trained, hence the name "training set". After being trained, an algorithm should be able to predict the class of an object in an image, but only for objects that it has already been trained on. In other words, as long as the classifier was trained on dresses, the predicted class, or label of a photo of a dress, should be one that refers to a dress.

In this case, supervised learning will be used. With supervised learning, the algorithm

already knows the class, or label, that each specific data element in the training set belongs to, and makes use of this for training purposes. If a classifier predicts that a data element belongs to the wrong class during the training phase, supervised learning lets the algorithm run again with the same data to attempt to correct its' previous mistakes[11]. Thus, a classifier learns by making predictions on the same training set continuously until it can accurately predict the classes of all or most data elements in the training set.

How can supervised learning help in predicting which clothing item is in an image? Since we do not have to get everything right the first time with supervised learning, we can use a method that makes mistakes each time (or each epoch) that it iterates through our dataset. When a classifier is in its' training phase, an epoch refers to one iteration through the training data. Depending on the way in which it is trained and with which and how much data, a classifier may need many many training epochs to be properly trained. The number of training epochs needed will depend on the type of classifier (neural network, nearest neighbour, SVM, etc), the number of possible classes, and the data. So how can we tell when a classifier is properly trained?

To measure how well a classifier has been trained we use a testing set of data (again, which in this case will be images) in which all elements are different from those in the training set, the classifier will iterate through the testing set. It will keep track of the actual prediction it makes for the element, and the correct prediction. Calculating the error of the previous epoch, and going back to fix the mistakes previously made by the classifier. There must be a type of classifier that make use of this idea.

Chapter 4

Neural Networks

Artificial Neural networks are an effective classifier for any data set that needs to be learned, but that will need several (or many) training epochs. The type of neural networks that we're interested in use deep learning which is a subfield of machine learning and Artifical Intelligence. Deep learning uses big data sets and deep neural networks with many hidden layers and neurons[2]. Like most machine learning algorithms (classifiers) neural networks take one data element as input (one image in this case), and outputs a probability that this element belongs to each class in a list of classes. What is unique about neural networks is that they are inspired by the human brain. Neural networks are made up of a series of layers, and each layer is made up of one of one or more neurons, like the neurons in a human brain. Every neural network has an input layer made up of input neurons, an output layer made up of output neurons, and 1 or more hidden layers composed of hidden neurons.

For the purpose of explanation, an example will be given as shown in figure 8, with only one hidden layer. There are three input neurons, or signals: x_1, x_2, and x_3. Each neuron in the input layer is connected to every neuron in first hidden layer. As can be seen in figure 8, every neuron in one layer is connected to every neuron in the next layer. Each one of these connections has its' own unique weight (weights are important) and a neuron in the first hidden layer takes a weighted sum of all input neurons. This weighted sum is then given to an activation function in each hidden layer. The output of this activation function is the output value of this hidden neuron. This is done for each neuron in the first hidden layer. If there are more hidden layers, the output of the previous hidden layer's neurons are fed into the neurons in these layers as input. This continues until the output layer is reached, where each output neuron represents a specific class and calculates the probability that the input belongs to this class.[14]

4.1 Activation Functions

First of all, every neuron in a neural network except for the input neurons, put their weighted sum of the previous layer into an activation function. An activation function takes this weighted sum mentioned earlier, and adds a bias. Equation 3.1 shows this as the output of

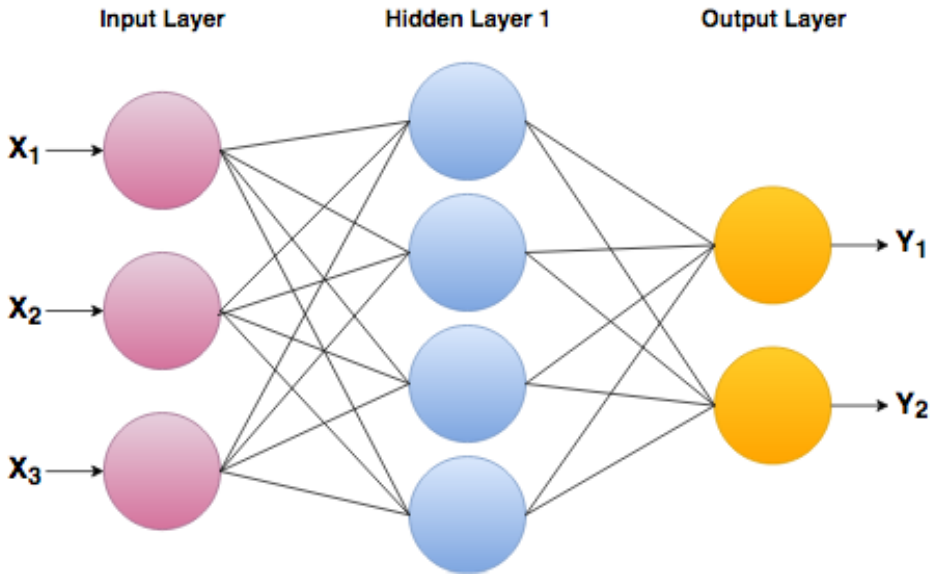

Figure 4.1: A simple neural network with 3 input neurons, 1 hidden layer with 4 neurons, and 2 output neurons

any neuron Y. The result of this function determines whether it should activate, or "fire" that neuron. The way in which a neuron uses the result of this function to decide whether it should activate is by declaring a threshold. If Y is over (greater than) this threshold then the neuron fires, if not, the neuron does not fire. This means that if a particular neuron does not fire, it does not contribute to the final output of the network (the classification)[10].

$$Y = \sum(weight * input) + bias \tag{4.1}$$

For a long time, people who worked with neural networks assumed the optimal solution would be one that closely mimics the neurons in a human brain. The activation function used by neurons in the brain is a sigmoid function(Equation 3.2)[14]. For many years this was the most commonly used activation function in artificial neural networks. In recent deep learning developments though, it has been discovered that there are several other functions that perform a lot better than the sigmoid function, depending on the type of deep neural network being used. This will be covered in another section.

$$f(\sum) = \frac{1}{1 + e^{-\Sigma}} \tag{4.2}$$

4.2 Training and Back-propagation

A Neural network has input neurons and output neurons, hidden layers and activation functions, but how does it learn, and how is it supervised? Back-propagation takes the same input through a network to correct any mistakes made in previous predictions (probabilities

of classification). This is where the training set comes in. During each Epoch of a training session, each element in the training set is forward-propagated through the neural network. Since we know the correct classification of all elements in our training set, after each epoch we can compute the error in the classification. This can be done using the weights. Since all neurons contribute independently to the classification of the input, each input or hidden neuron in a particular layer has a different weight to carry it to each neuron in the next layer. All weights should be updated by adjusting for the error in the previous epoch. A neural network should improve at predicting the actual class of its input after every training epoch. The error of an entire training set's classification can be calculated using accuracy, precision, and recall, but is most commonly calculated using accuracy. For binary classification (true or false classification) these calculations are done by using the number of true positives, false positives, true negatives and and false negatives. For non-binary classifications (deep neural networks) it is done a little differently. For neural networks, the error is most commonly calculated with MSE, or Mean Squared Error as shown in equation 3.3, where i is the ideal output (or class) for a training element, and a is the actual class, or what the neural network predicted for this element during this training epoch[7].

$$MSE = \frac{\sum_{i=1}^{n}(i_n - a_n)^2}{n} \tag{4.3}$$

Each one of these calculations can tell us how close the network was to making a correct prediction. All of the weights present in a network are adjusted after each epoch based on how much it contributes to the correct classification (its' error). Weights are usually initialized randomly, or randomly within a certain set of values, and then get udated accordingly during training[7]. Once a neural network is properly trained, the same weights can be used over and over to make predictions. This process of using error to update weights and train a network is called back-propogation and can help a deep neural network learn almost anything.[14]

Chapter 5

Convolutional Neural Networks

5.1 What are convolutional Neural Networks?

Since neural networks can learn almost anything, one would expect that they could learn which objects are in an image, or if an image contains a particular clothing item. Getting back to the theoretical fashion designer, it would be great if there was a type of neural network that could be used to classify images all of their clothing items. This can be done with deep neural networks, but with a bit of an adjustment. A specific type of deep neural network is used for image classification called a convolutional neural network.

Convolutional neural networks have a similar flow to other neural networks. The input for this type of network is all pixel values in an image and the output is again the probability that the input belongs to a certain class (for instance, the class "dress"). In this case the output will be the probability that the input image contains a certain object. The hidden layers are where CNNs get more complicated. There are a few possible types of hidden layers. The first and probably most important to the network is a convolutional layer. A convolutional layer applies a number of convolutions to an image each return a feature map[11].

5.2 Convolutional Layers & ReLU

Convolutional layers try to mimic the way that our eyes perceive what is around us. When we look at something, the neurons in our visual cortex are only really focused on a small area of the visual field called the local receptive field. Each neuron has a different local receptive field, although they may overlap. These local receptive fields of specific neurons, which are how our eyes process what they percieve, can be similar to filters in many ways. Some neurons have larger local receptive fields than others, whereas some can focus on and react to lines or other shapes. This is exactly what a convolutional layer attemps to do.[11]

The first convolutional layer will not have every input neuron connected to every other neuron in the layer like we see in the hidden layers of some other neural networks. Each neuron in the first convolutional layer is connected to a certain number of neurons in the

input image which are referred to as the neuron's local receptive field. The weights applied here are similar to a small image in equal size to the local receptive field of a neuron. This small image of weights that are applied to an image is a convolution kernel. A convolution happens here when this kernel is applied to all neurons in the local receptive field of the current neuron. This process is essentially a convolution when applied throughout an entire image. This process of convolution on local receptive fields from a previous layer, helps the network focus on low level features in the beginning stages of forward propogation through the network, and higher level features near the end of the network.[11]

When all of these convolutions are applied in a convolutional layer, we are left with feature maps (or filtered images) of the previous layer. Because these layers return a number of feature maps, the network learns which features are best at predicting the class of a particular image using back-propogation. A feature which helps in predicting the class will be more heavily weighted in the network. whereas a feature which does not contribute to a prediction will be minimally weighted.

Since neural networks apply an activation function after all data has entered a neuron, we would expect CNNs to do the same. There are several different activation functions that a CNN can use on the image data of a layer, but the most common one is the ReLU function. As mentioned in a previous section, in biological neurons, the sigmoid function gives the best learning results, but with deep neural networks, there are other activation functions that give better results and speed up the training time.[8]

For CNNs and some other deep neural networks, The ReLU (Rectified Linear Unit) function is preferred over other functions like sigmoid, because it does not saturate for positive values (sigmoid does) and gives 0 when any input is negative. Since the output of some neurons will undoubtably be 0 after computing ReLU, we can assume that many neurons will end up dropping out, or having no effect on the classification. This is okay for our purposes, but there are some good variations on ReLU that make sure neurons do not get killed or left out during the rest of the training process. [11]

$$(x) = max(0, x) \tag{5.1}$$

One variation on ReLU activation is LeakyReLU. It uses a hyperparameter (or tuneable parameter) α to determine the "leakage" of the function. In this case, the neuron may become temporarily inactive, but may eventually contribute to the network again, which is an improvement over ReLU. Another variation on ReLU is PReLU (Parametric Leaky ReLU) which takes the hyperparameter in LeakyReLU and makes it a parameter learned with backpropogation during the network's training. These variations perform well when the training set is large, but may overfit on smaller datasets. Since the dataset being considered for this project is quite small, ReLU will be fine for our training purposes. The activation function is important to note here because each convolutional layer in a CNN is usually followed by a layer in which the activation function (ReLU) is applied to all neurons in the previous layer.[11]

$$LeakyReLU_{\alpha}(z) = max(\alpha z, z) \tag{5.2}$$

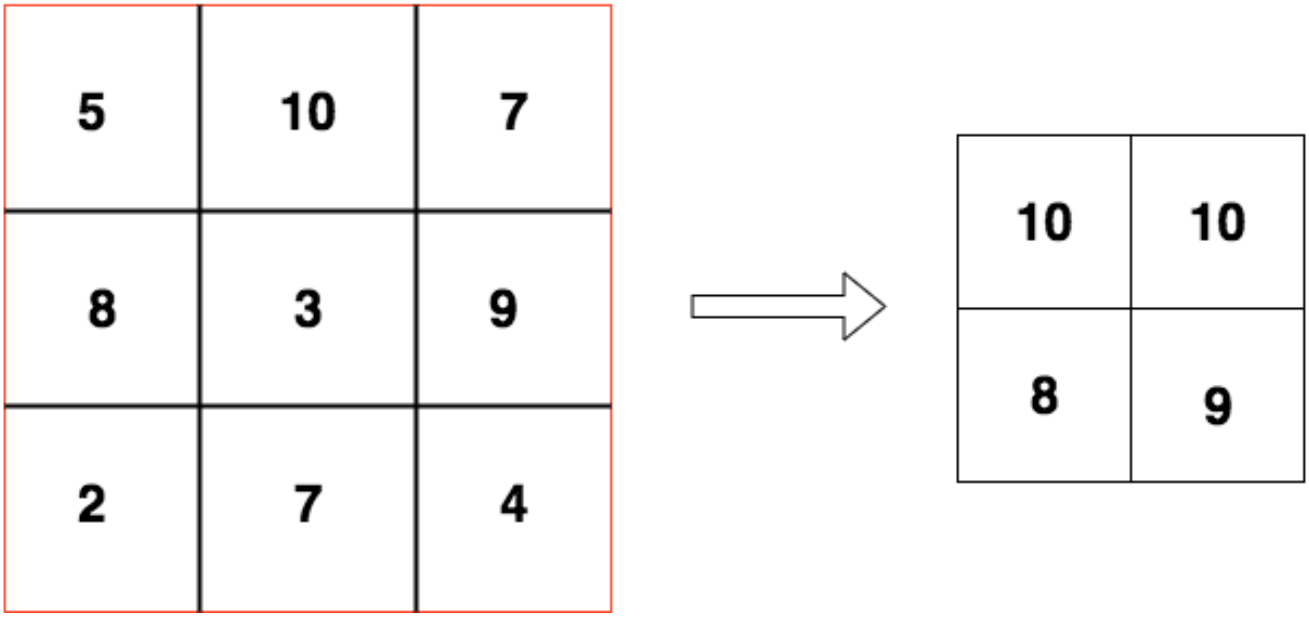

Figure 5.1: max pooling applied to a 3x3 section of pixels using a 2x2 kernel for pooling

5.3 Max Pooling Layer

The next hidden layer commonly used in convolutional neural networks, and other deep neural networks is a pooling layer. A pooling layer is used to summarize a large amount. In the case of images, this means reducing the size of the image, while still keeping certain parts or features of the image.[11]. What these features are will determine what type of pooling layer is used. In most CNNs, max pooling layers are used to keep the most prominent features in an image, which is usually desirable when attempting to recognize objects. Max pooling layers summarize the image data by taking the largest pixel value in a group of pixels and using this to represent the group (shown in figure 4.1) . Doing this across an entire image gives a smaller image with less noise. Max pooling layers are usually in between or after convolutional layers (after ReLU has been computed for the convolutional layer). Throughout the network the classification of an image becomes more clear, so it makes sense that after each convolution, we pool the feature maps for the most prominent pixels, which gets us closer and closer to finding the most important and defining features in an image.

5.3.1 Fully Connected Layers and Architectures

The last pooling layer is usually followed by two or more fully connected layers which again will use ReLU as an activation function. Fully connected layers are actually quite simple and are often placed at the end of the network. They work like the layers in a generic neural network as explained earlier in the paper. Each neuron in a previous layer is connected to every neuron in the fully connected layer, hence the name "fully connected"[3]. These layers help with training by giving the data from the last pooling layer a chance to propagate

Layer	Type
In	Input
C1	Convolution
S2	Max Pooling
C3	Convolution
S4	Max Pooling
C5	Convolution
C6	Convolution
C7	Convolution
F8	Fully Connected
F9	Fully Connected
Out	Fully Connected

Figure 5.2: Simple view of AlexNet architecture inspired by Table 13.2 on page 367[11]

through a few more hidden layers before outputting a prediction. Finally, as we already know, the output layer in a convolutonal neural network is similar to the output layer in most neural networks. Depending on the number of classes, the network returns a probability that the image belongs to each class, and the highest probability is the predicted class. A typical Convolutional neural network has an input layer, a few convolutional layers (each followed by a pooling layer) followed by two or three connected layers. Even though there is a general flow and consensus on which layers go where, the number of layers is dependent on the task itself and is usually discovered by trying different CNN (Convolutonal Neural Network) architectures. There are a few famous CNN architectures. These include LeNet-5, AlexNet, GoogLeNet, and ResNet. All of these architectures and their layers are outlined in the O'Rielly Book: Machine Learning with Scikit-Learn and TensorFlow.[11]. As an example, the AlexNet architecture, which won the 2012 ImageNet ILSVRC challenge and had the least amount of error by nine percent, was developed in 2012. A simple table outlining its' architecture can be seen in figure 5.2

Chapter 6

TensorFlow

We can in fact build something to recognize images of clothing and classify images that a fashion designer takes of their clothing. Convolutional neural networks and the right training and testing set a computer or an app on a phone can be trained to recognize images in photos. This is theoretically sound but how can it be implemented? Implementing a simple neural network that works as a binary classifier is a lot of work even for an experienced programmer. Someone looking to create an app that matches clothing in images may not have the time or experience to create a convolutional neural network from scratch.

6.1 What is TensorFlow?

TensorFlow is a library that allows for building deep neural networks (including CNNs) in Python (which is now also available in C++, Java, and Go). More formally, as defined in the TensorFlow documentation, "TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API."[9]

TensorFlow was originally developed with the intention of being used to build and work with Deep Neural Networks, but can now be used for other types of computation as well. With TensorFlow, defining layers, weights, and training a network is quite straightforward. The TensorFlow API provides methods for all of these things which take parameters specific to the graph or neural network being built. The documentation also provides tutorials on how to use TensorFlow. There are tutorials on building convolutional neural networks which can help tremendously with learning how they are actually implemented. A portion of this tutorial is shown in figure 6.1. It can also be seen that when using TensorFlow, "tensors" are created.[9]

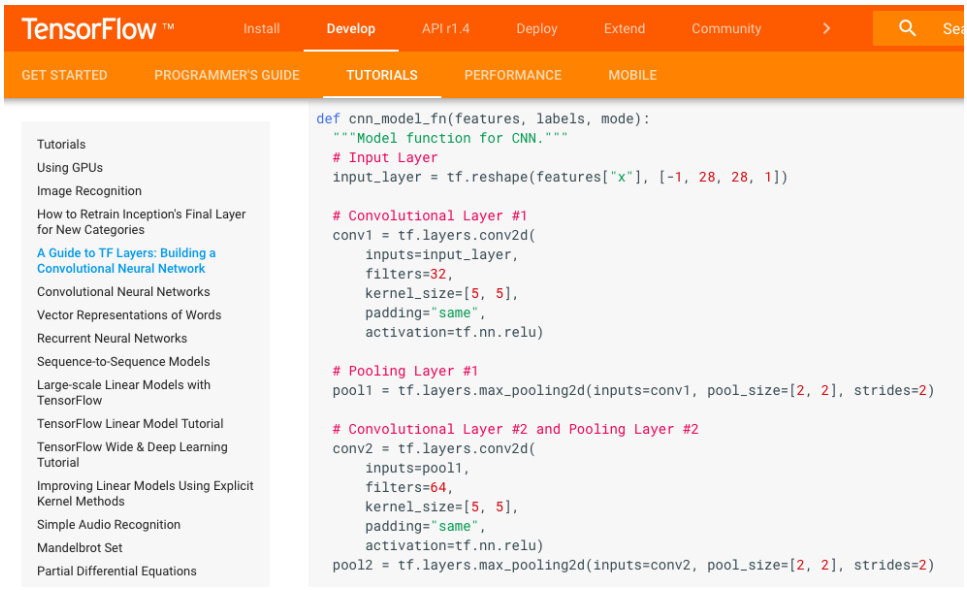

Figure 6.1: A portion of code from a CNN tutorial using TensorFlow[9]

6.2 What is a Tensor

Along with the previous definition taken from the TensorFlow documentation on the home page of TensorFlow's website, it is also defined as: "A framework to define and run computations involving tensors."[9]. Tensors generalize matrices and vectors, possible in higher dimensions. The are implemented as n-dimensional arrays. When using TensorFlow to write a program, a tensor is the object that is used by the rest of the program throughout the duration of the program, and is defined as a tf.Tensor. Usually, when a tensor is instantiated, it is only partially defined, as it may or not yet contain the data that the program will be working with. A tensor can be easily defined with its' two properties, a data type and a shape[9]. The data type will be consistent throughout every element of the array. The shape is the number of dimensions of the tensor, and the size of each of these dimensions. Each dimension contains an array of the given by the shape. There are several types of tensors, some of the most commonly used are Variable, . Tensors also have a rank which is determined by their shape. To see the value of a tensor we can use the .eval() function. As explained in the documentation, the rank of a tensor refers to an entity in math as shown in Figure 6.2. This table makes tensors a lot easier to visualize and understand[9].

6.3 Using TensorFlow to Build a CNN

TensorFlow can be used to build a convolutional neural network with any amount or sequence of layers. A set of images of any size can be used as long as the shape of the tensors are defined properly. After installing and importing TensorFlow, starting with the training phase, some tensors need to be defined to hold the data being worked within the network. For example,

Rank	Math entity
0	Scalar (magnitude only)
1	Vector (magnitude and direction)
2	Matrix (table of numbers)
3	3-Tensor (cube of numbers)
n	n-Tensor (you get the idea)

Rank 0

The following snippet demonstrates creating a few rank 0 variables:

```
mammal = tf.Variable("Elephant", tf.string)
ignition = tf.Variable(451, tf.int16)
floating = tf.Variable(3.14159265359, tf.float64)
its_complicated = tf.Variable((12.3, -4.85), tf.complex64)
```

Rank 1

To create a rank 1 `tf.Tensor` object, you can pass a list of items as the initial value. For example:

```
mystr = tf.Variable(["Hello"], tf.string)
cool_numbers  = tf.Variable([3.14159, 2.71828], tf.float32)
first_primes = tf.Variable([2, 3, 5, 7, 11], tf.int32)
its_very_complicated = tf.Variable([(12.3, -4.85), (7.5, -6.23)], tf.complex64)
```

Higher ranks

A rank 2 `tf.Tensor` object consists of at least one row and at least one column:

```
mymat = tf.Variable([[7],[11]], tf.int16)
myxor = tf.Variable([[False, True],[True, False]], tf.bool)
linear_squares = tf.Variable([[4], [9], [16], [25]], tf.int32)
squarish_squares = tf.Variable([ [4, 9], [16, 25] ], tf.int32)
rank_of_squares = tf.rank(squarish_squares)
mymatC = tf.Variable([[7],[11]], tf.int32)
```

Figure 6.2: Tensor ranks and the mathematical entities they represent followed by examples of tensors of different ranks.[9]

for a training set of 200 images, if all images in the training set are known to be 100x100 pixels and all images are RGB, we can define a tensor holding the input to a CNN as a tensor of type tf.float32. This means the pixel data of each image will be represented as a 3D array where each pixel is an element of a 2-D array and is represented a three element array with the RGB values. The definition of this tensor is shown in the following four lines of Python code.

```
batch_size = 200
img_height = 100
img_width = 100
num_channels = 3
input = tf.placeholder(tf.float32, shape=[batch_size, img_width, img_height, num_channel
```

A tensor must also be defined during the training phase containing the true class of each element in the training set, these true classes will be used to calculate the error of a training epoch, by seeing how many elements in the training set were given the correct prediction by the classifier (the CNN).

```
labels = tf.placeholder(tf.int32, shape=(batch_size))
```

After having defined placeholders for the input and its' classes, the network itself can be built. As mentioned in Chapter 5, a convolutional neural network has an input layer, convolutional layers, max pooling layers, and fully connected layers (one of which is the output layer). This may seem like a lot of work to write all of the code for these layers, and it can be to put it all together, but TensorFlow makes defining the layers easy. Luckily, the TensorFlow API defines layers of all of these types that can be used when building a convolutional neural network. For example, we can define the layers of a simple CNN (where the input layer is an element in the training set) with the following lines of code, assuming all variables being used as parameters for layers have been previously defined. We can also assume here that all initial weights for trainings have been defined and generated.

```
#first convolutional layer
layer_conv_1 = tf.nn.conv2d(input=input, filter=weights_kernel, strides=[1, 1, 1, 1], pa

#apply relu activation function to first convolutional layer
layer_ReLU_1 =  tf.nn.relu(layer_conv_1)

#pool the output of layer_ReLU_1
layer_max_pool_1 = tf.nn.max_pool(value=layer_ReLU_1, ksize=[1, 2, 2, 1], strides=[1, 2,

#second convolutional layer
layer_conv_2 = tf.nn.conv2d(input=layer_max_pool_1, filter=weights_kernel, strides=[1, 1

#apply relu activation function to second convolutional layer
```

```
layer_ReLU_2 = tf.nn.relu(layer_conv_2)

#pool the output of layer_ReLU_2
layer_max_pool_2 = tf.nn.max_pool(value=layer_ReLU_2, ksize=[1, 2, 2, 1], strides=[1, 2,

#first fully connected layer which takes layer_max_pool_2 as input and flattens the prev
layer_fully_connected_1 = tf.contrib.layers.flatten(layer_max_pool_2)
layer_fully_connected_1 = tf.layers.dense(layer_fully_connected_1, size)
layer_fully_connected_1 = tf.layers.dropout(layer_fully_connected_1, rate=dropout, train

#second fully connected layer
layer_fully_connected_2 = tf.layers.dense(layer_fully_connected_1, size)
layer_fully_connected_2 = tf.layers.dropout(layer_fully_connected_2, rate=dropout, train

#third fully connected layer which is the output layer
layer_fully_connected_out = tf.layers.dense(layer_fully_connected_2, num_classes)
```

Now that these layers are defined, the input can be given to the CNN and the network can be trained. The TensorFlow documentation also contains tips and tutorials on training neural networks[9]. After the network is trained, we can use these layers and the weights which were determined during training as a classifier, to make predictions about images.

Chapter 7

Working with a CNN

It has been shown that using TensorFlow, a convolutional neural network can be implemented by making full use of their high level API. This could be used in an app to classify images of clothing, as mentioned before, and this app could be used by a fashion designer. This fashion designer wants to be able to use this app give all of their images a classification, and use this label to match it to another image. This image which is input into the app (or the CNN) will be provided by a customer. Even with a straightforward way to build a convolutional neural network, and a theoretical app using the network as a classifier, there are still a few things missing. How can

Bibliography

[1] Bringing parallelism to the web with river trail.

[2] What is deep learning.

[3] Leonardo Aurajosantos. Artificial intelligence.

[4] David G Lowe. Object recognition from local scale-invariant features. 2:1150–1157, 1999.

[5] R. Fisher, S. Perkins, A. Walker, and E. Wolfart. Sobel edge detector, 2003.

[6] Paul Viola and Michael J Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

[7] Jeff Heaton. Neural network training (part 2): Neural network error calculation, 2010.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[9] Tensorflow documentation, 2017.

[10] Understanding activation functions in neural networks, 2017.

[11] Geron Aurelien. *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. O'Rielly Media, Inc., 2017.

[12] http://scienceworld.wolfram.com/physics/DeformableBody.html. *DeformableBody*.

[13] https://en.wikipedia.org/wiki/Rigid_body. *Rigid Body*.

[14] Miroslav Kubat. *An Introduction to Machine Learning*. Springer International Publishing Switzerland, 2015.

[15] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.