# More About Convolutional Neural Networks

Saahith Janapati

April 2019

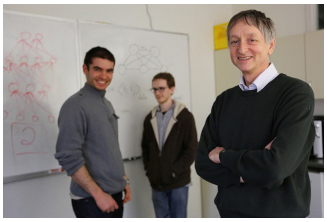
## Contents

# 1 Introduction

Convolutional Neural Networks (CNNs) have played a key role in the development of Deep Learning. A prior lecture covered the basics of CNNs. In this lecture, we will explore the aforementioned topics.

## 1.1 Background

Prior to getting into the main section of this lecture, I thought it would be cool to give some historical background about CNNs. They've actually been around since the 80s, but have only recently risen to popularity. This occurred when Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton [1] from the University of Toronto created a CNN that they called the AlexNet in 2012.



This model beat out past algorithms on the ImageNet benchmark challenge by a significant margin. The ImageNet challenge involves classifying an image into one of 10,000 labels (i.e dog, cat, car, plane, etc) [2]. The AlexNet paper spurred further research into CNNs and Deep Learning in general. It is said to be a seminal work in both the Computer Vision and Deep Learning research communities.

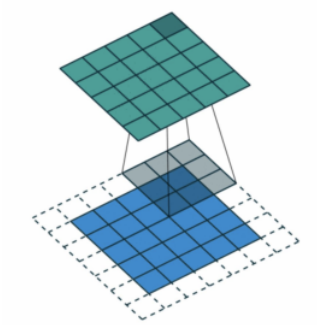
# 2 Review of CNNs

## 2.1 The Convolution Operation

As you may recall, a CNN works well on images because it is able to make use of spatial data. The convolution operation condenses information from several pixels into one output node in the following layer. This can be visualized by the following diagram.

[1] Geoffrey Hinton is commonly referred to as the godfather of Deep Learning. He recently won the ACM Turing Award, one of the highest awards in computer science, for his work on neural networks. He's a super cool person and you should definitely check out some of his interviews/talks if you have the time.

[2] The scoring actually involves the top five predictions of the model. So, let's say that for a given image, the model predicts cat, dog, ship, train, and bus in that order. If the correct label for the image was dog, then that image would be considered to be classified as correct. While it doesn't make sense in this example (as the categories are very different from each other), it is used in the ImageNet challenge because the categories are not very distinct from each other.



Let's walk through a simple convolution operation. Note that I use the multiplication operation for the convolution operation:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 37 & 47 \\ 67 & 77 \end{bmatrix} \tag{1}$$

Following standard naming conventions, I will refer to the second matrix as the kernel.

The top-left value (37) is calculated as the following: $1*1+2*2+4*3+5*4$.

The top-right value (47) is calculated as the following: $2*1+3*2+5*3+6*4$.

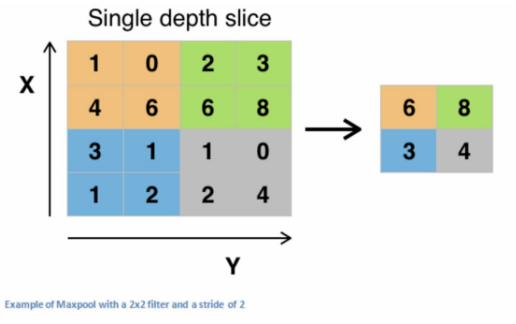
You can visualize the process as if you are sliding the kernel across the original matrix and multiplying the values that are on top of one other and subsequently adding all these values. The sum is then placed in the output matrix.

In practice, during backpropogation, the model would update the kernels for each layer, as well as its biases.

The convolution operation is the heart of the CNN (hence, the name). This operation allows for the network to view each pixel in its spatial context. That is, it can view each pixel with respect to the pixels close to it. This is helpful, since it allows the network to more easily detect features such as edges and more complex shapes. This is in contrast to a regular neural network, which would have to learn the relationships between nearby pixels (which unnecessarily wastes time and computation). In essence, by using the convolution operation, we are forcing the network to look at surrounding data.

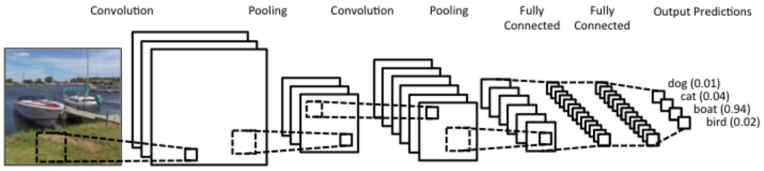
## 2.2 Pooling and Fully Connected Layers

In a CNN, convolutional layers are stacked together, along with pooling layers (a method of rapidly condensing the information in a layer). In max-pooling layers, a popular type of pooling layer, the maximum value in one section of a matrix is kept and the rest are discarded. A max-pooling layer is depicted here:




Fully connected layers are placed at the end of the networks. The activations of the previous convolutional layer are unrolled into these fully connected layers. If the problem that the programmer was trying to solve was a classification problem, perhaps she would add a softmax layer at the end.

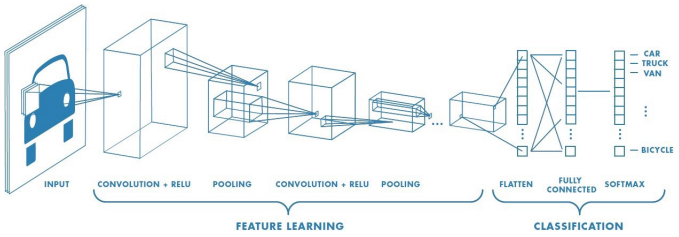
The end result looks something like this:




View the original CNN lecture for more information on pooling layers, activation functions, stride lengths, and other pertinent topics.

# 3 Transfer Learning

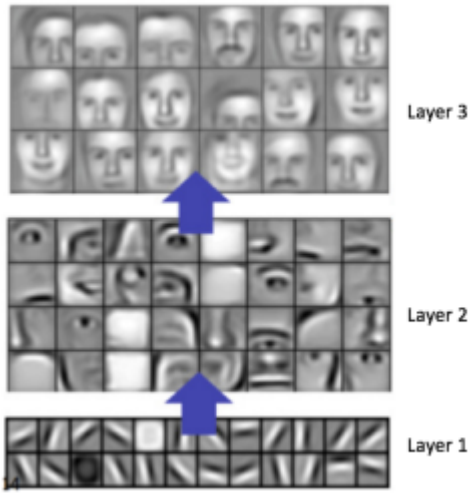
Practicing transfer learning is a great way to alleviate some of the computational and time requirements necessary to train a CNN (or a basic neural network). However, prior to exploring transfer learning, it is essential that one understands what a Convolutional Neural Network is learning at each layer.




## 3.1 What are CNNs learning?

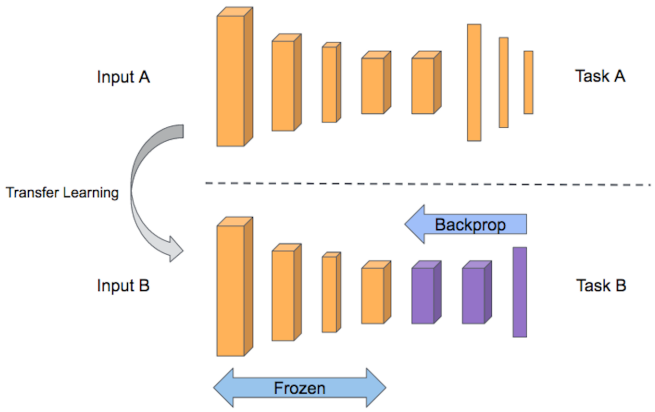
In the initial layers of a CNN, each output node in the layer corresponds to a small portion of the actual image. This can be seen in the first layer of the

CNN above. The node depicted in the first hidden layer only "views" a small portion of the car, specifically the left-middle section. So, the information gained in these layers about the image corresponds to small, minute features such as edges. In the later layers, all this information is then compounded together to detect more complex features. This can be seen in the following image.




## 3.2 Transfer Learning in Practice

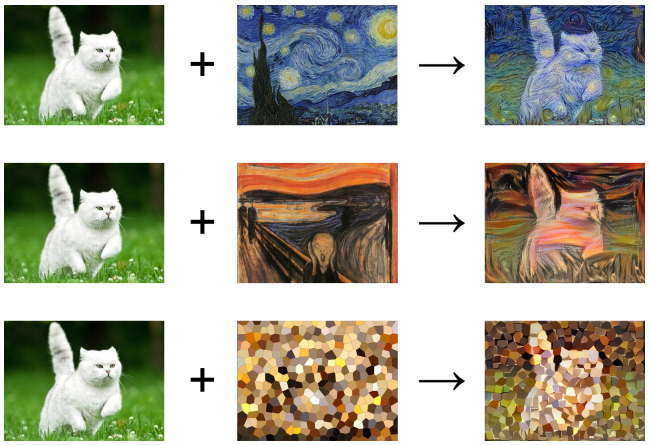
Transfer learning takes advantage of the fact that small features, such as edges, are found in many different types of images. So, the initial layers of a CNN trained on one dataset can be useful for classifying many images (even images from a different dataset). Libraries such as Tensorflow and Keras already have CNNs pretrained on datasets such as ImageNet. With just a couple of lines of code, you can import such a CNN and use it out of the box. However, let's assume that you want to train this CNN for your own task. Instead of training the whole network again on your images, which can be extremely computationally intensive, we can choose to only train the later layers of the network and freeze the initial layers. In other words, we will not be conducting backpropagation on the weights corresponding to the initial layers of the network.




Why does this work? Remember that the initial layers of a network can be viewed as feature extractors. The entire function of these initial layers is only to detect small features such as edges. The initial layers do not depend very much on the context of the dataset that you are training on. By freezing the initial layers, you can save time and computation during training. And you'll still be allowing your model to learn to detect the larger features that are specific to your problem (i.e faces, cats, dogs). This works very well if the datasets that the model was trained on initially is similar to the dataset you're trying to train you model on.

# 4 Neural Style Transfer

A very exciting and fun application of CNNs is Neural Style Transfer (NST). NST involves taking two images, a content image $C$ and a style image $S$ and combining them to generate an image $Y$. Here are several examples of NST in practice:



As you can observe, the content image (the cat) and the style painting (the paintings) are combined to create a hybrid image. Let's explore how this works.

## 4.1 Cost Function

Let's define a cost function $J(Y, C, S)$ that inputs the generated image and outputs how "good" it is, "good" being a measure of how well the styles of $C$ and $S$ were combined.

We will see later that by minimizing this cost function, we can generate an image with the styles/contents of both $C$ and $S$ taken into account.

Let's define $J(Y, C, S)$ more formally. It will have two components.

$$J(Y, C, S) = \alpha J_{content}(C, Y) + \beta J_{style}(S, Y)$$

$J_{content}(C, Y)$ measures the dissimilarity between the content image $C$ and the generated image $Y$. $J_{style}(S, Y)$ measures the dissimilarity between the style image $S$ and the generated image $Y$. $\alpha$ and $\beta$ are hyperparameters that the user can tune to adjust how much style is adopted from $C$ and $S$ respectively.

We will initialize the generated image $Y$ randomly. It will start off as noise. As we perform gradient descent on the aforementioned cost function, the dissimilarity, or error, of our image will be minimized. The image will go from random noise to a mix between the styles of $C$ and $S$. Let's now define the two components of the cost function.

## 4.2 Content Cost Function

This section will define the process for calculating $J_{content}(C, Y)$. To do this, we will pass both $C$ and $Y$ through a CNN pretrained on a dataset such as ImageNet. However, we will not pass the image through the whole CNN. Instead, we will stop the image in one of the hidden layers of the CNN (i.e. one of the middle layers). We will then extract the activations in this layer for both the $C$ and $Y$ images. We shall denote the activations for $C$ as $a^{(l)C}$ and the activations for $Y$ as $a^{(l)Y}$. Note that the superscipts are not exponents. Also note that $l$, the layer from which we are extracting the activations, remains the same for both $C$ and $Y$. We will define $J_{content}(C, Y)$ as the following:

$$J_{content}(C, Y) = \frac{1}{2} \sum_{i=1}^{n_H^l} \sum_{j=1}^{n_W^l} (a_{ij}^{(l)Y} - a_{ij}^{(l)C})^2$$

The derivative of this component of the loss with respect to the activation of the generated image is the following:

$$\frac{\partial J_{content}}{\partial a_{i,j}^{(l)Y}} = \begin{cases} (a_{i,j}^{(l)Y} - a_{i,j}^{(l)C}) & \text{if } a_{i,j}^{(l)Y} > 0 \\ 0 & \text{if } a_{i,j}^{(l)Y} < 0 \end{cases}$$

Note that the cost function is simply the sum of squared differences between the two activations. Also note that we did not index into the channels of the layers. Only one channel should be used.

If the two activations are similar, this output of this function will be minimized. If the two activations are dissimilar, the output of this function will be high. This is exactly the behavior we wanted!

Note that the regularization term $\frac{1}{2}$ is simply a regularization term.

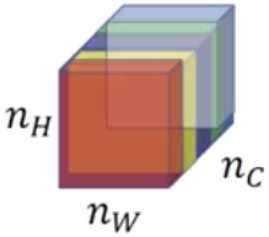
$l$ is a hyperparameter that can be adjusted by the user. Think back to the discussion about what the layers in a CNN are doing. If you set $l$ to be an earlier layer of the network, the two images will need to be very similar for $J_{content}(C, Y)$ to be low. Why? Because, as mentioned before, the initial layers of a CNN are detecting small features, such as edges. For the two activations to be similar, these features must be present in both images. However, if you pick a later layer to be $l$, the images do not need to be very similar, since the later layers of the network will be detecting larger features (such as faces, cars, buildings, etc.). Thus, $l$ is a hyperparameter that you should mess around with when implementing NST.

### 4.3 Style Cost Function

Let's now define the Style Cost Function $J_{style}(S, Y)$. The Style Cost Function is a bit more complicated. You might ask why it should be any different from the Content Cost Function. The Content Cost Function only needs to find the difference between the activations, since it's judging the difference in content. Content can be thought of as the specific objects in the image. Since the objects are most likely encoded as features in the CNN (possible features that the later layers of a CNN may be looking for are face, truck, bridge,etc.), it makes sense to compare activaitons. However, style is a bit more nuanced, since it relates to the texture and color of an image.

How should we define style then? The authors of the NST paper defined style as the correlation between activations across channels. What do they mean by this?

If you recall, as you progress into later layers of a CNN, the number of channels generally increases. For the purpose of understanding, imagine if each channel in a hidden layer is like a neuron corresponding to some feature.



Perhaps one channel (the red channel) is detecting horizontal lines in an image and another channel (the yellow channel) is detecting bluish-color. By measuring the correlation between these two channels in the activations of the network when an image was passed through, we can see if both horizontal lines and bluish color coexist in the image that was passed through.

If the two channels were not correlated, the two features they are detecting most likely do not coexist in the image that was passed through the network. On the other hand, if the two channels are correlated, then those two features do coexist together in the image.

To quantify the amount of correlation there is between the activations of all the channels in a hidden layer, the researchers used the Gram Matrix.

$$G^l = \begin{bmatrix} G^l_{11} & G^l_{12} & \dots \\ \vdots & \ddots & \\ G_{n^C 1} & & G_{n^C n^C} \end{bmatrix}$$

In the Gram matrix, $G^l_{kk'}$ is defined as the following:

$$G^l_{kk'} = \sum_{i=1}^{n^l_H} \sum_{j=1}^{n^l_W} a^l_{ijk} a^l_{ijk'}$$

Note the intuition behind this idea. We are essentially doing element wise multiplication on all the values in one channel of the hidden layer with the corresponding value in the second layer and summing all these values together. Notice that if both channels $k$ and $k'$ are activated by the image, the corresponding $G_{kk'}$ will be large as well.

You would calculate this Gram matrix for both the style image and generated image using the activations from the same layer. These matrices give us a style representation of both the image $S$ and the image $Y$.

Since we have the Gram matrices for both the style image $S$ and the generated image $Y$, we can quantify the difference between the two. You can now define $J_{style}(S, Y)$ as the following:

$$J^l_{style}(S, Y) = \sum_{k=1}^{n^l_C} \sum_{k'=1}^{n^l_C} (G^{[l](Y)}_{kk'} - G^{[l](S)}_{kk'})^2$$

Here, $G^{[l](S)}$ refers to the Gram Matrix calculated for the Style image at layer l. $G^{[l](Y)}$ refers to the Gram Matrix calculated for the generated image at layer l.

The authors of the NST paper also included a regularization term for this portion of the following. If you include it, the cost function looks like this:

$$J^l_{style}(S, Y) = \frac{1}{(2n^l_H n^l_W)^2} \sum_{k=1}^{n^l_C} \sum_{k'=1}^{n^l_C} (G^{[l](Y)}_{kk'} - G^{[l](S)}_{kk'})^2$$

A common practice that researchers have found to be effective is to sum together the expression above for several layers (some of them being initial layers of the CNN and some of them being later ones). Doing this has led to clearer images. You can define this final cost function as the following:

$$J_{style}(S,Y) = \sum_{l=0}^{n} \lambda_l J^l_{style}(S,Y)$$

where $l_1$ is the first layer in the list of layers you're using, $l_n$ is the last layer in the list of layers, and $\lambda_l$ is the parameter for each layer you are using. You should experiment with different numbers of layers that you use.

The derivative of $J_{style}(S,Y)$ with respect to the activations in layer $l$ are the following:

$$\frac{\partial J_{style}}{\partial a^{(l)Y}_{i,j}} = \begin{cases} \frac{1}{N_l^2 M_l^2}((a^{(l)Y}_{ij})^T(G^{[l](Y)}_{ij} - G^{[l](S)}_{ij})) & \text{if } a^{(l)Y}_{i,j} > 0 \\ 0 & \text{if } a^{(l)Y}_{i,j} < 0 \end{cases}$$

This looks messier than it is due to the notation that I used. The paper that introduced NST used a different notation, and the end derivative looked neater than this. I chose not to use the notation in the paper because the symbols they used seemed a bit arbitrary.

### 4.4 Wrapping Up

We now have all the components to implement NST. Using gradient descent (or some other optimization algorithm), we can alter the generated image $G$ to minimize the error for the cost function:

$J(Y) = \alpha J_{content}(C,Y) + \beta J_{style}(S,Y)$

We can do this by using the aforementioned derivatives calculated with respect to the activations of the generated image.

If you would like, you can mess around with this neat web-application implementing NST yourself at the following link:

https://tenso.rs/demos/fast-neural-style/

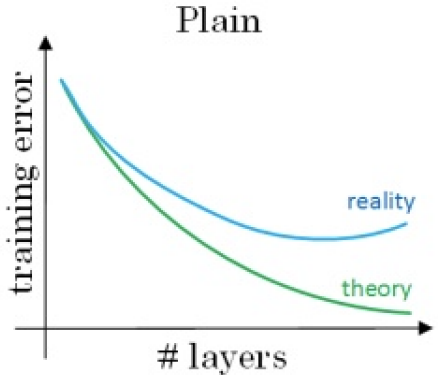
## 5 Residual Networks

As a Deep Learning practitioner, you may expect that as you make your neural network deeper (i.e. you add more layers), your model will be better able to approximate the mapping between your input and output. Unfortunately, this is not always the case. This is because of the vanishing gradient problem.

### 5.1 Vanishing Gradient Problem

It can become increasingly difficult for neural networks to learn useful features as their depth increases. This is because the gradients of the weights of the

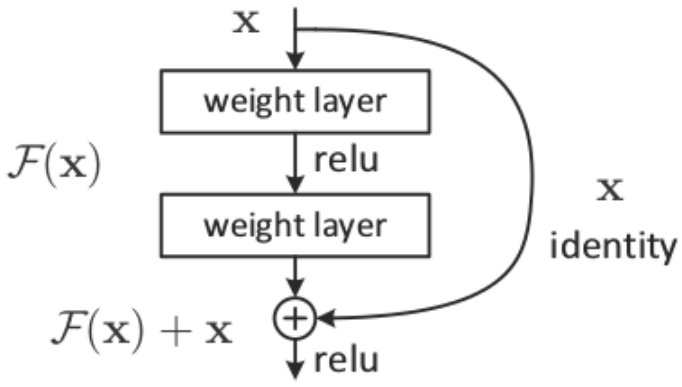
network can decrease exponentially as backpropagation modifies the weights of earlier and earlier layers. You can think of this as a compounding effect. If the gradient for a layer becomes too small, it will not learn much during training, as the updates to the weights will be incremental. And so, the network will not be able to learn the mapping from the input to the output. The effects of this can be visualized in the following image.




Instead of the network approximating the function better as the depth increases, the performance of the neural network actually gets worse. How can we fix this?

## 5.2 Residual Connections

In 2015, researchers from Microsoft came up with a solution to solve this problem: residual connections, or "skip" connections. In residual connections, the activations from one layer in the network are passed into a layer deeper into the network. This can be visualized by the following diagram:




More formally, the output from Layer n is being fed to another layer n + k such that the it is added to the activations before the nonlinearity function is

applied. To make this clearer, let us refer directly to the diagram above[3]. Let's refer to $X$ as $a_0$, referring to the fact that this is the input data.

$a_1$ is calculated as you would expect in a regular neural network.

$$a_1 = g(W_1 * a_0 + b_1)$$

where $g(x)$ is your activation function (in this case, RELU), $W_1$ is the weight matrix for the first layer, and $b_1$ is the bias vector for the first layer.

Now, the process for getting $a_2$ is a bit different, as this is the layer where the residual connection lies.

$$a_2 = g((W_2 * a_1 + b_2) + a_0)$$

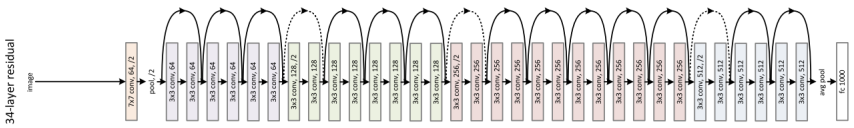
where $g(x)$ is your activation function (in this case, RELU), $W_2$ is the weight matrix for the second layer, $b_2$ is the bias for the second layer, and $a_0$ is the input data.

Note the key difference between the calculation of $a_2$ and the calculation of $a_1$. In the calculation of $a_2$, we add $a_0$ prior to applying the RELU function. This is the core idea of residual connections. Researchers have shown, through empirical evidence, that by adding these skip-connections, the neural network can overcome the vanishing gradient problem. One common line of thought as to why residual connections work so well is the following:

Even if the expression $(W_2 * a_1 + b_2)$ equates to zero (as can happen due to weight decay during gradient descent), the layer can still easily learn the identity function (assuming we are using the RELU as our activation function and that $a_0$ is positive), since we are also adding $a_0$. So, even in the worst case, we will not have hurt our model, since it will still maintain the same activation as the previous layer. In other words, adding residual connections makes sure that adding more layers does not hurt the performance of the model.

Note that by using residual connections, you will not be increasing the computational requirements of your model. Using residual connections does not require additional parameters (unless the activation you are feeding in is of different size to the output you are feeding to, i.e. if $a_0$ was a different size from $W_2 * a_1 + b_2$, in which case you would need a weight matrix to convert $a_0$ to the same dimensions as $W_2 * a_1 + b_2$). No major changes need to be made in order to make use of residual connections. You can still use Stochastic Gradient Descent or any other optimization algorithm that you are accustomed to using.

In practice, if you want to use residual connections in a CNN, you should use a pretrained version of the ResNet, found on Keras and Tensorflow. This model is visualized in the following image:



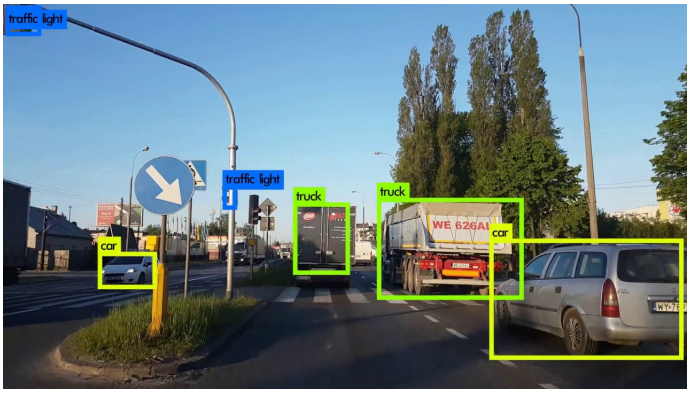

[3]Note that for this example, I treat the model as a regular neural network, not a CNN. The procedure is similar for a CNN.

You can make use of transfer learning to train this model on your own dataset.

You might be wondering why this content was included in this lecture. That's because the aforementioned researchers first introduced residual connections in the context of Convolutional Neural Networks. Residual connections are also mainly used in the context of Convolutional Neural Networks.

# 6 A High Level Overview of Object Detection

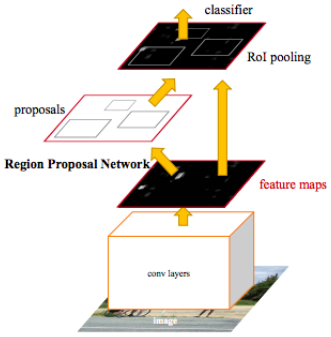
Up till now the networks that we've discussed can be used to perform classification, i.e. they can tell whether the image contains a cat or a dog. However, some problems require more than simply classifying what is in the image. One example of such a problem is the self-driving car. A self-driving car should not just detect whether or not the main object in an image is a person. A self-driving car should instead detect and classify every single important object in an image (i.e. car, truck, traffic light, pedestrian, etc.). Such an output is depicted below:




How do we go about this task? In the following section, we will discuss a popular object detection algorithm called the Faster-RCNN. For brevity (this lecture has already gotten pretty long), I will not go into the lowest level of detail. Instead, I will focus on the aspects of the model necessary to understand its functionality. Let's get started.

## 6.1 Faster-RCNN

The Faster-RCNN is an object detection model that arose from years of experimentation. Before the Faster-RCNN, there were the RCNN and the Fast-RCNN (such creative names). The Faster-RCNN is the latest iteration of these models. It is depicted in the following image:




### 6.1.1 Initial Convolutional Layers

The first component of this model is the convolutional block. The raw image is first fed through these convolutional layers. The output of these layers is a feature map, from which the rest of the model is based off of.

### 6.1.2 Region Proposal Network

The Region Proposal Network (RPN) is a core component of the Faster-RCNN. From the feature map generated from the initial convolutional layers, the RPN detects portions of the images which potentially contain distinct objects. This is done by training a small CNN on a multi-task loss function.

For each region that the RPN proposes, the RPN also outputs a classification score, which is how sure the RPN is that there is an object in that section of the image. The disparity between this score and the ground-truth classification score is quantified by the classification loss.

The RPN proposes regions by outputting for pixel coordinates. These pixel coordinates form a bounding box. The disparity between this predicted bounding box and the ground truth bounding boxes, provided by an annotated dataset, is quantified by the bounding box regression loss.

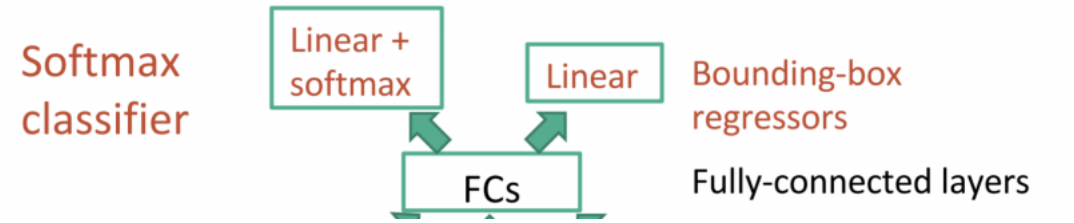
This multi-task loss-function takes into account both the classification loss and the bounding box regression loss while training.

Note that there was a lot glossed over in this section, including how the RPN uses anchors to output the prediction.

### 6.1.3 ROI Pooling

Once the RPN outputs regions of the image in which there are potentially objects, each of the regions goes through ROI pooling. This essentially warps each region proposal into a standard size so that they can be fed into a classifier for predictions. While this can be done very simply, the ROI pooling is used because it is differentiable, meaning it can be adjusted and improved as we conduct backpropagation.

#### 6.1.4 The Rest




The rest of the Faster-RCNN is the exact same as the end of the Fast-RCNN, as depicted above. From the ROI pooling layer, the data flows to fully connected layers which are connected to a softmax classifier and bounding box regressors. These are again trained on a multi-task loss function. And that's it!

Note that there are several other object detection algorithms including YOLO and SSD.

## 7 Conclusion

The topics discussed in this lecture are only several of many. These topics were simply an assortment of those that I found particularly interesting or useful. There are many more interesting topics related to CNNs that I have not covered.

CNNs have been a core part of Deep Learning research and will most likely continue to do so in the future.

## 8 Acknowledgements

I did not create any of the images found in this lecture. Here are some of the sources I used:

- Andrew Ng's deeplearning.ai course
- Stanford's CS231 Course
- ResNet Paper
- freeCodeCamp's Medium Post on CNNs
- Past ML CLub Lectures :)
- NST Paper