Examining the Structure of Convolutional Neural Networks

Serge Aleshin-Guendel

Computer Science Honors Thesis
Boston College

Advised by
Professor Sergio Alvarez

May 10, 2017

Abstract

Machine learning, in conjunction with large data sets, has seen many success stories in recent years, leading to increased interest in the field. One popular class of machine learning models is deep neural networks, where stacked layers of "neurons" are used to learn approximate representations of data. One particular model, the Convolutional Neural Network (CNN), is notable in that it's become the standard in most computer vision tasks. However, there is little to no theory surrounding how to best build these CNNs.

The aim of this thesis is two-fold. The first aim is to provide a brief introduction to the field of supervised machine learning, neural networks, and CNNs. The second aim is to explore how to best build CNNs, through an examination of structural properties related to the width, depth, and receptive field of networks.

Contents

1 Introduction

1.1 Overview: What is Machine Learning?

The field of machine learning is broadly concerned with the task of creating models and algorithms that "learn" from data. A somewhat naive, but convenient, way to divide the field of machine learning is between unsupervised and supervised learning (there are many other subfields in between and outside of this dichotomy, but they're outside the scope of this thesis). In the subfield of unsupervised learning you're given some input data and are tasked with finding some underlying structure in the data. This description is intentionally vague, as there's usually no good way to evaluate these methods given that there's no output data (and there lies some of its extreme difficulty!). In the subfield of supervised learning however, along with input data, you're also given associated output data. The task here is to find some "mapping" from the input to the output.

This thesis will be concerned with the task of supervised learning, and in particular image classification, where given an image your goal is to give it some basic label. For example, the CIFAR-10 data set [1] consists of $60,000$ images with 10 basic labels: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

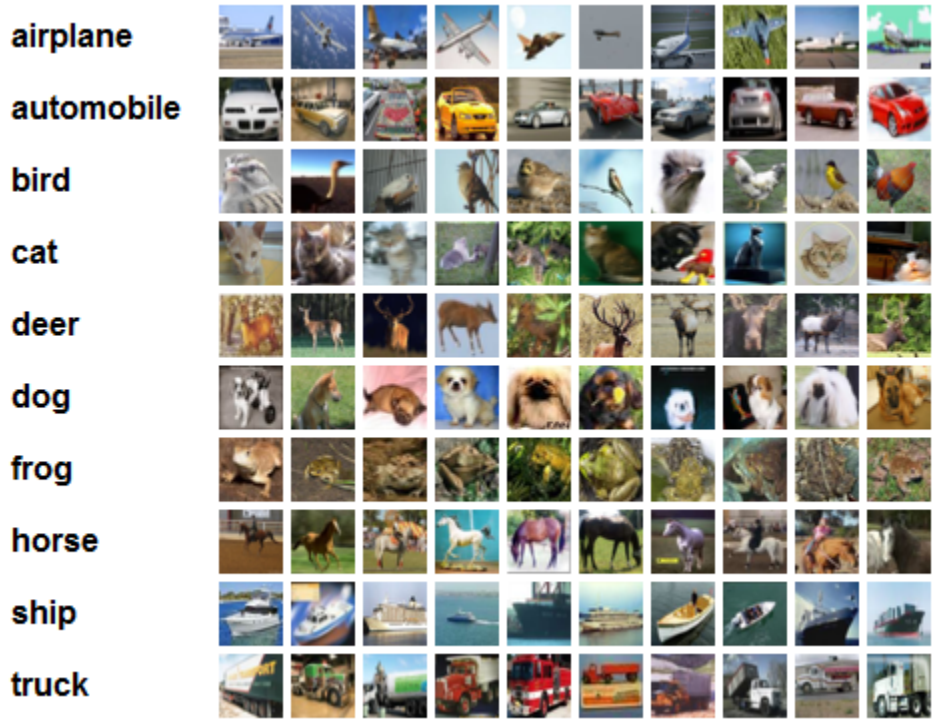

Figure 1: Example images from the CIFAR-10 data set [1]

We'd like to build some model so that if it sees any of these images, it'll be able to assign it the correct label. In order to build this model, we're going to need to show our model all of these images so that it can learn from them (in what's called the training process).

However, we'd further like to show this model a picture it hasn't seen before, and we'd once again like it to output the correct label (which would be supplied along with the unseen image). For

example, say we have an image of a cat such as the one right here, appropriately labeled "cat", that our model doesn't see since it's not in the CIFAR-10 data set:

Figure 2: An image labeled "cat" not in the CIFAR-10 data set

We still want our model to label it as "cat" rather than "frog" or "truck". This may be simple enough to me or you, the reader, but it's not immediately clear how we'd build such a model to do this automatically.

Much work has gone into this task, and associated tasks, in the past 20 or so years. The dominating models in image classification come from a what's known as deep learning, a field that uses a specific model called a neural network [2]. What follows in this thesis is an introduction to supervised learning, an introduction to neural networks, and my work on Convolutional Neural Networks, a specific class of neural networks.

1.2 Supervised Learning

The goal of supervised learning is, given an input space $\mathcal{X}$, an output space $\mathcal{Y}$, and a function $f : \mathcal{X} \to \mathcal{Y}$ which maps the input space to the output space, to approximate the function f. When the output space is discrete, the task is called classification, as in our motivating example of image classification. When the output space is continuous the task is called regression. In order to make learning feasible (since there could be infinitely many possible functions from $\mathcal{X}$ to $\mathcal{Y}$) we must restrict ourselves to choosing functions from a set of hypothesis functions $\mathcal{H}$. This choice of $\mathcal{H}$ will determine how we model the data.

In the real world, we don't have access to the underlying input and output spaces, or the actual function between the two spaces. Instead we have access to a finite data set $\mathcal{D} = \{(x_i, f(x_i) = y_i)\}_{i=1}^N$, called the training set, consisting of a particular set of realizations of the input and output spaces (i.e. $(x_i, f(x_i) = y_i) \in \mathcal{X} \times \mathcal{Y}$). We'll call the x_i our features, and the y_i our outputs. The question is now how do we choose a hypothesis $h^* : \mathcal{X} \to \mathcal{Y}$, where $h^* \in \mathcal{H}$, to approximate f, so that $h^* \approx f$?

A common framework is to use some error function $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$, which given a pair $(x_i, y_i) \in \mathcal{D}$ and a candidate function $h \in \mathcal{H}$, computes $L(h(x_i), y_i)$, which tells you how good (or bad) your current candidate function is. The goal in choosing a candidate function is to minimize $L(h(x_i), y_i)$. We're also usually more concerned with how a candidate function performs on the entire data set, rather than any single example in our data set, so we can consider the average of the loss on the data set for a candidate function h, $R(h) = \frac{1}{N} \sum_{i=1}^N L(h(x_i), y_i)$, called the empirical risk. Ideally we'd further like to measure the performance of our candidate hypothesis on the set of all possible examples so that we can find a hypothesis $h^* \in \mathcal{H}$ that performs as well as the underlying function f.

Putting this all together, given a data set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ and a set of hypothesis functions $\mathcal{H}$,

we wish to compute

$$h_{min} = \arg\min_{h \in \mathcal{H}} R(h) = \arg\min_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} L(h(x_i), y_i).$$

The output of this minimization, which we'll call the training process (hence the name training set), is considered to be our best hypothesis which we then use as our final model. But how exactly do we even go about performing this process? Well that depends on your model choice (i.e. what hypothesis space did you use), and there may still be many ways to either exactly or approximately minimize the empirical risk.

This then raises the question of: what is a good model? Just because we found the hypothesis $h_{min} \in \mathcal{H}$ that minimized our loss function, did we really find the best model? One way to view this question is through the lens of generalization. One way to tell how good or bad our model was by comparing $f(x)$ to $h_{min}(x)$ for every $x \in \mathcal{X}$. However, we don't have access to the underlying input space or function. As a proxy to f and access to the underlying input space, we use a separate data set, unseen to the model during the training process. We'll deem this set the test set, $\mathcal{D}^{test} = \{(x_i^{test}, y_i^{test})\}_{i=1}^{N^{test}}$. The goal now isn't strictly to find the h_{min} defined as above. Instead, we'd like to find a hypothesis that minimizes the empirical risk on the test set (rather than the training set), given that the model isn't trained or evaluated on the test set. It's important here to emphasize that the model is trained only on the training set, and not on the test set, even though we're now concerned with its performance on the test set. When a model preforms well on our test set, we say it generalizes well.

1.3 Example: Linear Models

In order to make this discussion of supervised learning clear, we'll consider two simple models for regression and classification: linear regression and logistic regression.

1.3.1 Linear Regression

As implied by the name, linear regression is a model for the task of regression, or approximating a continuous valued function. In particular, consider an input space of $\mathcal{X} = \mathbb{R}^d$ and an output space of $\mathcal{Y} = \mathbb{R}$. Then for linear regression, the set of hypothesis functions is of the form

$$\mathcal{H} = \left\{ h(x_1, \cdots, x_d) = w_0 + x_1 w_1 + \cdots + x_d w_d = w_0 + \sum_{i=1}^{d} x_i w_i \middle| w_0, \cdots, w_d \in \mathbb{R} \right\}.$$

Here, w_0 is called the bias (which can be thought of as the y-intercept), and each w_i is called the weight for the respective x_i. We'll then use the squared error loss function, given by $L(h(x), y) = (h(x) - y)^2$. This amounts to finding the "best" linear fit for our data.

Now suppose our data set is given by $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where $\mathbf{x}_i = (1, x_{i1}, \cdots, x_{id}) \in \mathbb{R}^{d+1}$ and $y_i \in \mathbb{R}$. We'll sometimes write our features in matrix form, $\mathbf{X}$, where the ijth entry is given by x_{ij}, and our outputs in vector form, $\mathbf{y}$ where the i^{th} entry is given by y_i. For a given hypothesis $h \in \mathcal{H}$ we denote the set of associated weights to be $\mathbf{w} = (w_0, w_1, \cdots, w_d)$. Note that our use of ones at the beginning of our feature vectors is for notational convenience when working with biases. Putting this altogether, our task is to find

$$\arg\min_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} L(h(x_i), y_i) =$$

$$\arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}} \frac{1}{N}\sum_{i=1}^{N}(\mathbf{w}^T\mathbf{x}_i - y_i)^2 =$$

$$\arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}} \frac{1}{N}||\mathbf{X}\mathbf{w} - \mathbf{y}||^2.$$

By taking the derivative of $\frac{1}{N}||\mathbf{X}\mathbf{w}-\mathbf{y}||^2$ with respect to the weight vector $\mathbf{w}$, and finding where this derivative is 0, we find that this expression is minimized when $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$. Thus we have a closed form solution for the best fitting linear function of the data!

1.3.2 Logistic Regression

Contrary to the implication of its name, logistic regression is a model for the task of classification, or approximating a discrete valued function. In particular, consider an input space of $\mathcal{X} = \mathbb{R}^d$ and an output space of $\mathcal{Y} = \{0, 1\}$, where this indicates that there are 2 possible classes that the input space can be mapped to (we can easily generalize to $m > 2$ classes with a slight tweak talked about later). Instead of the case of linear regression where the output of out model is our output, we'll be outputting a probability for the class labeled 1, and we can then classify an input as 1 if our model's output probability is greater than 1/2, otherwise we classify an input as 0 (we're glossing over some details here that aren't important for the purpose of this thesis).

For logistic regression, our set of hypothesis functions is of the form

$$\mathcal{H} = \left\{ h(x_1, \cdots, x_d) = \frac{e^{w_0+x_1w_1+\cdots+x_dw_d}}{1+e^{w_0+x_1w_1+\cdots+x_dw_d}} = \frac{e^{w_0+\sum_{i=1}^{d}x_iw_i}}{1+e^{w_0+\sum_{i=1}^{d}x_iw_i}} \middle| w_0, \cdots, w_d \in \mathbb{R} \right\}.$$

Here, we use the logistic function (also called the sigmoid function)

$$\sigma(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$$

as a non-linear squashing function to turn the real valued linear combination of weights and inputs into values in the range $[0, 1]$. So we get proper probabilities! We can thus rewrite $\mathcal{H}$ as

$$\mathcal{H} = \left\{ h(x_1, \cdots, x_d) = \sigma(w_0 + x_1w_1 + \cdots + x_dw_d) = \sigma\left(w_0 + \sum_{i=1}^{d} x_iw_i\right) \middle| w_0, \cdots, w_d \in \mathbb{R} \right\}.$$

Note that this set is roughly equivalent to the hypothesis space for linear regression, in that the functions considered in each space are completely determined by $d + 1$ real valued weights. We'll use the binary cross entropy loss function, given by $L(h(x), y) = y\frac{1}{\ln(h(x))} + (1-y)\frac{1}{1-\ln(h(x))}$.

Now suppose our data set is given by $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i = (1, x_{i1}, \cdots, x_{id}) \in \mathbb{R}^{d+1}$ and $y_i \in \{0, 1\}$. Putting this altogether, our task is to find

$$\arg\min_{h\in\mathcal{H}} \frac{1}{N}\sum_{i=1}^{N} L(h(x_i), y_i) =$$

$$\arg\min_{h\in\mathcal{H}} \frac{1}{N}\sum_{i=1}^{N} y_i\frac{1}{\ln(h(x_i))} + (1-y_i)\frac{1}{1-\ln(h(x_i))}.$$

However, unlike linear regression, when we take the derivative of this with respect to the weight vector $\mathbf{w}$ (which is encoded in the hypothesis h above), and try to find where this derivative is 0,

there is no closed form solution. Thus we need to resort to an optimization algorithm called gradient descent [3].

The idea behind gradient descent is that we can think of empirical risk as a function of a $d+1$-dimensional surface parameterized by our weights, and on this surface we're trying to find the lowest point, the minimal empirical risk. From calculus, we know that the direction of steepest descent from a given point $\mathbf{w}$ in our weight space is given by $-\nabla R(\mathbf{w})$, or the negative gradient of our empirical risk at the given point. If we start at a random point in our weight space, and iteratively shift this weight by some fraction of $-\nabla R(\mathbf{w})$, we're guaranteed to eventually reach a local minimum (if our fraction is suitably chosen). This fraction of the gradient is referred to as the learning rate. This gives rise to the algorithm known as gradient descent.

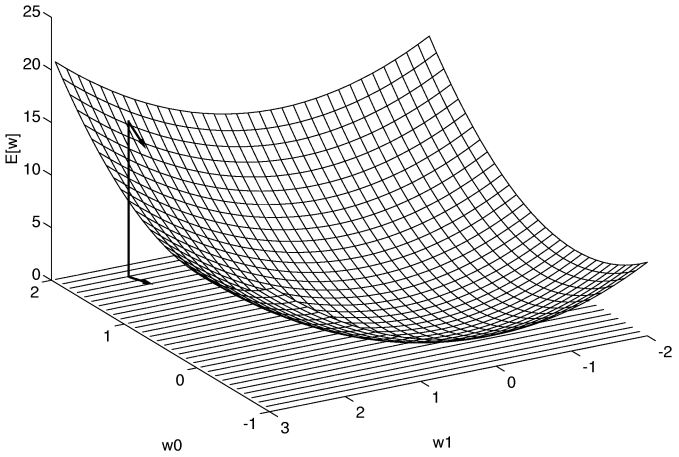

Figure 3: A visualization of a 2 dimensional weight surface, where $E[w]$ is the empirical risk for a given point in weight space [4]

Algorithm 1 Gradient Descent

Precondition: Function $f(\mathbf{w})$ of our weights (representing the empirical risk for logistic regression in this example for a given weight configuration), learning rate α

Initialize weights to some random $\mathbf{w}_{current}$

1: **while** not converged **do**
2: $\quad g \leftarrow -\nabla f(\mathbf{w}_{current})$
3: $\quad \mathbf{w}_{current} \leftarrow \mathbf{w}_{current} + \alpha g$
4: **end while**
5: **return** $\mathbf{w}_{current}$

2 Neural networks

2.1 The Perceptron

Neural networks and the field of deep learning started out of a (very) rough model of how the brain works [4]. The idea is that functions of the brain are carried out by composition of a large amount of neurons and synapses. In order to understand large neural networks, it helps to first start with their simplest form, a perceptron, which represents as single neuron.

Consider the same data set up as in logistic regression, In particular, consider an input space of $\mathcal{X} = \mathbb{R}^d$ and an output space of $\mathcal{Y} = \{-1, 1\}$ (we've changed 0 to -1 for notational convenience). Then for the perceptron, the set of hypothesis functions is of the form

$$\mathcal{H} = \left\{ h(x_1, \cdots, x_d) = \text{sgn}(w_0 + x_1 w_1 + \cdots + x_d w_d) = \text{sgn}\left(w_0 + \sum_{i=1}^{d} x_i w_i \right) \middle| w_0, \cdots, w_d \in \mathbb{R} \right\},$$

where the sign function is given by

$$\text{sgn}(x) = \begin{cases} 1 & x > 0 \\ -1 & x \leq 0 \end{cases}.$$

Thus our hypothesis functions predict an input to be 1 if their linear combination with the given weights is greater than 0, and -1 otherwise. Note that again, as in the case for logistic regression, the hypothesis space is roughly equivalent to that of linear regression. This stems from the fact that all three methods can only represent linear hypotheses. This is straight forward in the case of linear regression (after all linear is in the name, and we choose it so that we're essentially just line fitting). However, in the classification settings of logistic regression and the perceptron, this means that we can only correctly classify an entire data set if there exists a linear boundary between the two classes! This will motivate the generalization of the perceptron, the multilayer perceptron.

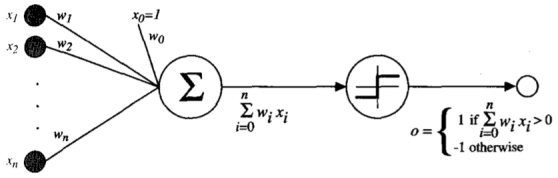

Figure 4: A graphical representation of a perceptron [4]

In order to train a perceptron, you can use what's known as the perceptron learning algorithm (PLA) (see [4] or [5]). It's similar to gradient descent in that we're iteratively shifting the weights, however, the PLA doesn't converge for non-linearly separable data. There are modifications to the PLA that can allow convergence, but they're not pertinent for this thesis. Another way to get around this would be to abandon the PLA and train the unthresholded hypotheses using gradient descent, which would be equivalent to attempting to regress onto -1 and 1.

2.2 Multilayer Perceptrons

All of the models we've examined so far have only been able to represent linear hypotheses. To go beyond linearity, we'll look at the generalization of the perceptron, the multilayer perceptron

(MLP), which is what most people call a neural network. We'll consider fully connected feedforward MLPs, which can be considered the standard or vanilla MLP. Most other neural networks can be realized as a special case of the standard MLP, as we'll see with convolutional neural networks.

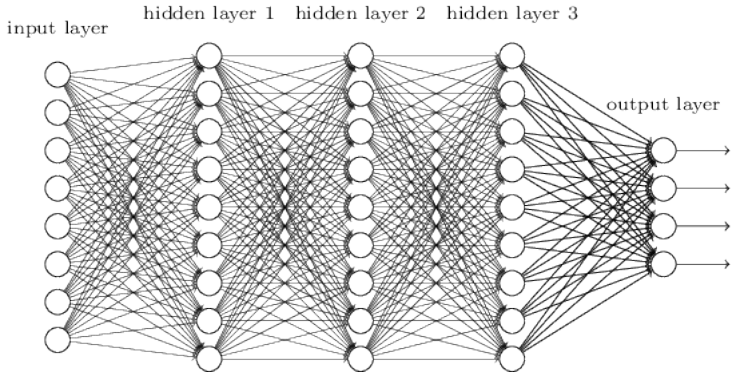

Figure 5: A three layer MLP [6]

As can be seen by this figure, an MLP is a network of perceptrons. And if we consider each perceptron to be a "neuron", we see where the name neural network comes from! From here on out we'll refer to the perceptrons as nodes or neurons interchangeably. The inputs are fed to a first "hidden layer" of perceptrons, where each perceptron receives all of the inputs (this is where fully connected comes from). The output of each perceptron is then passed through a non-linear thresholding function, such as the sign function discussed earlier. However, in practice, we'll choose thresholds that are differentiable, as this will allow us to train the networks (using a variant of gradient descent). The most common threshold in practice today is the ReLU function [7], which is defined as

$$\text{ReLU}(x) = \max(0, x).$$

Another common threshold is the sigmoid function, which we saw in the context of logistic regression. The outputs of each of the perceptrons in this first layer are then fed into a second layer of perceptrons (where the first layer output acts as the input for the second layer), which functions exactly as the first layer. This continues for as many layers as one would like (however in practice we'll only be able to build MLPs with finite numbers of layers).

To make this more concrete, we consider input data $(\mathbf{x}_i) \in \mathcal{X}$ given by $\mathbf{x}_i = (1, x_{i1}, \cdots, x_{id}) \in \mathbb{R}^{d+1}$, where 1 is added to the input data as a bias just like in all of our other models so far. We say our network has depth ℓ if it contains ℓ hidden layers. A given hidden layer k has size d_k, which is the number of perceptrons in that layer. So in the above figure we have a depth $\ell = 3$ network, where $d_1 = d_2 = d_3 = 9$, so each layer consists of 9 perceptrons. For a given layer k of the network, we have a given weight matrix $\mathbf{W}_k$, where the ith row $\mathbf{w}_{ki}$ is the weight vector of the ith perceptron in layer k.

When we reach the final layer, we then pass the final output (which is now a d_ℓ length vector) through one of two functions. If we're regressing, we pass this output through an unthresholded perceptron (we're preforming linear regression on the output!). If we're classifying, when we have two classes as in previous examples, we'll pass the output through a perceptron thresholded by the sigmoid function (we're preforming logistic regression on the output!). However, if we'd like to classify a data set with more than two classes, we'll pass the output through something called the softmax function, which is the multiclass generalization of the sigmoid function used for logistic

regression. Note that in this case, we require d_ℓ to be equal to the number of classes. Let $\mathbf{z}$ be a d_ℓ length vector. Then the ith, for $i \in \{1, \cdots, d_\ell\}$, component of the softmax function is given by

$$f(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{d_\ell} e^{z_j}}.$$

Note that if we didn't threshold any of the perceptrons in our network, we'd still only have a linear function of our data at the end of the network, and thus another linear hypothesis. By adding non-linear thresholds into our network, we allow our network to represent non-linear hypotheses. Indeed, it can be shown that a one layer network with sigmoid thresholds can arbitrarily approximate any decision region [8].

Much like the case for linear and logistic regression, depending on the task, we need different error functions. If we're regressing, we can continue to use the sum of squares loss. If we're classifying, and have only 2 classes, we can continue to use the binary cross entropy loss. However, if we're doing multiclass classification, with m classes, we'll use the generalization of binary cross entropy, called categorical cross entropy. Let $\mathbf{y}$ be an m length vector representing a one hot encoding for a particular class and $\mathbf{h}(\mathbf{x})$ be the candidate hypothesis, represented by a neural network, applied to the input $\mathbf{x}$ corresponding to the output $\mathbf{y}$ (where $\mathbf{h}$ is also a length m vector, but instead of being one hot encoded it contains the probabilities output by the final layer of our network). Then the categorical cross entropy loss function is given by $L(\mathbf{h}(\mathbf{x}), \mathbf{y}) = -\sum_{i=1}^{m} y_i \log(h(x)_i)$.

In order to train our MLP, we'd like to find the optimal weights of the network, which amounts to finding the optimal weight matrix $\mathbf{W}_k$ for each layer k of our network. Much like the case of logistic regression, there's no closed form solution for the weights of MLPs in general if we take the derivative of the entire network as a function and set it to 0 (it's not even clear how we'd even go about doing that in an arbitrary network). And again like logistic regression, we'll employ the gradient descent algorithm to train our network. However, it's not clear how to take the derivative of our loss function with respect to the all of the weights of the network. Instead we'll employ backpropagation, which is a modified version of gradient descent for feedforward MLPs [9]. The key to backpropagation is the chain rule from calculus. Note that calculating the derivative of the loss function with respect to the weights in the last layer of the network is something that's simple to do. In order to then calculate the derivative of the weights in the next to last layer, all we need is the derivatives of the weights in the last layer of the network. Then for the layer before that, we only need to derivatives from the next to last layer. This goes on till we reach the first layer. It can be seen that the name backpropagation comes from the manner in which we pass (propagate) the derivatives back through the network! To see a detailed treatment of backpropagation and its derivation, see [4] or [10].

2.3 Neural Networks As Feature Extractors

If it wasn't clear from the preceding section, there's a large connection between MLPs and the linear models discussed earlier. Namely, MLPs serve as a way to take repeated non-linear transformations of the input data. The transformed data is then passed through to a linear model! Although the entire network, including the linear model portion is trained end-to-end, we can see that MLPs are essentially automatic feature extractors!

That is to say, when using raw data of nearly any form, in most applications you want to hand make features from the data so that it has more meaning, before training a machine learning model on it. An example would in image classification, you might want to extract the number of holes in your images, or maybe extract the edges of the image, and then create a model using those

features rather than the raw pixels. The MLP sidesteps this laborious process through automatically extracting features!

2.4 Convolutional Neural Networks

Almost 20 years ago, convolutional neural networks (CNNs) were introduced as a variant of vanilla MLPs modeled after the visual cortex of animals [11]. The general motivation was that the visual cortex in mammals consists of layers of simple cells and complex cells, and as an image is being processed through the cortex, progressively richer features of the image are detected [12]. Appropriately, CNNs, as we'll see, consist of stacks of convolutional layers (simple cells) and pooling layers (complex cells) and have been shown to learn progressively higher-level features in the form of filters in convolutional layers [2]. CNNs are particularly well suited for handling grid-like data where the data's structure contains information, such as audio signals in 1D, images in 2D, and videos in 3D. In many machine learning models, such as vanilla MLPs, when handling data with structure, in order to learn based on the given data we're forced to vectorize our data, making it into a single vector (rather than a matrix or some other structure). Taking into account that structure is thought to help improve the performance of models.

In order to describe what a CNN is, it helps to first examine what a single layer of a CNN looks like. In general, each layer of a CNN consists of a set of learnable "filters" (which are the weights of the network), a non-linear threshold (the same as in MLPs), and possibly a pooling (also called subsampling) operation. The key operation is the convolution. Given an $n \times n$ filter (which is just a weight matrix)

$$\mathbf{F} = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nn} \end{bmatrix}$$

and an $n \times n$ patch of an image

$$\mathbf{I} = \begin{bmatrix} i_{11} & \cdots & i_{1n} \\ \vdots & \ddots & \vdots \\ i_{n1} & \cdots & i_{nn} \end{bmatrix},$$

the convolution of the filter and the image patch is defined as

$$\mathbf{F} \star \mathbf{I} = \sum_{j=1}^{n} \sum_{k=1}^{n} w_{jk} i_{jk}.$$

The idea is that you slide the filter over the image, convolving the filter with each patch of the image as it slides. We can vary the number of pixels the filter moves each time its slides by specifying a stride (so with a stride of 1 the filter sees every possible patch of the image). Each convolution produces an output, and the resulting outputs of all of the convolutions of a filter with an image produces a feature map, or the output of that layer which also has a matrix structure.

We can make this more concrete with an example. In the part a of the figure below, we see the result of the convolution of a 3×3 filter

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

with a patch of an image

$$\mathbf{I} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix},$$

given in the feature map by

$$\mathbf{F} \star \mathbf{I} = 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 = 4.$$

We then see in part b of the figure the resulting feature map when the filter is convolved across the entire image.

(a) The result of the first convolution

(b) The resulting feature map

Figure 6: An example of the convolution of an image and a filter [13]

A non-linear threshold is then applied element wise to the resulting feature map. As in the case of MLPs, the ReLU function is the most common threshold for CNNs. Finally, the feature map is passed through a pooling operation. Pooling operations work in much the same way as the filters, in that they slide across the feature map and preform operations involving patches of the feature map. The difference is that the pooling operations have no learnable weights. The most common pooling operation today is max pooling, which simply finds the maximum value of a patch of the feature map (usually 2×2 patches). In this way the feature map is downsized. The pooling is usually applied with a stride equal to the size of the patches being pooled, so that the patches don't overlap. The resulting feature map from the pooling operation is then the output of our layer!

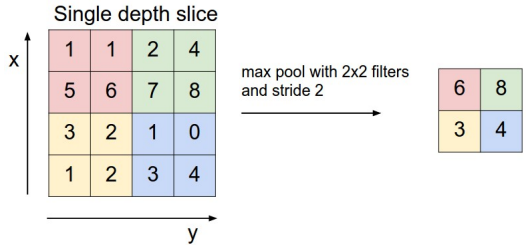

Figure 7: An example of the pooling operation [14]

When we have a multilayer network, successive layers then act on the feature maps output from previous layers, rather than the image itself. After the core convolutional layers in a CNN, it's common practice to then vectorize the final feature map, and pass this vector through a one layer MLP (usually called a fully connected layer), where we can then preform business as usual by passing the output through an appropriate function (e.g. identity, sigmoid, softmax), and the appropriate loss function. We can then train our CNN through backpropagation, just like for MLPs.

Putting this altogether, most CNN structures are made of multiple convolutional layers, each consisting of: filters, a non-linear threshold, and a pooling operation. Once we have the desired number of convolutional layers, the output is then vectorized and passed through a fully connected layer, after which it produces the desired output. In the figure below we can see this basic architecture in the first CNN, LeNet [11].

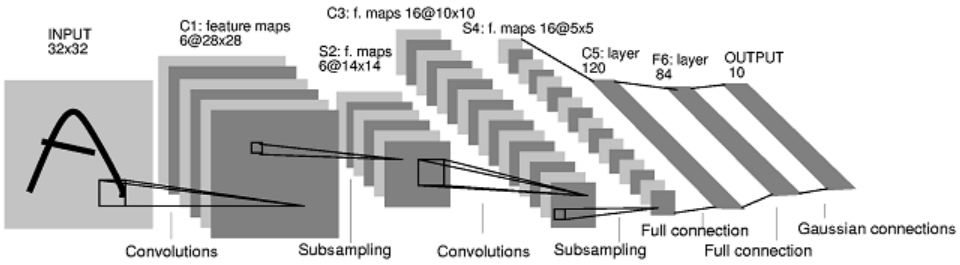

Figure 8: An example CNN architecture, LeNet [11]

Because the filters slide over the entirety of the image and feature maps, but only see a small window of it at a time, the filters capture low level representations of the image at first, like edges or other small motifs, and then farther along as you go down the network, the filters learn more high level representation, like objects or labels. In the figure below are learned filters from AlexNet [15].

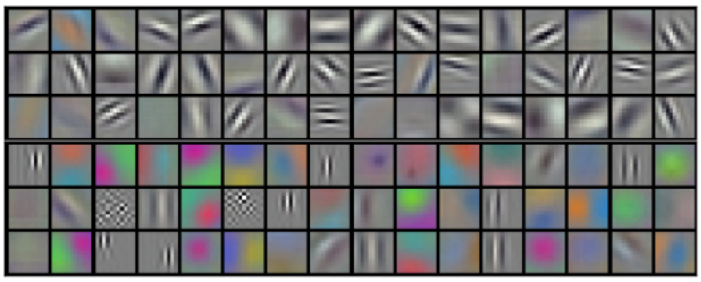

Figure 9: Learned filters from AlexNet [15]

2.5 Relation of MLPs to CNNs

While CNNs sort of resemble MLPs, it's not immediately clear how one gets from MLPs to CNNs. The key is in sparse connections and weight sharing. By sparse connections, we mean not every output of one layer is fed into every neuron of the next layer, as is the case in vanilla MLPs (which are fully connected). By weight sharing, we mean that we force the weights of some of the neurons of the same layer to be the same. It can be seen when we force these two restrictions on a vanilla MLP, we arrive at the convolutional operations of the filters in CNNs.

2.6 Inductive Biases

When considering a specific learning task, there are in most cases (such as image classification) an uncountable infinity of valid hypotheses. In order to construct models that are able to learn and

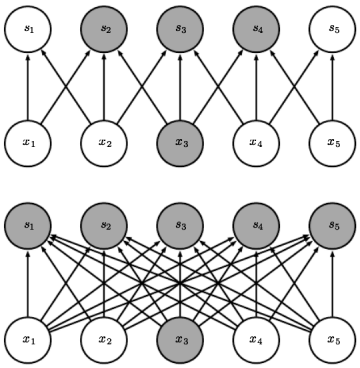

Figure 10: The top image shows sparse connections, in comparison to the fully connected bottom image [16]

generalize from data, we must bias the hypothesis space we're considering. The specific bias one bakes into a model is called the inductive bias. There's inductive bias in every machine learning model! In the case of linear and logistic regression and the perceptron, our inductive bias was that we could represent the desired task in the form of a linear combination of the data. It's not so clear what the inductive bias of an MLP is, since it can approximate nearly any function arbitrarily well. In [4] it's described as "smooth interpolation between data points".

A common idea as to why CNNs work so well in practice for images and similarly structured data is that the inductive bias of the networks restricts us to a class of hypotheses that work well specifically for structured data. This means that our inductive bias, which can be thought of as "local patterns in images manifest themselves at multiple portions in images", is a good inductive bias. We get better results for image classification (and any other task unvolving images) going from vanilla MLPs to CNNs because we're restricting the class of functions we're considering to a class better suited specifically for images.

3 Training Neural Networks in Practice

3.1 Practical Considerations

In order to succeed in training neural nets in practice, there are many different design choices that need to be taken into account. These include but are not limited to architecture choices (number of layers, size of each layer, etc.), initialization choices, training algorithms (mostly variants of gradient descent), hyperparameters for your training algorithms, and regularization choices. We'll consider a few of these in this section.

3.1.1 Initialization

A good initialization scheme is important in training neural networks, as it can potentially speed up the training process or allow the algorithm used during training to find a better network configuration (among other benefits) [10]. We can't initialize all of the weights in our network to the same value, like 0 or 1, as this would make all of the weights receive the same updates during the training process. What we'd like is to initialize the weights to small, random, non-zero values. The "best" choice of random initialization for a given network is largely based on the choice of threshold function. The most common initialization while using ReLU activations, is the He initialization [17], which says that weights should be initialized using a normal distribution with mean 0 and variance $2/fan_{in}$ (i.e. $\mathcal{N}(0, 2/fan_{in})$), where fan_{in} is the number of incoming connections to a neuron.

3.1.2 Data Splitting and Early Stopping

In nearly all applications of machine learning today, you won't train your models on all of your data. Instead you'll use a "train-test split" of the data (alluded to earlier in the introduction to supervised learning). Some large portion of the data (usually around 80%) is exclusively used to train your model. The remaining portion (usually around 20%) is used as test data to evaluate the model (it's important here to note that the test data is never used as part of the training process). In other settings, perhaps when performing cross validation or some form of hyperparameter tuning, it might make sense to split the data into three different sets instead, so that you train on one set, you use another portion as pseudo-test data to get an idea of how well your model is generalizing during training (while using this information to influence your training), and you use the final set as your actual test data.

One setting where a three way split makes sense is early stopping [18]. When training neural networks, and machine learning models in general, it's common to run into a situation, as illustrated in the figure below, where even though your performance is improving on your training data, it stops improving for your test data. This is known as overfitting [4]. In the setup of early stopping, we use a three way split of the data. The model is trained using the training set. The pseudo-test set is used to monitor the generalization performance of the model. When our model's accuracy doesn't improve on this pseudo-test set for a certain amount of training steps, or if the accuracy doesn't improve by some specified amount, we stop training. The model is then evaluated on the test set. This serves as a form of regularization, or a form of biasing our model in favor of improving generalization (and thus reducing overfitting).

3.1.3 Stochastic Gradient Descent

One of the most important practical considerations for training neural networks is using stochastic gradient descent instead of gradient descent [3]. Stochastic gradient descent is exactly the same

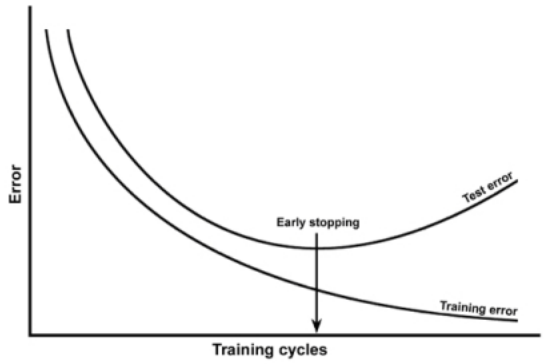

Figure 11: Overfitting and the role of early stopping [19]

algorithm as vanilla gradient descent except for one key difference: instead of updating the weights after each pass over the data set, we update the weights after seeing a certain mini-batch (i.e. some subset) of the data set (SGD technically refers to the case of a mini batch of size 1, while stochastic mini-batch gradient descent refers to any mini-batch size). This way, we make multiple weights updates per pass over the data set, instead of just one, and these weight updates serve as a noisy/stochastic approximation of the true gradient of our empirical risk over the entire data set. This allows for faster convergence in practice on larger data sets.

3.2 Modern Frameworks

If you want to train a small MLP on a small amount of data, there are many general machine learning frameworks out there which have MLPs as a built in model. While these frameworks aren't going to be optimized specifically for MLPs, they can be a decent way to start playing around with them. One widely use machine learning framework which you can train MLPs in, along with most other standard machine learning models, is Sci-kit Learn [20].

When one wants to experiment with different network architectures, training algorithms, or take advantage of optimized software that will let you train large networks with lots of data, you need a framework built specifically for neural networks. The two most common frameworks are currently Theano and Tensorflow [21][22].

Keras is a high level wrapper for both Theano and Tensorflow, which allows you to get off the ground quickly with neural networks [23]. The experiments performed for the purpose of this thesis were carried out in Keras, and so I'll follow now with two examples, training a vanilla MLP, and training a CNN with a standard architecture.

Let's say we have `d` dimensional input data, and there are `n` possible output classes. If we'd like to build an MLP, we have to make a few design choices, as discussed earlier in this section. I'll simplify the choices to deciding on the number of layers in our MLP, and the number of neurons per layer. Let's say we'd like a `3` layer network, with `9` neurons per layer (the structure of the example figure in the MLP section). Then defining this MLP in Keras is done as follows:

```
#Define our model
model = Sequential()
#Add in our first hidden layer, consisting of 9 neurons
#Note that we used He initialization as described earlier, and we have our input size of d
```

```
model.add(Dense(9, init='he_normal', input_shape=(d,)))
#Add a ReLU activation after the first layer
model.add(Activation('relu'))
#Add in our second hidden layer, consisting of 9 neurons
model.add(Dense(9, init='he_normal'))
#Add in another ReLU activation
model.add(Activation('relu'))
#Add in our third hidden layer, consisting of 9 neurons
model.add(Dense(9, init='he_normal'))
#Add in another ReLU activation
model.add(Activation('relu'))
#Add in our output layer of size n
model.add(Dense(n))
#Add in a softmax activation
model.add(Activation('softmax'))
```

We then need to train our MLP on some data, which we'll call `X_train` and `Y_train`. We'll use the categorical cross entropy error function as described earlier. We'll use stochastic gradient descent with learning rate `alpha` and mini-batch size `mini`, for `100` epochs. And just like that we've built an MLP ready to be trained!

```
#Declare the optimization algorithm we're using, in this case SGD
sgd = SGD(lr=alpha, momentum=0.0, decay=0.0, nesterov=False)
#Compile model
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
#Fit our model to the training data
model.fit(X_train, Y_train, batch_size=mini, nb_epoch=100, verbose=1)
```

We'll now build a CNN. Again, let's say we have `input_dim=(d×d)` dimensional input data, and there are `n` possible output classes. We'll make an architecture similar to that of LeNet [11]. In particular, we'll use two layers of `5×5` filters followed by `2×2` max pooling, with `20` filters in the first layer and `50` in the second. We'll then use one fully connected layer of `250` neurons. Then defining this CNN in Keras is done as follows:

```
#Define our model
model = Sequential()
#Add our first layer of 20 5x5 filters
#Note that we used He initialization as described earlier, and we have our input size of
    input_dim
model.add(Convolution2D(20, 5, 5, init='he_normal', input_shape=input_dim))
#Add a ReLU activation after the first layer
model.add(Activation('relu'))
#Add in 2x2 max pooling after the first layer
model.add(MaxPooling2D(pool_size=(2,2)))
#Add our second layer of 50 5x5 filters
model.add(Convolution2D(50, 5, 5, init='he_normal'))
#Add a ReLU activation after the second layer
model.add(Activation('relu'))
#Add in 2x2 max pooling after the second layer
model.add(MaxPooling2D(pool_size=(2,2)))
#Vectorize the feature maps
```

```
model.add(Flatten())
#Add in a fully connected layer of 250 neurons
model.add(Dense(250))
#Add in our output layer of size n
model.add(Dense(n))
#Add in a softmax activation
model.add(Activation('softmax'))
```

We can then train this CNN using the exact same code we used to train our MLP. And just like that we've built a CNN ready to be trained!

3.3 GPUs

Most common machine learning models can't be trained in parallel, which means they're bottlenecked by the fact that they have to train on a single processor. This includes neural networks. While small MLPs can run in a feasible amount of time using one processor, you're not going to be able to train a large CNN on your laptop, or even a computer with decent processing power. In order to train large networks, GPUs (graphics processing units) are needed. GPUs are good at simultaneously performing large amounts of simple operations. In particular, since most neural network operations can be framed as matrix operations, GPUs excel in taking advantage of these operations to speed up training and evaluation.

4 Examining CNN Architectures

While CNNs have been massively successful in image classification and many other similar tasks, there's no underlying theory as to why they actually work so well in practice. There are lots of intuitions as to why they work well, but there's largely only empirical evidence to back these up. One of the main ways in which CNNs aren't well understood is their architectures. Building a CNN requires making a number of design choices, including but not limited to the number of convolutional layers, the number of filters in the convolutional layers, the stride at which to apply filters, and the size and arrangement of filters in each convolutional layers. These can roughly be considered the core components of the architecture of a CNN. There's also a myriad of other design choices that can be made that go outside of the typical convolution, threshold, pool architecture, such as whether to use residual connections [24], whether to use some modified form of convolutions like dilated convolutions [25], whether to use a different pooling scheme than max pooling [26][27], or even whether to use pooling at all [28]. And then there are further regularization strategies such as drop out [29] or batch normalization [30]. Choosing the "best" CNN in practice comes down to experimenting with lots of these design choices and seeing which performs best for the task at hand.

It's clear from the highly parametric nature of CNN architectures that it's more than likely futile to try to understand some general theory of how to "best" build them by taking into account all of these choices. Instead, it seems more worthwhile to examine some specific structural choice in detail, and then see if we can discover some phenomenon that can be generalized to other structural choices.

To this effect, I examined three structural choices. The first is the idea of a so called "convolutional bottlenecking", where we replace large filters with stacks of smaller filters with the same receptive field, the second is the idea of the receptive field of the a network, and the third is the idea of the width of the network. In discussing the three structural choices, it helps to first discuss receptive fields, as bottlenecking builds on this notion. The width of a network is separate from these other notions, so will be discussed last. In the following chapter I'll detail my experiments with these choices.

4.1 Receptive and Effective Receptive Fields

The receptive field (RF) of a filter in a CNN is the dimension of the patch of the image (or feature map if it's not in the first layer) it operates on, e.g. the RF of a 3×3 filter is 3×3, the RF of a 5×5 filter is 5×5, and the RF of a 7×7 filter is 7×7. We can generalize this notion of a receptive field of a filter to the receptive field of a layer of a CNN, or to the receptive field of an entire CNN. We refer to this general notion of the receptive field as the effective receptive field (ERF). When referring to the ERF of a layer of a CNN, we mean the dimension of the patch of the image (or feature map if it's not in the first layer) that an output of that layer (located in the feature map) receives information from. When referring to the ERF of an entire CNN, we mean the dimension of the patch of the image that an output of the last layer (located in the final feature map being vectorized for the fully connected layer) of the network receives information from. We'll make both of these notions concrete in the following sections with examples.

4.1.1 Calculating the effective receptive field of a layer

We'll first examine the ERF of a layer of a network. Let's say we have a layer of a CNN consisting of a stack of two 3×3 filters (each with stride 1), followed by a ReLU non-linearity, followed by 2×2 max pooling (with stride 2). If we work our way sequentially through this layer it's simple to calculate the ERF. After the first filter, the ERF is just the RF of the filter, namely 3×3. After

the second filter however, the ERF is now 5×5. This can be visually checked in the figure below. Next, when we apply the ReLU to our current feature map, the ERF isn't affected, as the ReLU

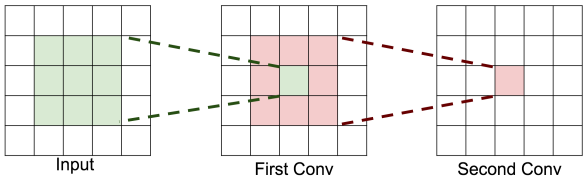

Figure 12: A stack of two 3×3 filters, the ERF of the red square is 5×5, as it's determined by a 5×5 patch of the input [14]

is applied elementwise to the feature map. After we apply the max pooling operation however, our ERF is increased to 6×6, as can be visually checked in the figure below, which requires a bit of explanation. The first grid represents the input, the second grid is the feature map after both filters are applied, and the final grid is the feature map after the max pooling is applied. Note that the blue (and green-blue) shaded boxes in the first grid correspond to the patch of the input seen by the blue box in the second grid (and similarly with the green and green-blue shaded boxes), which is a 5×5 patch (as calculated). The red outlined boxes in the second grid then correspond to the patch of the first feature map that one max pooling operation sees, which corresponds to the red outlined box in the third grid, the final feature map. Working backwards, we see that one node in the final feature map receives information from a 6×6 patch of the input image. Thus 6×6 is the final ERF of our layer!

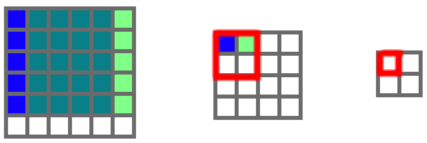

Figure 13: A visual representation of how pooling affects ERF

Finding the ERF of an entire network is a bit more involved (we'll have to make some assumptions about the structure of our network), but will build upon this basic example.

4.1.2 Calculating the effective receptive field of a network

For the purposes of this section, we'll assume that a layer of our network consists of a stack of stride 1 convolutions, followed by a ReLU threshold, followed by a 2×2 max pool with stride 2. We'll refer to the number of such layers as ℓ, and we'll refer to the ERF of the stack of convolutions in a each layer as $f \times f$ (in our example above, we found the ERF of a stack of two 3×3 filters is 5×5). We'll denote the effective receptive field of the network by $rf_{(\ell,f)} \times rf_{(\ell,f)}$. This type of network will only yield square ERFs, so we'll work instead with f and $rf_{(\ell,f)}$ instead of $f \times f$ and $rf_{(\ell,f)} \times rf_{(\ell,f)}$.

It's clear that for one layer networks, $rf_{(1,f)} = f + 1$. We arrived at this conclusion in our basic example in the last section (note that it was a network with $f = 5$ and $\ell = 1$, and we calculated $rf_{(1,5)} = 6$), but this can be seen more generally by noting that the two operations in a single layer are the convolutions of the filters and the max pooling. Thus if we know the ERF of the filters in

our layer is f, it follows that the max pooling only increases f by 1 (this can be seen with a similar argument as in the last section).

From here, we can define $rf_{(\ell,f)}$ recursively, in terms of $rf_{(\ell-1,f)}$ and $rf_{(1,f)}$ for $\ell > 1$. Namely,

$$rf_{(\ell,f)} = (rf_{(1,f)} - 2) + (2rf_{(\ell-1,f)}).$$

This can be motivated by the fact that if we know the ERF for a $\ell - 1$ layer network, we can work backwards from $rf_{(\ell-1,f)}$ to get $rf_{(\ell,f)}$ by noting that the pooling operation doubles $rf_{(\ell-1,f)}$, and the filter convolutions add $rf_{(1,f)} - 2$ to get the final ERF. To see this, suppose we have a network with $f = 5$ and $\ell = 2$, so we're adding a layer to the example from the last section. This is roughly sketched out in the figure below, which needs some explanation. Since we already know the ERF of a $f = 5$, $\ell = 1$ network is 6×6, we can see that a node in the final feature map has an ERF of 6×6 in the feature map following the first layer (represented by the red boxes). Each box in that feature map comes from a max pooling operation in the previous intermediate feature map (represented by the black and yellow outlined boxes), and thus we see our single output node has an ERF of 12×12 (this is where the $2rf_{(\ell-1,f)}$ factor comes from) in this intermediate feature map (i.e. the feature map after the first layer of filters). Each box in the 2×2 patch that the pooling operation acted on then comes from a $f = 5$ patch in the input image (represented by the blue and green shaded boxes). Since these blue and green boxes delimit the ERF of the node in the final feature map, we see that the final ERF of this network is 16×16 (this is where the $rf_{(1,f)} - 2$ factor comes from).

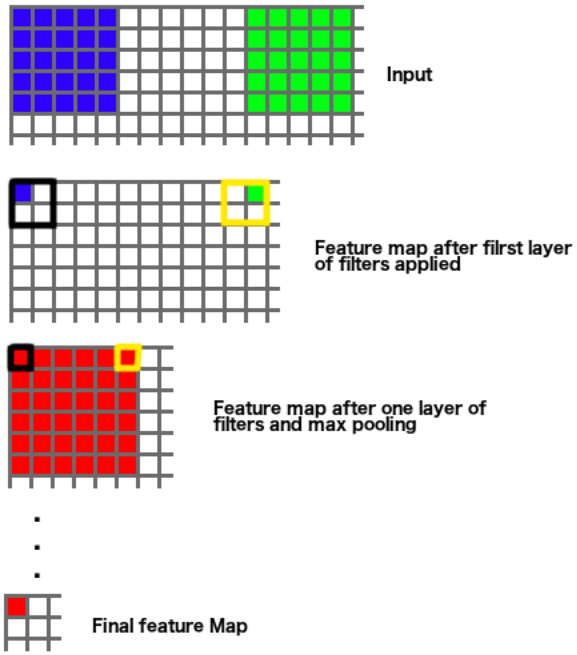

Figure 14: A visual representation of finding the ERF for an arbitrary network

It'd be nice to then have a closed form expression for $rf_{(\ell,f)}$ (even though most of our calculations will be for shallow networks where this recursive definition is enough). I claim that

$$rf_{(\ell,f)} = 1 + f \sum_{i=0}^{\ell-1} 2^i = 1 + f(2^\ell - 1)$$

(which holds for $\ell = 1$), which can be shown through induction. The base case is simple, as $1 + f(2^\ell - 1) = 1 + f = rf_{(1,f)}$. For the inductive step, suppose $\ell > 1$ and $rf_{(\ell-1,f)} = 1 + f(2^{\ell-1} - 1)$. Then using the recursive definition, we have as desired

$$\begin{aligned} rf_{(\ell,f)} &= (rf_{(1,f)} - 2) + (2rf_{(\ell-1,f)}) \\ &= (f-1) + (2(1 + f(2^{\ell-1} - 1))) \\ &= f - 1 + 2 + 2f(2^{\ell-1} - 1) \\ &= f + 1 + f2^\ell - 2f \\ &= 1 + f(2^\ell - 1). \end{aligned}$$

Thus we can easily find the ERF of any network that fits within our assumptions. The table below calculates the ERF for several architectures which we consider in our experiments (the top row denotes the receptive field of the filters in each layer, the left most column denotes the number of layers in the network).

$\ell \backslash f$	3	5	7
1	4	6	8
2	10	16	22
3	22	36	50
4	46	76	96

Figure 15: ERFs for given ℓ, f combinations

The idea of the effective receptive field is important as it's one rough way of encoding the representational power of our network. It's not clear whether there's a direct relationship between either the ERF, or some combination of ERF and other structural information, which can guide us in finding a CNN with good generalization performance.

4.2 Bottlenecking

One of the recent architectural techniques used in CNNs has been a so called "convolutional bottlenecking," in which large filters in the convolutional layers are replaced with stacks of smaller filters with the same ERF [24][31][32]. This was hinted at in our ERF calculations, as we saw that the ERF of a stack of two 3×3 filters is 5×5. Knowing this fact about the ERF, we can treat a stack of two 3×3 filters as roughly equivalent to one 5×5 filter. We can similarly show that a stack of three 3×3 filters is roughly equivalent to one 7×7 filter, and so on. This can also be generalized to incorporate 1×1 convolutions.

The justification for this practice so far has been that it reduces computation and the number of parameters, while maintaining the same ERF [14][32]. However, none of the studies which used bottlenecking examined systematically the effect of the technique on the generalization performance of CNNs. This question goes hand in hand with the question of whether the ERF can guide us to finding a CNN structure which generalizes well. I.e. if we're keeping the ERF of the network the same, should we expect a change in generalization performance when bottlenecking it?

4.3 Wide Networks

When talking about the width of a network, we simply mean the number of filters in each layer of the network (if we have a stack fo filters, we're not referring to the total number of filters in

the stack, but the number of filters in each slice of the stack). It was examined in [33] that CNNs could be trained to achieve state of the art results by sacrificing depth for width. This motivates a question of whether the interplay of width, depth, and ERF can guide us in building a CNN which generalizes well.

5 Examining CNN Architectures: Experiments

In experimenting with the structural choices discussed in the last chapter (bottlenecking, ERF, and width), we progressed as follows. We started by examining whether bottlenecking various networks with layers of 5×5 or 7×7 size filters aided generalization. We found that generalization accuracy wasn't affected by the bottlenecking. We posited that this was because the ERFs of the networks were maintained through the bottlenecking, so we began looking at different structural choices that are used to increase ERFs in networks (mainly strided convolutions, max pooling, and dilated convolutions), rather than keep them the same (bottlenecking). In this way we were looking at ERF explicitly as a potential way to increase generalization. We found max pooling to be the most effective choice with respect to increasing generalization accuracy, so we decided to stick to the convolution, threshold, pool architecture. We further decided to examine the interplay of width, depth, and ERF of a network. In this chapter I describe these experiments in detail.

The experiments in this chapter were all carried out on the CIFAR-10 data set [1]. All experiments were carried out in Keras [23], using a Theano backend [21], on a Linux machine with a NVIDIA Titan X GPU. Unless otherwise noted, we used batch sizes of 128, ReLUs were used as thresholds between layers, weights were initialized using He initialization [17], convolutions used same padding to preserve dimension [34], and networks trained using the Adadelta algorithm (a variant of SGD) with the default hyperparameters in Keras [35].

5.1 Bottlenecking Experiments

For the bottlenecking experiments, we examined the bottlenecking of both 5×5 and 7×7 filters in 1, 2, 3, and 4 layer networks. We use the notation of $XXXPXXXPXXXP$ to refer to a 3 layer network, where each layer consisted of stacks of 3 $X \times X$ filters (e.g. $333P333P333P$ is a 3 layer network consisting of stacks of 3×3 filters in each layer, $7P7P$ is a 2 layer network consisting of 7×7 filters in each layer). All the networks we looked at had a width of 16 in the first layer, 32 in the second layer, 64 in the third layer, and 128 in the fourth layer, and used a fully connected layer of 250 neurons as the deepest layer. Further, the networks were trained using early stopping with a patience of 5 epochs, where the network with a split of the training data of 10% (so we trained on 45000 images, and used 5000 to monitor early stopping). We'll refer to each batch of networks by their respective ℓ, f from the previous chapter. The results of the experiments are contained in the tables below.

As seen from the results, there's no clear pattern as to where bottlenecking filters in a network helps or hurts generalization accuracy. So in order to move forward, we had to frame our question differently. We came in wondering whether networks that are equivalent up to a bottlenecking of filters would generalize differently (namely would the bottlenecked network improve or hinder generalization). Our experiments suggest the bottlenecking neither improves or hinders generalization. So we posited that this was caused by the ERF of the networks remaining constant through each ℓ, f batch. This motivated us to ask whether we could find a direct relationship between ERF and generalization, which was the topic of our next set of experiments.

5.2 Playing Around with ERF

We began with experiments focused on all convolutional networks (valid padding [34], no intermediate operations other than thresholds). We found that adding more convolutional layers increased generalization accuracy to a point (a certain number of layers), at which accuracy began to decrease. We then examined how adding pooling layers and strided 2 convolutions into an all convolutional

$rf_{(1,5)} = 6$	Test Acc.
$33P$	0.61
$5P$	0.60

$rf_{(2,5)} = 16$	Test Acc.
$33P33P$	0.67
$33P5P$	0.69
$5P33P$	0.66
$5P5P$	0.67

$rf_{(3,5)} = 36$	Test Acc.
$33P33P33P$	0.69
$33P33P5P$	0.68
$33P5P33P$	0.70
$33P5P5P$	0.71
$5P33P33P$	0.67
$5P33P5P$	0.66
$5P5P33P$	0.69
$5P5P5P$	0.70

$rf_{(4,5)} = 76$	Test Acc.
$33P33P33P33P$	0.64
$33P33P33P5P$	0.67
$33P33P5P33P$	0.69
$33P33P5P5P$	0.65
$33P5P33P33P$	0.69
$33P5P33P5P$	0.69
$33P5P5P33P$	0.71
$33P5P5P5P$	0.70
$5P33P33P33P$	0.67
$5P33P33P5P$	0.62
$5P33P5P33P$	0.63
$5P33P5P5P$	0.64
$5P5P33P33P$	0.69
$5P5P33P5P$	0.67
$5P5P5P33P$	0.68
$5P5P5P5P$	0.69

Figure 16: Results of Bottleneck Experiments for 5×5 networks

$rf_{(1,7)} = 8$	Test Acc.
$333P$	0.61
$7P$	0.60

$rf_{(2,7)} = 22$	Test Acc.
$333P333P$	0.66
$333P7P$	0.68
$7P333P$	0.63
$7P7P$	0.65

$rf_{(3,7)} = 50$	Test Acc.
$333P333P333P$	0.66
$333P333P7P$	0.64
$333P7P333P$	0.70
$333P7P7P$	0.68
$7P333P333P$	0.65
$7P333P7P$	0.64
$7P7P333P$	0.68
$7P7P7P$	0.66

$rf_{(4,7)} = 96$	Test Acc.
$333P333P333P333P$	0.10
$333P333P333P7P$	0.55
$333P333P7P333P$	0.65
$333P333P7P7P$	0.63
$333P7P333P333P$	0.10
$333P7P333P7P$	0.63
$333P7P7P333P$	0.66
$333P7P7P7P$	0.66
$7P333P333P333P$	0.10
$7P333P333P7P$	0.57
$7P333P7P333P$	0.10
$7P333P7P7P$	0.61
$7P7P333P333P$	0.64
$7P7P333P7P$	0.63
$7P7P7P333P$	0.62
$7P7P7P7P$	0.66

Figure 17: Results of Bottleneck Experiments for 7×7 networks

network with same padding affected generalization accuracy (both operations increasing the ERF faster than an all convolutional network). We found that both pooling and strided convolutions increased generalization accuracy, but pooling increased it more. We also found a similar phenomenon of increasing layers helping generalization accuracy to a point.

We chose to examine pooling networks in greater detail (and in a more principled manner), due to the results from this section looking promising. These experiments are the subject of the following section.

5.3 Width, Depth, and ERF Experiments

Our goal in this set of experiments was to explore whether we could find a relationship between width, depth, ERF, and generalization accuracy. In particular, we consider networks with $f = 3$ (specifically one 3×3 filter in each layer), $\ell \in \{1, 2, 3, 4\}$, and width $k \in \{8, 16, 32, 64, 128\}$, and networks with $f \in \{5, 7\}$, $\ell \in \{1, 2\}$ (specifically stacks of 2 3×3 filters or stacks of 3 3×3 filters in each layer), and width $k \in \{16, 32, 64, 96, 128\}$. Further, for the $f = 3$ networks, we also considered $k \in \{68, 70, 72, 80, 96\}$ for $\ell = 1$, $k \in \{96, 112, 120, 122, 124\}$ for $\ell = 2$, and $k \in \{192, 256, 384, 448, 512\}$ for $\ell = 3$ and $\ell = 4$. Note that for a given network we train, the width is the same for every layer in the network. For each network considered, we trained for 75 epochs on the full training set of 50000 images, and we trained each network 10 separate times (in order to average the runs out). The results of the experiments are contained in the table and figure below.

k	$f:3, l:1$	$f:3, l:2$	$f:3, l:3$	$f:3, l:4$	$f:5, l:1$	$f:5, l:2$	$f:7, l:1$	$f:7, l:2$
8	0.57	0.61	0.59	0.56	-	-	-	-
16	0.58	0.65	0.67	0.65	0.59	0.68	0.59	0.67
32	0.58	0.66	0.69	0.67	0.58	0.67	0.58	0.67
64	0.60	0.66	0.70	0.70	0.60	0.70	**0.60**	**0.71**
68	0.60	-	-	-	-	-	-	-
70	**0.60**	-	-	-	-	-	-	-
72	0.55	-	-	-	-	-	-	-
80	0.56	-	-	-	-	-	-	-
96	0.51	0.68	-	-	0.61	0.72	0.56	0.60
112	-	0.67	-	-	-	-	-	-
120	-	0.68	-	-	-	-	-	-
122	-	**0.69**	-	-	-	-	-	-
124	-	0.68	-	-	-	-	-	-
128	0.26	0.68	0.73	0.73	**0.61**	**0.73**	0.45	0.29
192	-	-	0.74	0.75	-	-	-	-
256	-	-	0.75	0.75	-	-	-	-
384	-	-	0.76	0.76	-	-	-	-
448	-	-	**0.76**	**0.76**	-	-	-	-
512	-	-	0.63	0.76	-	-	-	-

Table 1: Results of width, depth, and ERF experiments, **bold** accuracies represent best width for a particular ℓ, f configuration

From these results we see two broad themes: deeper networks generalize better, and wider networks generalize better (to a point). We thought that this point, which we deemed the "breaking point", of a ℓ, f network configuration to be interesting, since it represents the maximum width of a

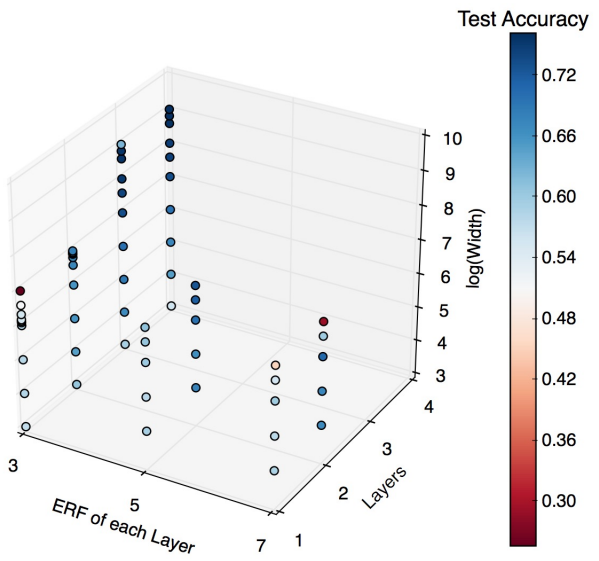

Figure 18: Visualization of width, depth, and ERF experiment results

network that we're able to train for a specific ℓ, f configuration (where the width doesn't affect the ERF). We searched for this breaking point in the $f = 3$ networks (we didn't have enough time to find it in the $f = 5, 7$ networks), but couldn't find a clear relationship between the ERF of a network and its breaking point. It does look like, however, that deeper networks have a larger breaking point.

We don't have a good answer as to what causes the breaking point of a network. Our basic idea is that it has something to do with the network complexity, which is hard to quantify (part of what we were trying to do with the ERF). It could possibly be the case that shallower networks with large widths might introduce poorer local minima into the the weight space of the network, causing the networks to get stuck during training. Finding the cause of this breaking point is left as further research.

6 Conclusions

In this thesis we provided a brief introduction to supervised learning, multilayer perceptrons, and convolutional neural networks. We then introduced some specific structural techniques used in building convolutional neural networks, and one way to try to capture the representational power of a network (the effective receptive field). The results of experiments that examined these structural techniques were then detailed. Although we couldn't meet our lofty goal of completely understanding the structural techniques, we did find a property of networks we examined in the "breaking point" of the networks, namely, the value of network width at which further increases no longer improve generalization performance. This looks to be an interesting avenue for future research.

References

[1] Krizhevsky, Alex, and Geoffrey Hinton. "Learning multiple layers of features from tiny images." Unpublished (2009).

[2] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *Nature* 521.7553 (2015): 436-444.

[3] Bottou, Lon. "Large-scale machine learning with stochastic gradient descent." *Proceedings of COMPSTAT'2010.* Physica-Verlag HD, (2010). 177-186.

[4] Mitchell, Tom M. *Machine learning.* McGraw-Hill, Inc., (1997).

[5] Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from data.* Vol. 4. New York, NY, USA:: AMLBook, 2012.

[6] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[7] Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks." *AISTATS.* Vol. 15. No. 106. 2011.

[8] Cybenko, George. "Approximation by superpositions of a sigmoidal function." *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (1989): 303-314.

[9] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Cognitive modeling* 5.3 (1988): 1.

[10] Bishop, Christopher M. *Neural networks for pattern recognition.* Oxford University Press, (1995).

[11] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324.

[12] Hubel, David H., and Torsten N. Wiesel."Receptive fields and functional architecture of monkey striate cortex." *The Journal of Physiology* 195.1 (1968): 215-243.

[13] http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/

[14] http://cs231n.github.io/

[15] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in Neural Information Processing Systems.* (2012).

[16] Goodfellow, Ian J., Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT Press (2016).

[17] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." *Proceedings of the IEEE International Conference on Computer Vision.* (2015).

[18] Prechelt, Lutz. "Early stopping-but when?." *Neural Networks: Tricks of the trade.* Springer Berlin Heidelberg, (1998). 55-69.

[19] http://documentation.statsoft.com/STATISTICAHelp.aspx?path=SANN/ImportedFromProceed/TestDataandEarlyStopping

[20] Pedregosa, Fabian, et al. "Scikit-learn: Machine learning in Python." *Journal of Machine Learning Research* 12.Oct (2011): 2825-2830.

[21] The Theano Development Team, et al. "Theano: A Python framework for fast computation of mathematical expressions." arXiv preprint arXiv:1605.02688 (2016).

[22] Abadi, Martn, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467 (2016).

[23] Chollet, François. Keras. GitHub repository, (2017). https://github.com/fchollet/keras.

[24] He, Kaiming, et al. "Deep Residual Learning for Image Recognition." arXiv preprint arXiv:1512.03385 (2015).

[25] Yu, Fisher, and Vladlen Koltun. "Multi-scale context aggregation by dilated convolutions." arXiv preprint arXiv:1511.07122 (2015).

[26] Graham, Benjamin. "Fractional max-pooling." arXiv preprint arXiv:1412.6071 (2014).

[27] Lee, Chen-Yu, Patrick W. Gallagher, and Zhuowen Tu. "Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree." *International Conference on Artificial Intelligence and Statistics.* 2016.

[28] Springenberg, Jost Tobias, et al. "Striving for simplicity: The all convolutional net." arXiv preprint arXiv:1412.6806 (2014).

[29] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

[30] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

[31] Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013).

[32] Szegedy, Christian, et al. "Rethinking the Inception Architecture for Computer Vision." arXiv preprint arXiv:1512.00567 (2015).

[33] Zagoruyko, Sergey, and Nikos Komodakis. "Wide residual networks." arXiv preprint arXiv:1605.07146 (2016).

[34] Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

[35] Zeiler, Matthew D. "ADADELTA: an adaptive learning rate method." arXiv preprint arXiv:1212.5701 (2012).