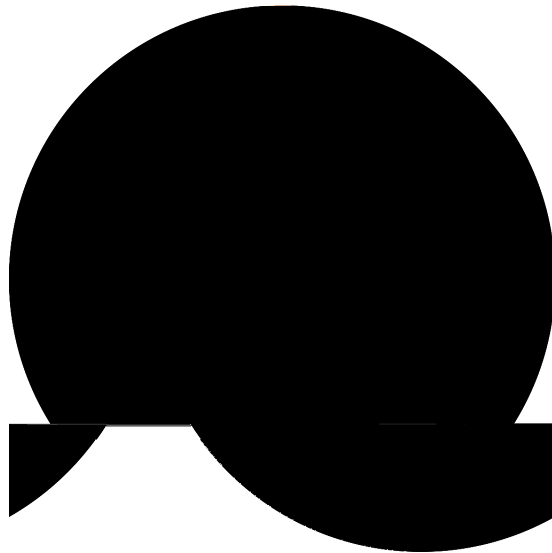
Examining the Structure of Convolutional Neural Networks

Serge Aleshin-Guendel



Computer Science Honors Thesis
Boston College

Advised by
Professor Sergio Alvarez

May 10, 2017

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.

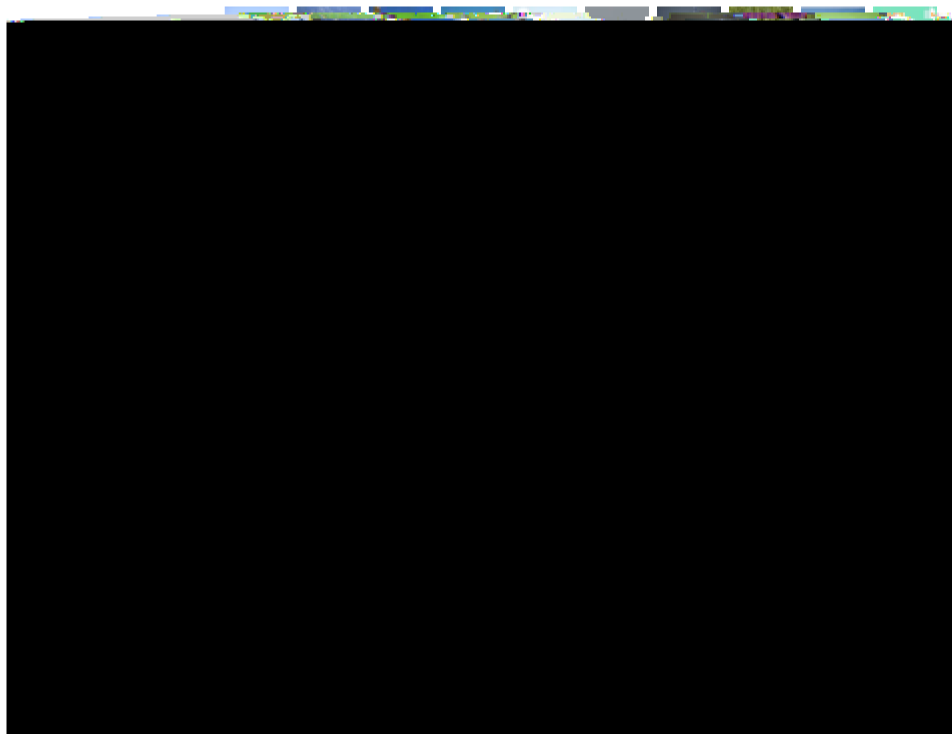


Figure 1: Example images from the CIFAR-10 data set [1]

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

we wish to compute

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

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 $h_{m} \in H$ 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 $h_{m}(x)$ for every $x \in X$. 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, $D^{test} = \{(x_i^{test}, y_i^{test})\}_{i=1}^{N_{test}}$. The goal now isn't strictly to find the h_{m} defined as above. Instead,

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.

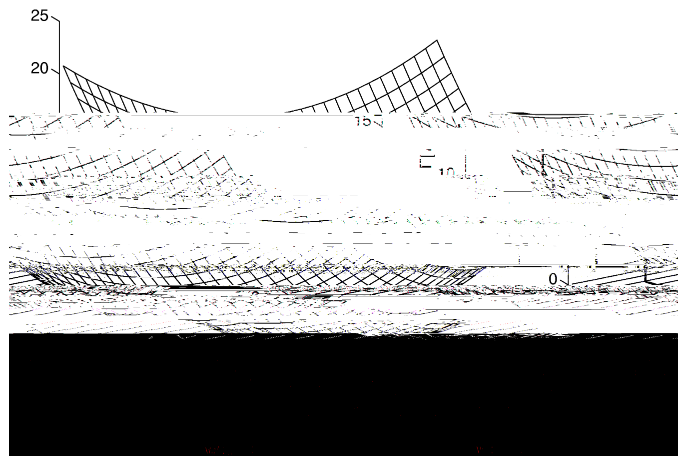


Figure 3: A visualization of a 2 dimensional weight surface, where $E[w]$ is the empirical risk for a given point in weight space [4]

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 η

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

1: **while** not converged **do**

2: $\quad g \leftarrow -\nabla f(\mathbf{w}_{curr})$

3: $\quad \mathbf{w}_{curr} \leftarrow \mathbf{w}_{curr} + \eta g$

4: **end while**

5: **return** $\mathbf{w}_{curr}$

---

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

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

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.

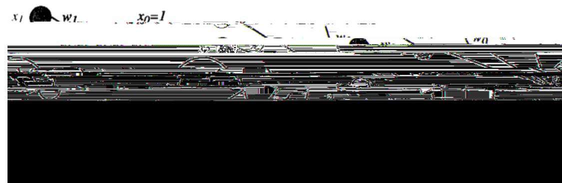


Figure 4: A graphical representation of a perceptron [4]

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 $i \in \{1, \dots, d_\ell\}$, 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

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

with a patch of an image

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

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].

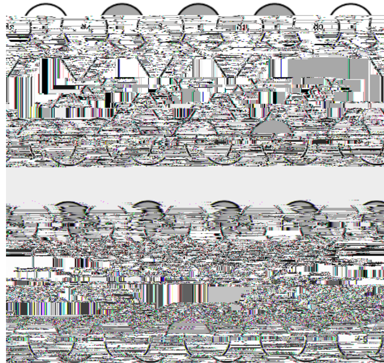


Figure 10: The top image shows sparse connections, in comparison to the fully connected bottom image [16]

generalize from data, we must bias the hypothesis space we're considering. The speci c 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

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. $N(0, 2/fan_{in})$), 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

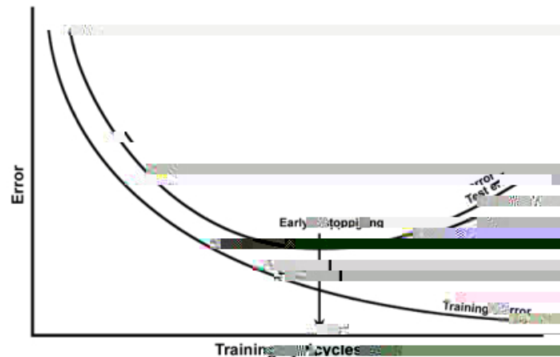


Figure 11: Over tting and the role of early stopping [19]

algorithm as vanilla gradient descent except for one key di erence: 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 speci cally 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 di erent 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 speci cally for neural networks. The two most common frameworks are currently Theano and Tensor ow [21][22].

Keras is a high level wrapper for both Theano and Tensor ow, which allows you to get o 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

```
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

```
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'))
```

---

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

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).

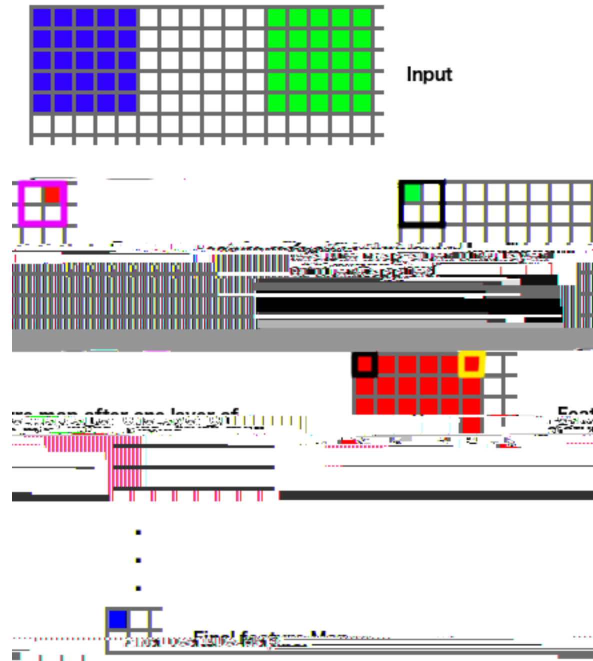


Figure 14: A visual representation of finding the ERF for an arbitrary network

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

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 exper-

$rf_{(1;5)} = 6$	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 a ected 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 nd a relationship between width, depth, ERF, and generalization accuracy. In particular, we consider networks with $f = 3$ (speci cally one 3

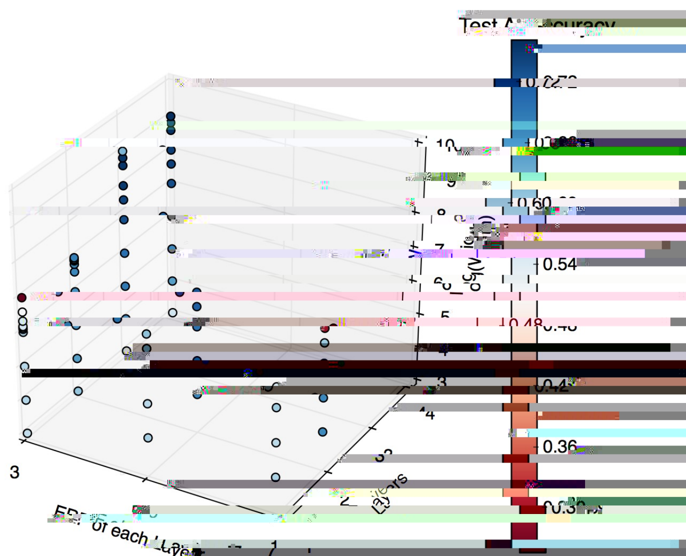


Figure 18: Visualization of width, depth, and ERF experiment results

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 Geo rey Hinton. \Learning multiple layers of features from tiny images." Unpublished (2009).

[2] LeCun, Yann, Yoshua Bengio, and Geo rey 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 Recti er 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., Geo rey 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 elds and functional architecture of monkey striate cortex." The Journal of Physiology

[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 Arti cial 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 over tting." Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[30] Io e, 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).