2019-01-01

A Novel Set Of Weight Initialization Techniques For Deep Learning Architectures

Diego Aguirre
University of Texas at El Paso

A NOVEL SET OF WEIGHT INITIALIZATION TECHNIQUES FOR DEEP LEARNING ARCHITECTURES

DIEGO AGUIRRE

Doctoral Program in Computer Science

APPROVED:

Olac Fuentes, Ph.D., Chair

Nigel G. Ward, Ph.D.

Amy Wagler, Ph.D.

Shahriar Hossain, Ph.D.

Stephen L. Crites, Jr., Ph.D.
Dean of the Graduate School

to my

MOTHER and GRANDMOTHER

with love

A NOVEL SET OF WEIGHT INITIALIZATION TECHNIQUES FOR DEEP LEARNING ARCHITECTURES

by

DIEGO AGUIRRE, BSCS, MSCS

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Doctoral Program in Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2019

Acknowledgements

I would like to express my gratitude to my mentor, Dr. Olac Fuentes of the Computer Science Department at The University of Texas at El Paso (UTEP). His guidance and support have shaped the person that I am. We have worked together for more than 9 years, and he has played a very important role in my academic career. He introduced me to the field of machine learning and taught me what I know. I attribute my success to his guidance and support. I also wish to thank the other members of my committee, Dr. Nigel G. Ward, Dr. Amy Wagler, and Dr. Shahriar Hossain. Their feedback, support, and valuable guidance were fundamental to the completion of this work.

As a special note, I wish to thank Dr. Nigel G. Ward for his support and direction. He has always taken the time to read my work and has provide me with invaluable feedback. His guidance has been crucial for my formation. Dr. Ward is one of the researchers that I admire the most, and I feel very fortunate to have worked under his supervision.

I would also like to use this space to thank Dr. Ann Gates. She has always believed in me, and has provided me with opportunities that have helped me grow personally and professionally. My career path would not be what it is if it was not for her. I am very happy to have met her and I will always be grateful for what she has done to help me. She is one of the leaders that I admire the most, and I feel privileged to have the opportunity to learn from her. She has had a tremendous impact in our community, and I hope I can do the same one day.

Other professors that have had a strong, positive impact in my career include Dr. Martine Ceberio, Dr. Natalia Villanueva-Rosales, Dr. Vladik Kreinovich, and Dr. Yoonsik Cheon. I am thankful for their teachings, insightful conversations, and support throughout my years at UTEP. I am also thankful for having had the opportunity to collaborate with other students, such as Ivan Gris, Gerardo Cervantes, and Anthony Ortiz. I feel privileged to have worked with them in various research and work opportunities. Their help and

support has been key for my professional development. I hope we can all collaborate in the future.

Lastly, I would like to thank the Computer Science Department professors, staff, and fellow students. Everybody's work has helped me and many other students achieve what we thought impossible. I feel very grateful to be part of this community, and I hope I can be of service to them in the future.

Abstract

The importance of weight initialization when building a deep learning model is often underappreciated. Even though it is usually seen as a minor detail in the model creation cycle, this process has shown to have a strong impact on the training time of a network and the quality of the resulting model (68). In fact, the implications of choosing a poor initialization scheme range from leading to the creation of a poorly performing model to preventing optimization techniques (like stochastic gradient descent) from converging (26).

In this work, we introduce and evaluate a set of novel weight initialization techniques for deep learning architectures. These techniques use an initialization data set (extracted from the training data set) to compute the initial values of a layer's weights. They also use properties about the problem and network architecture to better select initial weight values. The first scheme in our set focuses on the initialization of dense and convolutional layers that make use of the ReLU activation function. This technique aims to maximize neuron heterogeneity in a given layer while 1) keeping the standard deviation of the neurons' outputs uniform and 2) controlling the initial number of active neurons. Our second technique is inspired by the observation that, normally, the weights learned by convolutional neural networks closely resemble Gabor filters. We propose to initialize regular convolutional layers with weights chosen so that the result is that of Gabor filters. The third technique in our set also targets convolutional layers. It selects weight values that allow filters in the layer to activate only when processing interesting regions in the input space (as defined by feature extraction techniques, such as SIFT and FAST). The fourth technique in our set aims to initialize recurrent layers. Recurrent layers reuse the same weight matrix in succeeding matrix multiplication operations, which can lead to the amplification or dilution of output values and gradients. We mitigate this problem by selecting weight matrices that maintain a uniform output response across the range of possible inputs. All of these initialization techniques add an extra step to initialize the output layer. We propose to use

the ground truth information in the initialization data set to select weights that minimize the network's initial loss value. Experimentally, we show that our initialization schemes outperform state-of-the-art techniques (Glorot, He, and LSUV) by a considerable margin. Our methods allow for the creation of models that train faster and perform up to 40% better than if another technique was used for initialization.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep learning is a subfield of machine learning concerned with the creation of hierarchical models (artificial neural networks) that learn from data. These models have had great success solving challenging problems, which has allowed deep learning to become a very active area of research. Some examples of these problems include autonomous vehicles (9, 11, 31, 60), language recognition (15, 34, 54, 55, 83), machine translation (6, 13, 14, 36, 87), object detection and recognition (18, 19, 24, 27, 41, 51, 53), text and image generation (22, 39, 62, 71, 73, 75, 80, 82, 83), image segmentation (5, 10, 42, 49), advertising (12, 17, 77), autonomous vehicles (45), and disease detection (40, 46, 48, 50, 63, 67). The amount of conferences and events dedicated to the area clearly reflects its popularity. Examples include the conference and workshop on Neural Information Processing Systems (NIPS), IEEE's conference on Computer Vision and Pattern Recognition (CVPR), and NVIDIA's GPU Technology Conference (GTC). Tracks for autonomous machines, autonomous vehicles, supercomputing, and video analytics are gaining a lot of traction.

Because of the great impact deep learning has started to have, new techniques to train and improve the quality of deep learning models have been presented (32, 65, 81). The research community has been aggressively pushing the state-of-the-art by developing new architectures, regularization techniques, and more efficient and effective optimization algorithms (27, 30, 69). Examples of state-of-the-art deep learning models include Inception (69), ResNet (27), VGG (64), LSTM (28), and NASNet (86). These network architectures excel at tasks like object detection/recognition and speech processing. Systems like Alexa, Siri, and Google's Voice Assistant take advantage of these type of networks. Beyond network design, regularization techniques, like dropout, have allowed us to improve metrics

such as precision, recall, and accuracy across a wide variety of architectures.

Other contributions in the field lie around numerical optimization, where new variants of gradient descent have been proposed. The most noteworthy examples include the RMSProp (72) and Adam (35) optimizers, which have allowed us to train networks faster and converge at a better location in parameter space. These optimizers are the go-to options when building a deep learning model.

Besides advances in network architecture design, regularization techniques, and optimizers, companies like Nvidia, Intel, Google, and AMD have introduced new specialized hardware to train deep learning models. This has allowed deep learning practitioners to solve more complex problems by training deeper neural architectures. This is the case of networks such as Inception (69) on data sets like ImageNet (16).

Although steady progress has been made in the above-mentioned areas, insufficient attention has been given to weight initialization. This work focuses on this neglected topic. It presents a taxonomy of weight initialization techniques, surveys state-of-the-art approaches, introduces novel methods to initialize a wide variety of network architectures, and evaluates such methods on a variety of popular data sets.

1.1 Motivation

Training a deep learning model can be seen as a 5-phase process. Deep learning is a data-driven field, where data is the fuel that enables machines to learn. Thus, the first phase consists in gathering data. The second phase is devoted to preparing the data, so it is suitable for training. This includes tasks like cleaning the data, removing redundant information, standardizing/normalizing feature values, data set reduction, whitening, and data augmentation. Once the data is ready for training, the third step consists of building/selecting the network architecture. The fourth step consists in training the network on the collected data and evaluating its performance given a selected metric. In the last step, the results are analyzed and decisions on how to improve the network's performance are

made. This might include gathering more data, tweaking hyper-parameters, changing the network architecture, using another optimization algorithm, etc.

In this 5-step process, many decisions have to be made. To name a few, the deep learning practitioner has to determine the number of hidden layers, the number of units per layer, the type of layers, the cost function, and the optimization algorithm. With so many decisions to be made, selecting the initial weights of the network may, at first glance, seem like a minor detail.

However, weight initialization is an important step that has a strong impact on the training time of a network and the quality of the resulting model (68). In fact, improper weight initialization can prevent gradient descent from converging or lead to a poorly performing model (26). Understating this is important when experimenting with a neural network architecture, as poor results could be incorrectly attributed to other aspects of the task modeling process (such as the architecture itself or the quality of the data set). Training time is also affected by poor weight initialization (47). It is known that one of the weaknesses of deep learning is the amount of time and computational power required to build a model. Starting at a good area in the parameter space allows practitioners to run experiments faster and create better performing models.

Selecting the initial values of the parameters of a network is not an easy task. In fact, the whole point of training a neural network is to find good values for its parameters. To do this, we have been using first-order iterative optimization algorithms, such as gradient descent. We also employ a significant number of regularization techniques and other tricks in the process. This is because it is not trivial to find good parameter values at training time or at initialization time.

Researchers have found that good weight values meet certain properties. For example, small weight values are preferred over large ones. This is why people tend to use L2 regularization when training. Sparse weight matrices have also been shown to be beneficial in certain contexts (21). Parameter sharing is one of the reasons why convolutional neural networks outperform other architectures where spatial locality of the input values is relevant

(21). Additionally, analysis of the kernels learned while training a convolutional neural network on image data has shown their similarity to Gabor filters (44). Researchers, like He and Glorot, have proposed weight initialization techniques that aim to keep the variance of the input gradient and the output gradient the same by initializing the weights to numbers that are not too small nor too big. Others have proposed to force weight matrices to be orthonormal at initialization time to enhance the network's representational capacity by having heterogeneous units (47). The purpose of all this research is to understand what values the weights should have to prevent problems such as the vanishing and exploding gradient, train the network in a reasonable time, and build models that perform well.

Although multiple weight initialization techniques have been proposed, they come with a set of limitations and assumptions that might not hold true in practice. For example, some techniques do not take into account the type of activation functions that the units in a specific layer employ (such as the family of ReLU-like functions (4)). Others do not exploit the representational capacity of the network; while others do not take into account the type of layers the network employs. The reason why we have multiple types of networks and layers is because they are all meant to solve different types of problems. The differences in network architectures should also be reflected in how we initialize them. For example, we know that the weight values in a convolutional layer have different properties than the ones in an LSTM layer. To the best of our knowledge, there is no initialization technique that takes this into account.

1.2 Objective

The goal of this work is to introduce and evaluate a set of novel initialization techniques for state-of-the-art network architectures. Our main focus lies in building initialization techniques that minimize the network's error at initialization time, as minimizing the error is correlated with the performance of the network when solving the task it is meant to solve.

It is also this work's objective to compare our methods to state-of-the-art initialization techniques across a number of network architectures and types of data sets. More concretely, we focus on initializing feed-forward, convolutional, and recurrent neural networks. To do this, we use the following data sets: MNIST (38), CIFAR-10/100 (37), ImageNet (16), and MapTask (3).

1.3 Thesis Statement and Research Questions

Thesis statement: *Weight initialization in neural networks can be improved by 1) using statistical information extracted from a subset of the training data set, and 2) incorporating information about the architecture and the data set.*

Research Questions:

1. How can we exploit statistical information extracted from a training batch to initialize the parameters of a neural network? Is the overhead added to the process worth it?
2. What properties of the network architecture can we use to further tune parameter values at initialization time?
3. How can we incorporate information about the data set/problem to improve weight initialization techniques across network architectures?

1.4 Contributions

The contributions of this research are the following:

1. The development of four initialization techniques that reduce the starting error of dense, convolutional, and recurrent neural networks.
2. A deeper understanding of weight initialization across network architectures and types of data sets.

1.5 Summary and Outline

In summary, this research is focused on understanding and improving weight initialization across popular network architectures. This work is expected to enable deep learning practitioners to train neural networks faster, which will lead to more experimentation and advances in the field. The initialization schemes presented in this work are expected to help devices with limited resources run traditionally expensive training procedures by selecting weight values where the initial error is low.

This work is organized as follows: Chapter 2 presents a taxonomy of weight initialization techniques and surveys noteworthy examples of each class. It also provides the theoretical foundation necessary to understand the context of this work. Chapter 3 provides an overview of all initialization schemes introduced in this work and the challenges that motivated their creation. Chapters 4, 5, 6, and 7 describe each proposed initialization scheme in detail. Finally, Chapter 8 presents our conclusions and revisits our research questions.

Chapter 2

Background and Related Work

2.1 A Taxonomy of Weight Initialization Techniques

There are different ways in which weight initialization techniques can be grouped. For instance, one could categorize them based on the properties they force the initial weights to have. Another possibility is to classify them based on the type of assumptions they make about the training data (e.g. normally distributed) or the inputs/outputs a layer should produce. Additionally, we could categorize them based on the overhead they add to the overall training process. However, our proposed taxonomy divides weight initialization techniques into three groups: *Data Independent*, *Data Dependent*, and *Pre-Training* approaches. We believe this a natural classification of the techniques as techniques in each class share similar overhead and data set assumptions. The following subsections briefly describe the three groups in this taxonomy, and survey the most representative techniques in each group.

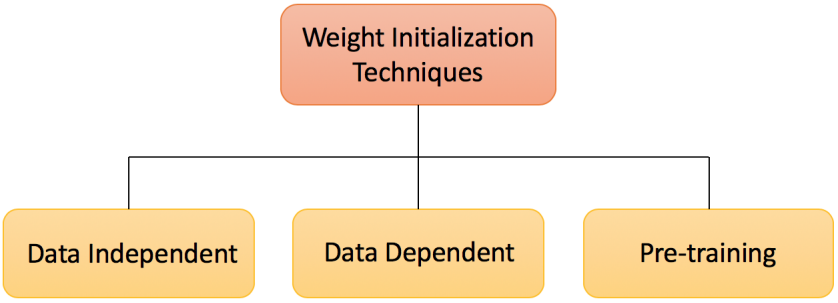

Figure 2.1: Weight Initialization Taxonomy

2.1.1 Data Independent Weight Initialization Techniques (DIWI)

Techniques in this category are the ones that do not make use of any training data to determine the initial parameter values of a network. They usually work by randomly sampling numbers from a normal or uniform distribution and using such numbers as the initial values of the network's parameters. Multiple heuristics have been proposed (20, 26) to select the variance and mean when a normal distribution is used, or the minimum and maximum values when sampling from a uniform distribution. A simple heuristic consists in initializing the network's parameters by sampling from a normal distribution with zero mean and unit variance (standard normal distribution). This type of initialization is one of the most common ways initialization is done. However, convergence rate and model quality are usually not the best. The reason for this is that the variance of the outputs produced by a neuron initialized using this approach grows with the number of inputs. To mitigate this problem, one can normalize the variance of the neurons' outputs to 1 by scaling a neuron's weights by the square root of its number of inputs.

Other approaches have been proposed to select better parameters for the normal distribution. For example, Glorot *et al.* (20) propose to use a standard deviation of $\sqrt{\frac{2}{n_{in}+n_{out}}}$ to initialize linear layers, where n_{in} is the number of inputs the layer is fed and n_{out} is the number of outputs it produces. The idea behind this method is to try to keep the variance of the input gradient and the output gradient the same by initializing the weights to numbers that are not too small nor too big. Let's assume we feed a vector $\boldsymbol{x}$ to a neuron that produces a linear output, where each of the n components of $\boldsymbol{x}$ is multiplied by a weight. The output produced by the neuron can be described as follows.

$$y = w_1x_1 + w_2x_2 + ... + w_nx$$

Glorot *et al.* aim to initialize the weights, so that $Var(y) = Var(x_i)$. Computing the variance of each pair of random variables w_ix_i yields:

$$Var(w_i x_i) = E[x_i]^2 Var(w_i) + E[w_i]^2 Var(x_i) + Var(w_i)Var(x_i)$$

Since both our inputs and weights are assumed to come from a normal distribution with zero mean, the expression can be simplified to:

$$Var(w_i x_i) = Var(w_i)Var(x_i)$$

If we also assume that all x_i and w_i are independent and identically distributed (iid), we can describe the variance of the layer's output y as follows:

$$Var(y) = Var(w_1)Var(x_1) + ... + Var(w_n)Var(x_n)$$

$$Var(y) = n * Var(w_i)Var(x_i)$$

Given that we want $Var(y) = Var(x_i)$, we need $n * Var(w_i) = 1$. As a result, the variance of the weights must be:

$$Var(w_i) = \frac{1}{n_{in}}$$

Analyzing the signals during backpropagation, Glorot *et al.* found that it is also desired that the following holds:

$$Var(w_i) = \frac{1}{n_{out}}$$

The two constraints can only be satisfied if $n_{in} = n_{out}$. Since this is not usually true in practice, they propose to use the following variance:

$$Var(w_i) = \frac{2}{n_{in} + n_{out}}$$

In practice, many implementations of Glorot initialization use a variance of $Var(w_i) = \frac{1}{n_{in}}$, which has shown significant advantages over unit-variance weight initialization (20).

A similar analysis was performed by He *et al.* (26) for ReLU activations. He *et al.* show that Glorot initialization does not work well for ReLU layers, and empirically demonstrate

it by building a 30-layer network that converges under He Initialization, but not under Glorot's. In this approach, a variance of $Var(w_i) = \frac{2}{n_{in}}$ is used for ReLU layers.

Although the analysis presented above focuses on sampling from a normal distribution, the same reasoning can be applied to uniform distributions with zero mean. Both Glorot and He initializations make reasonable assumptions about the input data. However, it is very likely that these assumptions do not hold true for a given data set. This limitation can be overcome by making use of available training data to initialize the network's weights.

2.1.2 Data Dependent Weight Initialization Techniques (DDWI)

In contrast to DIWI approaches, DDWI techniques use a subset of the training data to initialize the network's parameters. An example of this approach is the layer-sequential unit-variance (LSUV) initialization scheme proposed by Mishkin *et al.* (47). In this approach, weights of all units in a layer are pre-initialized using orthonormal matrices, and then an initialization set is extracted from the training data and used to force the variance of the output of each layer to be one. Algorithm 1 shows how this is achieved.

Algorithm 1: LSUV Initialization

```
// Algorithm Definitions:
// Y_l: output produced by layer l
// W_l: weight values associated with layer l
// ϵ: the algorithm's tolerance
// max_iterations: maximum number of iterations

Initialize all weights using orthonormal matrices
for each layer l:
    i := 0
    while |Var(Y_l) − 1| >= ϵ and i < max_iterations:
```

`Perform forward pass with initialization set taken from training data`

`Calculate` $Var(Y_l)$

$W_l \ := \ W_l \ / \ \sqrt{Var(Y_l)}$

$i \ := \ i \ + \ 1$

LSUV initialization was tested on different architectures, and the results showed that training very deep networks is possible if this initialization technique is used. The overhead added is minimal since it is equivalent as processing a single mini-batch, which is done thousands of times when training deep neural networks. LSUV initialization works well with multiple activation functions (such as linear, ReLU, and maxout), and outperforms more sophisticated systems such as FitNets (56).

Another technique that uses a DDWI scheme within a larger, more elaborate process is Weight Normalization (WN)(59). WN is a weight reparameterization process that allows gradient descent to converge more rapidly. This is done by reparameterizing each neuron's weight vector $\boldsymbol{w}$ in terms of another vector $\boldsymbol{v}$ and a scalar parameter g as follows:

$$\boldsymbol{w} = \frac{g}{||\boldsymbol{v}||}\boldsymbol{v}$$

The intuition behind WN is that the direction and magnitude of the weights are separated by reparametrizing vector $\boldsymbol{w}$. This allows optimization techniques (such as stochastic gradient descent) to tweak the network's weights through $\boldsymbol{v}$ and g instead of $\boldsymbol{w}$ directly. An important aspect of this reparametrization is that the norm of $\boldsymbol{v}$ grows monotonically during optimization. This means that if the gradients are noisy, the norm of $\boldsymbol{v}$ would quickly increase; however, this would also force g to decrease. Because of this, the gradient would self-stabilize, helping mitigate the vanishing and exploding gradient problems and allowing for faster convergence rates.

Given that $\boldsymbol{w}$ is defined in terms of g and $\boldsymbol{v}$, it is important to initialize them properly. To do so, Salimans *et al.* propose to use a DDWI approach, where elements of $\boldsymbol{v}$ are sampled from a normal distribution with 0 mean and a standard deviation of 0.05. To initialize a

neuron's g and bias b, they propose to perform an initial feedforward pass through the network using an initialization set, where each neuron computes the following:

$$y_{pre} = \frac{\boldsymbol{v} \cdot \boldsymbol{x}}{||\boldsymbol{v}||}$$

$$y = f(\frac{y_{pre} - \mu(y_{pre})}{\sigma(y_{pre})})$$

To fix the mini-batch statistics of all pre-activations in the network, parameters b and g are initialized as follows:

$$g = \frac{1}{\sigma(y_{pre})}$$

$$b = \frac{-\mu(y_{pre})}{\sigma(y_{pre})}$$

To test the performance of this technique, WN was applied to four different models covering applications in image recognition, generative modelling, and deep reinforcement learning. In all cases, WN showed a faster convergence rate.

2.1.3 Pre-Training Weight Initialization Techniques (PTWI)

Weight initialization techniques in this category start by initializing the weights randomly and pretrain the network on another task. An example of this approach is (66). Sudowe *et al.* propose to collect large amounts of unlabeled data and use it to pretrain the network in a self-supervised way. The artificial task they created to pretain an image classification network is called *Patch Task*, and it is described as follows. Given a patch of pixels extracted from the input, the model is trained to predict its origin out of k possible locations in the original image. The original image is divided into k locations, where each location is a discrete position within the image. Thus, the task is similar to solving a jigsaw puzzle with only one piece missing.

Once the network is trained on this artificial task, the weights are reused when training the network to solve its primary task. When evaluated, it was shown that this method outperforms traditional random initialization and closely matches reusing weight matrices obtained training on the ImageNet dataset.

Chapter 3

A Novel Set of Weight Initialization Techniques

The main contribution of this work is the introduction of four novel weight initialization schemes. This chapter presents an overview of these schemes and the main themes that motivated their creation. It is also this chapter's objective to present the challenges associated with the development of these techniques.

3.1 Overview

Four main weight initialization techniques are presented in this work. They were designed to initialize the following types of layers: dense, convolutional, output, and recurrent. The overarching theme among these techniques is that they 1) use an initialization set to select weight values that produce healthy gradient values and/or 2) employ heuristics that place initial weight values in a desirable region of the parameter space.

3.2 The Initialization Set and Heuristics

Existing weight initialization techniques aim to provide numerical stability during training. That is, the main objective of state-of-the-art weight initialization schemes is to generate weights that have the necessary mathematical properties to produce gradient values that do not explode nor vanish. Ultimately, effective learning depends on the quality of the gradients used during the backpropagation phase. As explained in Chapter 2, He and

Glorot initializations accomplish this by making reasonable assumptions about the data set. Although this has proven to be effective in practice, one can improve on this idea by using a subset of the training data set to aid the initialization process instead of making assumptions. This observation motivated the use of an initialization set in the schemes presented in this work.

Current initialization schemes focus on generating numerically stable weight values. They do not have additional goals that might be beneficial to the overall model creation process. The initialization methods presented in this work not only generate numerically stable values, but also select parameter values that minimize the starting network loss value. After all, the point of training a network is to find parameter values that allow the model to solve a task. Weight initialization strategies should also try to find parameter values that lie in a desirable region of the parameter space that help the network fulfill its purpose. This motivated the use of heuristics in the initialization strategies we present in this work.

One can analyze the weights learned by state-of-the-art models to find desirable mathematical properties or patterns that can guide the design of strong heuristics. For example, it has been found that early convolutional layers in a deep convolutional neural network learn simple filters that detect basic shapes, such as a corners and edges. This piece of knowledge should be exploited by initialization schemes. In the case of ReLU layers, it is highly desirable that neurons do not die during the training process because if they to, the gradient signal dies with them; affecting the whole backpropagation process. This piece of information should guide the weight initialization scheme to select strong and robust parameter values that prevent ReLU units from dying. In general, a good initialization scheme should select parameter values that not only allow for numerical stability during training, but also aid the network accomplish its goal. The initialization schemes presented in this work accomplish this.

3.3 Targeted Layers

The four initialization schemes presented in this work were designed to initialize very concrete types of layers. The first one aims to initialize *dense* and *convolutional* layers in any architecture that uses them in combination with the ReLU activation function. The second and third are tailored specifically to initialize *convolutional* layers. The fourth technique's goal is to initialize *recurrent* layers. We also introduce a complementary initialization scheme that targets the output layer(s). This complementary scheme is meant to be used in combination with the previous ones, as it assumes that good parameter values are selected for the network's hidden layers. The reasons why we chose these layers are the following:

1. Dense layers are used in almost all architectures. They are the foundational block of deep learning. This type of layers have been thoroughly studied. The research community has found desirable mathematical properties for dense layers that can be exploited by initialization techniques.

2. Convolutional networks are the go-to solution to many computer vision and natural language processing problems. Most state-of-the-art solutions to problems like object detection, recognition, and tracking make use of this type of networks. Creating specialized weight initialization techniques for convolutional layers is important due to their widespread use.

3. Recurrent networks are also the go-to solution to many problems that deal with sequences, such as speech recognition, turn-taking modeling, and machine translation. Voice assistants like the ones used by millions of people on their phones make use of this type of networks. Creating specialized initialization techniques for this type of layers has the potential of reducing training times and improve the network's performance.

4. Output layers are one of the most important blocks in a neural network. They map

the representation produced by the last hidden layer to the expected set of values that solve the task at hand. Because they play a crucial role in neural network architectures, the creation of specialized weight initialization for them is important.

3.3.1 Scheme 1 - Overview

The first initialization scheme in the set is designed to initialize dense and convolutional layers that use ReLU (or one of its variations) as their activation function. The idea is to improve on LSUV, the initialization technique proposed by Mishkin *et al.* (47). Like Mishkin *et al.* (47), we also propose to initialize the parameters of layers using orthonormal matrices, and force the output of a layer to have a predetermined standard deviation s using an initialization set. The innovation in our approach is the incorporation of a hyperparameter called the *active fraction* (f) that allows the deep learning practitioner to specify how likely it is for a ReLU unit to produce non-zero activation. The idea is to create ReLU layers with no dead units while still allowing them to introduce non-linearity into the architecture.

3.3.2 Scheme 2 - Overview

Gabor convolutional neural networks (GCNs) have shown to be robust architectures capable of tackling hard computer vision problems (44). They do so by building convolutional layers using hand-crafted Gabor filters. The network learns what Gabor filters to select when processing the input images. Because of the restriction imposed on the network, GCNs have many fewer parameters. Our second initialization scheme builds on this idea. In this technique, we initialize regular convolutional networks with weights chosen so that the result is that of Gabor filters. This allows regular convolutional networks to exploit the scale and rotation invariance properties of Gabor filters (as exploited by Luan (44)) while allowing the network to change the filters as backpropagation tweaks the network's weights.

3.3.3 Scheme 3 - Overview

In the 2000s and early 2010s, computer vision researchers introduced many feature extraction techniques for vision problems. Some of these techniques include SIFT (Scale-Invariant Feature Transform) (43), SURF (Speeded-Up Robust Features) (7), FAST (Features from Accelerated Segment Test) (57), BRIEF (Binary Robust Independent Elementary Features) (8), and ORB (Oriented FAST and Rotated BRIEF) (58). Our third initialization approach attempts to bring some of the benefits of these algorithms to the convolution space. The aim of this scheme is to select weight values for filters in convolutional layers that produce results similar to the ones obtained by these human-crafted feature extraction techniques.

3.3.4 Scheme 4 - Overview

The fourth scheme in the set focuses on initializing recurrent layers. The main idea behind this technique is to initialize the weights of recurrent layers using weight matrices with a condition number close to one. The purpose of this is to achieve numerical stability by having layers that produce controlled activations and therefore gradient values that do not vanish nor explode.

3.3.5 Complementary Scheme - Overview

The last scheme is meant to be used in combination with the previous ones. The intuition behind this scheme is the following. If the last hidden layer in a network is assumed to produce a good representation of the input data, the weights of the output layer can be initialized solving a system of linear equations. For example, assume that the output layer computes the following: $\boldsymbol{Y} = \boldsymbol{H_l} * \boldsymbol{W_o}$, where $\boldsymbol{H_l}$ is the output of the last hidden layer, $\boldsymbol{W_o}$ represents the output layer's weights, and $\boldsymbol{Y}$ represents either the outputs the network is to produce or the arguments to the monotonic non-linear activation function used in the output layer; the weights of the layer can be initialized as follows: $\boldsymbol{W_o} = \boldsymbol{H_l}^+ * \boldsymbol{Y}$, where $\boldsymbol{H_l}^+$ is the pseudo-inverse of $\boldsymbol{H_l}$. The idea is to select weight that are located in the a

region of the parameter space where the network's loss value is low.

Chapter 4

Scheme 1: Initialization of Dense and Convolutional Layers

4.1 Motivation

Dense and convolutional layers have been used in the area for a long time and are still the foundational blocks of modern deep learning architectures. They are the small pieces that form more complex structures. Modern constructs, such as residual blocks, are created using dense and convolutional layers. Robust weight initialization techniques need to consider how these layers are used in the hierarchical representation of data in a neural network.

For decades, the sigmoid activation function (see Figure 4.1) was used to introduce non-linearity into a neural network. Dense and convolutional layers were paired with this activation function to create non-linear hierarchical representations of the input data. Even though this activation function allowed for the creation of well-performing models, there were some disadvantages associated with its use. During backpropagation, the chain rule is used to calculate the gradients that are used to update weight values in the network. One of the advantages of using this activation function is that it is computationally easy to calculate its derivative; however, sigmoid units tend to saturate during training. As illustrated in Figure 4.1, if the input to these units is far from zero, the derivative will be close to zero. When this happens, we say that the unit saturates and gradient signals become zero during backpropagation, preventing weight values from being updated.

The ReLU activation function (see Figure 4.3) has shown to be a better alternative to

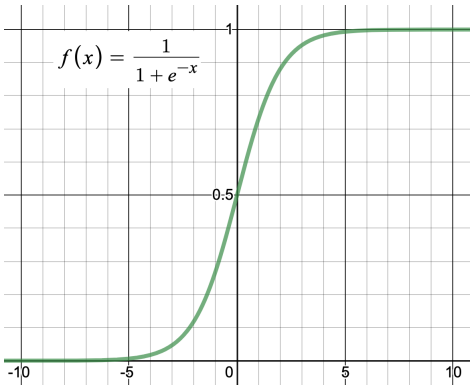

Figure 4.1: Sigmoid Activation Function

the sigmoid function. ReLU units, compared to sigmoid or similar units, allow for faster and effective training. ReLU units exhibit fewer gradient propagation problems since they do not saturate in both directions. Gradients are also easy to compute; they are either one or zero, depending on the sign of the input value. Despite its clear advantages, ReLU units exhibit some problems as well. During training, ReLU units can be pushed to states where they produce non-existent activations for all inputs. That is, the output if zero for all inputs. When this happens, we say that the unit dies and enters a perpetually inactive state where no gradient signals are produced during backpropagation. This is also a form of the vanishing gradient problem. If the number of dead ReLU units is high, the representational capacity of the network will be substantially affected. To mitigate this problem, variations of the ReLU activaton function have been introduced by the research community. Some examples include Leaky ReLU, Parametric ReLU, and ELU. The main idea behind this variations is to change the behavior of the activation function when the input is smaller than 0 to produce a small gradient when backpropagation runs.

4.2 Algorithm

Our first initialization technique aims to initialize any dense, convolutional, and residual layer/block that uses ReLU or one of its variations as its non-linear activation function.

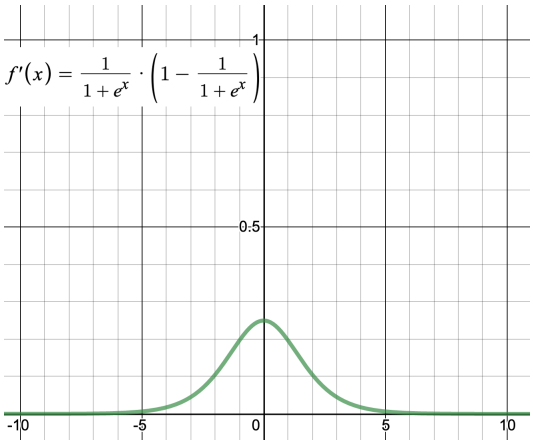

Figure 4.2: Sigmoid Activation Function Derivative

This is a new DDWI scheme that builds on the scheme presented by Mishkin *et al.* (47). The idea is to enhance LSUV by incorporating a series of steps that select weight values that prevent ReLU units from dying, mitigating the problems introduced by vanishing gradient signals. Like Mishkin *et al.*, we also propose to initialize the parameters of layers using orthonormal matrices, and force the output of a layer to have a predetermined standard deviation. The first innovation in our approach is the introduction of a hyperparamter called *goalstd*, which allows the deep learning architect to define the standard deviation that the layer's output is to have. The second innovation in our approach is addition of another hyperparameter called the *active fraction* that determines how likely it is for a unit in a ReLU layer to produce non-zero value. The third innovation in this approach is the use of the Moore-Penrose matrix pseudo-inverse to initialize weights of the network's output layer.

The four steps in this initialization process are: 1) Orthogonalization, 2) ReLU Preparation, 3) Standarization, and 4) Output Layer Initialization

4.2.1 Orthogonalization Step

An adequate initialization scheme should try to exploit the representational capacity of a network. To accomplish this, we initialize the units/filters of a layer using weight vectors

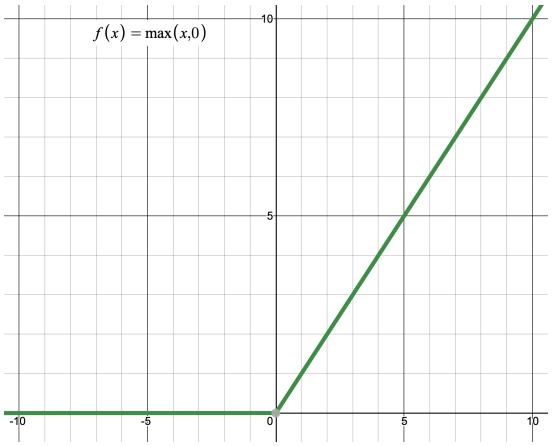

Figure 4.3: ReLU Activation Function

that are orthogonal to each other, as described by Saxe *et al.* in (61).

4.2.2 ReLU Adaptation Step

The ReLU activation function has shown to be a very effective and easy way to introduce non-linearity in a network (4). Due to their strong performance, ReLU layers and their variations (leaky ReLU, parametric ReLU, etc.) are very common in modern deep learning architectures. We propose to incorporate a hyperparameter called the *active fraction* f that can be used to determine how likely it is for a ReLU unit to produce a non-zero value.

To accomplish this, an initialization set of size n is fed to the network to compute the layer's output before the ReLU operation is applied. Let H be the pre-ReLU activation tensor, where $H_{i,j}$ is the output for instance i and unit j. To obtain the desired behavior, b_j, the bias for unit j must be equal to minus the $\lfloor (1-f)n \rfloor$-th order statistic of column j of h. [1]

[1] In practice, we find the kth statistic by sorting the columns of H, but a slightly faster ($O(n)$) vs. ($O(n\,logn)$) implementation is possible (Quickselect algorithm).

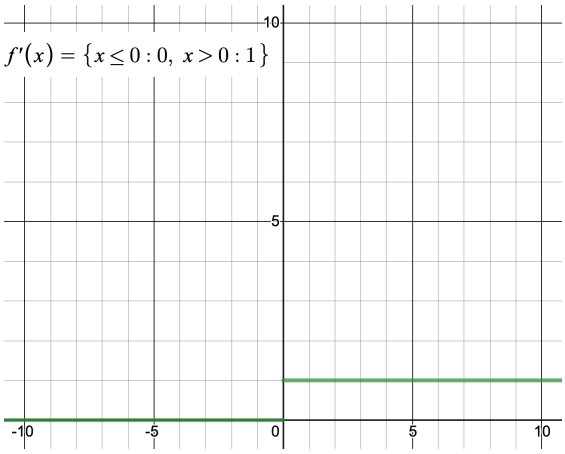

Figure 4.4: ReLU Activation Function Derivative

4.2.3 Standarization Step

Mishkin *et al.* (47) propose to initialize the weights of a layer in such a way that the activations produced by the layer have unit variance. We build on this idea by incorporating a hyperparameter *goalstd* that allows the deep learning practitioner to specify the desired standard deviation of a layer's activations. We do this by computing the layer's output after the ReLU operation is applied using the initialization set. We divide the weight and biases tensors by the corresponding standard deviation of the produced output and multiply that by *goalstd*.

4.2.4 Initialization of Output layers

To initialize the output layer(s) of the network, we also feed an initialization set to the network and compute the output H produced by the last hidden layer. If the network's output is $Y = HW$, we propose to initialize W by setting it to H^*Y, where Y is the network's expected output for the initialization set and H^* is the pseudo-inverse of H. If softmax is used, we replace Y with the corresponding logits the network is to produce. In essence, we are computing the best least-squares solution to $Y = HW$.

Pseudocode 1 shows the complete weight initialization algorithm. The input X, Y

represent the intialization set, h is a vector containing the layer sizes (where h_0 is the dimensionality of the input), f is the active fraction and σ_g is the goal standard deviation for all hidden units. Notice that the pseudocode assumes all layers are fully-connected, but the same initialization scheme can be applied to any layer that uses ReLU.

4.2.5 Overhead

Our initialization technique adds operations that are not performed by standard weight initialization schemes. To analyze this overhead, we need to consider the running time of: 1) the feed-forward pass performed on the initialization set; 2) the creation of orthonormal weight matrices; 3) the algorithm used to find the kth statistic in every column of activation matrices; 4) the operations performed on weight values to produce the predetermined goal standard deviation; and 5) the algorithm used to compute the pseudo-inverse necessary for the initialization of the output layer.

The size of the initialization set is considerably smaller than the size of the complete training set. In our experiments, we set the size of the initialization set to be about 4% of the size of the complete training set. Thus, the cost of the feed-forward pass performed on the initialization set is a small fraction of the time it takes to complete a full epoch. Generating orthonormal matrices using the approach described in (61) requires computing the singular value decomposition (SVD) of each weight matrix in the network. The running time of this is $O(min(h_i h_{i+1}^2, h_{i+1} h_i^2))$, where h_i represents the number of units in layer i. Finding the kth statistic in every column of activation matrices can be done using *quickselect*, which runs in $O(n)$, where n (in our context) is the size of the initialization set. To produce the goal standard deviation, we first compute the standard deviation of the activations of a given unit, which takes $O(n)$, where n is the size of the initialization set. Once this is done, every weight is divided by this value and multiplied by the goal standard deviation. This steps takes $O(n)$, where n is the number of inputs the unit receives. Finally, the cost of initializing the output layer is determined by the time it takes to compute the Moore-Penrose inverse, which is dominated by the cost of computing the

SVD of the activation matrix of the last hidden layer. As previously stated, the running time of performing this operation is $O(min(mn^2, nm^2))$ for an m-by-n matrix. In our context, n is the number of samples in the initialization set, and m is the number of units in the last hidden layer. Thus, for $n > m$, the initialization process takes time $O(n)$ with constant factors depending on the network's architecture.

Empirically, we observed that the running time of the initialization process is about 3 to 6 times of that of processing a training mini-batch of the same size as the initialization set.

Algorithm 1 Weight Initialization

procedure INITIALIZE(X, Y, h, f, σ_g)
  $H^{(0)} \leftarrow X$
  $n \leftarrow len\ (h)$
  for $i = 1$ to n **do**
    $W^{(i)} \leftarrow randn(0,1)_{[h_{i-1} \times h_i]}$ ▷ random matrix of size $h_{i-1} \times h_i$
    $W^{(i)} \leftarrow Orthonormalization(W^{(i)})$ ▷ orthogonalize
    $P \leftarrow H^{(i-1)}W^{(i)}$
    $P \leftarrow sort(P)$ ▷ sort each column of P in ascending order
    $b^{(i)} \leftarrow -P[\lfloor (1-f)h_i \rfloor]$
    $s = \sigma(P)$ ▷ vector of column-wise standard deviations
    ▷ Scale weights and bias to obtain σ_g standard deviation
    for $j = 1$ to h_i **do**
      $W^{(i)}_{[:,j]} \leftarrow W^{(i)}_{[:,j]}\sigma_g / s_j$
      $b^{(i)} \leftarrow b^{(i)}_j \sigma_g / s_j$
    $H^{(i)} \leftarrow relu(H^{(i-1)}W^{(i)} + b^{(i)})$
  $W^{(n)} \leftarrow (H^{(n)})^* Y$
  $b^{(n)} \leftarrow 0$
  return $W^{(1)}, \ldots, W^{(n)}, b^{(1)}, \ldots, b^{(n)}$

4.3 Experimental Results

Two sets of experiments were designed. The purpose of the first set was to analyze how the performance of our method is affected by its hyperparameters. In the second set, our proposed method was compared to popular weight initialization techniques, including Glorot, He, and LSUV. The following subsections describe in detail these two evaluation processes.

Table 4.1: Hyperparameter Values

Hyperparameter	**Values**
Initialization Set Size	128, 256, 512, 1024, 2048
Active Fraction	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
Learning Rate	0.01, 0.001, 0.0001
Goal Standard Deviation	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0

4.3.1 Hyperparameter Evaluation

This set of experiments was designed to analyze how the learning rate, the size of the initialization set, the goal standard deviation, and the active fraction hyperparameters affect the performance of our technique. To accomplish this, we constructed a small convolutional neural network and used our approach to initialize it multiple times using different hyperparameter values. After initialization, the network was trained for 50 epochs using gradient descent and categorical crossentropy as the loss function. We used a mini-batch size of 128 and used the CIFAR10/100 (37) data sets. We exhaustively ran experiments using all combinations of hyperparameter values defined in Table 4.1. The architecture of the convolutional network is described in Table 4.2. All convolutional layers are composed

of 3x3 filters with a stride of 1.

Table 4.2: Convolutional Neural Network Architecture

Layer #	Type	Properties
1	Convolutional 2D - ReLU	Number of filters: 32
2	Convolutional 2D - ReLU	Number of filters: 32
3	MaxPooling 2D	Pool size: 2X2
4	Dropout	Rate = 0.25
5	Convolutional 2D - ReLU	Number of filters: 64
6	Convolutional 2D - ReLU	Number of filters: 64
7	MaxPooling 2D	Pool size: 2X2
8	Dropout	Rate: 0.50
9	Dense - ReLU	Number of units: 512
10	Dropout	Rate: 0.50
11	Softmax	

4.3.1.1 Initialization Set Size Results

Due to the large number of conducted experiments, Table 4.3 only presents a subset of them. We kept the results where the learning rate, the active fraction, and goal standard deviation were set to 0.01, 0.8, and 1.0, respectively, as they are representative of the behavior observed with other hyperparameter values. We also only show the results obtained using CIFAR100 as they are very similar to the ones obtained using CIFAR10.

The results show that, as expected, larger initialization set sizes produce better initial accuracies and lower loss values. However, this behavior is not monotonic; when the initialization set size is exactly the same as the number of neurons in the last hidden layer, performance is extremely poor. This is due to the properties of the Moore-Penrose pseudo-inverse. When we have more instances in the set than units in the layer, the algorithm

Table 4.3: CIFAR100 - Initialization Set Results

Initialization Set Size	Loss after initialization	Loss after 1 epoch	Loss after 50 epochs	Test Accuracy after initialization	Test Accuracy after 1 epoch	Test Accuracy after 50 epochs
128	4.535	4.059	2.180	0.043	0.088	0.431
256	4.566	4.396	2.287	0.041	0.042	0.408
512	14.872	15.956	15.956	0.010	0.01	0.01
1024	4.479	4.459	2.254	0.065	0.033	0.412
2048	**4.410**	**4.050**	**2.132**	**0.110**	**0.104**	**0.439**

solves an over-determined system of equations. When this happens, the pseudo-inverse algorithm finds the least-squares fit solution. If we have more units than examples in the set and thus deal with an underdetermined system of equations, the pseudo-inverse algorithm finds a solution that minimizes the L2 norm of the solution variables. In both cases, the pseudo-inverse algorithm has a regularizing effect on the weight values. If the number of examples in the set is the same as the number of units, the matrix is square and (usually) invertible, thus the solution overfits the initialization set and does not have any regularization effect on the weights, which leads to poor generalization.

4.3.1.2 Active Fraction Results

Table 4.4 (CIFAR100) only shows a subset of the results as well. We kept the results where the learning rate, initialization set size, and goal standard deviation were set to 0.001, 2048, and 1.0, respectively. The results suggest that a value between 0.6 and 0.9 should be used. We speculate that higher active fraction values produce better results because gradients are richer as fewer zeros are used when calculating them. If most of the values produced by

ReLU units are non-zero, gradients do not vanish as backpropagation takes place because the signal does not die (become zero).

Table 4.4: CIFAR100 - Active Fraction Results

Active Fraction	Loss after initialization	Loss after 1 epoch	Loss after 50 epochs	Test Accuracy after initialization	Test Accuracy after 1 epoch	Test Accuracy after 50 epochs
0.1	4.674	4.593	4.572	0.026	0.018	0.027
0.2	4.521	4.564	4.342	0.052	0.035	0.075
0.3	4.476	4.537	3.996	0.067	0.049	0.122
0.4	4.452	4.514	3.649	0.084	0.057	0.164
0.5	4.448	4.494	3.502	0.093	**0.071**	0.188
0.6	4.436	**4.483**	3.392	0.099	0.063	0.205
0.7	**4.419**	4.495	3.313	**0.108**	0.046	0.215
0.8	4.420	4.504	**3.296**	0.106	0.042	**0.218**
0.9	4.438	4.498	3.320	0.098	0.049	0.213

4.3.1.3 Learning Rate Results

For Table 4.5, we kept the results where the active fraction, initialization set size, and goal standard deviation were set to 0.8, 2048, and 1.0, respectively. The results suggest that a large learning rate should be used with our approach. We hypothesize that gradient values are small when using our approach, as our weights are chosen to minimize the network's initial loss value. Therefore, a large learning rate is favored.

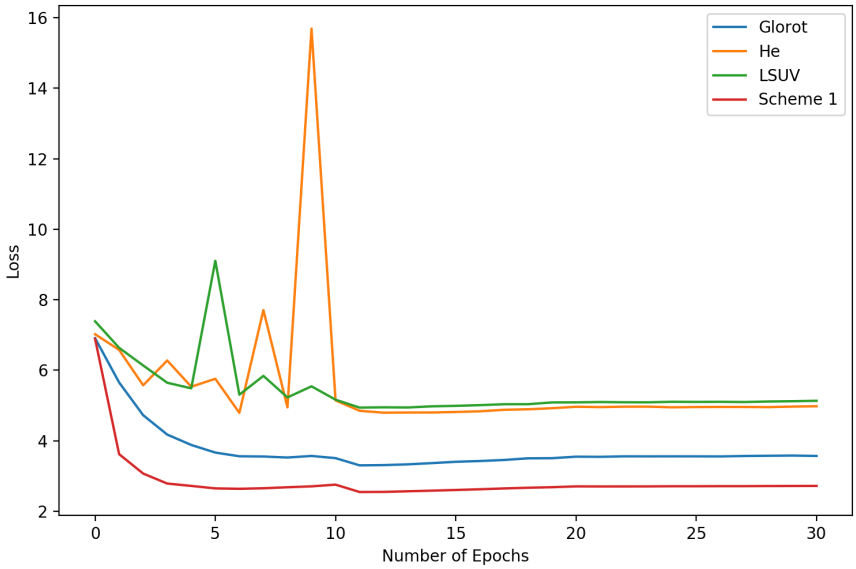

Figure 4.5: InceptionV3 - ImageNet - Loss Results

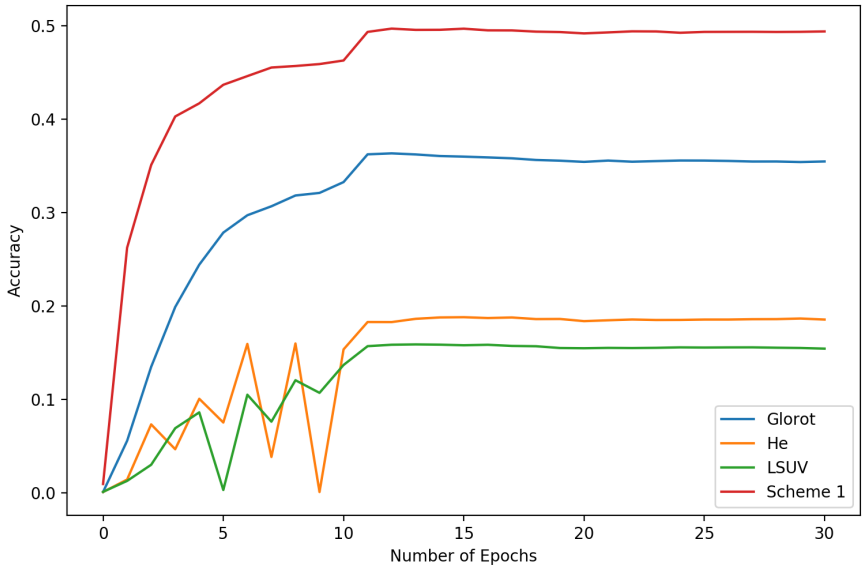

Figure 4.6: InceptionV3 - ImageNet - Accuracy Results

Table 4.5: CIFAR100 - Learning Rate Results

Learning Rate	Loss after initialization	Loss after 1 epoch	Loss after 50 epochs	Test Accuracy after initialization	Test Accuracy after 1 epoch	Test Accuracy after 50 epochs
0.01	4.410	**4.050**	**2.132**	0.110	**0.104**	**0.439**
0.001	4.410	4.504	3.296	0.110	0.042	0.218
0.0001	4.410	4.464	4.231	0.110	0.064	0.079

4.3.1.4 Standard Deviation Results

For Table 4.5, we kept the results where the active fraction was set to 0.8, the initialization set size was 2048, and the learning rate was kept constant at 0.01. The results, presented in Table 4.6, are very similar across all standard deviation values between 0.1 and 3.0. We tried values distant from 0.1 and 3.0, such as 0.001, 10, and 100, and we observed significantly worse performance. We conclude that a value between 0.8 and 3 is appropriate. The complete set of results is publicly available online (2).

4.3.2 Other Initialization Techniques

We compared our method to the following state-of-the-art initialization techniques: Glorot, He, and LSUV. We tested all initialization schemes on the following model architectures: VGG16 (64), VGG19 (64), and InceptionV3 (70). Table 4.11 shows the number of learnable parameters (weights) each model has. At the time of writing, these large, deep models are considered state-of-the-art. They are widely used by the research community to tackle a plethora of computer vision problems. For VGG16 and VGG19, we used exactly the same topology as described in the original paper, but we used filters of size 7x7 for all

convolutional layers. For InceptionV3, we made two changes to the original model: 1) we used filters of size 11x11 in the first layer (as opposed to 3x3), and 2) we applied batch normalization after ReLU (as opposed to applying it before). These changes were made since the research community has improved on these models since they were introduced by making changes similar to the ones we incorporated. We used ImageNet for all experiments with images resized to 128x128. When using our initialization approach, we used an initialization set of size 256 for hidden layers, an initialization set of size 16,384 for the output layers, a value of 0.8 for the *active fraction* hyperparameter, and a goal standard deviation of 1.0. For LSUV, we also used an initialization set of size 256 for hidden layers, and an initialization set of size 16,384 for the output layers. We only trained the models for 1 epoch as we were mostly interested in comparing the strategies in the early stages of the training process. Because the learning rate plays an important role, we repeated the experiments with the following learning rates: 0.1, 0.01, and 0.001.

Tables 4.7, 4.8, and 4.9 present the test loss and test accuracy after initialization (but before training) and after one epoch of training. Our experiments confirm the results obtained by He *et al.* (26). Glorot initialization does not work well with ReLU layers. It consistently performs worse than the rest of the techniques. Our results also show that our method initializes the network in a desirable area in the parameter space; where the initial test accuracy is high and the loss is low. In all experiments, our method outperformed the other techniques in both cost and test set accuracy.

We also trained InceptionV3 on ImageNet for 30 epochs using all initialization techniques to analyze how they compare as the training process advances. We used a mini batch size of 64, regular stochastic gradient descent, and categorical crossentropy as the loss function. We used a learning rate of 0.1 for the first 10 epochs; a learning rate of 0.01 for the next 10 epochs; and a learning rate of 0.001 for the last 10 epochs. Tables 4.10 and 4.12 present the results of these experiments. Figures 4.5 and 4.6 show how the loss and accuracy change with respect to the number of training epochs.

The results are remarkable. Not only was our initialization scheme able to produce

better cost and accuracy numbers after initialization, but it allowed the model to converge at a significantly better region in parameter space. Our final accuracy was almost 40% better than the second best initialization scheme. We expect the research community to adopt our contribution to the field by adding support to our initialization scheme in popular deep learning frameworks. Our Keras code is freely available in our GitHub repository (1).

4.4 Conclusion

In this chapter, we propose a new weight initialization technique, where we use statistics obtained from a subset of the training set to initialize the network's weights. We used orthonormal vectors to initialize the units of a layer, and introduce a hyperparameter called *active fraction* that allows the deep learning practitioner to define a prior on the behavior of ReLU layers. Similar to LSUV, we also incorporate a weight standarization step, where we force the output of each layer to have a predetermined standard deviation. Lastly, we conclude that initializing the last layer using the Moore-Penrose pseudo-inverse of the representation produced by the last hidden layer introduces a regularizing effect on the weights while minimizing the initial loss value. Our results show the advantages of using an initialization set to determine the starting weights of a network. This validates our hypothesis that weight initialization techniques can be improved by explicitly enforcing weight matrices to meet predefined statistical properties at initialization time. Future work includes studying the behavior of our method in other domains beyond image classification, analyzing its behavior in relation to optimization algorithms that employ adaptive learning rates such as Adam and RMSprop, and analyzing how it performs when residual blocks are used. Lastly, it is in our plans to conduct experiments using other activation functions, such as Leaky ReLU, Parametric ReLU, and ELU.

Table 4.6: CIFAR100 - Standard Deviation Results

Goal standard deviation	Loss after initialization	Loss after 1 epoch	Loss after 50 epochs	Test Accuracy after initialization	Test Accuracy after 1 epoch	Test Accuracy after 50 epochs
0.1	4.418	4.438	2.487	0.109	0.053	0.385
0.2	4.425	4.392	2.356	0.104	0.071	0.396
0.3	4.414	4.341	2.304	0.108	0.081	0.408
0.4	4.432	4.329	2.270	0.105	0.075	0.410
0.5	4.425	4.274	2.226	0.105	0.062	0.418
0.6	4.412	4.200	2.166	0.110	0.090	0.438
0.7	4.422	4.194	2.233	0.110	0.087	0.420
0.8	4.421	4.148	2.237	0.107	0.085	0.421
0.9	4.419	4.075	2.167	0.108	0.101	0.434
1.0	4.426	4.110	2.227	0.109	0.087	0.423
1.1	4.419	3.992	2.185	0.112	0.109	0.433
1.2	4.425	3.959	2.132	0.111	0.112	0.445
1.3	4.418	3.925	2.192	0.109	0.106	0.429
1.4	4.426	3.916	2.160	0.104	0.111	0.436
1.5	**4.411**	3.861	2.192	**0.119**	0.123	0.432
1.6	4.424	3.872	2.176	0.107	0.122	0.432
1.7	4.422	3.853	**2.126**	0.108	0.116	**0.448**
1.8	4.417	3.830	2.131	0.108	0.120	0.441
1.9	4.420	3.814	2.152	0.109	0.120	0.436
2.0	4.422	3.796	2.176	0.110	0.127	0.432
2.1	4.416	3.782	2.192	0.110	0.129	0.426
2.2	4.414	3.760	2.130	0.111	0.129	0.438
2.3	4.428	3.815	2.144	0.107	0.126	0.443
2.4	4.420	3.815	2.210	0.108	0.122	0.426
2.5	4.420	**3.737**	2.148	0.110	**0.136**	0.437
2.6	4.417	3.843	2.140	0.109	0.122	0.441
2.7	4.419	3.764	2.135	0.105	0.128	0.446
2.8	4.421	3.806	2.187	0.108	0.127	0.429
2.9	4.425	3.864	2.188	0.106	0.118	0.429
3.0	4.414	3.751	2.159	0.110	0.126	0.436

Table 4.7: VGG16 - ImageNet - Initialization and One Epoch Results

Learning Rate	Initialization Method	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.1	Scheme 1	**6.884**	16.096	**0.008**	0.001
0.1	LSUV	7.400	16.099	0.001	0.001
0.1	Glorot	6.910	**6.910**	0.001	$\widehat{0}.001$
0.1	He	7.389	16.099	$\widehat{0}.001$	0.001
0.01	Scheme 1	**6.884**	**3.915**	**0.008**	**0.233**
0.01	LSUV	7.400	4.043	0.001	0.205
0.01	Glorot	6.910	6.654	0.001	$\widehat{0}.006$
0.01	He	7.389	4.000	$\widehat{0}.001$	0.218
0.001	Scheme 1	**6.884**	**4.519**	**0.008**	**0.156**
0.001	LSUV	7.400	5.243	0.001	0.088
0.001	Glorot	6.910	6.909	0.001	0.001
0.001	He	7.389	5.144	0.001	0.098

Table 4.8: VGG19 - ImageNet - Initialization and One Epoch Results

Learning Rate	Initialization Method	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.1	Scheme 1	**6.888**	16.102	**0.006**	0.001
0.1	LSUV	7.430	16.105	0.001	0.001
0.1	Glorot	6.910	**6.910**	0.001	0.001
0.1	He	7.238	16.102	0.001	0.001
0.01	Scheme 1	**6.888**	**3.828**	**0.006**	**0.232**
0.01	LSUV	7.430	4.068	0.001	0.206
0.01	Glorot	6.910	6.909	0.001	0.001
0.01	He	7.238	4.138	0.001	0.196
0.001	Scheme 1	**6.888**	**4.487**	**0.006**	**0.159**
0.001	LSUV	7.430	5.120	0.001	0.095
0.001	Glorot	6.910	6.909	0.001	0.001
0.001	He	7.238	5.179	0.001	0.092

Table 4.9: InceptionV3 - ImageNet - Initialization and One Epoch Results

Learning Rate	Initialization Method	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.1	Scheme 1	**6.875**	**2.744**	**0.016**	**0.407**
0.1	LSUV	7.430	3.060	0.001	0.355
0.1	Glorot	6.911	2.986	0.001	0.372
0.1	He	7.160	2.977	0.001	0.367
0.01	Scheme 1	**6.875**	**3.326**	**0.016**	**0.300**
0.01	LSUV	7.430	3.904	0.001	0.222
0.01	Glorot	6.911	3.677	0.001	0.251
0.01	He	7.160	4.000	0.001	0.206
0.001	Scheme 1	**6.875**	**5.302**	**0.016**	**0.072**
0.001	LSUV	7.430	5.891	0.001	0.036
0.001	Glorot	6.911	5.803	0.001	0.039
0.001	He	7.160	11.605	0.001	0.002

Table 4.10: InceptionV3 - ImageNet - Loss Results

Initialization Method	After initialization	After 1 epoch	After 10 epochs	After 20 epochs	After 30 epochs
Scheme 1	**6.892**	**3.618**	**2.753**	**2.703**	**2.717**
LSUV	7.389	6.636	5.164	5.088	5.132
Glorot	6.910	5.653	3.505	3.547	3.567
He	7.021	6.577	5.143	4.963	4.978

Table 4.11: Number of Parameters - All Models

Model	Number of Parameters
VGG16	$\approx$ 138 million
VGG19	$\approx$ 143 million
InceptionV3	$\approx$ 23 million

Table 4.12: InceptionV3 - ImageNet - Accuracy Results

Initialization Method	After initialization	After 1 epoch	After 10 epochs	After 20 epochs	After 30 epochs
Scheme 1	**0.010**	**0.262**	**0.463**	**0.492**	**0.494**
LSUV	0.001	0.013	0.137	0.155	0.154
Glorot	0.001	0.056	0.333	0.354	0.355
He	0.001	0.014	0.154	0.184	0.186

Chapter 5

Scheme 2: Initialization of Convolutional Layers - Gabor Initialization

5.1 Motivation

Convolutional neural networks (CNNs) are a subset of network architectures in deep learning. These networks were designed to process data that has a grid-like topology, like images or time-series data. Because of their great success in tackling challenging problems, CNNs are very popular. These networks work by applying the *convolution* operation instead of the regular matrix multiplication operation used in fully connected networks. To create a weight initialization technique that works well for convolutional neural networks, one must fully understand how the *convolution* operation works in the deep learning context. Convolution is usually denoted with an asterisk. The convolution operation takes two arguments: the input and the kernel. The output is often called the feature map. In machine learning, convolution is usually applied to tensors (n-dimensional arrays), and the kernel is composed of parameters that are to be learned during the training process. Because of this, the convolution operation is applied over more than one axis. The convolution operation applied to a 2-D tensor I using a 2-D kernel K is defined as follows:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

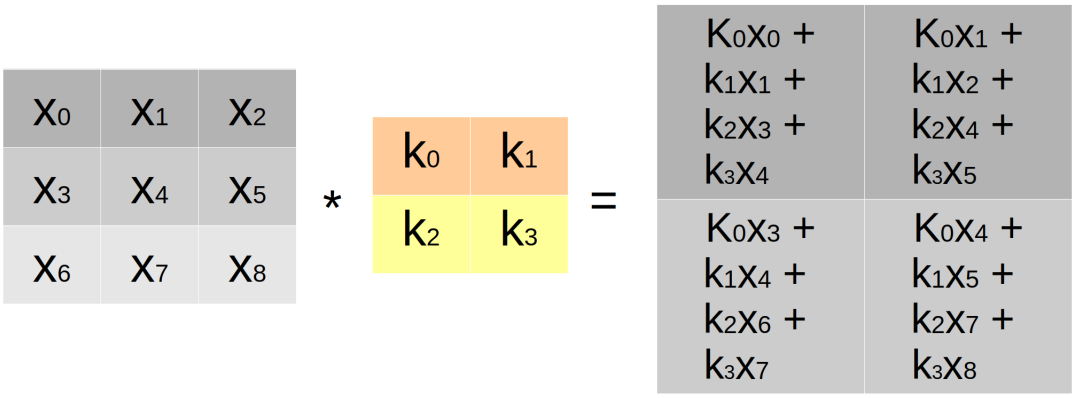

Figure 5.1: Convolution Example

Figure 5.1 shows how the convolution operation is applied to a 3x3 matrix using a 2x2 kernel.

Three properties are attributed to the great success of convolutional neural networks: 1) sparse connectivity, 2) parameter sharing, and 3) equivariant representations. Fully-connected layers have a weight value associated with each input value. In contrast, the kernels used in convolutional layers are a lot smaller than the input tensors they receive. This enables connections between input and kernel values to be sparse, which reduces the memory requirements of the resulting network and the number of operations needed to compute the output. This is clearly observed in modern architectures, where it is common to convolve 3x3 or 5x5 kernels with 224x224 input tensors.

In machine learning, parameter sharing occurs when the same parameter is used in multiple operations. In the case of convolution, kernel values are used at multiple positions of the input tensor. This allows learning algorithms to find weight values that work for multiple locations of the input instead of just one (as with fully-connected layers). This makes kernel values robust and useful for many types of inputs.

We say that convolution exhibits equivariant properties because if the input is changed by a very particular family of functions, the output changes in that same way. For example, if an image is shifted or its brightness is changed, the output produced by convolution is

affected in the same way by these operations. That is, one could apply the operations in any order, and the result would be the same (convolution and then brightness change, or brightness change and then convolution). There are operations, like changes in scale and rotation, where convolution does not exhibit equivariant properties. The reason why this is important is because these equivariant properties help convolutional layers learn filters that are useful when applied to neighboring pixels at multiple locations in the input. For example, the first layer of a convolutional network that deals with images usually learns filters that allow the network to detect edges. Identifying edges is important regardless of where in the image this process is done. It is also important to detect edges regardless of changes in brightness or pixel shifts.

In (84), Zeiler *et al.* propose a way to visualize trained convolutional neural networks. Their approach consists of using a deconvolutional network (deconvnet) (85). A deconvnet can be seen as a regular convolutional network that applies the same operations, but in reverse order. That is, instead of creating feature maps from pixel values, it does the opposite, it maps feature maps to pixel values. To better analyze what a layer in a regular convolutional network learned, they propose to attach a deconvnet to the layer to be examined, which allows to create a continuous path back form the layer to image pixels. Once the deconvnet is attached, a sample image is fed to the network and a feed forward pass is performed. To study one of the layer's activations, the other activations in the layer are set to zero. The resulting feature maps are passed to the attached deconvnet layer. The deconvnet then processes the information (performing regular convnet operations in reverse order) until pixel space is reached. Figures 5.2 and 5.3 show what convolutional neural networks learn when trained on ImageNet using the described visualization approach.

Analyzing what convolutional networks learn, one can see that basic filters that have been used by the computer vision community for decades are learned through gradient descent. These include edge and corner detection filters, as well as filters that closely resemble Gabor ones. It is interesting to see that deep learning models learn what researchers had been hand-crafting before the development of deep learning.

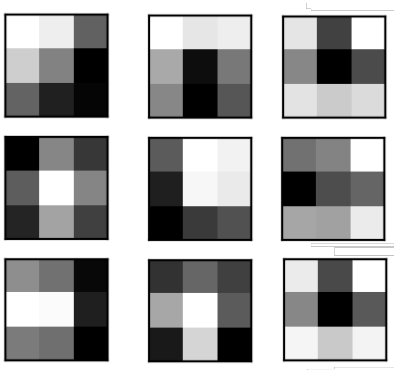

Figure 5.2: Nine VGG16 filters (first convolutional layer) after training on ImageNet

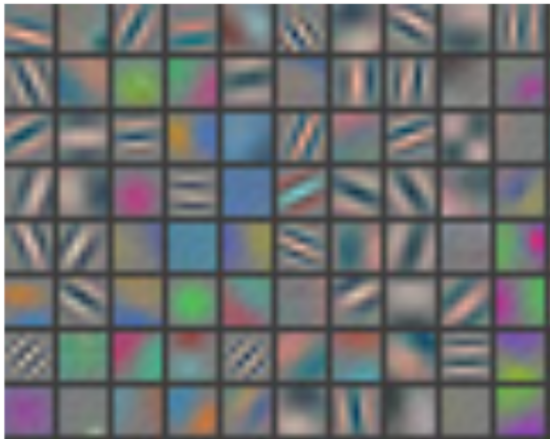

Figure 5.3: Visualization of the first layer features produced by Zeiler's convnet (84).

The filters learned by the first convolutional layers are well-suited for most computer vision tasks. Corner and edge detectors, for example, are of great value regardless of the vision task the network is to solve. This is the reason why many deep learning practitioners re-use weight values of convolutional networks trained on the ImageNet data set. These initial filters position the network in a great place in parameter space before training it on a new data set.

The idea behind our second initialization approach is to exploit the previously discussed observation. We know that convolutional layers end up learning filters that are very similar to the ones that computer vision practitioners used to hand-craft in the past. Our idea is to use algorithms that were developed before the popularization of feature-learning approaches to initialize weight values of convolutional networks. In a way, our intent is to provide an initial *guide* to the network by incorporating knowledge previously discovered by computer vision researchers. Of particular interest are Gabor filters since they are learned by convolutional layers often and have had great success in feature extraction, texture analysis, and segmentation tasks (23) (52).

The idea of explicitly using Gabor filters in CNNs is not new. In (44), Luan introduced Gabor convolutional neural networks (GCNs) in 2018 (44). These networks replace regular convolutional layers with layers that learn how to use hand-crafted Gabor filters. That is, instead of having the network learn filter values, it learns what filters to select from a predefined set of hand-crafted Gabor filters. This allows these networks to have much fewer parameters while producing results similar to the ones achieved with regular convolutional layers. Our initialization scheme takes a similar approach. However, instead of having the network learn what Gabor filters to use, we propose to keep regular convolutional layers but have them initialized with weight values chosen so that the result is that of Gabor filters. The objective is to take advantage of the scale and rotation invariance properties of Gabor filters starting at initialization time while still allowing gradient descent to tweak the filters appropriately for the task at hand. We limit this initialization scheme to the first convolutional layer in the network.

5.2 Algorithm

In the computer vision domain, two-dimensional Gabor filters are defined as follows:

$$G_c[i,j] = Be^{-\frac{(i^2+j^2)}{2\sigma^2}} \cos(2\pi f(i\cos\theta + j\sin\theta))$$
$$G_s[i,j] = Ce^{-\frac{(i^2+j^2)}{2\sigma^2}} \sin(2\pi f(i\cos\theta + j\sin\theta))$$

In this definition, B and C are normalizing factors determined by the practitioner; θ represents the direction of the filter; σ determines the size of the image region targeted by the filter; f is the function that defines the frequency the filter tries to capture. For our initialization, we decided to use a special subset of Gabor filters; the ones that satisfy the neurophysiological constraints for simple cells (33). Thus, the definition reduces to the following:

$$G(x;\omega,\theta,K) = \left[\frac{\omega^2}{4\pi K^2}\exp\{-(\omega^2/8K^2)[4(x\cdot(cos\theta,sin\theta))^2 + (x\cdot(-sin\theta,cos\theta))^2]\}\right]$$
$$\times\left[\exp\{iwx\cdot(cos\theta,sin\theta)\}exp(K^2/2)\right]$$

Our Gabor-based initialization scheme builds on the initialization presented in the last chapter. We keep the active fraction and standarization steps. However, instead of initializing filters of convolutional layers to be orthonormal vectors, we set them to be Gabor filters. We use the previous definition of $G(x;\omega,\theta,K)$ to do so, where K is set to π. Because we want all filters of a given convolutional layer to be different from each other, we use a different value for ω and θ for each of the layer's filters. For ω, we interpolate from 0.1 to 1, creating n different values, where n is the square root of the number of filters to be initialized. For θ, we interpolate from 0 to π, creating n different values, where n is the square root of the number of filters to be initialized. We then use all combinations of the values generated for θ and ω to create all filters using the previously described Gabor definition. We also kept the initialization scheme for the output layer as described in the previous chapter. The whole process is described in Algorithm 2. It is important to note that we

limit this initialization process to the first convolutional layer in the network, while the rest are initialized using the initialization approach described in Chapter 4. The reasoning behind this decision is that the first convolutional layer in the network is the one that is known to learn traditional filters previously proposed by computer vision researchers.

5.2.1 Overhead

This scheme is an extension of the one described in Chapter 4. The overhead analysis is identical for all layers in the network except for the first convolutional layer, which is initialized using the described approach. The computation time required to generate Gabor filters for the first convolutional layer does not add any extra computation time. Traditional initialization schemes sample random values from a given probability distribution. Computationally, our approach does the same by sampling values from the probability distribution that defines Gabor filters. Thus, the running time of this initialization scheme is $O(n)$ with constant factors depending on the network's architecture. Empirically, we observed that the running time of the initialization process is about 3 to 6 times of that of processing a training mini-batch of the same size as the initialization set.

5.3 Experimental Results

Two sets of experiments were designed. The purpose of the first set was to analyze how the performance of our method is affected by its hyperparameters. In the second set, we compared our method to popular weight initialization techniques, including Glorot, He, LSUV, and the initialization scheme presented in the last chapter. The following subsections describe in detail the evaluation process of each set of experiments.

Algorithm 2 Weight Initialization - Gabor

procedure INITIALIZE(X, Y, h, f, σ_g)

  $H^{(0)} \leftarrow X$

  $n \leftarrow len\ (h)$

  for $i = 1$ to n (where n is the number of convolutional layers) **do**

    $K \leftarrow \pi$

    $n_{filters} \leftarrow size(W^{(i)} -> filters\_dimension)$

    $\vec{\omega} \leftarrow range(0.1, 1, \sqrt{n_{filters}})$

    $\vec{\theta} \leftarrow range(0, \pi, \sqrt{n_{filters}})$

    $W^{(i)} \leftarrow G(W^{(i)}; \omega, \theta, K)$ using all values of ω and θ from $\vec{\omega}$ and $\vec{\theta}$

    $P \leftarrow H^{(i-1)} * W^{(i)}$

    $P \leftarrow sort(P)$ ▷ sort each column of P in ascending order

    $b^{(i)} \leftarrow -P[\lfloor (1-f)h_i \rfloor]$

    $s \leftarrow \sigma(P)$ ▷ vector of column-wise standard deviations

    ▷ Scale weights and bias to obtain σ_g standard deviation

    for $j = 1$ to h_i **do**

      $W^{(i)}_{[:,j]} \leftarrow W^{(i)}_{[:,j]} \sigma_g / s_j$

      $b^{(i)} \leftarrow b^{(i)}_j \sigma_g / s_j$

    $H^{(i)} \leftarrow relu(H^{(i-1)} W^{(i)} + b^{(i)})$

  $W^{(n)} \leftarrow (H^{(n)})^* Y$

  $b^{(n)} \leftarrow 0$

  return $W^{(1)}, \ldots, W^{(n)}, b^{(1)}, \ldots, b^{(n)}$

5.3.1 Hyperparameter Evaluation

To study how the hyperparameters used by our Gabor-based initialization scheme affect its performance, we trained a large network on a large data-set multiple times using different hyperparameter values. This is very different from what we did for the hyperparameter evaluation of the initialization scheme presented in Chapter 4. This is because Gabor filters are great at extracting useful properties from large images, like the ones produced by modern smartphones and consumer-grade cameras. This means that data-sets like CIFAR10 and CIFAR100 (which are composed of very low resolution images) are not useful for studying the performance of our Gabor-based initialization scheme. Because of this, we decided to use ImageNet with images resized to 128x128 and VGG16 with convolutional layers where all filters have a size of 7x7. The hyperparameters that we included in our analysis were the active fraction, the goal standard deviation, and the training learning rate. The following subsections describe how each of these hyperparameters was analyzed.

5.3.2 Active Fraction Results

The model was initialized using our Gabor-based initialization approach using the following active fraction values: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9. For all experiments, we used an initialization set of size 256 for all hidden layers, an initialization set of size 16,384 for the output layer, a learning rate of 0.01, and a goal standard deviation of 1.0. After initializing the model using a given set of hyperparameter values, the network was trained for 1 epoch using regular stochastic gradient descent, categorical crossentropy as the loss function, and a mini-batch size of 64. Table 5.1 presents the results of these experiments. Similar to what was found while analyzing the previous initialization scheme, an active fraction in the [0.6, 0.8] works well. We speculate that active fraction values greater than 0.5 are ideal since more non-zero values are used when computing the gradient, which allows the network to learn faster. However, values that are very close to 1.0 are to be avoided since the benefits of non-linearity in a network are removed as layers are forced to produce

almost linear activations.

Table 5.1: VGG16 - ImageNet - Active Fraction Results

Active Fraction	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.1	6.929	16.099	0.001	0.001
0.2	6.912	5.622	0.001	0.043
0.3	6.907	4.873	0.002	0.103
0.4	6.904	4.615	0.002	0.129
0.5	6.901	4.636	0.002	0.130
0.6	6.898	4.595	0.004	0.136
0.7	6.896	**4.552**	0.004	**0.142**
0.8	**6.895**	4.675	**0.005**	0.129
0.9	**6.895**	16.102	0.004	0.001
None	6.914	6.908	0.001	0.001

5.3.3 Standard Deviation Results

The model was initialized using our Gabor-based initialization approach using the following goal standard deviation values: 0.5, 1.0, 1.5, and 2.0. For all experiments, we used an initialization set of size 256 for all hidden layers, an initialization set of size 16,384 for the output layer, a learning rate of 0.01, and an active fraction of 0.7. After initializing the model using a given set of hyperparameter values, the network was trained for 1 epoch using regular stochastic gradient descent, categorical crossentropy as the loss function, and a mini-batch size of 64. Table 5.2 presents the results of these experiments.

Table 5.2: VGG16 - ImageNet - Standard Deviation Results

Standard Deviation	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.5	**6.896**	4.554	**0.004**	0.138
1.0	**6.896**	**4.552**	**0.004**	**0.142**
1.5	**6.896**	4.688	**0.004**	0.127
2.0	**6.896**	16.102	**0.004**	0.001
None	**6.896**	5.021	**0.004**	0.089

5.3.4 Learning Rate Results

To determine if the behavior of the initialization scheme is consistent across multiple learning rates, we initialized the model using our Gabor-based initialization approach using the following learning rates: 0.0001, 0.001, 0.01, and 0.1. For all experiments, we used an initialization set of size 256 for all hidden layers, an initialization set of size 16,384 for the output layer, a learning rate of 0.01, and a goal standard deviation of 1.0. After initializing the model using a given set of hyperparameter values, the network was trained for 1 epoch using regular stochastic gradient descent, categorical crossentropy as the loss function, and a mini-batch size of 64. Table 6.4 presents the results of these experiments.

5.3.5 Other Initialization Techniques

Our second initialization scheme was compared to the following state-of-the-art initialization techniques: Glorot, He, and LSUV. Since this scheme is a variation of the first one, we also compare our Gabor-based initialization to the first scheme presented in Chapter 4. To perform this comparison, we trained InceptionV3 on ImageNet for 30 epochs using

Table 5.3: VGG16 - ImageNet - Learning Rate Results

Learning Rate	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.0001	**6.896**	5.967	**0.004**	0.039
0.001	**6.896**	5.055	**0.004**	0.102
0.01	**6.896**	**4.552**	**0.004**	**0.142**
0.1	**6.896**	16.102	**0.004**	0.001

all initialization techniques. We used a mini batch size of 64, regular stochastic gradient descent, and categorical crossentropy as the loss function. We used ImageNet for all experiments with images resized to 128x128. For our two initialization approaches, we used an initialization set of size 256 for hidden layers, an initialization set of size 16,384 for the output layers, a value of 0.8 for the *active fraction* hyperparameter, and a goal standard deviation of 1.0. For LSUV, we also used an initialization set of size 256 for hidden layers, and an initialization set of size 16,384 for the output layers. We used a learning rate of 0.1 for the first 10 epochs; a learning rate of 0.01 for the next 10 epochs; and a learning rate of 0.001 for the last 10 epochs. Tables 5.4 and 5.5 present the results of these experiments. Figures 5.4 and 5.5 show how the loss and accuracy change with respect to the number of training epochs.

The results show a small improvement over our previous initialization scheme. Our Gabor-based approach managed to provide a modest increase in accuracy (≈2.4% over Scheme 1). Similar to Scheme 1, our second scheme outperforms all other techniques by a large margin. The small boost in performance that it introduces is welcome; specially considering that there is no added overhead. The implementation of Scheme 2 (in Keras) is freely and publicly available on GitHub (1).

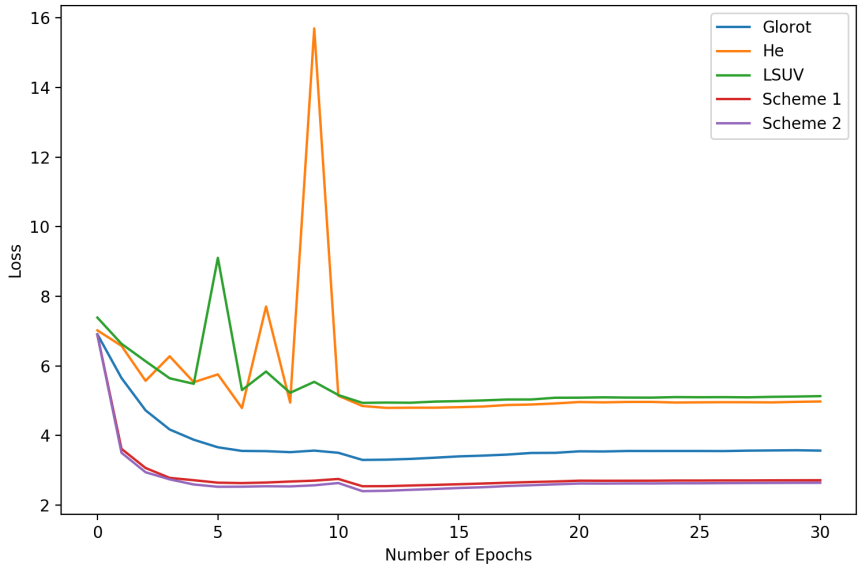

Figure 5.4: InceptionV3 - ImageNet - Loss Results

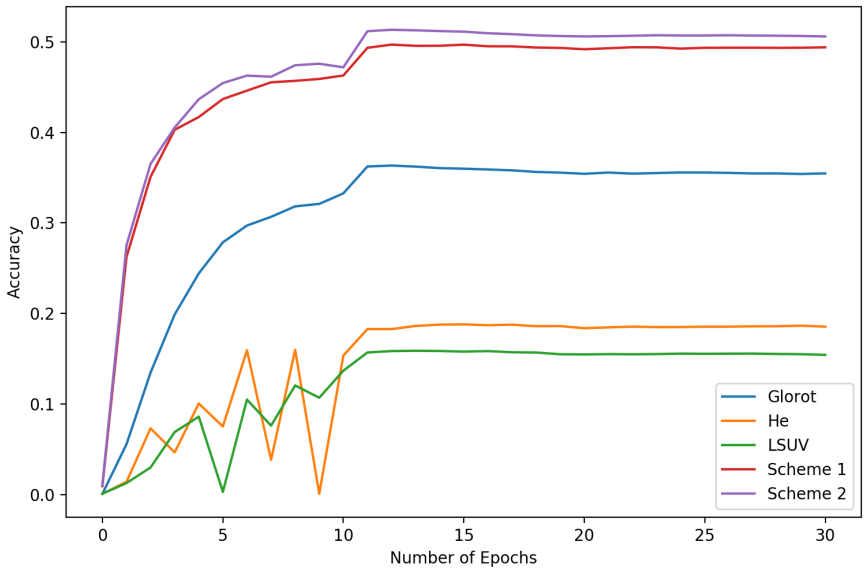

Figure 5.5: InceptionV3 - ImageNet - Accuracy Results

Table 5.4: InceptionV3 - ImageNet - Loss Results

Initialization Method	After initialization	After 1 epoch	After 10 epochs	After 20 epochs	After 30 epochs
Scheme 1	6.892	3.618	2.753	2.703	2.717
Scheme 2	**6.890**	**3.505**	**2.636**	**2.622**	**2.643**
LSUV	7.389	6.636	5.164	5.088	5.132
Glorot	6.910	5.653	3.505	3.547	3.567
He	7.021	6.577	5.143	4.963	4.978

Table 5.5: InceptionV3 - ImageNet - Accuracy Results

Initialization Method	After initialization	After 1 epoch	After 10 epochs	After 20 epochs	After 30 epochs
Scheme 1	0.010	0.262	0.463	0.492	0.494
Scheme 2	**0.011**	**0.275**	**0.472**	**0.506**	**0.506**
LSUV	0.001	0.013	0.137	0.155	0.154
Glorot	0.001	0.056	0.333	0.354	0.355
He	0.001	0.014	0.154	0.184	0.186

5.4 Conclusion

In this chapter, we introduced an extension of our first weight initialization technique. The innovation in this scheme is given by the use of Gabor filters. Since these filters are known to be learned by convolutional neural networks, we explored the idea of selecting weight values for the first convolutional layer, so the output of such layer is that of Gabor filtering. Results show that this idea does work, adding a modest performance boost over our previous scheme.

Future work includes creating a new type of convolutional layer that is forced to learn

Gabor filters only. The idea is that the weight values of this layer define the filters to be used by the layer. That is, instead of having the network learn filter values, we have it learn the best set of Gabor filters to use for the task at hand.

Chapter 6

Scheme 3: Initialization of Convolutional Layers - Keypoint Initialization

6.1 Motivation

Before deep learning became the preferred strategy to tackle computer vision problems, researchers introduced a variety of feature extraction algorithms that were used in combination with traditional machine learning techniques. Although not as accurate as modern deep learning models, these techniques allowed us to tackle problems like object detection, face recognition, and object tracking with great success. Since now the first layers in modern convolutional neural networks are tasked with extracting useful features from raw data, the popularity of these human-crafted feature extractors has decreased significantly.

Many of the most powerful, human-crafted keypoint detection and feature extraction algorithms were proposed in the 2000s decade. Some of the techniques that showed great performance include SIFT (Scale-Invariant Feature Transform) (43), SURF (Speeded-Up Robust Features) (7), FAST (Features from Accelerated Segment Test) (57), BRIEF (Binary Robust Independent Elementary Features) (8), and ORB (Oriented FAST and Rotated BRIEF) (58).

All of these techniques work by first finding keypoints in the images of interest. These keypoints are then used to create descriptors that provide a more compact representation of these images. SIFT was the first algorithm of this set of techniques and inspired the

creation of other approaches that follow the same scheme. The difference between these approaches lies in the complexity of the algorithms used to find keypoints and the properties that need to be met by a region to be considered of interest.

SIFT (43), introduced by Lowe is a feature extraction algorithm specifically designed for image data. This patented algorithm was shown to be very effective when tackling problems like object detection/recognition, robot mapping/navigation, modeling, gesture recognition, and object tracking. SIFT is a 4-step algorithm.

In its first step, SIFT is concerned with finding keypoints in images at different scales. The intuition is that images can be large, and it is important to find interesting regions at different spatial scales. To accomplish this, SIFT uses what is called *Difference of Gaussians* (DoG), which is an approximation of *Laplacian of Gaussian*. The idea is to blur the image of interest multiple times with different intensities at different scales (octaves). Differences between these resulting images are computed to identify regions (keypoints) that contain valuable information (local maxima). Figure 6.1 shows how this process is performed.

Once the DoG is computed, resulting images are analyzed to find local extrema over scale and space. Typically, one pixel in an image is compared with its 15 neighboring pixels as well as the 16 pixels in the next scale and the 16 pixels in the previous scale. If it is a local extreme, it is a *potential* keypoint. It is important to note that, although the number of neighboring pixels to be analyzed is to be defined by the practitioner, 16 is the value commonly used in this technique.

In the second step, keypoints are refined. This is done by using a Taylor series expansion of scale / space to accurately locate extrema. If the intensity of the extreme is below a certain threshold (commonly 0.03), the keypoint is discarded. When DoG is used, edges in images produce high responses. Thus, SIFT eliminates many of these keypoints by using a 2x2 Hessian matrix (a square matrix of second-order partial derivatives) to compute the principal curvature. The eigen-values of this matrix are compared and if the ratio between them exceeds a threshold (typically 10), the keypoints are discarded.

The third step in SIFT consists of assigning an orientation to every keypoint to achieve

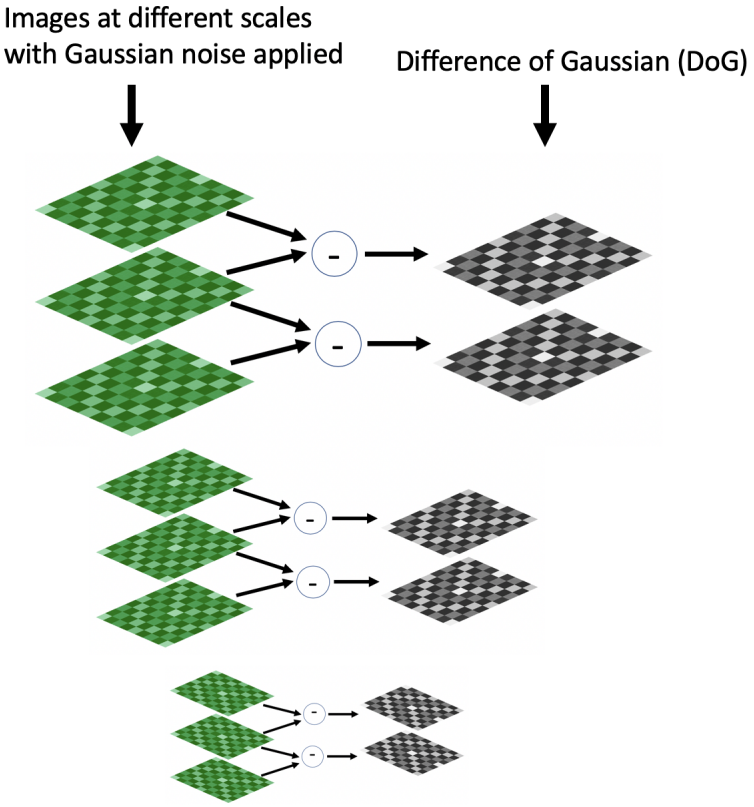

Figure 6.1: SIFT - Difference of Gaussian (DoG)

rotation independence advantages. The surrounding region of every keypoint is extracted and the gradient magnitude and orientation of the region are computed. With these values, an orientation histogram is constructed. The histogram has 36 bins (where each bin represents a 10 degree interval) and the magnitude is used as the weight when constructing it.

In the last step, SIFT constructs what is called *keypoint descriptors*. To construct them, the 16x16 neighborhood of the keypoint is divided into 16 4x4 sub-regions. For each sub-region, an 8-bin orientation histogram is constructed. All histograms are put together to form a vector of size 128. The resulting keypoint descriptors are the final, more compact representation of the image of interest.

The descriptors of the image can then be used for tasks like object recognition, tracking, and modeling. Two properties of SIFT motivated the creation of very similar algorithms: 1) it is patented; 2) it is slow. SURF, although also patented, introduced performance enhancements to SIFT's original steps. For example, it approximates LoG with a Box Filter, which can be efficiently calculated with the help of integral images (76). Plus, the process is highly parallelizable. Similar performance-oriented modifications (that make use of integral images) are applied in later steps to achieve similar results while minimizing the number of computations required to do so.

FAST, in contrast to SIFT and SURF, was designed to be used in real-time systems while being free. One of the key differences in FAST is how potential keypoints are identified. In order for a pixel p with intensity i to be a keypoint, at least n of its 16 surrounding pixels (circular) have to have an intensity bigger than $i + t$ or smaller than i - t, where t is a user-defined threshold. These n pixels have to be contiguous; and n is commonly set to 12. Figure 6.2 shows how FAST would try to determine if a given pixel is a keypoint.

As an open source and patent-free alternative, OpenCV Labs introduced ORB (Oriented FAST and Rotated BRIEF) in 2011 (58). For keypoint extraction, ORB uses the same process employed by FAST with some steps added to the process. After executing the same series of steps as FAST to find keypoints, ORB runs a Harris Corner Detector (25)

Figure 6.2: FAST - Keypoint Localization

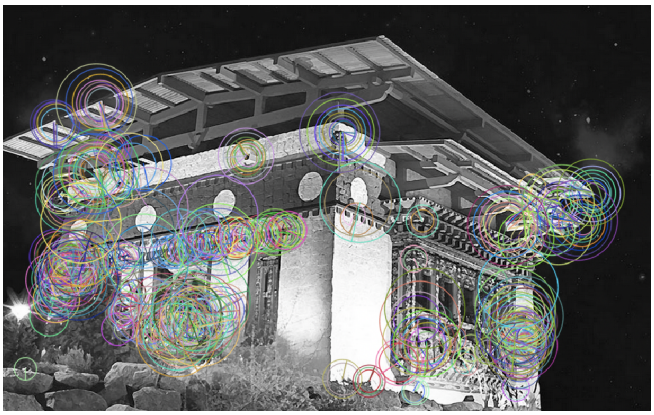

Figure 6.3: ORB - Keypoints Example

to find the best n keypoints. Similar to SIFT, ORB then uses a pyramid of octaves to find these keypoints at different scales. Figure 6.3 shows an example of the keypoints that ORB finds in a sample image.

Our next initialization method is inspired by these keypoint finding techniques.

6.2 Algorithm

The above-mentioned algorithms produce compact representations of images that allowed computer vision researchers to tackle non-trivial problems in the 2000s and early 2010s. Our

initialization scheme is inspired by the keypoint extraction mechanisms of these techniques. It is important to emphasize that the concepts of convolution and keypoint extraction are very different. Convolution is a well-defined and simple mathematical operation; whereas keypoint extraction is performed by more elaborate algorithms that are designed to identify points in an image that meet certain properties.

Our initialization approach attempts to bring some of the benefits of keypoint extraction to the convolution space. The intuition is the following. We aim to select weight values for filters in convolutional layers that produce a positive activation when a keypoint is processed during the convolution operation, and a non-positive value when other points are processed. That is, if a pixel in an image is a keypoint, the convolution operation (at that pixel) should produce a positive activation. If other pixels are at the center of the convolution operation, the output should be non-positive. Similar to the initialization approach presented in Chapter 5, this scheme is to be applied to the first convolutional layer only since it extracts features from raw image information.

To accomplish this, at initialization time, we perform a feed-forward pass on the network using a subset of the training data set (the initialization set). We perform this pass until the first convolutional layer is found. The input tensor of this layer (which is usually raw images) is used as the input of one of the keypoint extraction algorithms of the previously described methods, which identifies n keypoints in the input data. For each of these keypoints, we extract the k-by-k regions that surround them, where k-by-k is the size of the filters contained in the convolutional layer. With these n regions of size k-by-k, we create a matrix with n rows and k^2 columns, where each row in the matrix represents each of the n regions associated with each of the keypoints. For example, let's assume that the first layer in the network to be initialized is a convolutional layer composed of 3-by-3 filters. To initialize this layer, we run one of the keypoint extraction algorithms on a subset of the training images to identify n keypoints. After the keypoints are identified, the 3-by-3 image patches surrounding these keypoints are placed in a n-by-9 matrix. That is, each row in this matrix is composed of the pixel values surrounding a keypoint. We refer to this matrix

as the keypoint matrix.

After the keypoint matrix is constructed, the k-means clustering algorithm is run on it with k equal to the number of filters in the layer. The idea is to find as many clusters in the keypoint space as there are filters in the layer. The ultimate goal is to have each filter produce a positive activation when presented with a member of one of the clusters, and a negative activation when presented with something else. Our goal is to select initial weight values that allow filters to be different from each other (heterogeneous) while also producing activations that allow the network to identify keypoint regions in the input. To do this, after the clusters are found, a linear, multi-class support vector machine classifier is trained with the keypoint matrix as input and the cluster information as the desired classes. The learned coefficients are then used as the initial weight values associated with the layer's filters. Once this is done, the ReLU adaptation and standarization steps from the initialization scheme presented in Chapter 4 are performed on this layer. The rest of the layers are initialized using the scheme proposed in Chapter 4.

6.2.1 Overhead

The running time of this initialization scheme varies depending on the algorithm selected to find the n keypoints. For an r-by-c image, the fastest keypoint extractors, like FAST, run in $O(rc)$. Because the keypoint detection algorithm is applied to the p images that constitute the initialization set, the time it takes to find all keypoints is $O(prc)$. Once the n keypoints are found, the k-means algorithm is run. The time complexity of k-means is $O(knid)$, where k is the number of filters in the layer (clusters), n is the number of keypoints, i is the number of iterations, and d is the the number of weight values needed to represent a filter in the convolutional layer. Once the clusters are created, a linear support vector machine is trained, which is done in $O(n^2)$, where n is the number of training examples (and, in our case, equal to the number of keypoints). Thus, the final running time of our approach for the first convolutional layer is given by $O(prc + knid + n^2)$. The rest of the layers in the network are initialized using the approach described in Chapter 4. Our keypoint

initialization scheme for the first convolutional layer is highly parallelizable, which allows us to run it on modern computers in a few seconds. Empirically, we observed that the initialization process (including all layers in the network) takes about 5-8 times of that of processing a training mini-batch of the same size as the initialization set.

6.3 Experimental Results

Two sets of experiments were designed. The purpose of the first set was to analyze how the performance of our method is affected by its hyperparameters. In the second set, we compared our method to popular weight initialization techniques, including Glorot, He, LSUV, and the initialization schemes presented in Chapters 4 and 5. The following subsections describe in detail the evaluation process of each set of experiments.

6.3.1 Hyperparameter Evaluation

To study how the hyperparameters used by our keypoint initialization scheme affect its performance, we followed the same approach as the one described for our Gabor-based initialization in Chapter 5. That is, we trained a large network on a large data-set multiple times using different hyperparameter values. We used ImageNet with images resized to 128x128 and VGG16 with convolutional layers where all filters have a size of 7x7. The hyperparameters that we included in our analysis were the keypoint extraction algorithm used (FAST, ORB, and SIFT), the active fraction, the goal standard deviation, and the training learning rate. The following subsections describe how each of these hyperparameters was analyzed.

6.3.2 Keypoint Extraction Algorithm Results

We were interested in studying how our method was affected by the algorithm used to find keypoints in raw images. Even though all algorithms were designed with the same task in

mind, they approach the problem very differently. Not only is this reflected in their running times, but also in the types of *interesting* regions they identify. To study the influence of these differences in our initialization approach, we trained VGG16 (with filters of size 7x7) multiple times using the following keypoint finding algorithms: FAST, ORB, and SIFT. For all experiments, we used an active fraction of 0.7, a goal standard deviation of 1.0, an initialization set size of 256 for all hidden layers, an initialization set of size 16,384 for the output layer, and a learning rate of 0.01. After initializing the model with a given set of hyperparameters, we trained it for 1 epoch using regular stochastic gradient descent with a mini-batch size of 64 and categorical crossentropy as the loss function. Table 6.1 shows the results of this set of experiments.

Table 6.1: VGG16 - ImageNet - Active Fraction Results

Algorithm	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
FAST	**6.887**	4.379	**0.010**	**0.162**
ORB	**6.887**	4.495	0.009	0.151
SIFT	**6.887**	**4.362**	**0.010**	0.160

The results suggest that the keypoint finding algorithm has little effect on the results. They all perform about the same. However, FAST has a significantly better running time, which makes it ideal for this scheme.

6.3.3 Active Fraction Results

We initialized the model using our keypoint initialization approach using the following active fraction values: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9. For all experiments,

we used an initialization set of size 256 for all hidden layers, an initialization set of size 16,384 for the output layer, a learning rate of 0.01, and a goal standard deviation of 1.0. The keypoint extraction algorithm we used for all experiments was FAST. After initializing the model using a given set of hyperparameter values, the network was trained for 1 epoch using regular stochastic gradient descent, categorical crossentropy as the loss function, and a mini-batch size of 64. Table 6.2 presents the results of these experiments. Similar to what was found for the previous initialization scheme, a value between 0.6 and 0.8 works well. This confirms that the active fraction hyperparameter is a valuable addition across all initialization schemes that use it.

Table 6.2: VGG16 - ImageNet - Active Fraction Results

Active Fraction	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.1	6.936	16.102	0.001	0.001
0.2	6.911	16.102	0.002	0.001
0.3	6.905	4.655	0.002	0.128
0.4	6.899	4.579	0.003	0.139
0.5	6.895	4.425	0.005	0.157
0.6	6.892	**4.334**	0.006	**0.166**
0.7	6.887	4.379	0.010	0.162
0.8	**6.884**	4.664	**0.011**	0.133
0.9	**6.884**	16.102	0.010	0.001
None	6.91	6.908	0.002	0.001

6.3.4 Standard Deviation Results

We initialized the model using our keypoint initialization approach using the following goal standard deviation values: 0.5, 1.0, 1.5, and 2.0. For all experiments, we used an initialization set of size 256 for all hidden layers, an initialization set of size 16,384 for the output layer, a learning rate of 0.01, and an active fraction of 0.7. The keypoint finding algorithm employed for these experiments was FAST. After initializing the model using a given set of hyperparameter values, the network was trained for 1 epoch using regular stochastic gradient descent, categorical crossentropy as the loss function, and a mini-batch size of 64. Table 6.3 presents the results of these experiments. The results confirm what was stated before: a standard deviation of 1.0 works well.

Table 6.3: VGG16 - ImageNet - Standard Deviation Results

Standard Deviation	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.5	6.888	4.463	0.009	0.152
1.0	**6.887**	**4.379**	**0.010**	**0.162**
1.5	**6.887**	4.602	**0.010**	0.140
2.0	**6.887**	16.102	0.009	0.001
None	6.891	4.810	0.008	0.112

6.3.5 Learning Rate Results

To determine if the behavior of the initialization scheme is consistent across multiple learning rates, we initialized the model using our keypoint initialization approach using the following learning rates: 0.0001, 0.001, 0.01, and 0.1. For all experiments, we used an

initialization set of size 256 for all hidden layers, an initialization set of size 16,384 for the output layer, a learning rate of 0.01, and a goal standard deviation of 1.0. The keypoint finding algorithm employed for these experiments was FAST. After initializing the model using a given set of hyperparameter values, the network was trained for 1 epoch using regular stochastic gradient descent, categorical crossentropy as the loss function, and a mini-batch size of 64. Table 6.4 presents the results of these experiments.

Table 6.4: VGG16 - ImageNet - Learning Rate Results

Learning Rate	Loss after initialization	Loss after 1 epoch	Test Accuracy after initialization	Test Accuracy after 1 epoch
0.0001	**6.887**	5.558	**0.01**	0.063
0.001	**6.887**	4.799	**0.01**	0.125
0.01	**6.887**	**4.379**	**0.01**	**0.162**
0.1	**6.887**	16.102	**0.01**	0.001

6.3.6 Other Initialization Techniques

We compared our third initialization scheme to the following state-of-the-art initialization techniques: Glorot, He, and LSUV. Since this scheme is a variation of the first one, we also compared it to the two presented in Chapters 4 and 5. To perform this comparison, we trained InceptionV3 on ImageNet for 30 epochs using all initialization techniques. We used a mini batch size of 64, regular stochastic gradient descent, and categorical crossentropy as the loss function. For our three initialization approaches, we used an initialization set of size 256 for hidden layers, an initialization set of size 16,384 for the output layers, a value of 0.8 for the *active fraction* hyperparameter, and a goal standard deviation of 1.0. For

our keypoint initialization approach, we used FAST to find keypoints in input images. For LSUV, we also used an initialization set of size 256 for hidden layers, and an initialization set of size 16,384 for the output layers. We used a learning rate of 0.1 for the first 10 epochs; a learning rate of 0.01 for the next 10 epochs; and a learning rate of 0.001 for the last 10 epochs. Tables 6.5 and 6.6 present the results of these experiments. Figures 6.4 and 6.5 show how the loss and accuracy change with respect to the number of training epochs.

Table 6.5: InceptionV3 - ImageNet - Loss Results

Initialization Method	After initialization	After 1 epoch	After 10 epochs	After 20 epochs	After 30 epochs
Scheme 1	6.892	3.618	2.753	2.703	2.717
Scheme 2	**6.890**	**3.505**	**2.636**	**2.622**	**2.643**
Scheme 3	6.892	3.592	2.767	2.695	2.709
LSUV	7.389	6.636	5.164	5.088	5.132
Glorot	6.910	5.653	3.505	3.547	3.567
He	7.021	6.577	5.143	4.963	4.978

Table 6.6: InceptionV3 - ImageNet - Accuracy Results

Initialization Method	After initialization	After 1 epoch	After 10 epochs	After 20 epochs	After 30 epochs
Scheme 1	0.010	0.262	0.463	0.492	0.494
Scheme 2	**0.011**	**0.275**	**0.472**	**0.506**	**0.506**
Scheme 3	0.009	0.264	0.458	0.493	0.494
LSUV	0.001	0.013	0.137	0.155	0.154
Glorot	0.001	0.056	0.333	0.354	0.355
He	0.001	0.014	0.154	0.184	0.186

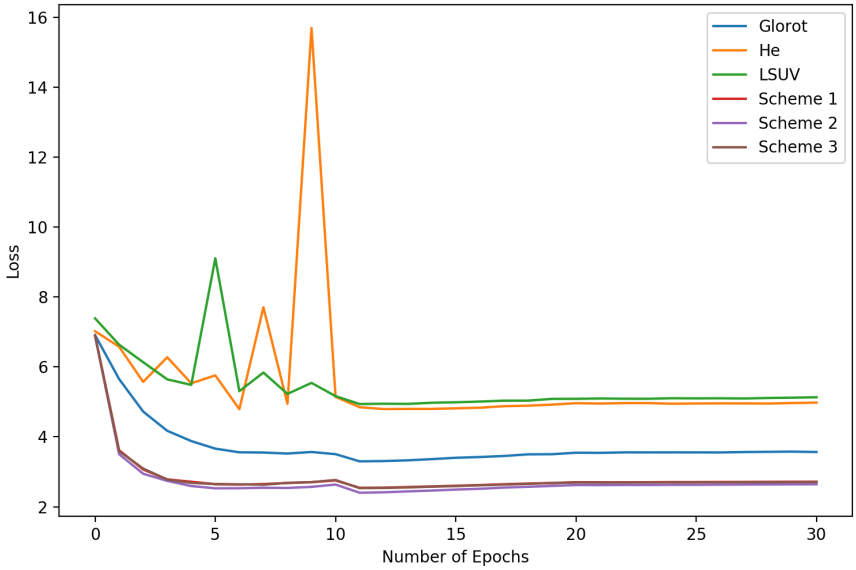

Figure 6.4: InceptionV3 - ImageNet - Loss Results

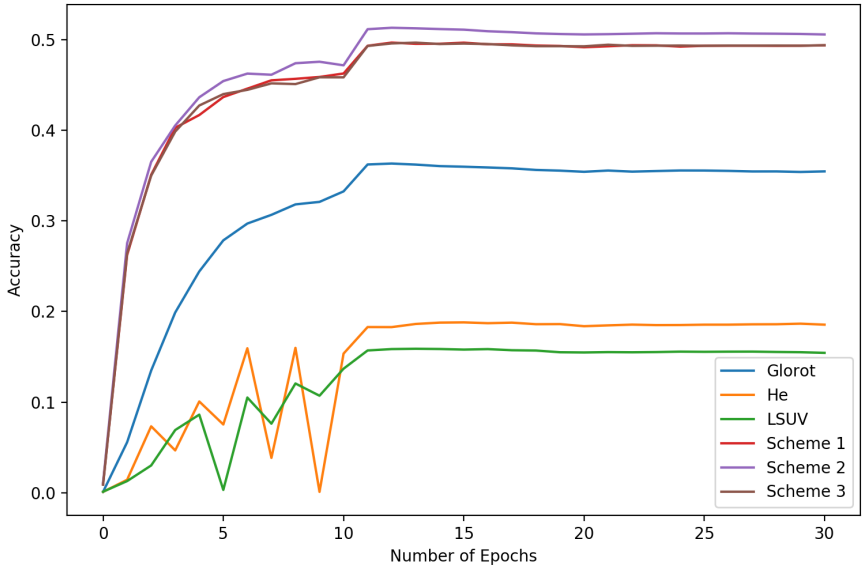

Figure 6.5: InceptionV3 - ImageNet - Accuracy Results

The results show that this initialization scheme performs almost the same as Scheme 1. No boost in performance was observed. We hypothesize that it is really hard for the support vector machine used to initialize the weights of the first convolutional layer to learn to distinguish different types of keypoints. After all, keypoints are extracted using the same rules employed by the keypoint extraction algorithm. It is very likely that all keypoints are so similar that it is just not possible to learn to tell them apart. Even though this technique did not improve the performance of the original scheme, it is worth noting that it also did not affect its performance negatively. This can be attributed to the way SVMs are trained. Even though the task of learning to identify different types of keypoints might have been impossible to solve, the resulting support vectors learned by the SVM are forced to be very different from each other since SVMs are tasked with separating classes through the use of support vectors. This means that filters initialized with these vectors are very different from each other as well, making them produce very different outputs when presented with the same input. This, at least, allows the network to exploit its representational capacity, which is something that Scheme 1 also enforces by using orthonormal filters. This explains why their performance is almost identical. However, the overhead of this keypoint-based approach is higher than the one for Scheme 1. Because of this, we discourage its use. If a convolutional layer is to be used to solve a computer vision task, we recommend the use of our Gabor-based approach. If a non-vision task is to be tackled, we recommend the use of Scheme 1. Our implementation of Scheme 3 is freely available in our GitHub repository (1).

6.4 Conclusion

In this chapter, we introduced another extension of Scheme 1. The innovation in this scheme is given by the use of popular keypoint extraction techniques used by computer vision researchers in the past. The idea is to select weight values for the first convolutional layer that allow the network to identify regions in input images that are believed to be of

interest. Results show that this idea does not provide any advantages over Scheme 1. The results are nearly identical while being more computationally expensive to run.

The objective of this approach was to incorporate what computer vision researchers learned about feature extraction before the popularization of deep learning in modern approaches. Future works includes finding other ways of achieving this objective. Approaches like SIFT, ORB, and FAST ultimately aim to find *image descriptors*. This idea is similar to that of *embeddings* in the deep learning context. More research on the similarities of these two ideas and how they can be leveraged needs to be conducted.

Chapter 7

Scheme 4: Initialization of LSTM Layers

7.1 Motivation

Modern speech recognition and machine translation systems make use of recurrent neural networks (RNNs). RNNs are capable of processing input tensors of variable size, which makes them ideal for processing sequence data, like text and audio information. As the name suggests, RNNs make use of *recurrent* layers. What makes recurrent layers special is the way they are connected within the network architecture. Besides propagating the output forward to the next layer in the network, they also propagate the output back to themselves, which is to be used by the layer as another input when the next tensor is fed to the network. This allows the network to *remember* information of the previously processed data. This is especially important when dealing with information that exhibits sequential dependencies.

Figure 7.1 shows the basic configuration of a recurrent layer. The layer produces an output h_t for every input tensor X_t it receives. Since these layers are meant to process sequence data, X_t represents the section of the input data at time step t. The output produced by the layer h_t is reused when the next input tensor is fed to the layer. Figure 7.2 shows how the tensor information flows through the layer as discrete time-steps occur.

Multiple types of RNNs have been introduced by the research community. Out of these types, the most widely used is the Long Short-Term Memory (LSTM) variant. This variant was introduced by Hochreiter and Schmidhuber (29) in 1997, and since then, it has been

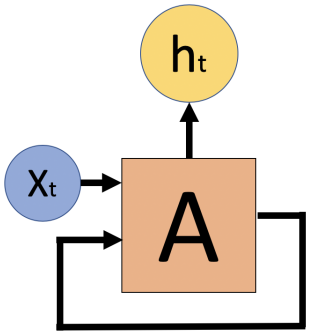

Figure 7.1: Recurrent Layer

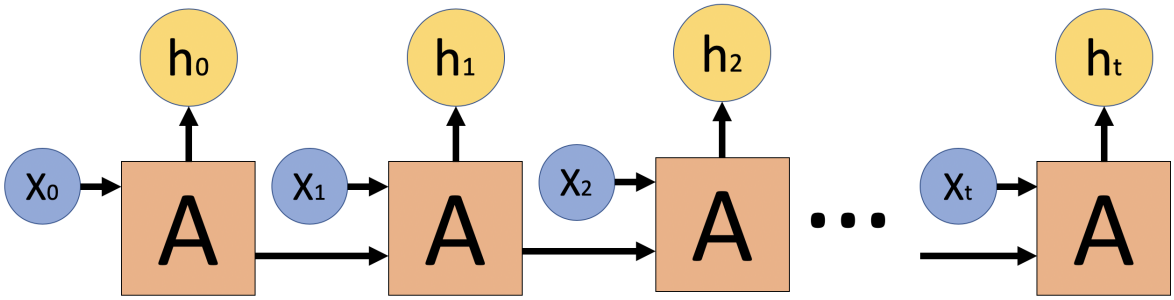

Figure 7.2: Recurrent Layer Unrolled

thoroughly studied and improved by the community due to its ability to learn long term dependencies in sequence data. These networks are widely used by well-known systems such as Siri and Alexa.

Regular RNNs exhibit a problem that prevents them from being used in real world scenarios: they cannot capture long-term dependencies in the input data. Even though their inner *loop* allows regular RNN layers to learn dependencies across time, these connections are only capable of capturing short-term dependencies. This is due to the fact that only the output from the previous time-step is used when computing the output of the present time-step. In text processing, for example, it is important to *remember* information across many words. For instance, consider the following piece of text: "Mexico is a fun country! The people there speak Spanish." Remembering the word *Mexico* is important to later understand that the word *there* refers to this country. LSTMs were introduced with the

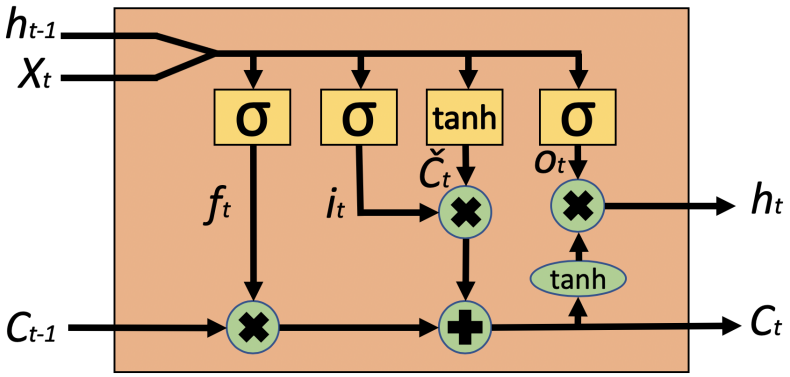

Figure 7.3: LSTM Layer

purpose of mitigating this long-term dependency problem. To do so, LSTMs are equipped with *memory* (represented with the letter C) that allows them to remember information for extended periods of time. The idea is that the network learns what to remember as the sequenced data is being fed. This *memory* is called the layer's *cell*, and is usually denoted by the letter C. This cell contains the information that the LSTM decides to keep as data is processed by the layer. The information kept in this cell is removed or added through the use of regulating structures called the layer's gates.

These gates determine how information flows within the layer. Figure 7.3 shows how the data is processed within the layer every time a new input tensor X_t is fed to the network at some time-step t. Besides X_t, the network receives as input the tensor h_{t-1}, which is the output of the layer from the previous time-step, and C_{t-1}, which is the cell produced by the layer at time $t-1$. In this representation, lines represent tensors flowing through the layer. If two lines merge, the tensors they represent are concatenated. If one line is divided into multiple ones, a copy of the tensor is assigned to each of these lines. Light orange boxes represent dense layers, and green circles represent element-wise operations performed on the input.

The light orange squares in the figure represent regular sigmoid and tanh layers with their own independent weight matrices. These dense layers have very concrete objectives. The first sigmoid layer (from left to right) represents the *forget* gate. The purpose of this

gate is to determine what information to remove (*forget*) from the memory cell. Because it employs the sigmoid activation function, the output tensor contains values in the [0,1] range. Each cell value is associated with one of the output values of the forget layer. If the value is 0, the corresponding cell entry is to be *forgotten*, while if the value is 1, the information is to be kept. The output of the layer and the cell are combined through point-wise multiplication. Mathematically, this is what the layer produces:

$$f_t = \sigma(W_f * concat(h_{t-1}, X_t) + b_f)$$

The next two layers (sigmoid and tanh) are used to determine what new information will be stored in the memory cell. The sigmoid layer represents the *input* gate, which determines what values should be updated; while the tanh layer produces candidate values to be added to the memory cell. The point-wise multiplication operation performed on the outputs of these two layers creates a tensor that contains the final information that is to be added to the memory cell. This final step is done by simply adding the values from the memory cell (after previously *forgetting* some information) and the output of the point-wise multiplication operation. Mathematically, this is how this process is performed:

$$i_t = \sigma(W_i * concat(h_{t-1}, X_t) + b_i)$$

$$\hat{C}_t = tanh(W_c * concat(h_{t-1}, X_t) + b_c)$$

$$C_t = f_t * C_t + i_t * \hat{C}_t$$

Finally, the layer's output is computed as follows. First, the last sigmoid layer produces its output. The idea is to modify parts of this output tensor by combining it with information coming from the memory cell. This is done by applying the tanh operation on the memory cell to produce values in the [-1, 1] range. Once the memory cell has been transformed to the [-1, 1] range, the two tensors are combined via point-wise multiplication, creating the final layer's output. This process is expressed mathematically as follows:

$$o_t = \sigma(W_o * concat(h_{t-1}, X_t) + b_o)$$

$$h_t = o_t * tanh(C_t)$$

LSTM layers are used in many state-of-the-art architectures. Because of their popularity and interesting topology, our last initialization technique focuses on them.

7.2 Scheme 4: Initialization for LSTM Layers

The intuition behind our initialization scheme for LSTM layers is the following. The output tensor produced by the LSTM layer at time-step $t-1$ is fed back to the layer when processing another input tensor at time-step t. What this means, mathematically, is that the weight matrices associated with each of the gates in the layer are used to compute matrix multiplication operations every time a new input X_t is fed. In an ideal scenario, the outputs produced by these matrix multiplication operations should be numerically stable. That is, we do not want values to explode or vanish, as we know that this prevents networks from learning. The problem is that we do not know exactly the properties of the input tensor X_t at time-step t. However, we do have control over the initial values given to weight matrices. Thus, we want weight values that produce numerically stable outputs without knowing exactly what type of input tensors (e.g. audio or text) will be used when the multiplication occurs. To accomplish this, we propose to initialize weight matrices with random matrices that have a condition number that is low (one or close to one).

The condition number of a matrix is defined as the ratio between its largest and smallest singular values. The singular values and singular vectors of a rectangular matrix A are defined as follows (note that we use σ to define a singular value and u and v to define its corresponding singular vectors):

$$Av = \sigma u$$

$$A^H u = \sigma v$$

In this definition, A^H is the conjugate or Hermitian transpose of the rectangular matrix A. In our context, we only deal with real matrices, so the Hermitian transpose of A is the same as A^T. The definition states that the result of multiplying A or A^T times one of A's singular vectors produces the same result as multiplying the singular value and the other singular vector. We say that a matrix is *well-conditioned* if the ratio between its largest and smallest singular values (the condition number) is low. When a matrix with a low condition number is used in a mathematical operation, the output values are stable. That is, a small change in the input of the operation is known to produce small changes in the output values. This is a desirable property in LSTM weight matrices, as we do not want output values to explode or vanish as these same output values are fed back to the layer when the next input tensor is fed.

To take advantage of these desirable properties, we propose to initialize weight matrices of LSTM layers with matrices that have a pre-determined condition number c. The deep learning practitioner has to specify the desired condition number c, and the desired average of singular values s_{avg} that the LSTM weight matrices will have. To do this, we first create an initial matrix M by randomly sampling from a normal distribution. The idea is to use M as a building block to create another matrix M' that possesses the desired condition number c and s_{avg}. The first step in this process is to decompose M using a singular value decomposition (SVD) of the form $M = USV^T$. The matrix S, in this decomposition, stores the singular values of M as diagonal entries, arranged in descending order. We create a new matrix S' where the smallest singular value (s_{min}) is given by $\frac{2*s_{avg}}{1+c}$ and the largest singular value (s_{max}) is given by $s_{min} * c$. The remaining singular values are calculated by taking the linear interpolation between s_{min} and s_{max}. Once S' is constructed, the final matrix M', which is to be used as the layer's initial weight matrix, is created as follows, $M' = US'V^T$. The whole process is described in Algorithm 3. If there are dense layers in the network that use the ReLU activation function, we initialize them as described in Chapter 4.

Algorithm 3 Weight Initialization - LSTM

1: **procedure** INITIALIZE(c, s_{avg})
2: **for** $i = 1$ to n (where n is the number of convolutional layers) **do**
3: $M \leftarrow randn(W^{(i)}.n_{rows}, W^{(i)}.n_{cols})$
4: $U, S, V \leftarrow svd(M)$
5: $s_{min} \leftarrow \frac{2 * s_{avg}}{1+c}$
6: $s_{max} \leftarrow s_{min} * c$
▷ linear interpolation is used to compute intermediate values
7: $S' \leftarrow diag(s_{min}, ..., s_{max})$
8: $W^{(i)} \leftarrow U * S' * V^T$
9:

7.2.1 Overhead

Most initialization techniques initialize weight matrices with values sampled from a given probability distribution. Our approach does the same as its first step, so no extra computation is introduced at this step. After creating the initial m-by-n matrix M, our approach computes the singular value decomposition of such matrix, which takes $O(min(mn^2, m^2n))$. Computing the values of s_{min} and s_{max} takes constant time, as only basic arithmetic operations are employed. The creation of S' takes $O(min(m, n))$, as the singular values in the diagonal are given by linear interpolation. Lastly, the final weight matrix is computed by performing two matrix multiplication operations, each of which take $O(max(m, n)^{2.4})$. In practice, we observed that this process takes a few seconds to run on an i7 laptop when initializing an LSTM layer with 30 hidden units, making the overhead cost negligible.

7.3 Experimental Results

LSTM RNNs are commonly initialized using multiple approaches. Because the weights in an LSTM layer play different roles (some are used to control the gates; while others are

used to perform a numeric transformation on the received input), each weight matrix is initialized differently. Glorot initialization is commonly used to initialize the kernel weight matrix, which is used for the linear transformation of the input tensor. Orthonormal matrices are used to initialize the recurrent kernel of the layer, which is used for the linear transformation of the layer's recurrent state. Finally, biases are usually initialized to zero. Our approach was designed to initialize the kernel matrix. Thus, we only compared our approach to Glorot's. To perform this comparison, we trained an LSTM RNN multiple times using our approach and Glorot's on the same task and dataset. To analyze how the selection of the desired condition number affects the performance of our scheme, we initialized the network using the following goal condition numbers: 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 3.0, and 4.0. For all experiments, we used s_{avg} of 1.0 and an *active fraction* of 0.8. The task that the network was trained to solve was that of turn-taking. The purpose of the model is to predict if a given person (participating in a two-people dialogue) will speak or not in the immediate future given a sequence audio features. To train this model, we used the dataset produced by Ward *et al.* in (79). This dataset, originally built from MapTask (3), is composed of 1,168 training and 410 testing instances. Each instance is composed of 1,200 time-steps. Each time-step is composed of 12 prosodic features that were computed in a 50ms window; 6 of those features represent the activity of one of the speakers, and the other 6 capture the activity of the second speaker. These 6 feature are:

1. Absolute Pitch
2. Relative Pitch
3. Voicing (binary - indicates if the frame was voiced)
4. Energy
5. Voice Activity (binary - indicates if participant was speaking)
6. Cepstral flux (proxy measure for lengthening)

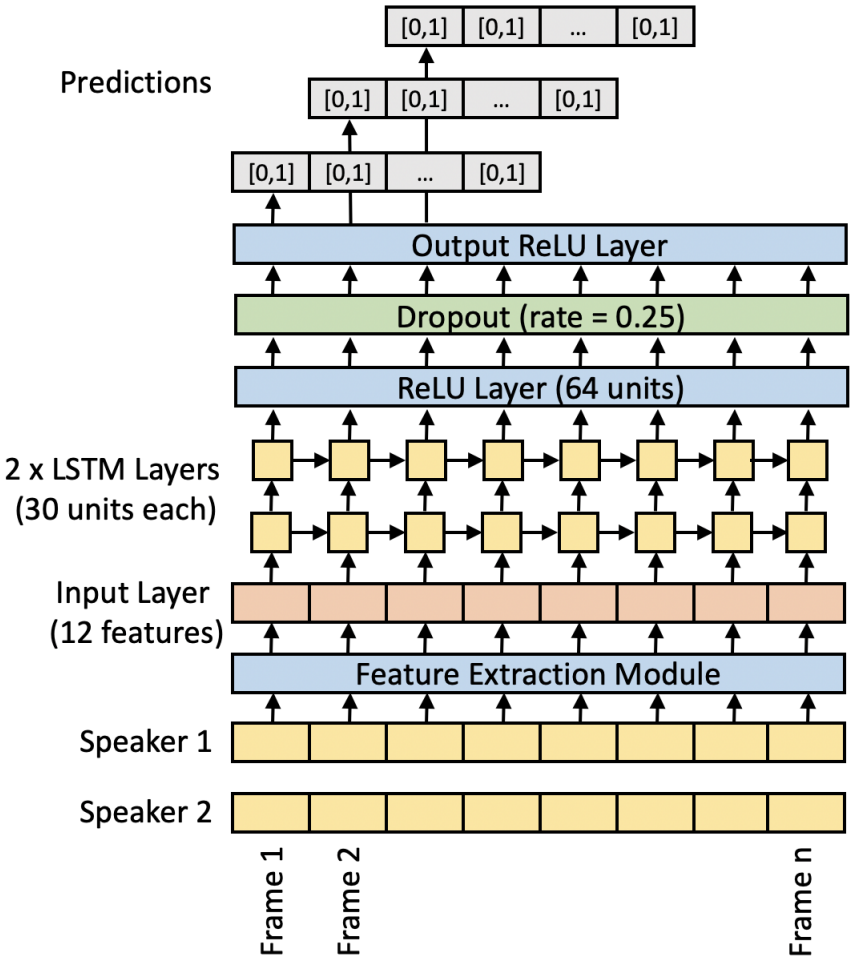

Figure 7.4: Architecture of the LSTM RNN

The network is to predict whether the first participant in the dialogue will speak (1) or not (0) in the upcoming n 50ms frames. The code to extract the used features can be found in (78). The architecture of the LSTM RNN used in this set of experiments is described in Figure 7.4.

For every experiment, the network was initialized using a different initialization technique and was trained for 30 epochs. During training, the data was fed to the network in mini-batches of size 32. The RMSProp optimizer was used with an initial learning rate of 0.001. The mean-squared error (MSE) between the ground truth and the network's pre-

dictions was used to measure performance. Each experiment was repeated for five different prediction window sizes: 0 to 250ms, 0 to 500ms, 0 to 1s, 0 to 2s, and 0 to 3s. The results of these experiments are presented in Tables 7.1, 7.2, 7.3, 7.4, and 7.5.

Table 7.1: LSTM Initialization - MAE Results - 250ms

Initialization Method	After initialization	After 1 epoch	After 5 epochs	After 15 epochs	After 30 epochs
Scheme 4 - 1.0	**0.228**	0.219	0.203	0.190	0.182
Scheme 4 - 1.2	0.229	0.219	0.202	0.189	0.181
Scheme 4 - 1.4	0.229	0.219	0.201	0.189	0.181
Scheme 4 - 1.6	0.229	**0.218**	**0.200**	0.188	0.181
Scheme 4 - 1.8	0.230	0.219	**0.200**	0.188	0.181
Scheme 4 - 2.0	0.230	0.220	0.201	0.189	0.181
Scheme 4 - 3.0	0.236	0.225	0.202	0.189	0.181
Scheme 4 - 4.0	0.238	0.229	0.204	0.190	0.181
Glorot	0.411	0.296	0.206	**0.186**	**0.178**

After training the networks for 30 epochs, the results show that our approach works better if the prediction window is large. For smaller windows, Glorot is favored. The results, nonetheless, are so similar that it is fair to conclude that both approaches perform identically when it comes to network performance. There is an important and noteworthy distinction though. Our initialization scheme produces good results even before training the network. After initialization and before training, our approach produces MAE values that are close to the one achieved after training for 30 epochs. For a window size of 250ms, the MAE obtained after initialization and before training using our approach is only 12% worse than the final MAE obtained after training the network. This is also consistent across other window sizes. If our initial MAE is compared to the ones produced by Glorot, we see improvments of up to 35%. It is important to notice that the main disadvantage of LSTM

Table 7.2: LSTM Initialization - MAE Results - 500ms

Initialization Method	After initialization	After 1 epoch	After 5 epochs	After 15 epochs	After 30 epochs
Scheme 4 - 1.0	**0.276**	0.271	0.261	0.252	0.246
Scheme 4 - 1.2	0.277	**0.270**	0.261	0.252	0.247
Scheme 4 - 1.4	0.277	**0.270**	**0.260**	0.252	0.247
Scheme 4 - 1.6	0.278	**0.270**	**0.260**	0.252	0.247
Scheme 4 - 1.8	0.278	0.271	**0.260**	**0.251**	0.246
Scheme 4 - 2.0	0.279	0.271	**0.260**	**0.251**	0.246
Scheme 4 - 3.0	0.281	0.274	0.261	0.252	0.245
Scheme 4 - 4.0	0.282	0.274	0.262	0.253	0.246
Glorot	0.413	0.349	0.274	0.253	**0.244**

RNNs is their slow training time. It is hard to parallelize the computations performed by an LSTM layer since the output of such layer at time t depends on the output of the same layer at time $t-1$. Our approach offers a way of substantially improving the training time at negligible overhead. In our experiments, the initialization process took less than one hundredth of the time it takes to complete a whole training epoch.

7.4 Conclusion

In this chapter, we introduced a new weight initialization scheme for recurrent layers. The idea is to initialize weight values with matrices that have a low condition number. The motivation behind this decision is that a lot of matrix multiplication operations are performed using these matrices as operands. If their condition number is small, values are not likely to explode or vanish. Future work includes using this approach to tackle larger, more complex problems as the problem that we used to analyze the method does not

Table 7.3: LSTM Initialization - MAE Results - 1s

Initialization Method	After initialization	After 1 epoch	After 5 epochs	After 15 epochs	After 30 epochs
Scheme 4 - 1.0	**0.331**	**0.328**	0.322	**0.313**	**0.308**
Scheme 4 - 1.2	**0.331**	**0.328**	0.322	**0.313**	**0.308**
Scheme 4 - 1.4	0.332	**0.328**	0.322	0.316	0.310
Scheme 4 - 1.6	0.332	**0.328**	**0.321**	0.315	0.309
Scheme 4 - 1.8	**0.331**	0.329	0.322	0.315	0.309
Scheme 4 - 2.0	**0.331**	0.329	0.323	0.314	0.309
Scheme 4 - 3.0	**0.331**	0.330	0.324	0.316	0.310
Scheme 4 - 4.0	0.332	0.331	0.325	0.318	0.311
Glorot	0.446	0.386	0.333	0.315	**0.308**

seem to be challenging enough. We were not able to confirm our hypothesis that matrices with low condition numbers mitigate the effects of the vanishing and exploiting gradient problems. However, good results were achieved even if the network is not trained for a large number of epochs.

Table 7.4: LSTM Initialization - MAE Results - 2s

Initialization Method	After initialization	After 1 epoch	After 5 epochs	After 15 epochs	After 30 epochs
Scheme 4 - 1.0	**0.374**	**0.372**	0.369	0.361	**0.355**
Scheme 4 - 1.2	0.375	**0.372**	**0.368**	**0.360**	**0.355**
Scheme 4 - 1.4	0.375	**0.372**	**0.368**	**0.360**	0.356
Scheme 4 - 1.6	0.375	0.374	0.369	0.361	**0.355**
Scheme 4 - 1.8	0.375	0.375	0.371	0.361	0.357
Scheme 4 - 2.0	0.375	0.375	0.370	0.362	0.358
Scheme 4 - 3.0	**0.374**	0.375	0.371	0.363	0.359
Scheme 4 - 4.0	**0.374**	0.373	0.370	0.365	0.359
Glorot	0.451	0.391	0.375	0.364	0.359

Table 7.5: LSTM Initialization - MAE Results - 3s

Initialization Method	After initialization	After 1 epoch	After 5 epochs	After 15 epochs	After 30 epochs
Scheme 4 - 1.0	**0.392**	0.390	0.387	0.380	**0.375**
Scheme 4 - 1.2	0.393	**0.389**	**0.385**	**0.378**	**0.375**
Scheme 4 - 1.4	0.393	0.391	0.387	0.379	0.377
Scheme 4 - 1.6	0.393	0.391	0.387	0.380	0.376
Scheme 4 - 1.8	0.393	0.393	0.390	0.382	0.378
Scheme 4 - 2.0	0.393	0.392	0.391	0.384	0.378
Scheme 4 - 3.0	**0.392**	0.394	0.391	0.386	0.380
Scheme 4 - 4.0	**0.392**	0.394	0.391	0.386	0.381
Glorot	0.416	0.392	0.392	0.385	0.380

Chapter 8

Conclusion and Future Work

8.1 Contribution Analysis

In this work, four novel initialization techniques for deep learning models were introduced and evaluated. The common theme among these techniques is the use of an initialization set to select weight values that minimize the network's error at initialization time. We also used properties of the network architecture (e.g. ReLU activation function and recurrent connections) and the problem they aim to solve (e.g. computer vision tasks and Gabor filters) to build heuristics that improve the initialization process.

The first initialization technique can be used to initialize any layer in a network that uses the ReLU (or similar) non-linear function. It allows deep learning practitioners to define how likely it is for a neuron to produce a non-zero activation. Our experiments showed that this initialization scheme yields good results. Our scheme was able to reduce the initial loss value of the model (before training) with little overhead. And, more importantly, models trained using this scheme were shown to perform significantly better (up to 40% in terms of accuracy) when compared to other, state-of-the-art initialization approaches.

The second and third initialization schemes were designed to initialize convolutional models that tackle computer vision tasks. These schemes extend the first one. The difference resides in the way the first convolutional layer is initialized. The second scheme uses randomly generated Gabor filters to initialize the layer. The third scheme initializes the filters in the layer to produce non-zero activations when processing interesting regions of input images. These regions are identified using popular keypoint extraction techniques, such as SIFT, ORB, and FAST. Our experiments showed that the Gabor-based initializa-

tion improve the accuracy performance of the first scheme by up to 3%. This was not the case with our keypoint-based initialization, which did not manage to outperform Scheme 1.

Our fourth and last scheme was designed to initialize recurrent layers. This approach initializes weight values with matrices that have low condition numbers. This is done to mitigate the vanishing and exploding gradient problems at training time. Our experiments show that this technique allows networks to learn faster while introducing negligible overhead at initialization time. This approach produces results (after initialization, but before training) that are up to 35% better than the state-of-the-art.

This work's contributions are noteworthy. We expect the research community to adopt our initialization schemes. With little overhead and less training time, better performing models can be created. As our initialization techniques become more widespread, we speculate that they will allow for the creation of record-breaking models in the many areas where deep learning is employed.

8.2 Thesis Statement and Research Questions Analysis

Our experimental results support our thesis statement (copied below for convenience). **Thesis statement:** *Weight initialization in neural networks can be improved by 1) using statistical information extracted from a subset of the training data set, and 2) incorporating properties of the architecture and the data set.*

The design of our initialization schemes was guided by our thesis statement and research questions. Similarly, all experiments were designed to answer our research questions while producing convincing evidence to support our claims. For convenience, our research questions are copied below and short-answers are provided.

Research Questions:

1. How can we exploit statistical information extracted from a training batch to initialize the parameters of a neural network? Is the overhead added to the process worth it?

   In all schemes, we used statistical information extracted from an initialization set when determining the initial values of all biases in the network. This was done to satisfy the constraints imposed by the *active fraction* hyperparameter. We also used statistics from this set to force activations to have a given standard deviation. In the second and third schemes, we used information extracted from training images to construct the filters of the first convolutional layer.

   The overhead is worth it since it is minimal and allows for the creation of better performing models that can be trained faster.

2. What properties of the network architecture can we use to further tune parameter values at initialization time?

   In all schemes, we exploited the fact that most modern deep learning architectures heavily use ReLU-based functions to introduce non-linearity into the model. This was done through the use of the *active fraction* hyperparameter. For the last initialization scheme, weight matrices with low singular values were used since the loops in the architecture of recurrent layers introduce a significant amount of matrix multiplication operations.

3. How can we incorporate information about the data set/problem to improve weight initialization techniques across network architectures?

   When designing schemes 2 and 3, we decided to tackle convolutional models that deal with computer vision tasks. For scheme 2, we exploited the fact that the first layer in convolutional networks is known to learn weight values that closely resemble Gabor filters. For scheme 3, we used popular, human-crafted feature extraction mechanisms to improve weight initialization.

8.3 Future Work

Future work for each of the presented schemes is discussed in their corresponding chapters. This section is devoted to talk about the future work of this line of research from a broader perspective. This work focused on the creation of a novel set of initialization approaches. Even though we compared the presented schemes against others by training well-known networks on well-known datasets, it was never the main intent of this research to train well-performing models. However, we suspect that, if our initialization approaches are used in combination with modern regularization and data augmentation techniques, we can re-train existing state-of-the-art models to improve their performance even further. We plan to explore this in the near future.

We also intend to develop new initialization techniques for other types of layers. As shown by our experimental results, specialized initialization schemes significantly outperform general ones. We are interested in improving the way *attention* (74) modules are initialized. The concept of attention was recently introduced in the deep learning domain, and its use has become widespread. By adding attention mechanisms to network architectures, researchers have been able to tackle problems like image captioning and machine translation more effectively. We plan to develop strategies to formally analyze and understand what attention modules learn, in order to design specialized initialization schemes for them.

Finally, we plan to explore creating new regularization techniques based on the same concepts that guided the creation of the presented initialization schemes. We believe that some of the numerical properties that our initialization schemes provide should be kept as the training process updates weight values. This can be done through regularization or direct enforcement.

References

[1] D. Aguirre. Weight initialization code repository. https://github.com/aguirrediego/weight-initialization-relu-and-output-layers, 2019.

[2] D. Aguirre. Weight initialization results. http://bit.ly/2W3iEIr, 2019.

[3] A. H. Anderson, M. Bader, E. G. Bard, E. Boyle, G. Doherty, S. Garrod, S. Isard, J. Kowtko, J. McAllister, J. Miller, et al. The HCRC map task corpus. *Language and Speech*, 34:351–366, 1991.

[4] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee. Understanding deep neural networks with rectified linear units. *arXiv preprint arXiv:1611.01491*, 2016.

[5] V. Badrinarayanan, A. Kendall, and R. Cipolla. SegNet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, 2015.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[8] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF: Binary robust independent elementary features. In *European conference on computer vision*, pages 778–792. Springer, 2010.

[9] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. DeepDriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[10] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2018.

[11] X. Chen, K. Kundu, Z. Zhang, H. Ma, S. Fidler, and R. Urtasun. Monocular 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2156, 2016.

[12] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 7–10. ACM, 2016.

[13] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[14] J. Chung, K. Cho, and Y. Bengio. A character-level decoder without explicit segmentation for neural machine translation. *arXiv preprint arXiv:1603.06147*, 2016.

[15] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier. Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083*, 2016.

[16] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[17] A. M. Elkahky, Y. Song, and X. He. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *Proceedings of the 24th International Conference on World Wide Web*, pages 278–288. International World Wide Web Conferences Steering Committee, 2015.

[18] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov. Scalable object detection using deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2154, 2014.

[19] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[20] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

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

[22] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. DRAW: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[23] M. Haghighat, S. Zonouz, and M. Abdel-Mottaleb. Identification using encrypted biometrics. In *International Conference on Computer Analysis of Images and Patterns*, pages 440–448. Springer, 2013.

[24] J. Han, D. Zhang, G. Cheng, N. Liu, and D. Xu. Advanced deep-learning techniques for salient and category-specific object detection: a survey. *IEEE Signal Processing Magazine*, 35(1):84–100, 2018.

[25] C. G. Harris, M. Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. CiteSeer, 1988.

[26] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[28] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] S. Hochreiter and J. Schmidhuber. LSTM can solve hard long time lag problems. In *Advances in neural information processing systems*, pages 473–479, 1997.

[30] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*, 2017.

[31] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[32] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[33] J. P. Jones and L. A. Palmer. An evaluation of the two-dimensional gabor filter model of simple receptive fields in cat striate cortex. *Journal of neurophysiology*, 58(6):1233–1258, 1987.

[34] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[35] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[36] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. OpenNMT: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810*, 2017.

[37] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[38] Y. LeCun. The MNIST database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.

[39] J. Li, W. Monroe, T. Shi, S. Jean, A. Ritter, and D. Jurafsky. Adversarial learning for neural dialogue generation. *arXiv preprint arXiv:1701.06547*, 2017.

[40] R. Li, W. Zhang, H.-I. Suk, L. Wang, J. Li, D. Shen, and S. Ji. Deep learning based imaging data completion for improved brain disease diagnosis. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 305–312. Springer, 2014.

[41] T.-Y. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie. Feature pyramid networks for object detection. In *CVPR*, volume 1, page 3, 2017.

[42] Z. Liu, X. Li, P. Luo, C.-C. Loy, and X. Tang. Semantic image segmentation via deep parsing network. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1377–1385, 2015.

[43] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[44] S. Luan, C. Chen, B. Zhang, J. Han, and J. Liu. Gabor convolutional networks. *IEEE Transactions on Image Processing*, 2018.

[45] S. Makridakis. The forthcoming artificial intelligence (AI) revolution: Its impact on society and firms. *Futures*, 90:46–60, 2017.

[46] G. Manogaran, R. Varatharajan, and M. Priyan. Hybrid recommendation system for heart disease diagnosis based on multiple kernel learning with adaptive neuro-fuzzy inference system. *Multimedia tools and applications*, 77(4):4379–4399, 2018.

[47] D. Mishkin and J. Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[48] L. Nie, M. Wang, L. Zhang, S. Yan, B. Zhang, and T.-S. Chua. Disease inference from health-related questions via sparse deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2107–2119, 2015.

[49] H. Noh, S. Hong, and B. Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[50] A. Ortiz, J. Munilla, J. M. Gorriz, and J. Ramirez. Ensembles of deep learning architectures for the early diagnosis of the Alzheimer's disease. *International journal of neural systems*, 26(07):1650025, 2016.

[51] W. Ouyang, X. Wang, X. Zeng, S. Qiu, P. Luo, Y. Tian, H. Li, S. Yang, Z. Wang, C.-C. Loy, et al. DeepID-NET: Deformable deep convolutional neural networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2403–2412, 2015.

[52] A. Ramakrishnan, S. K. Raja, and H. R. Ram. Neural network-based segmentation of textures using gabor features. In *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing*, pages 365–374. IEEE, 2002.

[53] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (6):1137–1149, 2017.

[54] F. Richardson, D. Reynolds, and N. Dehak. Deep neural network approaches to speaker and language recognition. *IEEE Signal Processing Letters*, 22(10):1671–1675, 2015.

[55] F. Richardson, D. Reynolds, and N. Dehak. A unified deep neural network for speaker and language recognition. *arXiv preprint arXiv:1504.00923*, 2015.

[56] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[57] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.

[58] E. Rublee, V. Rabaud, K. Konolige, and G. R. Bradski. ORB: An efficient alternative to sift or surf. In *ICCV*, volume 11, page 2. Citeseer, 2011.

[59] T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

[60] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017.

[61] A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

[62] S. Semeniuta, A. Severyn, and E. Barth. A hybrid convolutional variational autoencoder for text generation. *arXiv preprint arXiv:1702.02390*, 2017.

[63] D. Shen, G. Wu, and H.-I. Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[64] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[65] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[66] P. Sudowe and B. Leibe. Patchit: Self-supervised network weight initialization for fine-grained recognition. In *BMVC*, 2016.

[67] H.-I. Suk, S.-W. Lee, D. Shen, A. D. N. Initiative, et al. Hierarchical feature representation and multimodal fusion with deep learning for AD/MCI diagnosis. *NeuroImage*, 101:569–582, 2014.

[68] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[69] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[70] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[71] Y. Taigman, A. Polyak, and L. Wolf. Unsupervised cross-domain image generation. *arXiv preprint arXiv:1611.02200*, 2016.

[72] T. Tijmen and H. Geoffrey. Lecture 6.5 - RMSprop. *COURSERA: Neural networks for machine learning*, 2012.

[73] A. van den Oord, N. Kalchbrenner, L. Espeholt, O. Vinyals, A. Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems*, pages 4790–4798, 2016.

[74] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[75] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.

[76] P. Viola, M. Jones, et al. Rapid object detection using a boosted cascade of simple features. *CVPR (1)*, 1(511-518):3, 2001.

[77] H. Wang, N. Wang, and D.-Y. Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1235–1244. ACM, 2015.

[78] N. G. Ward. Midlevel prosodic features toolkit. https://github.com/nigelgward/midlevel, 2017.

[79] N. G. Ward, D. Aguirre, G. Cervantes, and O. Fuentes. Turn-taking predictions across languages and genres using an lstm recurrent neural network. In *2018 IEEE Spoken Language Technology Workshop (SLT)*, pages 831–837. IEEE, 2018.

[80] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.

[81] Y. Yamada, M. Iwamura, and K. Kise. Shakedrop regularization. *arXiv preprint arXiv:1802.02375*, 2018.

[82] F. Yan and K. Mikolajczyk. Deep correlation for matching images and text. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3441–3450, 2015.

[83] T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018.

[84] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[85] M. D. Zeiler, G. W. Taylor, R. Fergus, et al. Adaptive deconvolutional networks for mid and high level feature learning. In *ICCV*, volume 1, page 6, 2011.

[86] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017.

[87] B. Zoph, D. Yuret, J. May, and K. Knight. Transfer learning for low-resource neural machine translation. *arXiv preprint arXiv:1604.02201*, 2016.

Vita

Diego Aguirre is an Assistant Professor of Practice at the University of Texas at El Paso (UTEP). His research interests lie in the areas of deep learning, computer vision, and reinforcement learning. Diego has had the opportunity to collaborate alongside companies like Lockheed Martin, NASA, IBM, Leidos, and Google. He currently teaches multiple courses at UTEP and actively participates in the university's Vision and Learning Lab.

Contact Information: diego4.aguirre@gmail.com