© 2016 Raymond Alexander Yeh

# STABLE AND SYMMETRIC CONVOLUTIONAL NEURAL NETWORK

BY

RAYMOND ALEXANDER YEH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Advisers:

Professor Minh N. Do
Professor Mark Hasegawa-Johnson

# ABSTRACT

First we present a proof that convolutional neural networks (CNNs) with max-norm regularization, max-pooling, and Relu non-linearity are stable to additive noise. Second, we explore the use of symmetric and antisymmetric filters in a baseline CNN model on digit classification, which enjoys the stability to additive noise. Experimental results indicate that the symmetric CNN outperforms the baseline model for nearly all training sizes and matches the state-of-the-art deep-net in the cases of limited training examples.

*To my parents, for their love and support.*

# ACKNOWLEDGMENTS

I would like to thank Professor Mark Hasegawa-Johnson and Professor Minh Do for their guidance and support throughout the research process. Also, I would like to thank the members in the Statistical Speech Technology Group and the Computational Imaging Group for their valuable suggestions and discussions. Lastly, I thank my parents and sister for their love and support.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| CNN | Convolutional Neural Network |
| SGD | Stochastic Gradient Descent |
| ReLU | Rectified Linear Unit |
| PCA | Principal Component Analysis |
| MNIST | Mixed National Institute of Standards and Technology Database |

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

In recent years, deep learning has demonstrated state-of-the-art performance a wide range of areas and tasks. Speech and image processing task benchmarks are now dominated by deep learning models. In particular, deep learning has the most significant impact on image recognition and speech recognition research [1, 2, 3]. Many companies (e.g. Google, Facebook) are now applying deep learning techniques to their services. The main advantage or attraction of deep learning is the ability to learn the useful *features* from the data; through the hierarchical structure, features of higher levels can be learned. This is very different from the classical approach, in which human experts design specific feature extraction processes based on domain knowledge.

Empirically, deep learning has outperformed the classical approach; however, it is not without a cost: (1) Deep learning requires a lot of labeled training data, due to the usually large number of parameters. (2) Deep learning is notorious for difficulty in training process, which is a non-convex optimization problem, resulting in problems in local minimum and long training time with gradient methods. (3) The learned features are hard to interpret and lack theoretical support. In this thesis, we will attempt of tackle some of the issues surrounding deep learning.

## 1.2 Background

### 1.2.1 Brief Overview of Statistical Learning

In this section, we will briefly review the setting of a learning problem to understand the trade-off of model selection with Vapnik-Chervonenkis (VC) dimension.

The following review is based on [4].

Given data set $D_n = \{(X_1, Y_1), (X_2, X_2), ..., (X_n, Y_n)\}$, which comprises independent identically distributed (i.i.d.) random pairs drawn from a distribution $P(\mathcal{X}, \mathcal{Y})$, the goal for a machine learning algorithm is to find a function $g : \mathcal{X} \mapsto \mathcal{Y}$ classifier that minimizes a loss function as $l(g(X), Y)$ that measure how "good" the classifier is at approximating the $\mathcal{X} \mapsto \mathcal{Y}$ relationship.

We define the expected risk as follows:

$$\mathcal{L}(g) = \mathbb{E}_P[l(g(X), Y)] = \int l(g(x), y) dP(x, y) \tag{1.1}$$

Then the best classifier, $g^*$, can be denoted as

$$g^* = \arg\min_g \mathcal{L}(g) \tag{1.2}$$

In theory, if we know the distribution $P$, then we will just directly solve for the best classier. However, in practice, we only get access to the dataset that is drawn from $P$. We define the empirical risk of $n$ samples as follows:

$$\mathcal{L}_n(g) = \frac{1}{n} \sum_{i=1}^{n} l(g(x_i), y_i) \tag{1.3}$$

As minimization over all families of functions is impractical, a learning algorithm will be restricted to certain families of functions, which we denote $\mathcal{G}$. Then the error of a classifier, $g \in \mathcal{G}$, can be divided into two parts, the estimation error, $E_{est}$, and the approximation error, $E_{approx}$:

$$E_{est} = [\mathcal{L}(g^*_{\mathcal{G}}) - \mathcal{L}(g_n)] \tag{1.4}$$

$$E_{approx} = [\mathcal{L}(g^*) - \mathcal{L}(g^*_{\mathcal{G}})] \tag{1.5}$$

where $g^*_{\mathcal{G}} = \arg\min_{g\in\mathcal{G}} \mathcal{L}(g)$, and $g_n$ is the produce function from $n$ examples of a machine learning algorithm.

As can be seen, there is a trade-off between $E_{est}$ and $E_{approx}$. If we choose a very large family for $\mathcal{G}$, then $E_{approx}$ will more likely be small, but $E_{est}$ will more likely be large. It turns out that the $E_{est}$ can be upper bounded by the VC dimension, which in a way describes the "size" of $\mathcal{G}$ [5].

Deep neural networks essentially choose a large $\mathcal{G}$ in order to get a closer approximation to the true $g*$, while still having a finite VC dimension, meaning that learning is possible given enough examples. While the theoretical bounds on $E_{est}$ are too loose to be used in practice, it shows that learning is possible with deep neural networks.

### 1.2.2 Neural Network

In this section, we will motivate the use of neural networks and review the model definition.

#### Motivation

One of the incentives to use a neural network is that it is a universal approximator. The universal approximation theorem states that a feed-forward neural network with a single hidden layer, with a finite number of nodes, can approximate any continuous function with a certain precision [6]. The larger the network, the more accurate the approximation. However, this theorem only shows the existence of such a network and does not give an algorithm to find it; generally, the larger the network, the more difficult it is to optimize the parameters.

#### Definition of Neural Network

The most vanilla version of a neural network can be thought as repetition of linear transformation followed by an element-wise non-linear operator. Using the notation in [7], we can define a neural network iteratively as

$$z^{(l)} = W^{(l-1)}a^{(l-1)} + b^{(l-1)} \tag{1.6}$$

$$a^{(l)} = f(z^{(l)}) \tag{1.7}$$

where $\mathbf{a}^{(l)}$ is the input vector from layer $l$, $W^{(l)}$ are the weights at layer $l$, $b^{(l)}$ are the biases at layer $l$, and the term $f$ is an element-wise non-linear function such as *sigmoid*, *tanh* or $Relu(x) = \max(0, x)$.

The computation of $a^{(l+1)}$ is commonly called the forward operation of the neural network. Figure 1.1 is a visualization of a neural network with two hidden layers, where each circle depicts an activation, $a$, and each connect represents a weight, $w$.

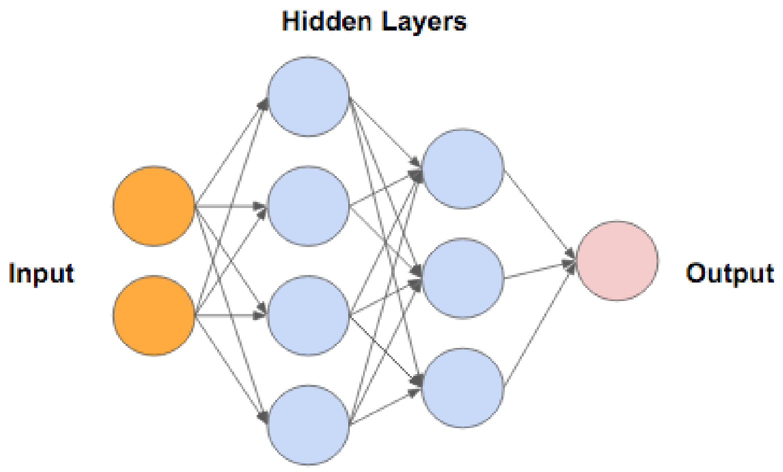

Figure 1.1: Neural network with two hidden layers

### 1.2.3 Convolutional Neural Network

#### Motivation

In this section, we will motivate the use of convolutional neural network (CNNs) and review the model definition [8]. Consider the case in which we are applying neural network to an image classification task. Assume the input image to be $250 \times 250 \times 3$; this is small compared to a photo taken by a typical smart phone, which consists of approximately $3264 \times 2448$ pixels. This image size means that the input dimension of the neural network is 187500, which means for each hidden node in the next layer it will need 187500 weights. Further, assume one half dimension at each layer. Then, a 2-hidden-layer neural network would consist of $187500 \cdot \frac{187500}{2} + \frac{187500}{2} \cdot \frac{187500}{4} = 21,972,656,250$, which is an impractical number of parameters. One idea to reduce the dimension is to assume stationarity in image patches (i.e. each image patch is from the same distribution regardless of the location in the overall image). Once we make this assumption, then it make sense that the weights will only be the dimension of the local patch. Additionally, weights are "shared" across the patches, which creates the sliding window operation of a convolution, more formally defined below. Note that this stationarity assumption is generally not true for images, but empirically CNN is the state-of-the-art model for image related tasks.

## Convolution Layer

We will follow the notation from [7].
Denote the following:

- $a_j^{(l)}$ = the $j^{th}$ channel of the activation map at $l^{th}$ layer, where $a_j^{(l)}$ is a matrix.
- $W_{ij}^{(l)}$ = the $i^{th}$ channel of the $j^{th}$ filter at $l^{th}$ layer, where $W_{ij}^{(l)}$ is a matrix.
- $b_j^{(l)}$ = the bias for $j^{th}$ filter at $l^{th}$ layer, where $b_j^{(l)}$ is a scalar.
- $f(\cdot)$ = a element-wise non-linear function.
- $\star$ = convolution

Then the forward operation of a convolution layer is

$$z_j^{(l)} = (\sum_i a_i^{(l-1)} \star W_{ij}^{(l-1)}) + b_j^{(l-1)} \tag{1.8}$$

$$a_j^{(l)} = f(z_j^{(l)}) \tag{1.9}$$

Note that $b_j^{l-1}$ is added to all outputs from the convolution.

As can be seen, one can think of a CNN as a stack of linear filter operations followed by an element-wise non-linearity; again, *Relu* is the typical choice.

### Pooling Operation

Pooling operation can be thought of as linear or non-linear filtering followed by down-sampling. The main idea behind the use of a pooling layer is dimension reduction, to reduce the number of activations in the next layer [9].

In particular, a common pooling operation is max pooling, which is a maximum filter followed by a down-sampling operation.

Maximum filter is defined as:

$$y_{i,j,d} = \max_{k,l \in N} x_{(i+k),(j+l),d} \tag{1.10}$$

where $N$ denotes the neighborhood for pooling.

Max pooling is also advantageous for its approximate spatial invariant property (i.e., if the max value does not shift outside the pooling window, then the output remains the same). The spatial invariant property is particularly desirable when the location of the signal is irrelevant (e.g. object recognition).

Similarly, average pooling is a mean filter, followed by down-sampling. Mean filter is defined as

$$y_{i,j,d} = \frac{1}{|N|} \sum_{k,l \in N} x_{(i+k),(j+l),d} \tag{1.11}$$

where $N$ denotes the neighborhood for pooling.

## 1.2.4 Training Neural Networks

In this section, we will review the common tools for optimizing neural networks [10, 7].

### Gradient Descent

Gradient descent is a first-order optimization method. The method is to iteratively move in the direction of the negative of the gradient at the current point of the function we wish to minimize.

Denote the following:

- $\Theta$ = The space of all possible parameters in the model.
- $\theta \in \Theta$ = A specific instance of the model parameters.
- $\mathcal{L} : \Theta \mapsto \mathbb{R}$ = The loss function to minimize.
- $\alpha \in \mathbb{R}$ = the learning rate or step size.

The gradient descent algorithm starts at a random initial point $\theta_0$, then repeatedly updates $\theta_t$ until convergence.

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial}{\partial \theta_t} \mathcal{L}(\theta_t) \tag{1.12}$$

Figure 1.2 is a visualization of the gradient descent algorithm on a convex $\mathcal{L}$.

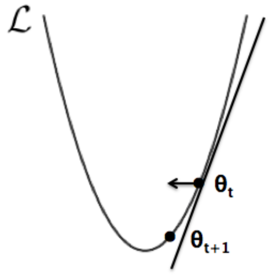

Figure 1.2: Gradient descent visualization

Note that the learning rate has to be chosen with care; Figure 1.3 shows that large step-size can lead to divergence.

### Stochastic/Batched Gradient Descent

The gradient descent algorithm described above is not practical from a computational point of view. In a machine learning problem the loss function

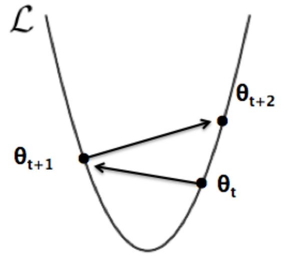

Figure 1.3: Gradient descent large step size visualization

takes the form $\mathcal{L}(\theta) = \sum_{i=0}^{n} l(x_i, y_i; \theta)$, where $n$ is the number of training examples. Hence, for every gradient step we will need to compute the gradient for each training example; this is computationally unfeasible when we have large amount of training data.

A solution is to approximate the gradient by using subsets of the training data:

$$\mathcal{L}_{D_t}(\theta) = \sum_{x_i, y_i \in D_t} l(x_i, y_i; \theta) \tag{1.13}$$

where $D_t$ is some subset of the training data and $|D_t|$ is the batch-size.

Again, stochastic gradient descent starts at a random initial point $\theta_0$; then repeatedly update $\theta_t$ until convergence.

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial}{\partial \theta_t} \mathcal{L}_{D_t}(\theta_t) \tag{1.14}$$

## Back Propagation

We need an algorithm to compute the gradient for each of the parameters. This is referred to as the backward operation in a neural network, where the loss is "back propagated" from the top of the network to the input. Essentially, back propagation is an application of the chain rule [11, 7].

Recall that the computation of the neural network involves

$$z_j^{(l)} = \sum_i w_{ji}^{(l-1)} a_i^{(l-1)} + b_j^{(l-1)} \tag{1.15}$$

$$a_j^{(l)} = f(z_j^{(l)}) \tag{1.16}$$

Denote the back-propagated errors, $\delta$, as

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \tag{1.17}$$

Next, we can observe that

$$\frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l-1)}} = a_i^{(l-1)} \tag{1.18}$$

Then using the chain rule, we can compute the gradient with respect to the weights

$$\frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial w_{ji}^{(l-1)}} \tag{1.19}$$

Next, to compute the $\delta_j$, again using the chain rule,

$$\delta_j = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \tag{1.20}$$

Then, substituting in the definition of *delta* and expanding the $\frac{\partial z_k}{\partial z_j}$ term using the chain rule,

$$\delta_j^l = f'(z_k^{(l)}) \sum_k w_{kj}^{(l)} \delta_k^{(l+1)} \tag{1.21}$$

This is referred to as the back-propagation formula. This setup is particularly computation-efficient, as one will compute the $\delta$ back from the end of the network to be used in computation for gradients earlier in the network.

## Loss Functions

In the previous section, we used the term loss function, $l(x_i, y_i; \theta)$, as some function we would like to optimize; here we will formalize the concept for various tasks. Assume we have a machine learning model, $g$, with parameters $\theta$, that takes in an input, $x$, and makes a prediction, $\hat{y}$.

For a regression problem, one of the most common loss functions is the squared difference,

$$l(x, y; \theta) = \|\hat{y} - y\|^2 = \|g(x; \theta) - y\|^2 \tag{1.22}$$

For a classification problem, the common loss function is the cross entropy,

$$H(p,q) = -\sum_x p(x) \log q(x) \tag{1.23}$$

where $p$ is the true distribution and $q$ is the predicted distribution.

Consider a multi-class classification with $K$ classes. Then the model outputs a predicted probability for each class, denoted $y_k$. Then the loss function will be

$$l(x,y;\theta) = -\sum_{k=0}^{K} [y=k] \cdot \log(y_k) \tag{1.24}$$

where $[\cdot]$ is the indicator function.

### 1.2.5 Practical Guide for Deep Learning

In this section we will review some of the common practices when training a neural network with gradient descent, as suggested in [12].

Input Preprocessing

- Input Standardization

  One of the commonly used preprocessing techniques is input standardization. The main idea is to have all the input features in a similar range, which leads to faster training convergence in practice.

  Assume that we are given $k$ dimensional input feature vectors denoted as $\mathbf{x} = [x_1, ..., x_k]^\intercal$ and $N$ data samples denoted as $D_N = \{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}\}$. The feature standardization is defined as

$$\mu_{x_i} = \frac{1}{N} \sum_{\mathbf{x} \in D_N} x_i \tag{1.25}$$

$$\sigma_{x_i} = \sqrt{\frac{1}{N} \sum_{\mathbf{x} \in D_N} (x_i - \mu_{x_i})^2} \tag{1.26}$$

$$\tilde{x}_i = \frac{x_i - \mu_{x_i}}{\sigma_{x_i}} \tag{1.27}$$

  where $\tilde{\mathbf{x}}$ is the standardized feature input.

- Principle component analysis (PCA) is also another popular preprocessing technique as motivated in [13]. The main motivation for using PCA is to decorrelate the input features, called principal components [14]. Below, we visualize how PCA works.

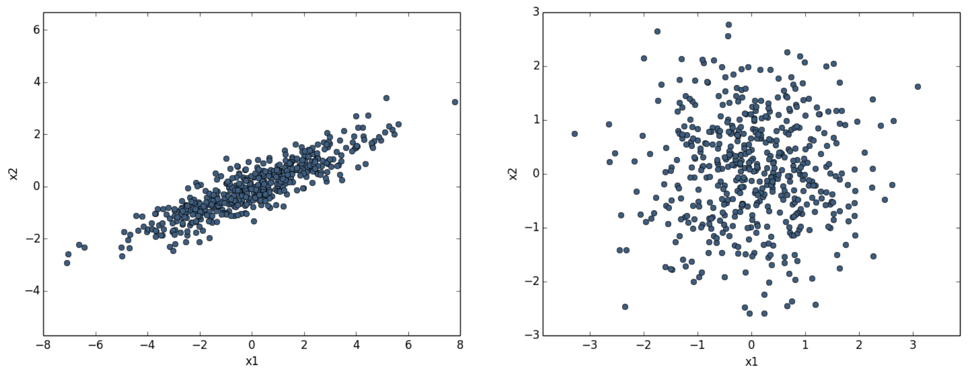

Figure 1.4: PCA visualization

In Figure 1.4 (a) is the original data, plotted on axes $x_1$ and $x_2$; as can be seen, the two dimensions are correlated. Figure 1.4 (b) shows the transform features after PCA. The two variables are now decorrelated. This is useful if we wish to reduce the feature dimensions.

Now we will formally go over how to apply PCA and how to use it for dimension reduction.

Given $N$ samples of $k$ dimension feature vectors, we can write this set as

$$X = \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(N)} \\ x_2^{(1)} & \dots & x_2^{(N)} \\ \vdots & \ddots & \vdots \\ x_k^{(1)} & \dots & x_k^{(N)} \end{bmatrix} \tag{1.28}$$

We hope to find a transformation $W$ such that

$$(WX)(WX)^{\intercal} = I \tag{1.29}$$

$$W(XX^{\intercal})W^{\intercal} = I \tag{1.30}$$

$$WCov(X)W^{\intercal} = I \tag{1.31}$$

As the $Cov(X)$ is symmetric, then we can factorize it into

$$Cov(X) = U\Lambda U^{\intercal} \tag{1.32}$$

where $U$ are the eigenvectors of $Cov(x)$ and $\Lambda = diag(\lambda_i)$, where $\lambda_i$ is the eigenvalue of the $i^{th}$ eigenvector.

As can be seen,

$$W = \Lambda^{-1/2} U^{\intercal} \tag{1.33}$$

is a solution to decorrelate the variables.

Next, for dimension reduction, we should keep the dimension starting from the largest eigenvalues, as these principal components preserve the most information from the original signal, in the least-squares sense. Proof is shown below.

Recall that we have $N$ examples $\mathbf{x} = (x_1, ...x_k)$, a $k$ dimensional vector, and we are mapping it to a lower dimension space of $m$, $\mathbf{z} = (z_1, ..., z_m)$, where $m < k$ (i.e. we only keep $m$ eigenvectors).

Next, we can represent the vector $\mathbf{x}$, zero-mean, as a linear combination of the orthonormal eigenvectors.

$$\mathbf{x} = \sum_{i=1}^{k} z_i \mathbf{u_i} \tag{1.34}$$

We define the approximation vector with only $m$ dimensions as

$$\tilde{\mathbf{x}} = \sum_{i=1}^{m} z_i \mathbf{u_i} \tag{1.35}$$

where $z_i = \mathbf{u_i}^{\intercal}\mathbf{x}$.

Then the sum of squares error is defined as

$$
\begin{aligned}
E &= \sum_{n=1}^{N} \left\| \mathbf{x^{(n)}} - \mathbf{\tilde{x}^{(n)}} \right\|^2 && (1.36)\\
&= \sum_{n=1}^{N} \sum_{i=m+1}^{d} \left\| z_i^{(n)} \mathbf{u_i} \right\|^2 && (1.37)\\
&= \sum_{n=1}^{N} \sum_{i=m+1}^{d} (z_i^{(n)})^2 && (1.38)\\
&= \sum_{n=1}^{N} \sum_{i=m+1}^{d} (\mathbf{u_i}^\intercal \mathbf{x}^{(n)})^2 && (1.39)\\
&= \sum_{i=m+1}^{d} \mathbf{u_i}^\intercal Cov(X) \mathbf{u_i} && (1.40)\\
&= \sum_{i=m+1}^{d} \lambda_i && (1.41)
\end{aligned}
$$

Here, $\mathbf{u_i}$ are the eigenvectors of the covariance matrix; thus, $Cov(x)\mathbf{u_i} = \lambda_i \mathbf{u_i}$. Hence, to minimize the error, the dimension with the smallest eigenvalues should be removed.

Hyperparameter

For a deep neural network to work properly, many hyperparameters have to be chosen properly. Below, we will review the tuning procedure for several of the common hyperparameters.

- Learning rate
  Learning rate, or step-size parameter, is crucial to the success of training a neural network. A very high learning rate leads to divergence, and a very lower learning rate leads to very slow convergence. A suggested in [12], choose a large learning rate; if it diverges, decrease the learning rate by a factor of 3 and repeat until a smooth learning curve is observed. Note that before starting to tune the learning rate, one should fix the batch-size. As a different batch-size will affect the learning rate, this is typically chosen to be 64. Also, another common practice is to have a learning rate schedule, where the learning rate is decreased de-

pending on the number of iterations; this heuristically leads to better convergences.

- Weights initialization
  The weights in the neural network need to be initialized carefully to avoid symmetry in the network, leading to all the weights being exactly the same. Typically, this is done by initializing with small random values. Work in [15, 16] suggested various useful initialization schemes. Next, biases are typically initialized with zeros. When using *Relu*, one should begin with an initialization that results in positive activation, to avoid all zero outputs after the *Relu* non-linearity.

- Hyperparameter searching
  Hyperparameters in neural networks are typical selected through tuning with a validation set. That is, one should try different hyperparameter configurations, and settle on the best based on the performance of the validation set. One typical method is doing grid search or random search [13], which are usually computationally expensive as they involve training multiple deep networks.

# CHAPTER 2

# STABLE CONVOLUTIONAL NEURAL NETWORK

## 2.1 CNNs from Signal Processing Perspective

In the previous sections, we have reviewed deep learning and CNNs from a machine learning point of view. In this section, we will present them from a signal processing and feature engineering perspective.

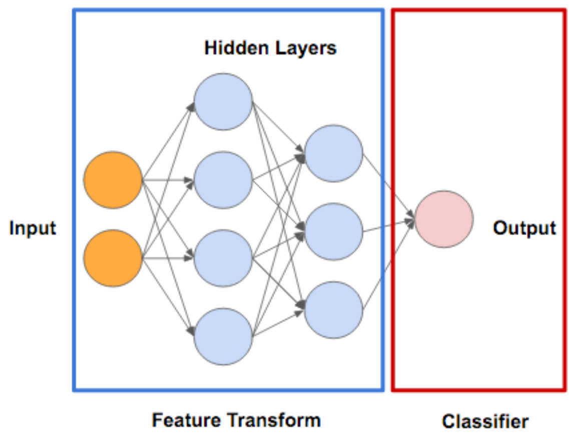

Figure 2.1: Neural network visualization

Instead of viewing a deep neural network as a single model, we can view it as two parts: (1) feature transformation/extraction, followed by (2) a simple linear classifier, as illustrated in Figure 2.1.

Consider the "toy" example shown in Figure 2.2 (a); we see that a linear classifier cannot perfectly separate the two classes, indicated by red and blue. However, if we take the square-root of $x1$ and $x2$, then the feature has been transformed into a space that is linearly separable, as seen in Figure 2.2 (b). Thus, if we can find a feature transform that maps from the input space to a space that is linearly separable, then all classification problems can be

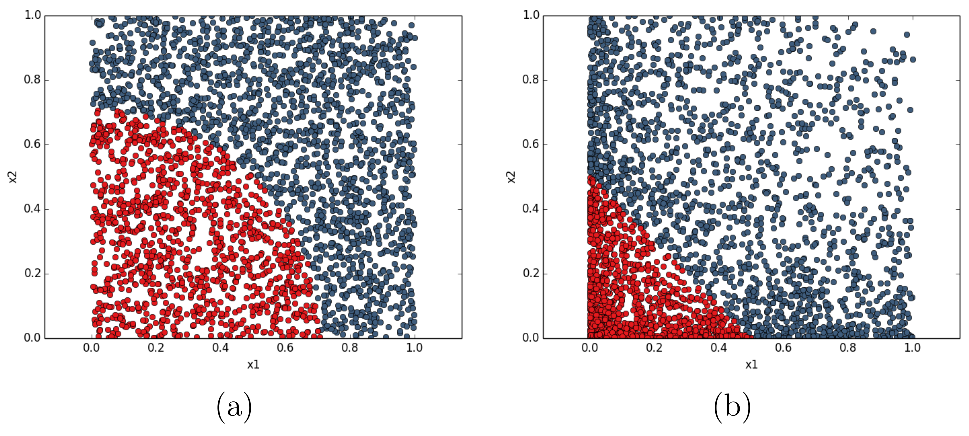

Figure 2.2: Feature transform illustration

solved with a linear classifier. The main difficulty is how to design this feature transformation. For the two-dimensional case, we can visualize the data, and if the pattern is simple then we can cleverly craft a transformation. However, for high-dimensional data it is much more difficult. Thus, designing these feature transformations is difficult and often takes expert domain knowledge (e.g., MFCC in speech, or SIFT features in images [17, 18]).

Deep learning takes a different approach; instead of having expert designing these features, we use data to optimize the feature extraction by having many layers of non-linear transformation. However, we now have less understanding of features we learned and why the features work, besides minimizing the loss.

There are many studies attempting to understand the effectiveness of these networks, both empirically by visualizing the activations, and theoretically through analyzing the network properties [19, 20, 21]. In particular, Bruna and Mallat addressed the scattering network [22], a specific type of CNN, in which the weights in the hidden layers are fixed wavelet coefficients. With this setup, they showed that the network creates a transformation that is translation-invariant and stable to deformation/additive noise. They matched state-of-the-art results in the MNIST dataset [23]. In the rest of the chapter, we will provide a proof that a traditional CNN is stable to additive noise.

## 2.2 Stability to Additive Noise Motivation

For a transformation, $\Phi$, to be stable to additive noise $x'(u) = x(u) + \epsilon(u)$, it needs a Lipschitz continuity condition as defined in [22]

$$\|\Phi x - \Phi x'\|_2 \leq C \cdot \|x - x'\|_2 \tag{2.1}$$

for a constant $C > 0$, and for all $x$ and $x'$. $\Phi x$ denotes the transformed feature.

The intuition is that when noise is added to the signal, the transformed feature is perturbed in a controlled manner.

## 2.3 Stability to Additive Noise in CNN

A standard CNN's forward operation is a combination of the following operations: (1) convolution with max-norm regularization, (2) element-wise Relu non-linearity, and (3) max-pooling. We will prove that the sequence of these operations can satisfy the Lipschitz continuity condition, as defined in Equation 2.1.

### 2.3.1 Stability of Convolution with max-norm Regularization

Denote the output of a convolution as $w \star x$. With $l_1$ max-norm regularization, the weights are renormalized to constant norm, $\kappa$. This means $\|w\|_1 \leq \kappa$. By Young's inequality for convolutions [24],

$$\|w \star x\|_2 \leq \|w\|_1 \cdot \|x\|_2 \leq \kappa \cdot \|x\|_2 \tag{2.2}$$

Then by linearity of convolution

$$\|w \star x - w \star x'\|_2 \leq \kappa \cdot \|x - x'\|_2 \tag{2.3}$$

If $x$ is multi-channel, the convolutional layer sums the convolution outputs of each channel. Then, by triangle inequality, Lipschitz continuity condition holds with $C = \kappa \cdot$ (number of channels).

### 2.3.2 Stability of Element-wise Relu non-linearity

Relu is an element-wise operation defined as

$$Relu(x_i) = max(0, x_i) \tag{2.4}$$

where $x_i$ denotes an element in the input signal $x$. Next, it can be verified that $|Relu(x_i) - Relu(x'_i)| \leq |x_i - x'_i|$, by considering the four cases with $x_i$ and $x'_i$ each positive or negative.

Case 1 $x_i > 0, x'_i > 0$ :

$$|relu(x_i) - relu(x'_i)| = |x_i - x'_i| \tag{2.5}$$

Case 2 $x_i > 0, x'_i < 0$ :

$$|relu(x_i) - relu(x'_i)| = |x_i - 0| < |x_i + |x'_i|| \tag{2.6}$$

Case 3 $x_i < 0, x'_i > 0$ :

$$|relu(x_i) - relu(x'_i)| = |relu(x'_i) - relu(x_i)| < |x'_i + |x_i|| \tag{2.7}$$

Case 4 $x_i < 0, x'_i < 0$ :

$$|relu(x_i) - relu(x'_i)| = 0 < |x_i - x'_i| \tag{2.8}$$

Therefore, $\|Relu(x) - Relu(x')\|_2 \leq \|x - x'\|_2$. Thus, satisfying the Lipschitz continuity condition holds with $C = 1$

### 2.3.3 Stability of Max-pooling

Max-pooling operation divides the input signal into a set of overlapping or non-overlapping windows, and for each window outputs the maximum value.

First consider the windows to be non-overlapping; then we only need to show that the max operation for each window, following a Relu operation, is Lipschitz continuous. Recall that the max-pooling operation follows the Relu operation, and therefore $x > 0$ and $x' > 0$, where $x$ and $x'$ denote the signals in each window.

Denote $i^* = \arg\max_i x_i$ and $j^* = \arg\max_j x'_j$. We claim that

$$|\max_i x_i - \max_j x'_j| \le \max(|x_{i^*} - x'_{i^*}|, |x_{j^*} - x'_{j^*}|) \tag{2.9}$$

Then,

$$\begin{aligned} |\max_i x_i - \max_j x'_j| &\le \max_i |x_i - x'_i| \\ &= \|x - x'\|_\infty \le \|x - x'\|_2 \end{aligned} \tag{2.10}$$

If the inequality is true, then the max-pooling operator satisfies the Lipschitz continuous condition. The inequality can be proved by considering the following two cases:

1.
$$x_{i^*} > x'_{j^*} \rightarrow |\max_i x_i - \max_j x'_j| = x_{i^*} - x'_{j^*} \le x_{i^*} - x'_{i^*} \tag{2.11}$$

   As $x$ and $x'$ are all greater than 0 and $x'_{j^*}$ is the largest in $x'$.

2.
$$x'_{j^*} > x_{i^*} \rightarrow |\max_i x_i - \max_j x'_j| = x'_{j^*} - x_{i^*} \le x'_{j^*} - x_{j^*} \tag{2.12}$$

Therefore, max-pooling operation with non-overlapping windows satisfies the Lipschitz continuity condition with $C = 1$

Next, consider overlapping windows, with $k$ overlaps, where $k$ is less than the window size. As the contribution of each overlapping term to the norm is less than or equal to $\|x - x'\|_2$, we can show that

$$\|maxPool_k(x) - maxPool_k(x')\|_2 \le (k+1) \cdot \|x - x'\|_2 \tag{2.13}$$

where $k$ is the number of overlapping elements in the pooling window.

### 2.3.4 Summary

In this section, we have shown that each of the operations (1) convolution with $l_1$ max-norm regularization, (2) *Relu*-non-linearity and (3) Max-pooling satisfies the Lipschitz continuity condition. As CNNs are stacks of these operators, they satisfy the Lipschitz continuity condition and thus are stable to additive noise.

# CHAPTER 3

# SYMMETRIC FILTER CONVOLUTIONAL NEURAL NETWORK

## 3.1 Symmetric Constraint and Motivation

The symmetric filter CNN is motivated by recent results showing that a scattering net, with weights set equal to wavelet coefficients and untrained, was able to reach state-of-the-art performance in handwritten digit recognition [22]. As wavelets have symmetric or antisymmetric structure, we speculate that the hypothesis space of the CNN model can be restricted to only symmetric and antisymmetric convolution layers. Let $W$ denote a weight (filter) coefficients centered at $(0, 0)$. By "antisymmetric" we mean $W(i, j) = -W(-i, -j)$ and by "symmetric" we mean $W(i, j) = W(-i, -j)$; as illustrated in Figure 3.1.

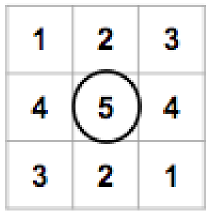

Figure 3.1: Symmetric filter illustration

Symmetric and antisymmetric filters with odd height and width have generalized linear phase.

Consider the symmetric case,

$$
\begin{aligned}
H(e^{j\omega}) &= \sum_{\mathbf{n}} W[\mathbf{n}]e^{-j\omega\mathbf{n}} && (3.1)\\
&= \sum_{\mathbf{n}>=\mathbf{0}} W[\mathbf{n}]e^{-j\omega\mathbf{n}} + W[-\mathbf{n}]e^{j\omega\mathbf{n}} && (3.2)\\
&= \sum_{\mathbf{n}>=\mathbf{0}} W[\mathbf{n}](e^{-j\omega\mathbf{n}} + e^{j\omega\mathbf{n}}) && (3.3)\\
&= \sum_{\mathbf{n}>=\mathbf{0}} W[\mathbf{n}]2\cos(\omega\mathbf{n}) && (3.4)
\end{aligned}
$$

As can be seen, the symmetric filter has linear phase. A similar derivation can be done for the anti-symmetric case.

This ensures that no phase distortion occurs at the convolutional layer; hence, the structure of the signal is maintained. This is a very common practice in filter design [25]. Furthermore, when enforcing this symmetric constraint, the number of parameters to train is reduced and the potential to accelerate training and decoding by using a symmetric convolution operator is gained, as convolution on symmetric filters requires half as many multiplications as convolution with arbitrary filters.

## Model Reasonableness

At first glance, a constraint of symmetric and antisymmetric filters with respect to the origin seems like a very strong condition. However, the overall model can represent approximately the same set of functions as a model that has symmetric or antisymmetric weights with respect to a certain point, not necessarily the origin; many of the learned CNN filters published as examples in image recognition papers have approximately this property, as do the many of the filters learned in our own baseline experiments. The reasoning is as follows: Denote the translation operator $T_{\vec{c}}$, such that $T_{\vec{c}}(W(\vec{x})) = W(\vec{x}-\vec{c})$. Then, for a filter symmetric to some point $\vec{c}$, we can translate the filter to be centered at the origin. From the translation invariance property of convolution, $T_{\vec{c}}(W) \star g = T_{\vec{c}}(W \star x)$, i.e., the output from the convolution layer is translated. Therefore, we see that centering the filter will result in a translated output and no loss of information.

*Relu* is an element-wise operator, and thus the output continues to be a translated version. For max-pooling of window size $N \times N$, if $\vec{c}$ happens to be a multiple of $N$, then the output from the centered model will again be the translated version of the non-centered model. On the other hand, if $\vec{c}$ is less than $N$, then as long as the max element does not move out of the max-pooling window, the output will be equivalent; hence the output is approximately the same as the one from the non-centered model. Lastly, the fully connected layers are not affected by reordering of the inputs, as reordering the weights in the same manner will give equivalent output.

## Gradient for Symmetric Convolutional Layer

In this section, we derive the gradient formula for the symmetric/antisymmetric convolutional layer, using back propagation notation of [7] reviewed in the background section. Denote the following:

- $J$ = overall loss function
- $\delta_j^{(l)}(u,v) = \frac{\partial J}{\partial z_j^{(l)}(u,v)}$ = backprop error

Recall that the forward convolutional operation without symmetric constraint can be defined as

$$z_j^{(l)} = (\sum_i a_i^{(l-1)} \star W_{ij}^{(l-1)}) \tag{3.5}$$

$$a_j^{(l)} = f(z_j^{(l)}) \tag{3.6}$$

Then the gradient of $J$ with respect to the filter weight is

$$\frac{\partial J}{\partial W_{ij}^{(l-1)}(u,v)} = \sum_{u'}\sum_{v'} \frac{\partial J}{\partial z_j^{(l)}(u',v')} \cdot \frac{\partial z_j^{(l)}(u',v')}{\partial W_{ij}^{(l-1)}(u,v)} \tag{3.7}$$

From equation 3.6, we can see that

$$\frac{\partial z_j^{(l)}(u',v')}{\partial W_{ij}^{(l-1)}(u,v)} = a_i^{(l-1)}(u'-u, v'-v) \tag{3.8}$$

as the gradient is non-zero when $\hat{u} = u$, and $\hat{v} = v$. Lastly, for the simplicity of indexing, let $\widetilde{W}_{ij}^{l-1}(u,v)$ be the weights of the symmetric convolution layer, constrained so $\tilde{W}_{ij}^{(l-1)}(-u,-v) = \tilde{W}_{ij}^{(l-1)}(u,v)$.

The gradient with respect to the symmetric filter weight can be written in terms of the gradient of the general convolution layer as follows:

$$\frac{\partial J}{\partial \widetilde{W}_{ij}^{(l-1)}(u,v)} = \sum_{u'}\sum_{v'} \frac{\partial J}{\partial z_j^{(l)}(u',v')} \cdot \frac{\partial z_j^{(l)}(u',v')}{\partial W_{ij}^{(l-1)}(u,v)} + \sum_{u'}\sum_{v'} \frac{\partial J}{\partial z_j^{(l)}(u',v')} \cdot \frac{\partial z_j^{(l)}(u',v')}{\partial W_{ij}^{(l-1)}(-u,-v)} \quad (3.9)$$

The gradient for the antisymmetric convolution layer can be derived similarly.

## 3.2 Experiments

### 3.2.1 Dataset and Experiment Setup

MNIST dataset

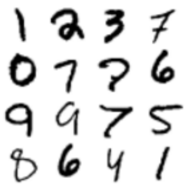

Figure 3.2: MNIST training examples

The MNIST database of hand-written digits contains 60,000 training samples and 10,000 test samples [23], some of which are visualized in Figure 3.2. We evaluated the model on different training sizes and report the results in Table 3.1. For each training size, we randomly sampled from the training set with a constraint that all digits occur the same number of times [22], which is to avoid very skewed distribution possibly resulting from random sampling. Also, no distortion of any kind was used to enhance the training data; preprocessing done to the data was normalization.

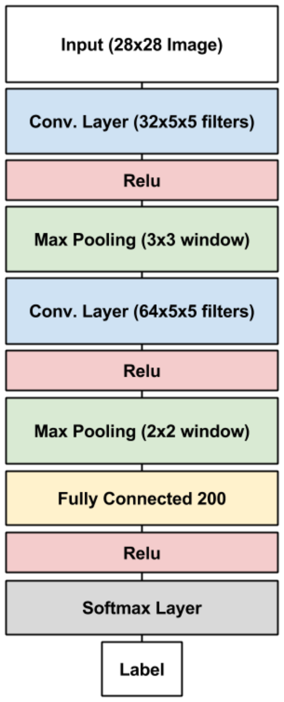

Figure 3.3: CNN architecture used in experiments

### 3.2.2 Model architecture

The model architecture we used is shown in Figure 3.3. In order to directly observe the effect of the proposed symmetric filter CNN model, we have chosen a simple network architecture modified from LeNet [8], using *Relu* as the non-linearity operation, and max-pooling with 2x2 non-overlapping windows. We used this model as the baseline benchmark. Experimental results for the comparison are shown in Table 3.1.

The symmetric filter CNN model follows the same architecture as the baseline described except that, at each convolutional layer, half of the filters are forced to be symmetric and the others to be antisymmetric. The model weights were randomly initialized. No pre-training or dropout was used.

### 3.2.3 Learning procedure

We trained our networks using stochastic gradient descent with momentum. We used 10,000 random examples from the training data as a holdout set for tuning hyperparameters; this includes learning rate, regularization parameters (max-norm and $l_2$ regularization), momentum and batch size. The

Table 3.1: Percentage of Errors on MNIST Test Set vs. Training Size

| Training Size | Sym-Conv. Net | Base-line. Conv. Net | State-of-art Conv. Net |
|---|---|---|---|
| 300 | **9.95** | *10.30* | 10.63 |
| 1,000 | **4.31** | *4.40* | 4.48 |
| 2,000 | 3.25 | *3.20* | **3.05** |
| 5,000 | *2.15* | 2.21 | **1.98** |
| 10,000 | 1.45 | *1.30* | **0.84** |
| 20,000 | *1.01* | 1.06 | **0.70** |
| 40,000 | *0.82* | 0.85 | **0.64** |
| 60,000 | *0.70* | 0.74 | **0.62** |

tuning procedure follows the suggested techniques mentioned in [12]. The identical tuning procedure is performed on the baseline model and the symmetric model, to control for the effect of tuning on the performance of the models.

### 3.2.4 Results and Discussions

Table 3.1 reports the results from the symmetric convolutional network, the baseline network without symmetric filters, and the state-of-the-art convolutional network model (5-layers) with no pre-training, no image distortion, and no other improvement techniques [26], which is a reasonable comparison to our model.

The symmetric convolutional network outperforms the baseline model for nearly all training sizes; the difference in error rates between the symmetric and the baseline models decreases as training set size increases.

For small training size (e.g. 300 and 1,000), both the symmetric and baseline models outperform the state-of-the-art deep-net in [26]. These results support the intuition that more complex models are more prone to overfitting, and simpler models perform better with limited training data.

Next, Figure 3.4 presents the weights visualization of the first convolutional layer from the trained symmetric CNN on size 20,000, where the first four rows are for symmetric filters, and the bottom four rows are for antisymmetric ones. These weights are matched with our signal processing intuition. Denote $(i, j)$ as the $i^{th}$ row $j^{th}$ column in Figure 3.4. Consider $(3, 4)$; the middle $3 \times 3$ pixels exactly resemble a high pass filter. Overall, the weights are

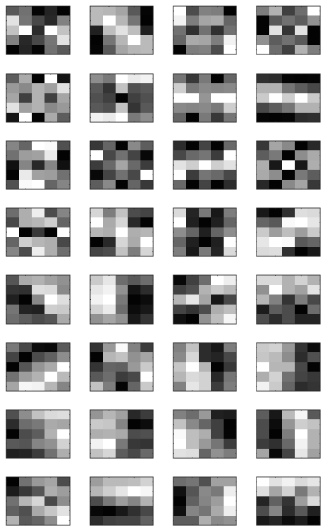
Figure 3.4: First convolutional layer weights visualization from symmetric CNN

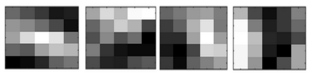
Figure 3.5: Selected first convolutional layer weights visualization from baseline CNN

very interpretable, i.e., they are all roughly edge detections in a particular direction, which is very reasonable as the edges of a digit are likely the most discriminant classification features. Furthermore, we compared the weights learned from the symmetric CNN and baseline CNN; we observed that some of the weights, Figure 3.5, are identical, but symmetric CNN has the weights centered at the origin (e.g. $(5, 1)$, $(5, 2)$).

Lastly, we have also examined models with only symmetrical filters and with only antisymmetric filters. Overall, the convolutional network with half antisymmetric filters and half symmetric filters outperforms the models with only antisymmetric filters or with only symmetric filters. Furthermore, the network with only antisymmetric filters outperforms the one with only symmetric filters. These findings lead to the conclusion that antisymmetric filters are important for correctly identifying the digits, but antisymmetric filters are not sufficient without the complementary information provided by symmetric filters.

# CHAPTER 4

# CONCLUSION

## 4.1 Stable Convolutional Neural Network

We present a proof that CNNs with max-norm regularization, *Relu* non-linearity, and max-pooling are stable to additive noise. This proof provides a reasonable explanation for why the CNN feature transform works in practice, and a mathematical guarantee on stability of the extracted feature.

## 4.2 Symmetric Filter Convolutional Neural Network

We investigate the use of symmetric and antisymmetric filters in CNN model on the MNIST dataset. State-of-the-art results were achieved for handwritten digit classification in the cases of very small training sizes. We also show that the network with symmetric and antisymmetric filters is generally better than the baseline benchmark model. Lastly, we analyzed the model weights and verified our understanding that the set of functions that the symmetric models have learned are empirically similar to those of the baseline model.

## 4.3 Summary and Future Work

Deep learning has demonstrated strong empirical performance in image and speech recognition, and other applications; however, very often these are "black-box" models, where we do not have strong understanding of the internal workings of the learned classifier. We believe that understanding the effectiveness of the model could lead to future architectural improvements. In particular, we hope to incorporate traditional signal processing tools to have interpretable deep models with strong performance.

# REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos et al., “Deep speech 2: End-to-end speech recognition in english and mandarin,” *arXiv preprint arXiv:1512.02595*, 2015.

[3] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] P. Liang, “Statistical learning theory lecture notes,” Stanford University, Winter 2016.

[5] V. Vapnik, *The Nature of Statistical Learning Theory.* Springer Science & Business Media, 2013.

[6] K. Hornik, “Approximation capabilities of multilayer feedforward networks,” *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[7] C. M. Bishop, *Neural Networks for Pattern Recognition.* New York, NY, USA: Oxford University Press, Inc., 1995.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[9] D. Scherer, A. Müller, and S. Behnke, “Evaluation of pooling operations in convolutional architectures for object recognition,” in *Artificial Neural Networks–ICANN 2010.* Springer, 2010, pp. 92–101.

[10] I. Goodfellow, Y. Bengio, and A. Courville, “Deep learning,” 2016, book in preparation for MIT Press. [Online]. Available: http://goodfeli.github.io/dlbook/

[11] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural Networks: Tricks of the Trade.* Springer, 2012, pp. 9–48.

[12] Y. Bengio, "Practical recommendations for gradient - based training of deep architectures," in *Neural Networks: Tricks of the Trade.* Springer, 2012, pp. 437–478.

[13] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, 2012.

[14] I. Jolliffe, *Principal Component Analysis.* Wiley Online Library, 2002.

[15] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249–256.

[16] G. Hinton, "A practical guide to training restricted Boltzmann machines," *Momentum*, vol. 9, no. 1, p. 926, 2010.

[17] S. B. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 28, no. 4, pp. 357–366, 1980.

[18] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer Vision, 1999, The Proceedings of the Seventh IEEE International Conference on*, vol. 2. IEEE, 1999, pp. 1150–1157.

[19] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision–ECCV 2014.* Springer, 2014, pp. 818–833.

[20] I. Goodfellow, H. Lee, Q. V. Le, A. Saxe, and A. Y. Ng, "Measuring invariances in deep networks," in *Advances in Neural Information Processing Systems*, 2009, pp. 646–654.

[21] I. J. Goodfellow and O. Vinyals, "Qualitatively characterizing neural network optimization problems," *arXiv preprint arXiv:1412.6544*, 2014.

[22] J. Bruna and S. Mallat, "Invariant scattering convolution networks," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1872–1886, 2013.

[23] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," 1998.

[24] W. Beckner, "Inequalities in Fourier analysis," *Annals of Mathematics*, vol. 102, pp. 159–182, 1975.

[25] J. Kovacevic, V. Goyal, and M. Vetterli, in *Fourier and Wavelet Signal Processing.* Cambridge University Press, 2014.

[26] M. A. Ranzato, F. J. Huang, Y.-L. Boureau, and Y. LeCun, "Unsupervised learning of invariant feature hierarchies with applications to object recognition," *Computer Vision and Pattern Recognition (CVPR), 2007. IEEE Conference on*, pp. 1–8, 2007.