İLKER ÇAM

M.S. Thesis

2019

LEARNING FILTER SCALE AND ORIENTATION IN CONVOLUTIONAL NEURAL NETWORKS

İLKER ÇAM

IŞIK UNIVERSITY

2019

LEARNING FILTER SCALE AND ORIENTATION IN CONVOLUTIONAL NEURAL NETWORKS

İLKER ÇAM
B.S., Computer Engineering, IŞIK UNIVERSITY, 2019

Submitted to the Graduate School of Science and Engineering
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Engineering

IŞIK UNIVERSITY
2019

IŞIK UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

LEARNING FILTER SCALE AND ORIENTATION IN CONVOLUTIONAL NEURAL NETWORKS

İLKER ÇAM

APPROVED BY:

Asst. Prof. F. BORAY TEK (Thesis Supervisor)	Işık University	________________
Prof. Dr. OLCAY TANER YILDIZ	Işık University	________________
Prof. Dr. FİKRET S. GÜRGEN	Işık University	________________

APPROVAL DATE: ..../..../....

LEARNING FILTER SCALE AND ORIENTATION IN CONVOLUTIONAL NEURAL NETWORKS

Abstract

Convolutional neural networks have many hyper-parameters such as filter size, number of filters, and pooling size, which require manual tuning. Though deep stacked structures are able to create multi-scale and hierarchical representations, manually fixed filter sizes limit the scale of representations that can be learned in a single convolutional layer. Can we adaptively learn to scale the filters on training time?

Proposed adaptive filter model can learn the scale and orientation parameters of filters using backpropagation. Therefore, in a single convolution layer, we can create filters of different scale and orientation that can adapt to small or large features and objects. The proposed model uses a relatively large base size (grid) for filters. In the grid, a differentiable function acts as an envelope for the filters. The envelope function guides effective filter scale and shape/orientation by masking the filter weights before the convolution. Therefore, only the weights in the envelope are updated during training.

In this work, we employed a multivariate (2D) Gaussian as the envelope function and showed that it can grow, shrink, or rotate by updating its covariance matrix during backpropagation training. We tested the model with its basic settings to show the collaboration of weight matrix with envelope function is possible. A deeper architecture was used to show the performance on deeper and wider networks. We tested the new filter model on MNIST, MNIST-cluttered, and CIFAR-10 datasets. Compared the results with the networks that used conventional convolution layers. The results demonstrate that the new model can effectively learn and produce filters of different scales and orientations in a single layer. Moreover, the experiments show that the adaptive convolution layers perform equally; or better, especially when data includes objects of varying scale and noisy backgrounds.

Keywords: Adaptive CNN, filter learning, filter scaling

EVRİŞİMSEL SİNİR AĞLARINDA FİLTRE ÖLÇEĞİ VE ORYANTASYONUNUN ÖĞRENİLMESİ

Özet

Evrişimsel sinir ağlarında filtre boyutu, sayısı ve ortaklama boyutu elle seçilmektedir. Derin katmanlı sinir ağları hiyerarşik çok ölçekli temsiller öğrenebilmesine rağmen, sabit filtre boyutları farklı ölçekteki öğrenilebilecek filtre sayısını sınırlamaktadır. Aynı katmanda farklı ölçeklerde filtreleri eğitim aşamasında öğrenebilen bir mimari olabilir mi?

Önerilen filtre modelimizde filtre ölçek ve oryantasyonları geriye yayılım ile öğrenilebilir. Bu şekilde, aynı evrişimsel katmanda farklı ölçek ve oryantasonlarla büyük ve küçük objeleri tanımlayabiliriz. Önerilen model, nispeten büyük filtre (ızgara) boyutlarına sahiptir. Türevi olan bir çevreleyici fonksiyon ile filtrelerin efektif ölçeklerini ve oryantasyonlarını, evrişim işlemine girmeden, katsayı matrislerini maskeleyebiliriz. Bu sayede, sadece çevreleyici fonksiyon içerisindeki katsayılar eğitilecektir.

Bu çalışmamızda, çok değişkenli (2 Boyutlu) Gaussian fonksiyonunu çevreleyici fonksiyon olarak kullandık. Kovaryans matrisinin geriye yayılım yöntemiyle eğitilmesiyle, çevreyeliyici fonksiyonun büyüyüp, küçüldüğünü ve dönebildiğini gösterdik. Çevreliyici fonksiyonun eğitilebildiğini ve katsayılarla işbirliğini, modelin en basit haliyle deneyimledik. Derin katmanlardaki performansını, derin ve geniş mimariler üzerinde çalıştırdık ve performansını izledik. Önerilen modeli, MNIST, MNIST-cluttered ve CIFAR-10 veri kümelerinde çaıştırdık ve geleneksel evrişimsel sinir ağ mimarilerindeki çalışma performanslarıyla karşılaştırdık. Sonuçlar, önerdiğimiz modelin, farklı ölçek ve oryanyasyonlarda, aynı katmanda, filtreler öğrenebildiğini gösterdi. Ayrıca, deneylerimiz, adaptif evrişimsel katmanının aynı, özellikle veri kümesinde farklı ölçeklerde obje ve gürültülü arkaplan içeren veri kümelerinde daha iyi çalıştığını gösterdik.

Anahtar kelimeler: Adaptif CNN, filtre öğrenmesi, ölçeklendirilebilir filtre

Acknowledgements

I would like to express my gratitude to my advisor Asst. Prof. F. Boray Tek for his continuous support of my study. His guidance helped me to develop my research, writing and coding skills. I could not have imagined having a better advisor and mentor for my MSc study.

I would not forget to remember Hattori Hanzō and Ichigo Kurosaki for their encouragement.

And finally, last but by no means least, I am thankful to and fortunate enough to get constant encouragement, support and guidance from Diliara Umiarova, Esra Çam and Koral El.

I dedicate this work to my mother; De familia mihi vires.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CNN	**C**onvolutional **N**eural **N**etwork
ACNN	**A**daptive **C**onvolutional **N**eural **N**etwork
ReLu	**Re**ctified **L**inear **U**nit

Chapter 1

Introduction

Naming or describing real-life objects is only meaningful with respect to a relevant scale [1]. For instance, a view can be described as a leaf, branch, or a tree depending on the distance of the observer. Natural and casual scenes are generally composed of many different entities/objects at various scales. During image acquisition, the true physical scale is usually ignored. However, the relative scale of the objects is somehow implicitly captured and stored in the image grid and pixels.

An automated method to identify or describe objects in images can be analyzed in two parts: representation + classification. Basic algorithms without add-ons cannot successfully handle variation and complexity of raw pixel-level representation of objects, instead, they rely on functions that map image pixels into different constructs, -named features-, which are seeking to represent the image content more briefly and invariant to various geometric and intensity changes.

Although recent studies stack different sizes of convolution kernels in a single layer, these are highly engineered hyper-parameters. Existing convolutional neural networks use fixed kernel sizes. Can we introduce a trainable parameter to set kernel size (scale) on run-time? It is known that sparse connectivity has no negative effect in terms of computation. Can large kernels with limited but growable effective area automatize this scale? How will having different scaled filters affect the overall performance? Our study tries to answer these questions with an adaptive model of the convolution layer where the filter size (actually scale) and orientation are learned during training. Therefore, a single convolution layer can have distinct filter scales and orientations. Broadly speaking, this corresponds to extracting multi-scale information using a single convolution layer. However, our aim is not to fully replace the stacked architectures and deep networks for multi-scale information. Instead, our approach improves the information that can be extracted from an input (may be an image), in a single layer. Additionally, the model removes the necessity of fixing convolution kernel sizes, so that the filter size can be removed from the list of hyper-parameters of deep learning networks.

This study will introduce the related work behind the proposed solution. A Literature survey will explain the basics of classification, machine learning, and convolutional neural networks. The method will introduce the implementation details of the proposed solution. Lastly, we will support our study with various experiments on filters such as autoencoder, performance on deeper layers and how the scale and orientation are changed during the training.

1.1 Related Work

Traditionally, computer vision researchers relied on manually designed feature extractors for representation. Recently, we are witnessing the success of algorithms which can self-learn appropriate feature extractors. In either case, the size of an operator or a probe usually determines and fixes the scale of the entities that can be represented. However, even in the self-learn case, the size of the probes or operators is often manually selected.

On the other hand, the last two decades have seen many automated object detection/recognition algorithms that were superior to their counterparts because they have comprised multi-scale processing of images [2], [3]. Multi-scale feature extractors gather and present the inherent scale information of image pixels to a subsequent classifier. In SIFT [4] and wavelets [5], this is done by creating a multi-scale pyramid from the input image and then applying a fixed size probe-kernel to each scale. In an application of Gabor filters for object recognition, Serre et al. [6] used a hierarchy of stacked Gabor filtering layers, where the filters have predetermined scales and orientations. However, Chan et al. [7] showed that the adaptation of handcrafted filters to low-level representations is difficult. On the other hand, convolution neural networks (CNN) rely on stacked and hierarchical convolutions of the image to extract multi-scale information. Convention of CNNs for filter size selection is to use small fixed size weight kernels in the lower levels. However, thanks to the stacked operation of convolutional layers, sandwiched by pooling layers which down-sample intermediate outputs, the deeper levels of a network are able to learn representations of larger scales.

Though the optimality of fixed size kernels has not proven, the convention is to use filters small as 3×3 in the first layer, which can be larger 5×5 or 7×7 in the later stages [8]. During backpropagation training, filters are evolved to imitate lower level receptive fields in biological vision which are sensitive to certain shapes and orientations. Another justification for avoiding large filter sizes is that, while certainly increasing computation time, they may also increase over-fitting.

Nevertheless, the number of filters and their sizes in convolution layers are usually selected intuitively, researchers are seeking alternatives to improve representation capacity of the network in deeper architectures. For example, Szegedy et al. [9] handcrafted their "inception’" architecture to include mixing of parallel and wide convolution layers which use differently sized filter kernels. In a deep architecture, this approach allows multi-scale, parallel and sparse representations. In summary, existing CNN based methods use fixed size convolution kernels and then rely on the fact that the shape and orientation of the filters can be inferred from the training data. Additionally, CNNs employ stacked convolution layers to successfully create multi-scale representations.

On the other hand, Hubel and Wiesel [10] discovered three types of cells in visual cortex: simple, complex, hyper-complex (i.e. end-stopped cells). The simple cells are sensitive to the orientation of the excitatory input, whereas the hyper-complex cells are activated with a certain orientation, motion, and *length* of the stimuli. Therefore, it is biologically plausible to assume that filters of different scales next to orientations and direction may also work better in CNNs.

Chapter 2

Literature Survey

2.1 Machine Learning

Computer-aided systems can solve many tasks. Each of them is designed for the purpose of solving a problem with predefined procedures. Set of these creates algorithms. An algorithm can be finding the largest element in a list, or prime factors of a number, and many more. Complex mathematical problems can be easily solved via algorithms no matter how complex we think they are. Once we define all the steps carefully, an algorithm can calculate an output in seconds. Yet how can we classify a happy person from a sad one? One can heuristically create an algorithm for a picture which does some operations on the image. Although, the position of a person can change and writing lots of algorithms for different poses is virtually impossible. One of the problems with our modern world is spams. We need to manually read and understand what information is vital for us. Therefore the question arises, what kind of algorithm will classify an important e-mail among lots of spams?

Machine learning algorithms try to transform the given input such that output can be a class or decision (i.e happy or sad person in the image.). The transformation between input and output usually depends on repeated steps which are mostly predictable. These repetitions are called patterns. Nature is based on symmetries, trees, leave tissue, waves and many more. Even though it's impossible to exactly model these patterns, computer-aided algorithms can approximate them.

2.1.1 Classification

Classification algorithms solve the problem of assigning correct labels of given input data. For instance, given tumor size data, classifying it as malignant or not. Assuming historical data on tumor sizes exists, we can approximate whether a given tumor is malignant or not. Confidence in this classification relies on how descriptive the features are and how big the training sample set is. Formally; Classification is a process of approximating a continuous function f using input variables X which outputs a discrete output variable(s) y. Depending on the training inputs, estimating new observations category is identified as classification.

There are many classification algorithms available to solve the problem. Depending on the problem and feature space these can be solved via; linear (i.e Logistic Regression, Naive Bayes Classifier, Perceptron etc.) and non-linear (Multi-layer Perceptron, K-Nearest Neighbor etc.) methods. The example given above which classifies an input with few features may be separated using a combination of linear separators (Hyper-planes). Projecting the input space to non-linear and solving the problem linearly as illustrated on figure 2.1

Linear classifiers are considered as baseline solutions for many problems. They have relatively fewer parameters than non-linear classifiers, thus over-fitting (an approximation method having close parameters to training data set) is less likely to happen. Figure 2.2(a) shows an example of two features of two classes separated via plane. Such example is a linearly-separable problem. Although the line may separate classes, there are infinity many possible lines to draw.

Real world problems usually have higher dimensional features or lower dimensional representations of raw data (i.e PCA [12]). Feature space of these problems

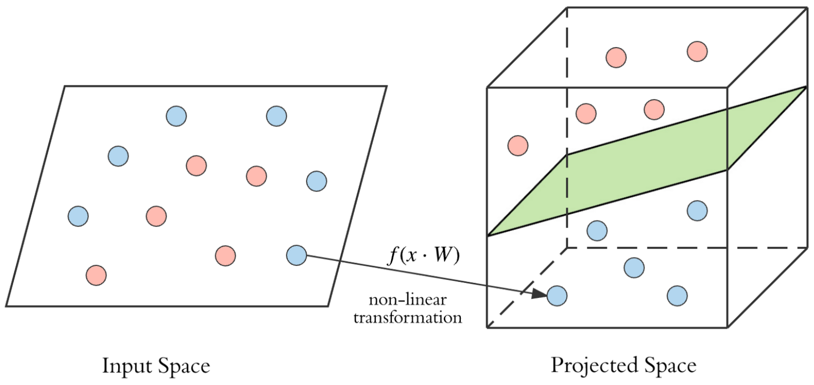


Figure 2.1: An input space is projected to a non-linear space. Figure taken from [11]

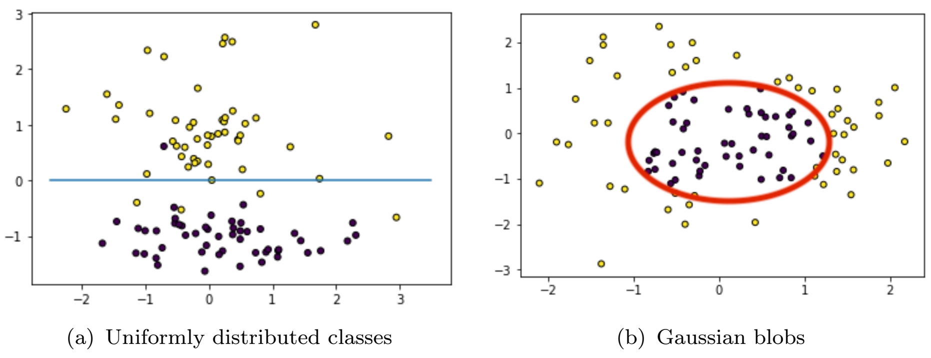


Figure 2.2: a) Randomly generated feature set of two classes. A linear classifier places a separating line. There are infinitely many lines separating these two classes. b) Data is not linearly separable.

might not be linearly separable. In other words, there is no linear line to separate classes. **Non-linear classifiers** can divide classes into locally linear segments, but in overall structure, these decision boundaries have complex shapes that have no linear shapes.

2.1.2 Neural Networks

The aim of a neural network is to learn a function f which maps the given input x to a desired output y. $y = f(x)$. It is done by learning set of parameters σ, $y = f(x; \sigma)$. As illustrated in figure 2.3, neural networks consist of an input layer, an output layer and stacked hidden layers. Each hidden layer has multiple neurons which consist of n trainable parameters w (weight). Every neuron j

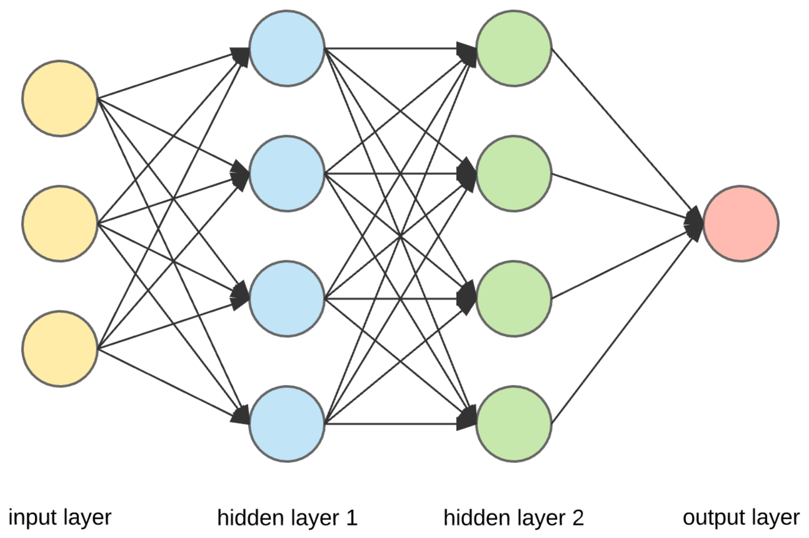


Figure 2.3: A Neural Network model. Consists of one input, two hidden and one output layer.

receives all inputs from the previous layer and calculates a weighted activation a_j. A neuron's mathematical calculation can be written as:

$$a_j = f\left(\sum_{i=1}^{m} w_{ij}x_i + w_{0j}\right)$$

Weights (w_{ij} and w_{0j}) are the parameter space of neuron j and constructs the solution space. In feed forward process, the neuron collects activation inputs from m other neurons and calculates a_j via activation function f. The function is usually chosen to be a non-linear function such as sigmoid. An error function calculates the error and back-propagates through the network.

2.1.2.1 Back-propagation

Back-propagation calculates the partial gradients of an error function E (cost function) for each neuron with respect to weights, biases and changes the weights by η steps on opposite direction on the gradient. This process can be written as:

$$E = \sum_{j=1}^{n} \frac{1}{2}(t_j - a_j)^2 \qquad (2.1)$$

$$\frac{\partial E}{\partial a_j} = (a_j - t_j) \qquad (2.2)$$

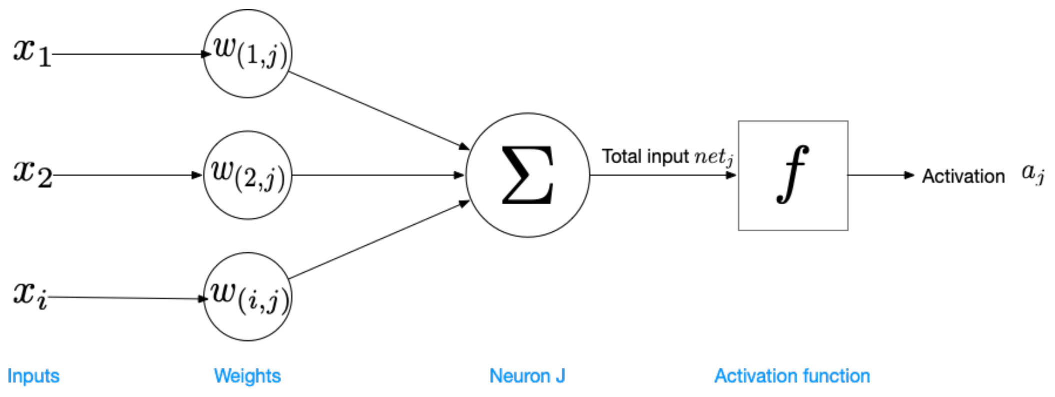


Figure 2.4: Artificial neuron model.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j} * \frac{\partial a_j}{\partial net_j} * \frac{\partial net_j}{\partial w_{ij}} \tag{2.3}$$

$$w_{ij} -= \eta * \frac{\partial E}{\partial w_{ij}} \tag{2.4}$$

2.1.3 Auto-encoders

A neural network is a supervised learning algorithm where input x has a target (label) t. Auto-encoders are a type of unsupervised learning algorithm trained to learn a function f to reconstruct x with a parameter set σ. As illustrated in figure 2.5 the network consists of two elements; a) Encoder: Takes the input and projects it to a higher or lower dimension. b) Decoder: Projects the encoded space back to the original input. Encoding step can learn highly descriptive features from the input. Having $m > k$ units will transform the input to a lower dimensional space and learned features will be highly descriptive whereas having $k > m$ units will transform the input to sparse space and hidden descriptive features might be learned in the space.

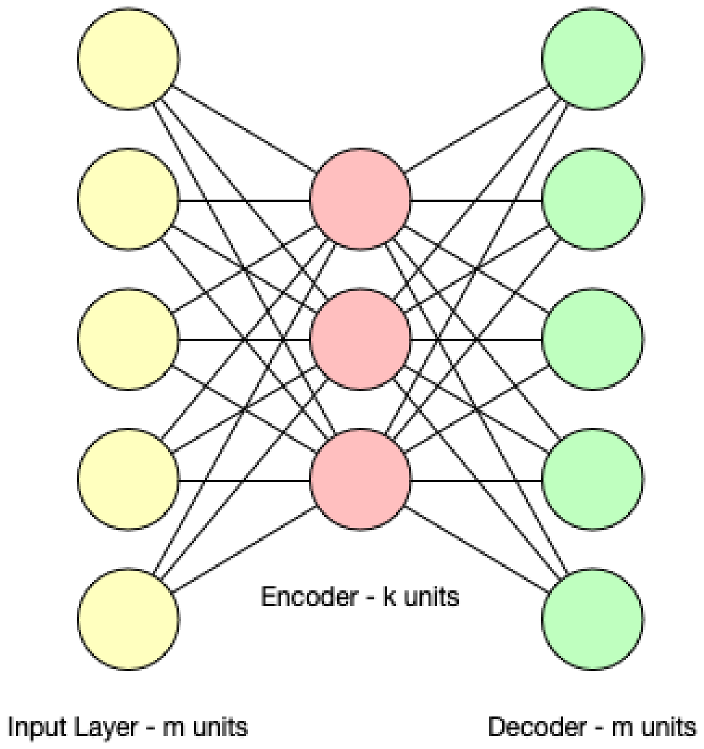


Figure 2.5: An auto encoder model. First hidden layer is the encoder layer. Encoder layer transforms the input to higher or lower dimensional space. Decoder layer reconstructs an approximation of the input.

2.1.4 Convolutional Neural Networks

CNN (Convolutional neural networks) are a type of artificial neural network popularly used in solving problems on the image domain. They can also be used for other data analysis and classification problems as well. Conventional neural networks have hidden layers which are fully connected to the incoming layer. A single neuron has connections to all inputs. Assuming an input of $32 \times 32 \times 3$, a fully connected layers every neuron has $3,072$ $(32 \times 32 \times 3)$ weights. While inputs such as images have large amounts of pixels, fully connected layers will have thousands or millions of parameters in one single layer. Such numbers of parameters require high computation power and could easily lead the model to overfit. CNN's, leverage three concepts that power the overall system; Parameter sharing, sparse connectivity, translation invariance.

CNN's, introduce **parameter sharing** by connecting on a small region of the

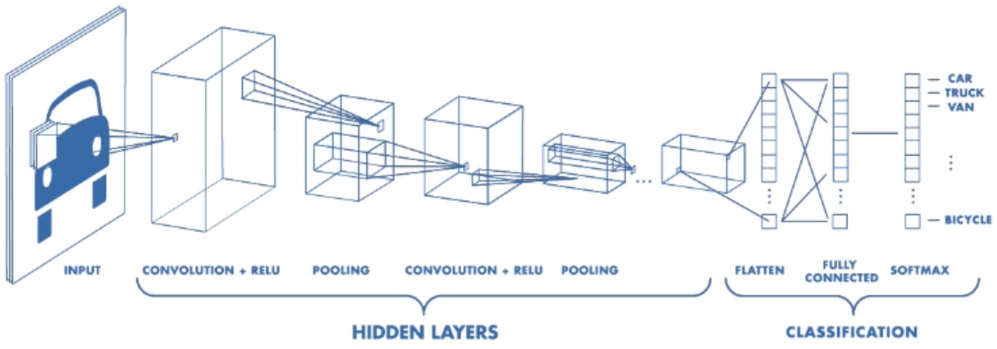


Figure 2.6: A Convolutional Neural Network architecture solving a classification problem. Typical CNN consists of convolutional, non-linearity and a pooling layer.

input, thus requiring less computational power and less prone to overfitting. Single kernel (weight matrix) of size 5×5 will only have 75 ($5 \times 5 \times 3$) parameters. CNN's has a special type of layers "Convolutional Layer" and has local connectivity to the previous layer. Local connections (by making the kernel smaller than the input) lead to **sparse connectivity**. The learned filter will only activate on certain patterns and output of a convolution will be sparse. Assuming a input with size $m \times n$, the total parameter calculation will run on $O(n \times m)$. Whereas, sparsely connected neurons k will only require $O(n \times k)$ calculations on run time.

Translation invariance is one of the most important aspects of CNN's. A convolution layer followed by a pooling layer will approximate to translation invariance. A filter can still detect a pattern if the image is shifted to a different position. For instance: An edge filter is detecting the edges of an input image. The edge can be on the upper left or right corner of the image. Same representation will still be detected but activation will be on different iterations. Pooling also provides translation invariance. For instance, max pooling operation will still get the largest value within its receptive field. So convolution followed by a pooling operation is approximating translation invariance to CNN's.

Figure 2.6 shows a Convolutional Neural Network architecture having stacked convolutional, non-linearity and pooling layers. We will discuss their roles in a CNN on the following sections.

2.1.4.1 Convolutional Layer

Given input with size $m \times m$ and kernel size $k \times k$, $m > k$. The convolutional layer consists of multiple kernels and performs the operation "convolution" to the input using kernels (weight matrices) and outputs feature maps. As illustrated in figure 2.7 every kernel slides over the input and performs a dot product. A convolutional layer has the following hyper-parameters:

- **Number of filters**: A Convolutional layer consists of n filters[1] every filter is convolved on the input feature map. Typical choices are, $32, 64, 128$
- **Filter size** k: Determines the receptive field size.
- **Stride**: After every dot product, the kernel is slid by one unit. Changing stride to 2 will lead to sliding two units thus will less overlap between neurons.
- **Padding**: As shown on figure 2.7 convolving a $5x5$ input with $3x3$ kernels outputs $3x3$ feature map. This might lead to losing in dimension over stacking many layers. Padding will surround the input with zero-valued pixel so that the input feature map will grow on the desired size. For instance, using 2 zero padding on $5x5$ input will pad the image to $7x7$.

2.1.4.2 Non-linearity Layer

Convolution operations produce linear activations. To introduce non-linearity CNN's activate the outputs of convolution with a non-linear function such as Rectified Linear Unit (ReLu). Formally, ReLu calculates output y_i for input x_i:

[1]Also known as kernels or weight matrices

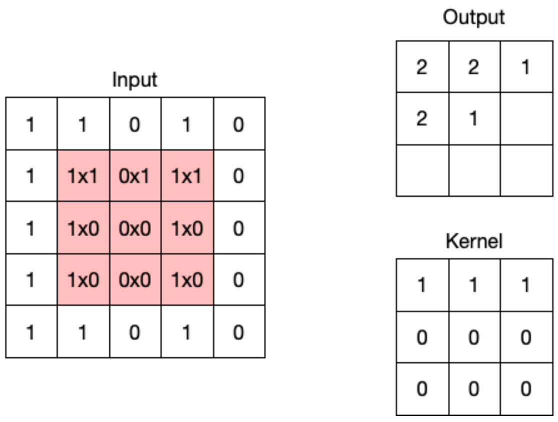


Figure 2.7: A not finished convolution operation on $5x5$ input with $3x3$ kernel. Using no zero padding will calculate an output feature map with $3x3$ dimensions.

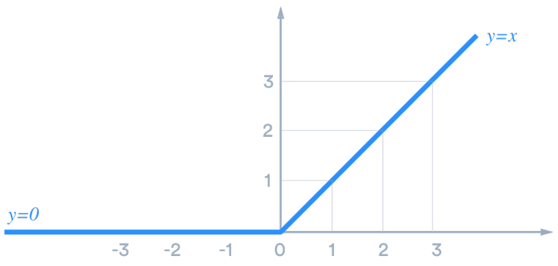


Figure 2.8: Rectified Linear Unit activation function. All negative values are transformed to zero and positive values do not saturate.

$y_i = max(0, x_i)$. Although it transforms the input from linear to non-linear space, the function remains very close to a piece-wise linear function. ReLu is widely used among researches over other activation functions (tanh, sigmoid) while their easy optimization on linear models. As shown on figure 2.8, ReLu, converts all negative values to zero thus non-positive activation will not activate a neuron.

Choosing an activation function for a deep architecture is crucial. Gradient-based learning algorithms depend on increasing the performance by changes in parameters. Small gradients may lead to very little change in parameters thus performing poorly. Vanishing gradient problem appears when the activation function squashes the activations. A popular solution is to use ReLu, while there is no saturation on positive ranges.

2.1.4.3 Pooling Layer

Pooling layers perform downsampling to input feature maps. There are a variety of pooling layers such as max-pooling and average-pooling. For instance, a max-pooling operation with size 2×2 iterates over the input with same stride size (2 in this example) and calculates the maximum value of the area. Whereas the average pool does the same operation but calculates the average. This operation has two important roles in the CNN stack. **Translation invariance**: A pattern might be activated on a different location of the image. Pooling will ensure that the maximum activation[2] will be transformed to the next layer. **Dimension reduction**: Input with size $8 \times 8 \times 10$ will be reduces to $4 \times 4 \times 10$ which is 75% less than the input.

2.1.5 Related Adaptive Neural Networks

Classic convolutional neural network stacks have typical structures as convolutional layer followed by spatial pooling layer and stacked them together. GoogleNet stacks [9] different sizes of convolution kernels on one single output. Figure 2.9 illustrates the inception module used in GoogleNet architecture. 1×1, 3×3, 5×5 convolutions and 3×3 max pooling are done on the previous layer and outputs are stacked all together as one single layer. Convolving the input with 1×1 kernels are inspired by the study "Network In Network" [13]. Authors state that linear convolutions are sufficient for abstracting when data is linearly separable. However, meaningful abstractions are usually non-linear representations of the input data. Thus, they use 1×1 convolutions to introduce more non-linearity and increase the representational meaning of the data. However, in GoogleNet's 1×1 convolutions are used as dimension reduction hence reduce the computation bottleneck.

[2]On Max-pool

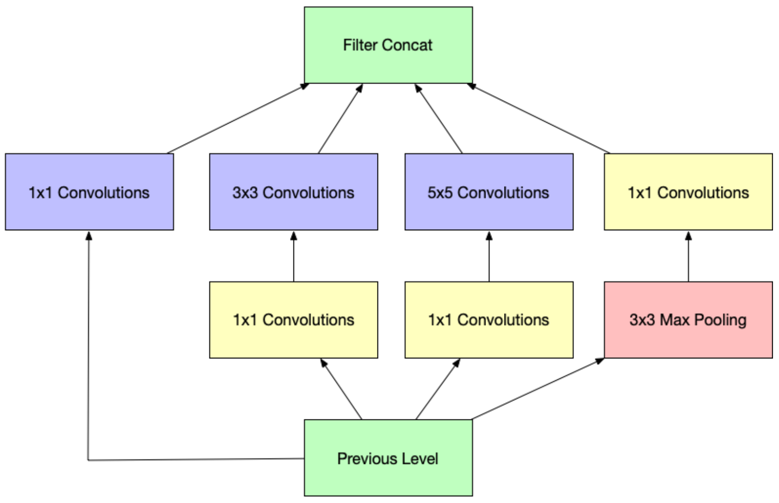


Figure 2.9: Inception module from GoogleNet

Focusing neuron [14] presents an artificial neuron learning its receptive field. Both field size and location are learned during training. Authors used a Gaussian function to change the field size and location. The synthetic Gaussian blob experiments show that focus location can steer out from the redundant inputs and focus on descriptive inputs.

Chapter 3

Method

In deep network architectures, stacked convolution layers perform convolutions with fixed size kernels where sandwiched pooling layers perform down-sampling operations to achieve a multi-scale and hierarchical representation. Fixed size convolution kernels put a limit on the scale of features which can be extracted from a single layer.

Though it is possible to mix several kernels of different size in a single convolution layer, the convention is to use a fixed size for all the kernels in a layer. Here, we introduce a new filter model which can adapt its scale and orientation. Therefore, it allows the development of multi-scale and differently oriented filters in a single convolution layer. To realize this, we need a smooth function that can grow, shrink, or rotate during training which acts as an envelope to guide filter scale and orientation. The following sections explain the role of an envelope, selecting an appropriate envelope function and its partial derivatives which are used in back-propagation.

3.1 Envelope Function

The role of the envelope function is to guide the filter scale and orientation development. As illustrated in figure 3.1, a base grid acts as the envelope and filter

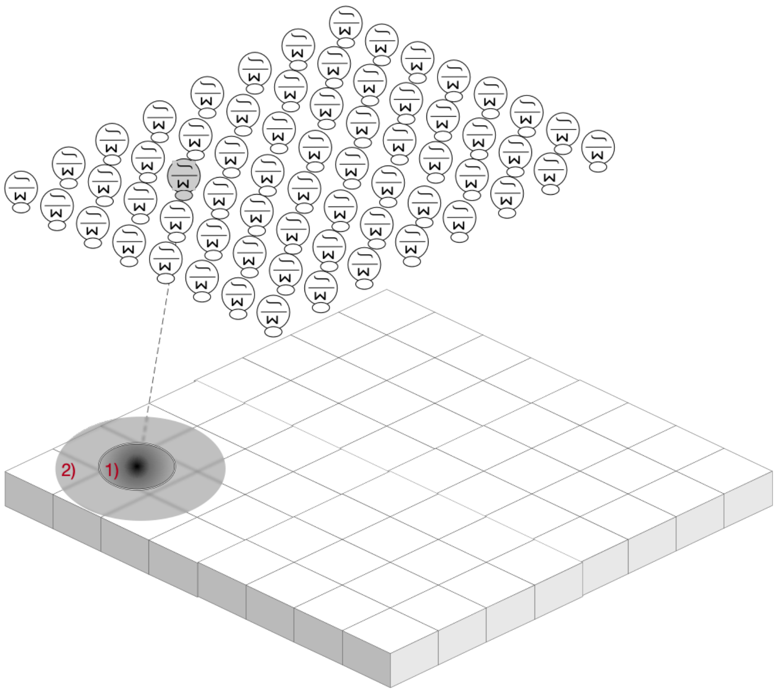


Figure 3.1: Demonstration of an envelope function. Label 2 shows the size of kernel and Label 1 shows the effective area. Effective area can grow or shrink via varying Θ.

domain. Since it is the most common case, we will assume a two-dimensional domain. Generalizations to higher dimensions are straightforward. In this domain, the envelope function must be differentiable and smooth. Let us assume a base grid for an $n \times n$ odd sized and square filter (3.1); and let U be a (continuously) differentiable function defined in grid g (coordinate space) and parameter vector $\boldsymbol{\Theta} \in R^i$ to define its shape (3.2).

$$\begin{gathered} A = \{1, 2, .., n\}, \\ g \in A \times A = \{(a, b) \mid a \in A \, and \, b \in A\} \end{gathered} \tag{3.1}$$

$$u = U(g, \boldsymbol{\Theta}), \qquad \boldsymbol{\Theta} = \{\theta_1, \theta_2, ..\theta_i\} \tag{3.2}$$

By updating the parameters in $\boldsymbol{\Theta}$, envelope function must be able to grow or shrink its effective area and change its orientation. The feed forward model of a single neuron with an input x and transfer function f can be written as:

$$o = f\Big(\sum_{g \in A \times A} x_g \, w_g \, U(g, \mathbf{\Theta})\Big) \tag{3.3}$$

Or if we think of a whole convolution layer of input matrix (image) X and weight matrix W and envelope matrix U. Simply, an element-wise multiplication of U with the weight matrix W will mask and scale the weights before the convolution.

$$O = f(X * (W \circ U)) \tag{3.4}$$

Since the weights can not grow out of envelope U, the filter size and orientation will be bounded and determined by U. Assuming that the partial derivatives of G with respect to continuous parameter θ_i is defined using the chain rule, the update can be performed using the standard back-propagation algorithm with the learning rate η. However, note that the weight update also gets u as a scaler.

$$w_g^{'} := w_g - \eta \frac{\partial E}{\partial o} \frac{\partial o}{\partial w_g} \qquad \theta_i^{'} := \theta_i - \eta \frac{\partial E}{\partial o} \frac{\partial o}{\partial U} \frac{\partial U}{\partial w_g} \tag{3.5}$$

3.1.1 Selecting an Envelope Function

It is well-known that continuous Gaussian kernel has unique properties which are important for generating a scale space. Simply put, the Gaussian kernel does not create new local extrema, nor enhance existing extrema, whilst smoothing the image with a variable continuous parameter [1]. Some of these properties are proven to exist in discrete space if the sample size is sufficiently large. Therefore, Gaussian is an ideal candidate for the envelope function U (As illustrated in figure 3.2):

$$U(g, \mu, \Sigma) = \frac{1}{A} e^{-\frac{1}{2}(g-\mu)'\Sigma^{-1}(g-\mu)} \tag{3.6}$$

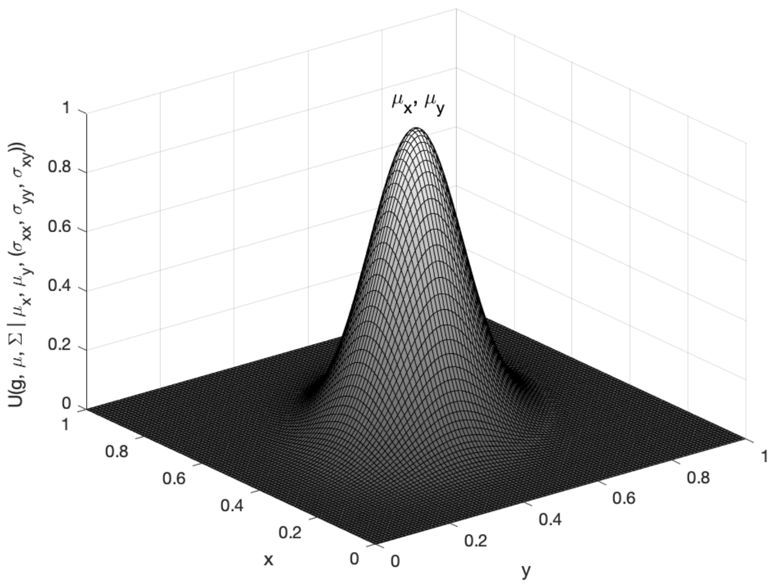


Figure 3.2: Demonstration of an envelope function. Label 2 shows the size of kernel and Label 1 shows the effective area. Effective area can grow or shrink via varying Θ

Here, parameter A is an optional normalization parameter; μ controls the center of the envelope, whereas the covariance Σ controls the scale and orientation of the kernel.

$$\mu = \left\{ \mu_x, \mu_y \right\} \quad \Sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{xy} & \sigma_{yy} \end{bmatrix} \tag{3.7}$$

During the feedforward execution the envelope function is calculated on the grid coordinates g with the current covariance Σ; and then element-wise multiplied with the weight matrix W, prior to the convolution. This is illustrated in figure 3.3(b)-3.3(d). Note that this operation not only bounds the weights and adjusts the effective area, it also scales the weights. To implement the convolution operation appropriately, we set $\boldsymbol{\mu}$ as a vector of constants that is initialized with the center point coordinates of the grid g. Therefore, it is not updated during training. However, the covariance Σ must be updated to learn the filter scale and orientation.

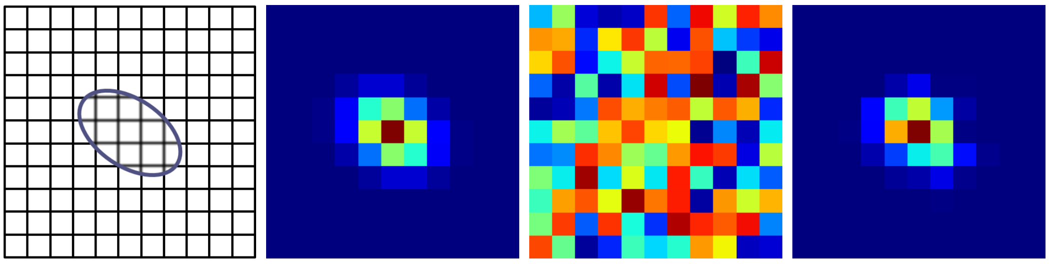


Figure 3.3: Illustration of the proposed weight envelope (a) an arbitrary differentiable envelope function controls the weight spread and shape on a regular relatively large base grid, (b) an example, initial Gaussian kernel with centered μ and initial Σ, (c) initial weights of the filter that are randomly generated, (d) weights are masked with the envelope (b) by simple element-wise multiplication.

In order to keep the symmetric property of Σ we calculate the gradients for each σ and apply update rule.

$$\forall\sigma \in \Sigma, \quad \sigma := \sigma - \eta\frac{\partial E}{\partial U}\frac{\partial U}{\partial \sigma} \tag{3.8}$$

Covariance Σ must be kept as a symmetric and positive definite matrix. A symmetric matrix is positive definite if for all non-zero vectors: $x^T\Sigma x \geq 0$; which imposes the following conditions:

$$\big(\sigma_{xx} > 0,\ \ \sigma_{yy} > 0,\ \ \sigma_{xx}\sigma_{yy} > \sigma_{xy}^2\big) \quad \texttt{or} \quad \lambda_i \geq 0 \tag{3.9}$$

where λ_i denotes the eigenvalues of the covariance matrix, which can be checked to ensure positive definiteness. However, during training the diagonal sigma terms are ensured to be positive (and nonzero) by setting bounds, e.g. σ_{xx} =min(ϵ, σ_{xx}); whereas σ_{xy} is constrained by σ_{xy}=max$\big(\sigma_{xy}, \sqrt{\sigma_{xx}\sigma_{yy}}\big)$. However, experiments show that the covariance behaves well during training when it is initialized properly and updated with a small learning rate, and thus it removes the necessity for these constraints.

3.1.2 An alternative envelope function

Keeping the covariance symmetric and positive definite matrix creates constraints which can be assured with different methods such as regularizing. In spite of keeping constraints, calculating Σ^{-1} can be computationally expensive and keeping covariance as a symmetric and positive definite matrix has additional constraints to the overall cost function. A fast working alternative is to use a full-with-at-half-maximum parameter to change the width of a curve function.

$$U(\mu, \sigma) = e^{-\frac{(x-\mu)^2+(y-\mu)^2}{2\sigma^2}} \tag{3.10}$$

Eq. 3.10 has one trainable parameter σ which varies the with of the curve function. Where x, y are the normalized indices of weight matrix and μ is the center and constant (0.5).

3.2 Initialization

Since the improvements in computational instruments rise, training deeper neural networks became popular among researchers. Studies showed that randomly initialized weights result with poor performance [15]. Recent studies show that proper initialization for deeply layered networks is crucial [16]. Gradients populated via back-propagation move weights regardless of the scale difference between layers. In such a case when the network is relatively deep enough, the scale difference between weights in different layers may hamper the learning. We observed scale difference when using recent initialization methods (i.e [15], [17]) on weights, resulting in vanishing variance between layers.

Conventionally, initial weights are being drawn randomly from a distribution $Wi, j \sim [-a, a]$ where a is the upper and lower bounds of the distribution. The range of a affects the total variance on weights, thus changing the input variance. On deep neural networks, being able to keep the input variance can be compelling.

Ignoring envelope functions scaling on weights will result in irreversible variance change across the whole network. Ensuring $Var(x) = Var(y) = 1$ where x, y are the input and output receptively, can be done via keeping the variance change on weights W over inputs to unit.

$$Var(W) \cdot n_{in} = 1 \tag{3.11}$$

Authors of [15] take this to one step further and also took output connections in account.

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{3.12}$$

Where n_{in} and n_{out} is the total number of connected and fed neurons. Deriving 3.12 to a solution yields with two possible solutions 3.13.

$$Var(W) \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \tag{3.13}$$

Our initialization method is based on Eq.3.13 and still take the envelope functions scaling effect in account. As described above, adding U must still keep the variance unit. Considering U, the formula can be rewritten with the following steps;

$$Var(WU) = U^2 Var(W) \tag{3.14}$$

$$Var(W) = \frac{2}{U^2(n_{in} + n_{out})} \tag{3.15}$$

$$Var(W) = E(X^2) - E(X)^2 = \frac{(b-a)^2}{12} \tag{3.16}$$

$$\frac{2}{U^2(n_{in}+n_{out})} = \frac{(b-a)^2}{12} \tag{3.17}$$

- Eq. 3.14: W and U are independent variables. We used a variance property to take out U from the calculation: $Var(aX+b) = X^2 Var(a)$

- Eq. 3.16: W is drawn from uniform distribution. Thus variance of W equals to a form of interval ranges a and b
- We used the initialization ranges calculated in Eq. 3.18

$$W \sim \left[-\sqrt{\frac{6}{U^2(n_{in}+n_{out})}}, \sqrt{\frac{6}{U^2(n_{in}+n_{out})}} \right] \tag{3.18}$$

3.3 Implementation

We implemented the adaptive convolution filters using a base CNN implementation available on Lasagne [18]. Lower level details such as gradient manipulation and learning rate feeding for different parameters are done using Theano backend.[19] Tests ran on Nvidia Tesla K40 and P100 GPU boards. In terms of computational complexity, using a Gaussian function as envelope function adds an extra overhead in training. While most costly operations are on the inverse calculation of covariance, an approximation function is also used in some stages of late implementation and tests as described in Section 3.1.2

However, during feed forward execution, the trained and enveloped final weights can be stored and used immediately without any overhead. Compared to conventional filters, we use relatively larger (e.g. 11x11) base filter sizes to observe adaptive growth and rotations. Please note that the grid can be selected as large the input image. A-CNN ran one epoch (500 examples) in $6.1 \pm .2$ seconds whereas CNN-11 ran in $3.4 \pm .15$ on MNIST dataset without an optimized implementation(available online [1].)

[1]github.com/ilkerc/AdaptiveCNN

Chapter 4

Experiments

We conducted three types of experiments to compare and contrast the adaptive layers performance on the visual domain. Firstly, auto-encoder tests are discussing the necessity and collaboration of envelope functions with weights. ILSVRC14 winner GoogleNet uses a deeper and wider network, inception module introduces descriptive and highly non-linear features using different kernel sizes concatenated on a single depth channel. We replaced these engineered filters with ACNN and discussed results on the second part.

Lastly, using a shallow network we experimented the filters scale and orientation changes. We tested the adaptive filter model with three different datasets, also compared the results against two conventional CNN configurations that used different fixed size filters.

4.1 Filter Guide Experiments

One can argue that the filter guide is unnecessary because a relatively large CNN-layer can learn any filter. To disprove this idea we conducted a test with an encoder where the input is an image the filter is expected to learn a simple filtering operation.

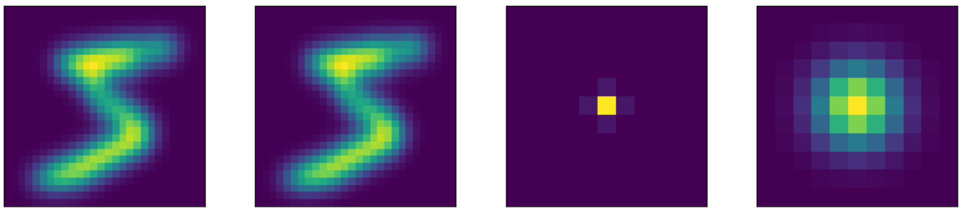

(a) Target (b) Output (c) Initial Envelope Function (d) Final Envelope Function

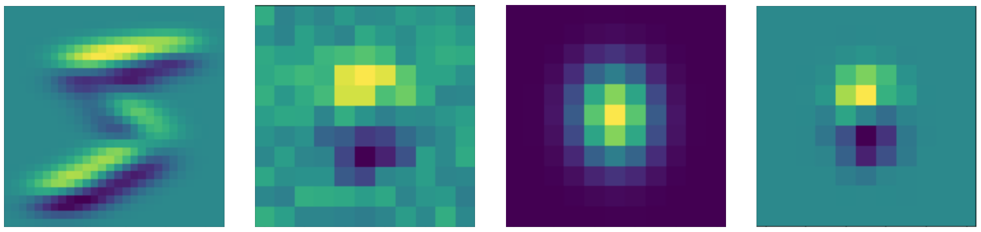
Figure 4.1: An example chosen from hand-written digits. Applied a gaussian filter, the only envelope function learns to enlarge the filter.

4.1.1 Learning a Gaussian filter

We conducted a test on a simple auto-encoder stack where the input is 28×28 image and output is the same input applied edge filter on one axis. The adaptive convolutional layer is expected to learn the filter without any weights. Figure 4.1 demonstrates how A-CNN can learn a Gaussian filter. The final σ stops to change on almost identical precision to targets Gaussian wideness. As trivial this problem might seem, the only trainable parameter is σ of the filter. No weights or biases are present on the training model.

4.1.2 Learning a Gaussian and edge filter

Perhaps one of the biggest obstacles in this study was the cancellation of envelope function by the weights. Experiment on Section 4.1.1 can learn a Gaussian filter with and without training weights. Although, weights collaboration with envelope function is not illustrated. The input image used in this experiments is convolved with a Gaussian and edge filters respectively. An envelope function can not regenerate an output for both filters. Figure 4.2(b) and 4.2(c) shows the finals weights and envelope function respectively. In addition to Gaussian's scale and orientation features, the envelope function also limits the effective area of weights. We can conclude that weights and envelope function can be trained together.



(a) Target (b) Weights (c) Envelope Function (d) Weights * Envelope Function

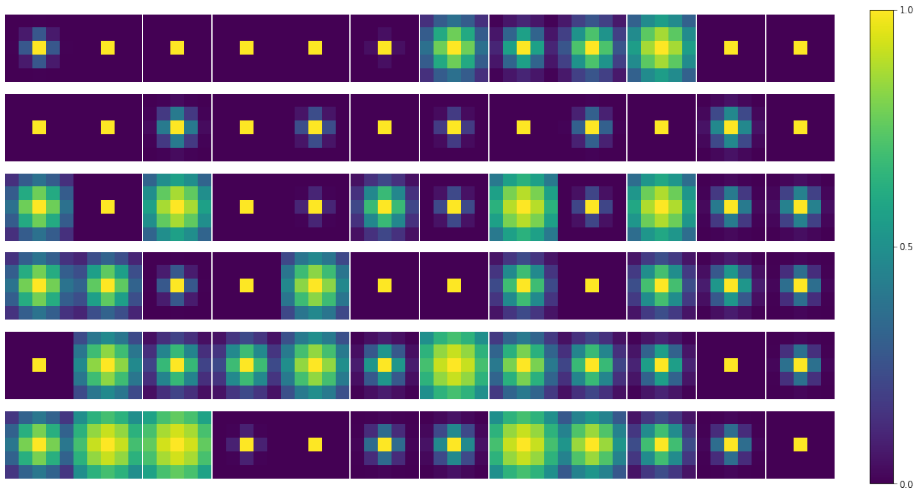
Figure 4.2: An example chosen from hand-written digits. Applied a gaussian and sobel filter respectively.

4.2 Application on deeper and wider networks

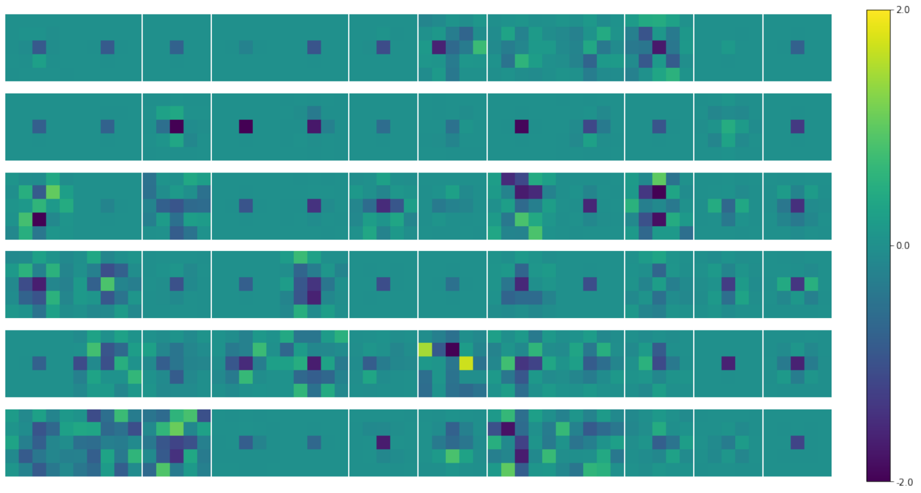
GoogleNet, a deep convolutional neural network architecture, uses "inception modules" which stacks different sizes of convolutional kernels and pooling layer into one output layer. The main idea of the study is to improve computational utilization and increase the levels of deepness and wideness. ACNN has a similar idea to train different scaled kernels in the same layer. We replaced the inception module's 3×3 and 5×5 convolutional layers with a single ACNN layer and conducted a similar test case on shallower network architecture. Figure 4.4 illustrates the adaptive-module used in replacement for inception-module described in section 2.1.5.

In order to keep the computational performance faster, we used an alternative envelope function which only scales up and down on the kernel described in section 3.1.2. The original architecture is 22 layers (layers only with trainable parameters) in total whereas our implementation has 11 layers in order to keep the test simple and faster. The network details are described in table 4.1. Our test setup uses nesterov momentum as optimization method with 0.9 momentum, fixed learning rate 0.01. Trained 150 epochs with 500 mini-batches. Input dataset is CIFAR-10 [20] $32 \times 32 \times 3$ image-set is divided into $50,000$ training and $10,000$ validation sets.

Validation accuracy is shown on figure 4.5. The adaptive module is performing less or equally whereas the last stacked adaptive-module filters are illustrated in



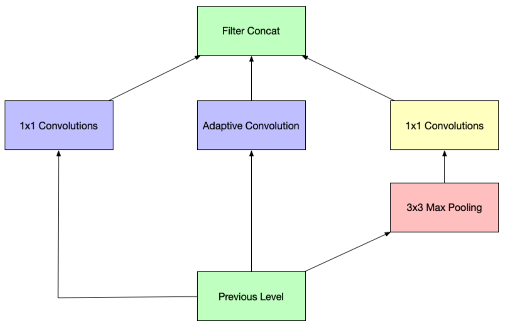
(a) Envelope functions



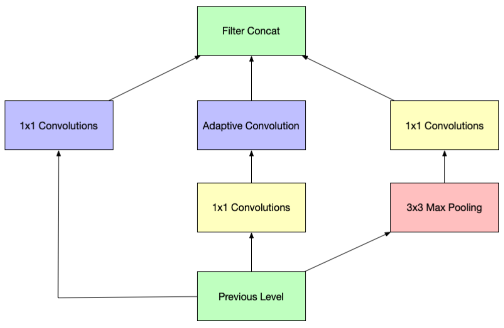
(b) Effective weights

Figure 4.3: a) Envelope functions of first 72 filters (144 filters in total) in ACNN at the deepest level in adaptive module. b) Weights summed in depth channel and multiplied with corresponding envelope function.

figure 4.3 has a variety of scaled kernels. We initiated Σ such that they have approximately 3×3 effective area. Adaptive kernels can grow up to 5×5 when envelope function grows. Scaled weights in figure 4.3(b) shows scaled weights, their effective area is limited by their corresponding envelope function outputs shown on 4.3(a). The figure shows that filter scales are trained to scale up and down. We used L-2 norm regularization on weights and as the figure shows the inactive regions of the weight matrices are very close to zero.



(a) Naive Adaptive Module



(b) Adaptive Module

Figure 4.4: a) Naive adaptive module does not implement 1×1 convolutions. b) 1×1 convolutions are used to increase non-linearity and reduce dimension.

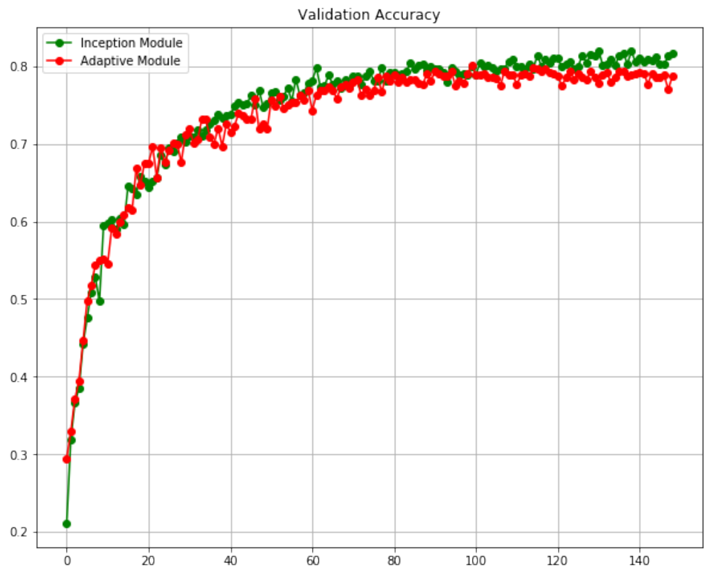


Figure 4.5: Validation accuracy plot for inception and adaptive modules.

type	**patch size/ stride**	**output size**	**depth**	#1×1	#3×3 **reduce**	#3×3	#5×5 **reduce**	#5×5	**pool proj**
convolution	7×7/1	32×32×16	1						
max pool	3×3/2	30×30×16	0						
convolution	3×3/1	30×30×48	2		16	48			
max pool	3×3/2	28×28×48	0						
inception (3a)		28×28×64	2	16	24	32	4	8	8
max pool	3×3/2	14×14×64	0						
inception (4a)		14×14×128	2	48	24	52	4	12	16
max pool	3×3/2	7×7×128	0						
inception (5a)		7×7×240	2	64	40	80	8	32	32
avg pool	7×7/1	1×1×240	0						
dropout (40%)		1×1×240	0						
softmax		1×1×10	0						

Table 4.1: GoogLeNet incarnation of the Inception architecture

Layer	Units	Filters	Filter Size	Pool Size	Activation
1- input	-	-	-	-	-
2- convolution-1	-	8/16*	5x5/11x11	-	ReLu
3- max pool-1	-	-	-	2x2	-
4- convolution-2	-	8/16*	5x5/11x11	-	ReLu
5- max pool-2	-	-	-	2x2	-
6- dropout(50%)	-	-	-	-	-
7- fully connected - 1	256	-	-	-	ReLu
8- fully connected - 2	10	-	-	-	Sigmoid

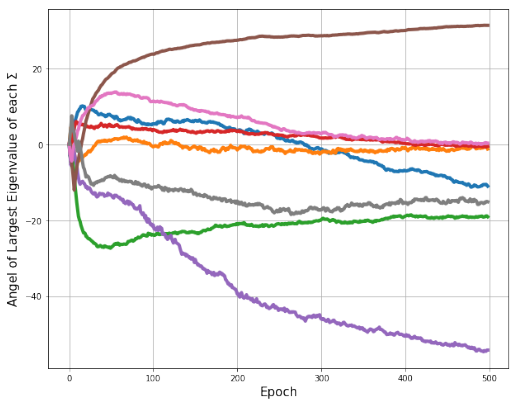
Table 4.2: The network topology that is used to test our method. All three networks were comprised of 8 layers. In conv-1 and conv-2 layers, the proposed adaptive model (ACNN) used an 11×11 base grid for filters, whereas 'cnn-5' and 'cnn-11' used 5×5 and 11×11 filter sizes, respectively. *CIFAR-10 experiments used 16 filters instead of 8.

4.3 Scale and orientation experiments

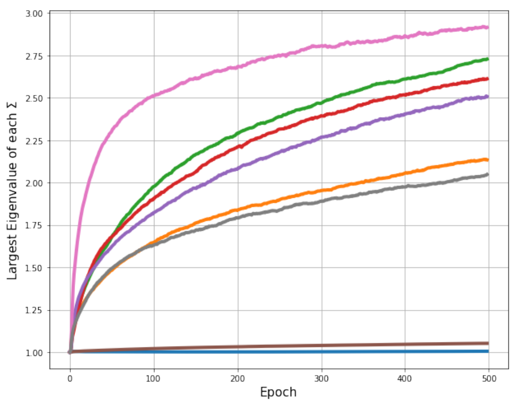
We tested the adaptive filter model with three different datasets, compared the results against two conventional CNN configurations that used different fixed size filters. All three networks have the same architecture stacked with two convolutions followed by pooling layers, a dropout layer, and two fully connected layers as described in table 4.2. The only difference between the adaptive CNN (ACNN) and conventional CNNs (cnn-5, cnn-11) is the replacement of convolution layers. The hyperparameters we used are as follows; Learning rate: 0.01, momentum: 0.95, batch size: 500. We trained each model 500 epochs and compared classification errors, scale and orientation changes in filters. Though the learning rate for Σ could be adjusted separately it was not necessary.

4.3.1 MNIST

MNIST [21] is a database of handwritten digits, widely used in machine learning research to test models. It has 50,000 training and 10,000 testing images from 10 different categories. To observe the change in Σ, we calculated its eigenvalues and eigenvectors. The maximum eigenvalue represents the scale, whereas the

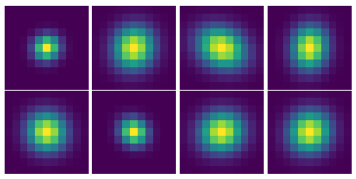


(a) Orientation Change.

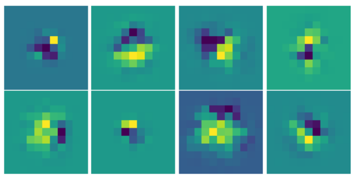


(b) Scale Change.

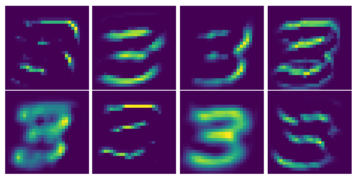
Figure 4.6: Covariance matrix Σ change in MNIST dataset. Depicted by the (a) angle of largest eigenvector and (b) largest eigenvalue.



(a) Gaussian envelope functions.



(b) Scaled filters.



(c) Convolution outputs.

Figure 4.7: The first layer (conv-1) filters at the end of training with MNIST. (a) Gaussian envelopes, (b) scaled filters, (c) output of a sample that was convolved with each filter.

tangent between eigenvectors shows the orientation as illustrated in Figure 4.6. The early stages of training orientation changes happen rapidly whereas scale tends to change continuously. In Figure 4.7, we can observe the learned envelope functions scale and orientation effects on filters. Smoothing effect of the envelope function over the input is also observed in some outputs (4.7(c)).

Figure 4.8 shows the classification error plot. Although, the adaptive filters had no performance gain against conventional CNN-11, CNN-5, a variety of scale and orientation effects can be observed on the final filters.

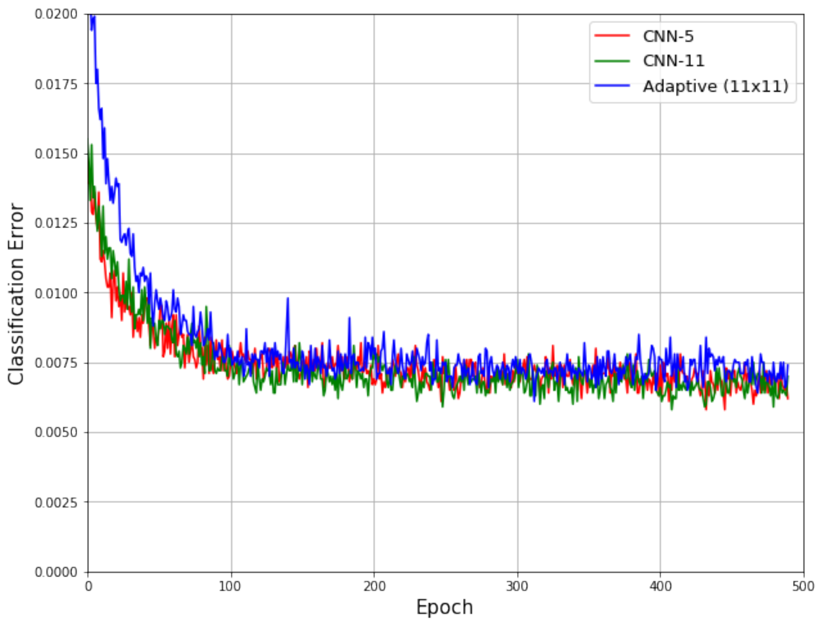


Figure 4.8: Classification error for MNIST dataset.

4.3.2 MNIST Cluttered

Cluttered MNIST dataset [22] consists of 60, 000 samples in 10 classes. We split this dataset into 50, 000 and 10, 000 for test and train purposes, respectively. Randomly selected 8 samples are illustrated in Figure 4.9. It contains 60×60 images generated from the original MNIST database with numerous distractions. Projecting the original MNIST 28×28 pixel space onto 60×60 also caused changes in scale. Thus, the dataset has the scale and rotational variances, in addition to cluttered background noise which makes it a suitable test case to demonstrate the use of adaptive filters. Figure 4.10 shows classification error. Compared with 5×5 and 11×11 kernel sizes, adaptive kernels perform better than conventional neural networks.

4.3.3 CIFAR-10

This dataset [20] is a relatively small $(32 \times 32 \times 3)$ image set with 60, 000 samples from 10 different classes. We divided this dataset into 50, 000 train and 10,000 test sets, respectively. Other than MNIST dataset, color channels are present, and

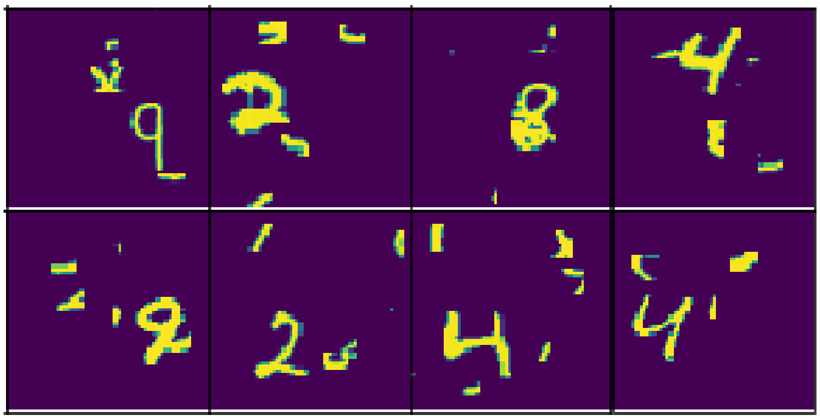


Figure 4.9: Eight randomly selected samples from the cluttered MNIST dataset.

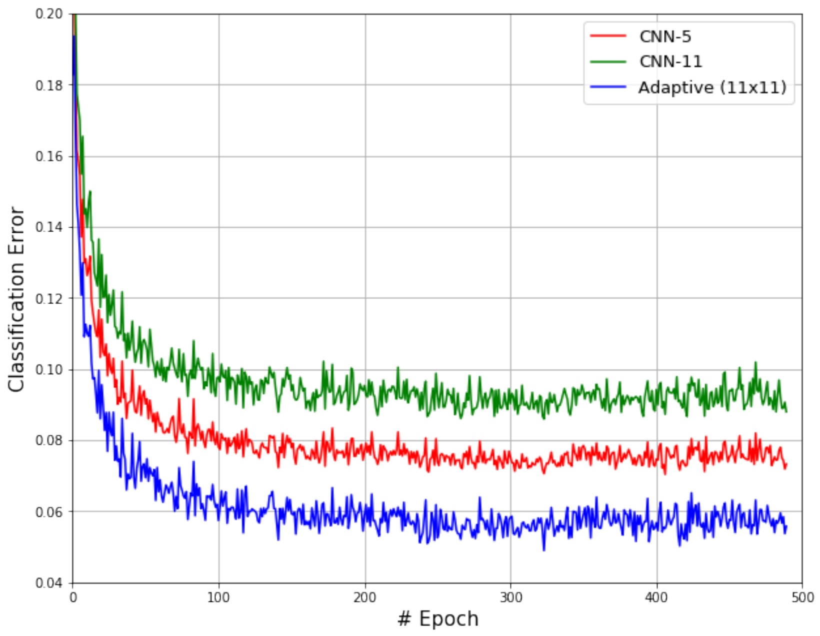
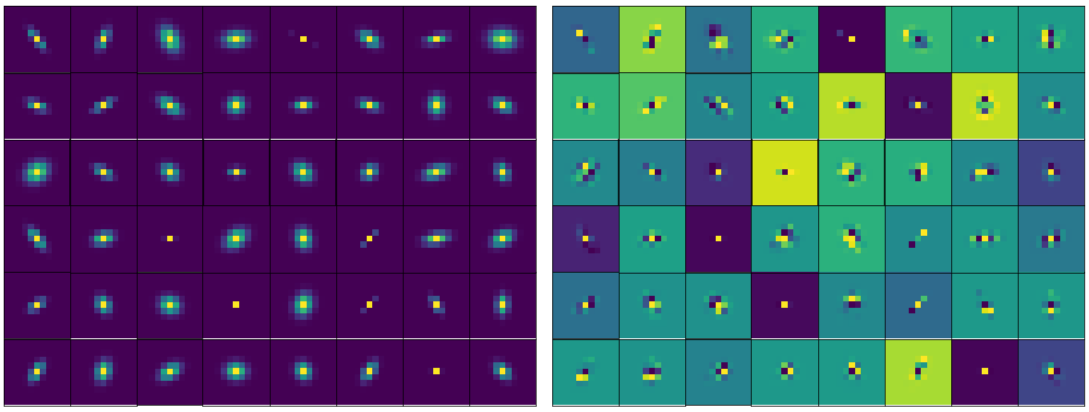


Figure 4.10: Classification error for cluttered MNIST dataset.

objects are much more in need of multi-scale features. The classification results that are shown in Figure 4.12. ACNN performed better in terms of classification performance. For further investigation, we also included the change of Σ for learned envelope functions and scaled filters in Figure 4.11. Compared to MNIST, the envelope functions are observably different; and included both large, small, rotated filters. The change in scale and orientation is shown in Figure 4.13. Compared with change on Σ in MNIST test, scales and orientations have more variation and some filters tend to shrink, whereas some were enlarged their scales.



(a) Gaussian envelope functions.

(b) Scaled filters.

Figure 4.11: The first layer filters at the end of training in CIFAR-10 database.

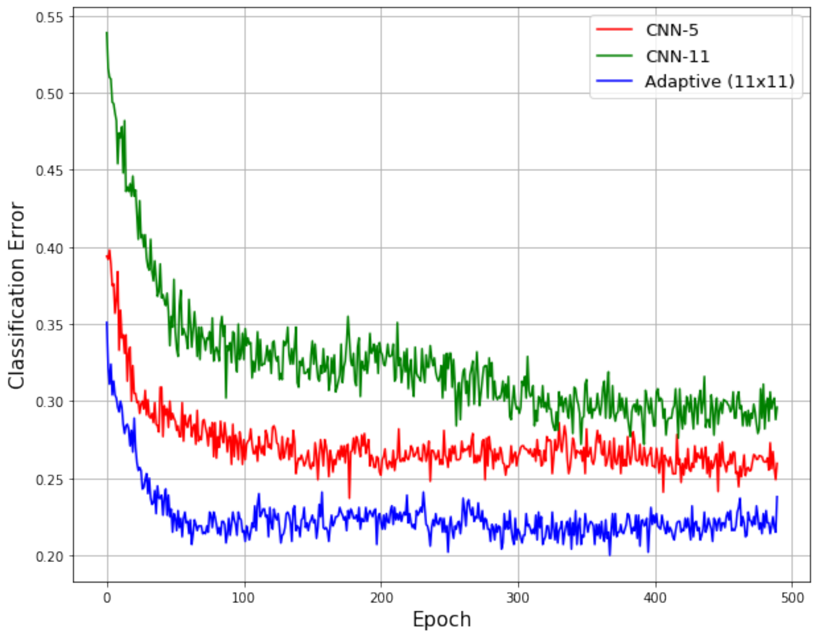
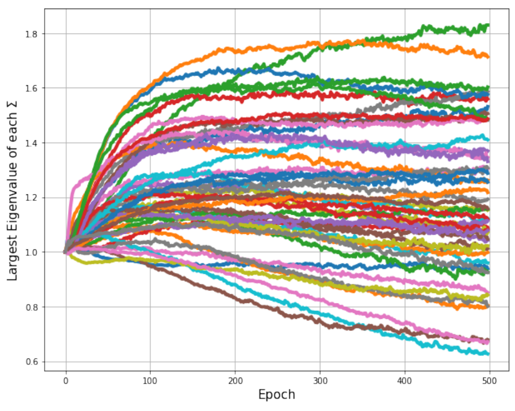
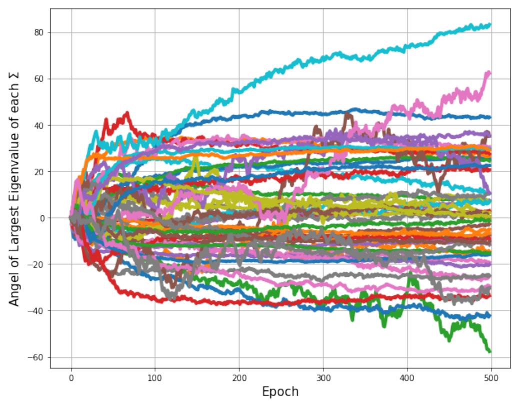


Figure 4.12: Classification error for CIFAR-10 dataset.



(a) Orientation Change.



(b) Scale Change.

Figure 4.13: Plots for covariance matrix Σ change in CIFAR-10 dataset, depicted by the (a) angle of largest eigenvector, (b) largest eigenvalue.

Chapter 5

Conclusion

In this study, we propose an adaptive convolution filter model based on a Gaussian kernel that is acting as an envelope function on shared filter weights. The plots of scale and orientation changes during the training epochs show that the adaptive model is capable of generating differently scaled and oriented filters in a single convolution layer. However, besides bounding and scaling of convolution weights, the Gaussian kernel tends to perform smoothing on input. Such that, if all weights were set to 1 and not trainable, the kernels perform only a Gaussian smoothing operation on input.

The initial setting of variance terms to 1.0, enables an initial filter of 5x5 size. During training the effective size of the filters is gradually increased. This is because enlarging filters enable more weights to be included in the convolution, which will allow further reduction in the network error. Therefore, the adaptive filter may be prone to over-fit more than a conventional fixed sized filter of the same initial size. However, because the envelope rescales weights (max 1.0), it has a regulative effect on their magnitudes, which shall create an advantage. In overall, training of the adaptive filter model did not require very fine tuning of the parameters. However, we observed that the use of dropout layer encouraged the development of filters of different scale and orientation. This can be explained by that the parallel and sparse network configurations induced by the dropout mechanism forces filters to prevent co-adaptation and become independent. We will investigate other ways of inducing independent filters, perhaps with an additional cost term for the network which punishes co-adaptation.

A clear benefit of our model is that it removes the filter size from the list of hyper-parameters of deep learning networks. However, our main purpose is to add an adaptive multi-scale representation capacity to convolution layers. The results show that the advantage of using the new model depends on the complexity and variations in training and test data. Among the three datasets, MNIST is the simplest where digits are size-normalized and centered. The adaptive filters have less or no need for scale adaptation in pixel space, which resulted in no improvement in classification error when compared to conventional CNNs. However, MNIST cluttered and CIFAR-10 include examples of arbitrary scale, orientation, and center, which allowed the filters to adapt their scale and orientation to improve training while not over-fitting. Therefore, we can conclude the adaptive filters expressive power is revealed in datasets with variations in scale and orientation. It is worthwhile to investigate its applications to other domains.

The new and adaptive model of convolution layers allows filters' scale and orientation to be learned during training. Therefore, a single convolution layer can have filters at various scales and orientations. As a result, a single convolution layer can adapt to extract multi-scale information from its input. State-of-the-art deep networks have many layers and more complex designs compared to the networks that were tested in this study. An interesting question which we will investigate further is whether using the adaptive filter layers can shorten the depth of state-of-the-art architectures, such as inception [9], highway [23] or thin [24]. Though our aim is not to fully replace stacked and deep architectures, the new model may help reduce redundancy and improve accuracy. Another question is whether placing the adaptive layer in deeper levels of a network can produce additional gains by focusing on the higher level representations.

References

[1] T. Lindeberg, *Scale-Space Theory in Computer Vision.* Norwell, MA, USA: Kluwer Academic Publishers, 1994.

[2] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[3] 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, Nov 1998.

[4] 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.

[5] S. Mallat and S. Zhong, “Characterization of signals from multiscale edges,” *IEEE Transactions on pattern analysis and machine intelligence*, vol. 14, no. 7, pp. 710–732, 1992.

[6] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, “Robust object recognition with cortex-like mechanisms,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 3, pp. 411–426, March 2007.

[7] T. Chan, K. Jia, S. Gao, J. Lu, Z. Zeng, and Y. Ma, “Pcanet: A simple deep learning baseline for image classification?” *CoRR*, vol. abs/1404.3606, 2014. [Online]. Available: http://arxiv.org/abs/1404.3606

[8] M. D. Zeiler and R. Fergus, *Visualizing and Understanding Convolutional Networks*, 2014, pp. 818–833. [Online]. Available: https://doi.org/10.1007/978-3-319-10590-1_53

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[10] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of physiology*, vol. 195, no. 1, pp. 215–243, 1968.

[11] "Applied deep learning - part 1: Artificial neural networks," accessed: 2019-02-02. [Online]. Available: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

[12] S. Boccaletti, G. Bianconi, R. Criado, C. I. Del Genio, J. Gómez-Gardenes, M. Romance, I. Sendina-Nadal, Z. Wang, and M. Zanin, "The structure and dynamics of multilayer networks," *Physics Reports*, vol. 544, no. 1, pp. 1–122, 2014.

[13] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[14] F. B. Tek, "An adaptive locally connected neuron model: Focusing neuron," *CoRR*, vol. abs/1809.09533, 2018. [Online]. Available: http://arxiv.org/abs/1809.09533

[15] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, vol. 9, May 2010, pp. 249–256.

[16] D. Mishkin and J. Matas, "All you need is a good init," *CoRR*, vol. abs/1511.06422, 2015. [Online]. Available: http://arxiv.org/abs/1511.06422

[17] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[18] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri *et al.*, “Lasagne: First release.” Aug. 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.27878

[19] Theano Development Team, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[20] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[21] Y. LeCun and C. Cortes, “MNIST handwritten digit database,” 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[22] c. Christopher, “Cluttered mnist dataset,” https://github.com/christopher5106/mnist-cluttered, 2015.

[23] R. K. Srivastava, K. Greff, and J. Schmidhuber, “Highway networks,” *CoRR*, vol. abs/1505.00387, 2015. [Online]. Available: http://arxiv.org/abs/1505.00387

[24] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, “Fitnets: Hints for thin deep nets,” *CoRR*, vol. abs/1412.6550, 2014. [Online]. Available: http://arxiv.org/abs/1412.6550