Functionally Weighted Convolutional Neural Networks

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Dimitris Triantis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN TECHNOLOGIES - APPLICATIONS

University of Ioannina

November 2017

Examining Committee:

- **Konstantinos Blekas**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)
- **Isaac Lagaris**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Aristidis Likas**, Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

Dedicated to my family and friends.

Acknowledgements

I would like to thank my thesis advisor, Prof. Konstantinos Blekas for the guidance and the valuable advice he provided me throughout the completion of this thesis. Working beside him has been a great expeprience and a big influence for developing a strong work ethic.

I would also like to thank Prof. Isaac Lagaris and Prof. Aristidis Likas for agreeing to participate in my examination committee and for their helpful comments and insight on my thesis.

Furthermore, I gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

Finally, I would like to express my sincere gratitude towards my family and friends. Their great help and support kept me going through all these years. Without them this thesis could not have been possible.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Dimitris Triantis, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, November 2017.
Functionally Weighted Convolutional Neural Networks.
Advisor: Konstantinos Blekas, Assistant Professor.

In this thesis we introduce a new convolutional model where the weights are functions of a continuous variable, instead of a discrete indexed kernels. Consequently, we create an infinite size kernel for the convolutional layer, thus we obtain infinite feature maps as the output of the layer which become an integral over the introduced continuous variable. The gain is a drastic reduction of parameteres, accompanied by a superior generalization performance. To evaluate the quality of this new network, we conducted a series of experiments among some established computer vision datasets obtaining some very promising results.

Εκτεταμενη Περιληψη

Δημήτρης Τριάντης, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Νοεμβριος 2017.
Functionally Weighted Convolutional Neural Networks.
Επιβλέπων: Κωνσταντίνος Μπλέκας, Αναπληρωτής Καθηγητής.

Τα Νευρωνικά δίκτυα βαθιάς μάθησης αποτελούν έναν τομέα της Μηχανικής Μάθησης με έντονο επιστημονικό ενδιαφέρον τα τελευταία έτη. Τα δίκτυα αυτά έχουν παρόμοια αρχιτεκτονική με τα κλασσικά νευρωνικά δίκτυα, αλλά διαφέρουν στον τρόπο και την τοπολογία της εισόδου των δεδομένων. Η ιδιαιτερότητα των δικτύων αυτών εντοπίζεται στην ικανότητά τους να παράγουν «πλούσια» και ποιοτικά χαρακτηριστικά των αρχικών δεδομένων, χρησιμοποιώντας ένα μηχανισμό αποτελούμενο από ένα πεπερασμένο πλήθος φίλτρων. Τα συνελικτικά νευρωνικά δίκτυα είναι η πιο γνωστή μορφή νευρωνικών δικτύων βαθιάς μάθησης και χρησιμοποιούνται σε ποικίλες εφαρμογές μηχανικής όρασης και αναγνώρισης. Σημαντικό ρόλο στην ανάπτυξη αυτού του τύπου των νευρωνικών δικτύων διαδραμάτισε η εξέλιξη της υπολογιστικής ισχύος των πληροφοριακών συστημάτων.

Στην παρούσα εργασία μελετάται μια νέα μεθοδολογία κατασκευής συνελικτικών δικτύων χρησιμοποιώντας φίλτρα ή kernels συναρτησιακής μορφής. Αυτό έχει ως αποτέλεσμα την ταυτόχρονη επιβολή στα δεδομένα άπειρων σε πλήθος φίλτρων, που οδηγούν σε ποιοτικότερα χαρακτηριστικά με σημαντικά μικρότερο αριθμό παραμέτρων, διευκολύνοντας έτσι τη γενικευτική ικανότητα της μεθόδου. Μία παραλλαγή της μεθόδου είναι η χρήση kernels συναρτησιακής μορφής, όχι στο συνελικτικό επίπεδο παραγωγής χαρακτηριστικών, αλλά στο fully-connected layer του CNN, όπου επιτελείται το τελικό στάδιο του μηχανισμού αναγνώρισης.

Η προτεινόμενη μεθοδολογία αξιολογείται πειραματικά πάνω σε προβλήματα ταξινόμησης χρησιμοποιώντας γνωστά σύνολα δεδομένων υπολογιστικής όρασης

(MNIST και CIFAR-10), ενώ συγκρίνεται και με γνωστές αρχιτεκτονικές CNNs της βιβλιογραφίας.

Chapter 1

Introduction

The mystery of how brain sifts signals from senses and elevates them to the level of conscious awareness drove much of the early interest in deep neural networks among artificial intelligence pioneers, who hoped to build an intelligent system through reverse-engineering the brain's learning rules. The first known computational model of neural networks was formalized in 1943 by neurophysiologist Warren McCulloch and mathematician Walter Pitts [2] whose work discusses how neurons in the brain might work. They used a combination of algorithms they called "threshold logic" to mimic the thought process. Since then, deep learning is evolving steadily. The first convolutional neural networks were used in 1979 by Kunihiko Fukushima [3] who designed an artificial neural network called Neocognitron, employing multiple pooling and convolutional layers. Neocognitron networks resembled modern versions, but were trained with a reinforcement strategy of recurring activation in multiple layers.

When Rumelhart, Williams, and Hinton [4] demonstrated back propagation brought a breakout in the field, and showed how some of the neural networks limitations may be overcome. It had been invented a decade sooner, but the first computer experiments demonstrating it can generate useful representations were published in 1986. In 1989, Yann LeCun [5] provided the first practical demonstration of backpropagation on convolutional neural networks, constructing a system which was eventually used to read the numbers of handwritten checks. Around the year 2000, The Vanishing Gradient Problem appeared. It was discovered that for deeper archichitectures, the gradient signal was deminishing as it propagated to the lower layers of the net-

work. This was a fundamental problem for neural networks with gradient-based learning methods. The source of the problem were multiple reasons, from which the most significant turned out to be certain activation functions. One early approach to the solution were used to solve this problem were layer by layer pre-training using autoencoders [6] and restricted boltzman machines [7].

Within the next decade, parallel architecture found its use in the gaming industry. Gaming industry represents a big market, which provided generous funding for the development of high performance chips with parallel architecture. Computationally poweful chips became relatively cheap and available for the masses which greatly fueled research in the artificial intelligence field. The boosted computing speed came along with an immense increase on volume, speed and different sources of data, allowing that way to construct deeper neural network models capable of recognizing very complicated patterns. An iconic work that formed the modern trend towards deep learning methods, was published in 2012 by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, titled "ImageNet Classification with Deep Convolutional Networks" [8] and is widely regarded as one of the most influential publications in the field. This was the first time a model performed so well on the historically difficult ImageNet dataset, utilizing techniques that are still used today, like dropout and non-saturating activation functions. With all those astonishing achievments, deep neural networks have established as very effective methods and have made prominent contributions across a broad spectrum of industries. Their value is widely appreciated in a variety of applications, among which are medical imaging, autonomous driving, and weather forecasting to name but a few.

Motivated by the great success of convolutional neural networks, in this thesis we propose a new convolutional model. We observe that convolutional neural networks, despite their parameter sharing properties, utilize an extensive number of parameters in order to achieve fair results on demanding problems, thus issues like over-fitting emerge. To this end we propose a functionally weighted convolutional neural network which employs weights as functions of a continuous variable, instead of a discrete index. This way we create infinite feature maps through an infinite kernel, gaining a drastic reduction in the number of parameters, accompanied by an improved generalization performance.

This thesis consists of three chapters. In the first chapter, we make a thorough analysis behind the theory and the structure of a neuron which is the fundamental

building block of the network and we continue with a discussion of the learning process and the basic concepts behind related to feedforward neural networks and convolutional networks. We analyze their main properties and some widely used practices. In the second chapter we introduce and mathematically define our proposed method for a functionally weighted neural network. We give examples of the how it is structured, the way it can employ different activation functions and how it engages the convolutional operation. In the last chapter we give a brief description of the framework, the benchmark datasets and the implememententation details we chose for the evaluation of our method and, consequently, we present the results for the chosen dasets for a variety of configurations of our network.

Chapter 2

Deep Neural Networks

2.1 Neural Networks

Aritificial Neural Networks are algorithms within the machine learning used for the construction of models for supervised and unsupervised learning. Inspired by biological nervous systems, an Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. The learning proccess of biological systems involves adjustments to the synaptic connections that exist between the neurones, likewise Neural Networks learn in a similar way.

A feedforward neural network is a network composed of computational units which are called neurons. An artificial neuron is a computational unit which makes a particular computation based on other units it is connected to. Neurons are organized

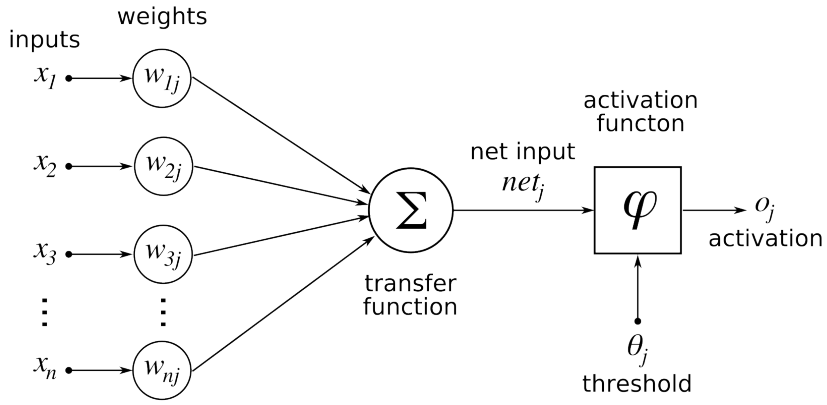


Figure 2.1: Artificial Neural Network.

in layers, where there are no connections between neurons of the same layer, but are fully connected to each other in succesive layers. They are called feedforward neural networks because all the computations follow the same direction. The data flows in the direction of oriented edges and ends at the output neurons. The result is interpreted from the values obtained in the output neurons, hence they can be interpreted as a directed acyclic graph. Neurons that receive stimuli from outside the network are called input neurons, those whose outputs are used externally are called output neurons. Neurons that receive stimuli from other neurons and whose output is a stimulus for other neurons are known as hidden neurons.

The objective of a feedforward network is to approximate some function f^*. For example for a classifier, $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation. An artificial neuron has d inputs represented as a vector $x \in \mathbb{R}^d$, where each input $i \leq i \leq d$ has an assigned weight $w_1 ... w_d$, it performs computations and feeds the result to the next layer neurons. Weighted input values are combined and run through an activation function producing some output y as shown in figure 2.1.

Assume we have a neuron j with input $\boldsymbol{x_j} = (x_{1j}, ..., x_{dj})$, weights $w_1 ... w_d$ and bias θ_j the pre-activation of the neuron is computed as:

$$\xi_j = 1 + \sum_{j=1}^{d} w_{ij} x_{ij} + \theta_j \,. \tag{2.1}$$

Where we consider the sigmoid as activation function:

$$f(a) = \frac{1}{1 + e^{-a}} \,. \tag{2.2}$$

Then the output y_i of the neuron j is computed:

$$y_j = f(\xi_j) = f(\sum_{j=1}^{d} w_{ij}x_{ij} + \theta_j)\,. \tag{2.3}$$

Considering an Artificial Neural Network contains k output neurons in the output layer, we obtain the output of the network as $\boldsymbol{y} = (y_1 ... y_m)$. A more general definition of an ANN is given in [1].

2.1.1 Perceptron

The simplest neural network architecture is the perceptron that consists of one fully functional neuron. A perceptron is a single neuron which computes a linear combination of an input vector with some weights and then passεσ it through a nonlinearity to make decisions. Its output is taken directly by Equation 2.3. The most basic activation function is the binary step function, which produces strictly binary outputs. By definition the single perceptron forming a trivial network uses the step function, though the most commonly function used as an activation function is the sigmoid. In 2.3b we can see the sigmoid function which produces a continuous output.

Single perceptron neuron is too trivial for solving complex tasks. Simple ANNs can be extended to the multilayer perceptron network (MLP) to address more complex tasks. As we will discuss in the next section, for the training of an MLP, continuous functions are more suitable for gradient based learning, because we can easily calculate their derivative which is important for the weight adjustment.

2.1.2 Multilayer Perceptrons

The Multilayer perceptrons (MLPs) is a feed-forward neural network consisting of multiple mutually interconnected layers of neurons. The motivation behind it is overcoming the limitation of the perceptrons which is solvινγ problems strictly on linear decision surfaces. Multilayer perceptrons overcome this problem by combining multiple neurons organized in layers in order to obtain more compex functions composed through several simpler ones.

MLPs have the advantage of creating nonlinear decision surfaces which are suitable for solving nonlinearly separable problems. The hidden layers transform the input space by computing feature representations of the input attempting to make

the problem linearly separable. These feature representations are fed to the output layer which is essentially a linear classifier solving the problem βυ using linear decision surfaces in the new space. Therefore, feature representations make the problem easier for the linear classifier at the output layer to solve. Activation functions in the hidden units make the neural network able to construct functions with higher degree of nonlinearity; thus, the decision surfaces are more complex which makes the network more flexible into solving more challenging problems.

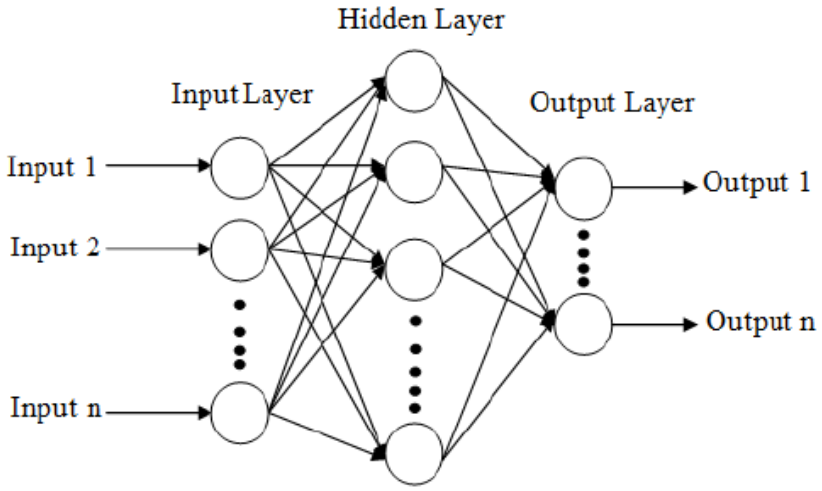


Figure 2.2: Structure of one hidden layer MLP.

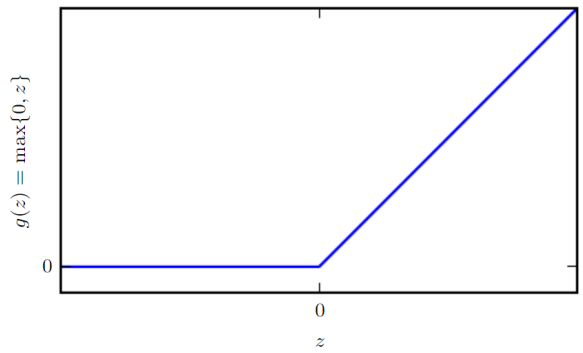
Considering the structure of an MLP, perceptrons are arranged into $k \geq 2$ layers. Let us consider a network M with k layers. The set of neurons is split into mutually disjunct subsets called layers $L_1, ..., L_k$. The network layers are stacked one onto each other, L_1 being the inut layer, $L_2, ...L_{k-2}$ being the hidden layers and L_k being the output layer. All edges are oriented in the direction from the input layer L_1 towards the output layer L_k. Each neuron in layer L_i is connected to every neuron in layer L_{i+1}. In other words, all neighboring layers form complete bipartite graphs. The output of the network is computed sequentially, layer by layer, starting with the input layer by directly assigning $\boldsymbol{y}^0 = \boldsymbol{x}$, $\boldsymbol{x}^0 = \boldsymbol{y}^{-i-1}$ for the layer L_i and $\boldsymbol{y}^{-k}$ for the output layer L_k. The weights and the activation function are given by the network, thus the output of each layer depends only on the output of the previous layer.

Rectified Linear Units

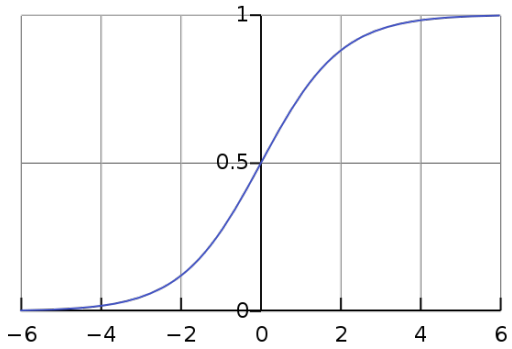
As we previously discussed in 2.1.1 with the introduction of the concept of the hidden layer, it is required to choose an activation function that will be used to compute the hidden layer values. In modern neural networks, the default recommendation is to use the rectified linear unit, or ReLU [9], defined as:

$$g(z) = max(0, z)\,. \tag{2.4}$$

Applying this function which is illustrated in figure 2.3a to the output of a linear transformation yields a nonlinear transformation. The function remains very close to linear, however, in the sense that is a piecewise linear function with two linear pieces.



(a) The rectified linear activation function (RELU).



(b) Graph of the Logistic function.

Figure 2.3: Graphs for the logistic and rectified linear activation functions.

The necessity for rectified linear units appeared due to the fact that logistic and hyperbolic tangent networks suffer from the vanishing gradient problem, where the gradient essentiallly becomes 0 after a certain amount of training, which stops all learning in that section of the network. ReLU units are only 0 gradient on one side, thus they produce a strong signal during the training procedure. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods, whereas they also preserve many of the properties that make linear models generalize well.

2.2 Neural Network Training

In this thesis, we will focus on training examples from labelled datasets. Such datasets consists of input data and their corresponding labels witch are expected as network outputs. The process of learning from a labelled dataset is referred as supervised learning. The aim of the learning process is to find the optimal parameters and structure of the network for solving the given task. Before the training process starts, network parameters need to be initialized. Initial values are often chosen randomly, however using some heuristics may lead to a faster parameter adjustment towards the optimal values. Learning is then carried out on the training set by feeding the training data through the network. It is an iterative process, where the outputs produced on each input from the training set are analyzed and the network is repeatedly being adjusted to produce better results. The network is considered to be trained after reaching the target performance on the training data.

2.2.1 Cost function

In order to evaluate the abilities of a machine learning algorithm, designing an aquantitative measure of its performance is needed. For tasks such as classification, the measurement of the model's accuracy is being obtained. Accuracy is the proportion of examples for which the model produces the correct output. The equivalent information measures the error rate, i.e the proportion of examples for which the model produces an incorrect output. The most common approach on modern networks is maximizing the log-likelihood for whom a given set of parameters θ of the model, can result in a prediction of the correct class at each input sample.

The output of the model $y = (z)$ can be interpreted as a probability y that input z belongs to one class $(t = 1)$, or probability $1–y$ that z belongs to the other class $(t = 0)$ in a two-class classification problem. We note this down as: $P(t = 1|z) = (z) = y$. The neural network model will be optimized by maximizing the likelihood that a given set of parameters of the model can result in a prediction of the correct class of each input sample. The parameters θ transform each input sample i into an input to the logistic function z_i. The likelihood maximization can be written as:

$$\underset{\theta}{\operatorname{argmax}} \mathcal{L}(\theta|t, z) = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{n} \mathcal{L}(\theta|t_i, z_i). \tag{2.5}$$

The likelihood $\mathcal{L}(\theta|t, z)$ can be rewritten as the joint probability of generating t and z

given the parameters $\theta : P(t, z|\theta)$. Since $P(A, B) = P(A|B)\square P(B)$, this can be written as $P(t, z|\theta) = P(t|z, \theta)P(z|\theta)$. Since we are not interested in the probability of zz we can reduce this to: $\mathcal{L}(\theta|t, z) = P(t|z, \theta) = \prod_{i=1}^{n} P(t_i|z_i, \theta)$. Since t_i is a Bernoulli variable, and the probability $P(t|z) = y$ is fixed for a given we can rewrite this as:

$$\begin{aligned} P(t|z) &= \prod_{i=1}^{n} P(t_i = 1|z_i)^{t_i} * (1 - P(t_i = 1|z_i))^{1-t_i} \\ &= \prod_{i=1}^{n} y_i^{t_i} * (1 - y_i)^{1-t_i} \end{aligned} \tag{2.6}$$

Since the logarithmic function is a monotone increasing function we can optimize the log-likelihood function $\underset{\theta}{\text{argmax}}\ log\mathcal{L}(\theta|t, z)$. This maximum will be the same as the maximum from the regular likelihood function. The log-likelihood function can be written as:

$$\begin{aligned} log\mathcal{L}(\theta|t, z) &= log \prod_{i=1}^{n} y_i^{t_i} * (1 - y_i)^{1-t_i} \\ &= \sum_{i=1}^{n} t_i log(y_i) + (1 - t_i) log(1 - y_i) \end{aligned} \tag{2.7}$$

Minimizing the negative of this function (minimizing the negative log likelihood) corresponds to maximizing the likelihood. This error function $\xi(t, y)$ is known as the cross-entropy error function or the log-loss:

$$\begin{aligned} E(t, y) &= -log\mathcal{L}(\theta|t, z) \\ &= -\sum_{i=1}^{n} [t_i log(y_i) + (1 - t_i) log(1 - y_i)] \\ &= -\sum_{i=1}^{n} [t_i log(\sigma(z) + (1 - t_i) log(1 - \sigma(z))] \end{aligned} \tag{2.8}$$

In multi-class classification, there is some need for probability distribution over a discrete variable with n possible values, which corresponds to n different classes. The 2.8 function can be generalized as:

$$L(t_i, y_i) = -\sum_{i=1}^{C} t_i log(y_i) \tag{2.9}$$

where C is the number of classes, in order to output a multiclass categorical probability distribution by the softmax function. The softmax function takes a C-dimensional vector z as input and outputs a C-dimensional vector y of real values between 0 and

1. This function is a normalized exponential and is defined as:

$$y_c = softmax(\mathbf{z})_c = \frac{e^{z_c}}{\sum_{d=1}^{C} e^{z_d}} \quad \text{for } c = 1 \cdots C. \tag{2.10}$$

Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature, where we normalize over unnormalized probability distributions. Here, the unnormalized probability distribution is z_c which is the preactivation of the output layer neurons. Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish. In particular, squared error is a poor loss function for softmax units and can fail to train the model to change its output, even when the model makes highly confident incorrect predictions (Bridle, 1990).

2.2.2 Back Propagation

When we use a feedforward neural network to accept an input x and produce an output $\hat{y}$, information flows forward through the network. The input x provides the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{y}$. This is called forward propagation. During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$. The back-propagation algorithm [4] allows the information from the cost to then flow backward through the network in order to compute the gradient.

Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure. The term back-propagation is often misinterpreted as the whole learning algorithm for multi layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function.

We will describe how to compute the gradient $\nabla_x f(x, y)$ for an arbitrary function f, where x is a set of variables whose derivatives are desired, and y is an additional

Algorithm 2.1 Backpropagation learning algorithm

for d in data **do**

Forwards Pass

Starting from the input layer, use eq. ?? to do a forward pass trough the network, computing the activities of the neurons at each layer.

Backwards Pass

Compute the derivatives of the error function with respect to the output layer activities

for layer in layers **do**

Compute the derivatives of the error function with respect to the inputs of the upper layer neurons

Compute the derivatives of the error function with respect to the weights between the outer layer and the layer below

Compute the derivatives of the error function with respect to the activities of the layer below

end for

Updates the weights.

end for

set of variables that are inputs to the function but whose derivatives are not required. In learning algorithms, the gradient we require is the gradient of the cost function with respect to the parameters $\nabla_\theta J(\theta)$. We start by deriving the gradient of the loss at the output of the neural network. We use the softmax activation of the ith output unit as described in 2.10, and the cross entropy error function for multi-class output as stated in 2.9.

Starting from the output layer the derivation of the classification loss would be given by:

$$\frac{\partial}{\partial f(x)_{ij}}[-\log(f(x)_y)] = -\frac{1_{y=c}}{f(x)_y} \tag{2.11}$$

where $1_{y=c}$ is the indicator function, c a given class and y the true class of x. The numerator is multiplied by the indicator function because if $y \neq c$, then $-logf(x)_y$ is constant with respect to $f(x)_c$. Hence its gradient is given by:

$$\nabla_{f(x)}[-\log(f(x)_y)] = \frac{-e(y)}{f(x)_y} \tag{2.12}$$

where $e(y)$ is the one-hot vector of y which is everywhere 0 except the position where

$y = c$. The derivative gradient of the loss with respect to the preactivations of the input layer is given by:

$$\frac{\partial}{\partial a^{(L+1)}(x)_c}[-\log(f(x)_y)] = f(x)_c - 1_{y=c} \tag{2.13}$$

It is the same formula as in the case of the logistic output units. The values themselves will be different, because the predictions y will take on different values depending on whether the output is logistic or softmax, but this is an elegant simplification. The same procedure as before could be followed to obtain an analytic expression for each neuron of each hidden layer, but in this case, things are getting complicated at the lower layers of the network. Thus, the chain rule of calculus is used. It obtains the gradients of each layer in a highly efficient way. With the use of the chain rule the gradient for weights in the top layer would be:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} \tag{2.14}$$

By recursively computing the gradient of the error with respect to the activity of each neuron, we can compute the gradients for all weights in a network.

2.2.3 Optimization techniques

The goal of a machine learning algorithm is to minimize the expected generalization error, or equivalently to minimize the corresponding objective function where the expectation is taken across the data-generating distribution. This quantity is known as *risk*. If we knew the true distribution of the data, risk minimization would be an optimization task solvable by an optimization algorithm. The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This replaces the true distribution $p(x, y)$ with the empirical distribution $\overline{p}(x, y)$ defined by the training set. We now minimize the empirical risk. The training process based on minimizing this average training error is known as empirical risk minimization.

Gradient Descent

Feedforward networks can be seen as efficient nonlinear function approximators based on using gradient descent to minimize the error in a function approximation. The gradient descent optimization method calculates, as the name suggests, the gradient

of the function containing the optimization parameters. The parameters are then updated in the direction where the magnitude of the gradient descend is the steepest. This is optimally iterated until the magnitude of the gradient is close to zero which means that the parameter space has reached a local minimum. In reality the optimization is terminated when the loss is below a threshold value. The parameter θ_t is updated according to:

$$\theta_t = \theta_{t-1} - \alpha f(\theta_{t-1}) \tag{2.15}$$

where f is the function containing the parameters and α is the step size for the optimization, which is often called learning rate. It determines how quickly the θ parameters are updated. Gradient descent has the shortcoming that update of parameters is always exactly proportional to change of gradient. This might become a problem when the gradient change slows down. Another downfall of this method is that the whole data set has to be processed to perform one update of the parameters.

Stochastic Gradient Descent

To improve the optimization the Stochastic Gradient Descent (SGD) [10] method is able to update the parameters in every iteration according to:

$$\theta_t = \theta_{t-1} - \alpha f(\theta_{t-1}; x_i; y_i) \tag{2.16}$$

This method could improve the convergence rate as opposed to the gradient descent method, but also make it fluctuate more and it may have trouble finding the exact minimum.

Mini-batch optimization

A compromise of these two methods is the minibatch optimization method. The minibatch optimization uses the data samples of a small batch and computes the gradient as an average of the gradients of each data sample according to:

$$\theta_t = \theta_{t-1} - \alpha f(\theta_{t-1}; x_{i:i+n}; y_{i:i+n}) \tag{2.17}$$

where n is the size of the batch.

Momentum

Stochastic Gradient Descent has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common

around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum. Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_\theta f(\theta) \\ \theta_t &= \theta_{t-1} - v_t \end{aligned} \tag{2.18}$$

where α is the learning rate, γ is the momentum constant and v is the updated vector. Essentially, when using momentum, it can be interpreted like pushing a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster on the way, until it reaches its terminal velocity, if there is air resistance, i.e. $\gamma < 1$. The same happens to the parameter updates. The momentum term increases for dimensions whose gradient's point is in the same directions and reduces updates for dimensions whose gradients change directions. As a result, it gains faster convergence and reduced oscillation.

ADAM

The ADAM (Adaptive Moment Estimation) optimization [11] is a development of the momentum optimization which adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. This is done by using two momentum constants β_1 and β_2 and two momentum vectors m_t and v_t. Then the weight update in each step is:

$$g_t = \nabla_f(\theta) \tag{2.19}$$

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{2.20}$$

where g_t^2 should be interpreted as the element-wise multiplication of the gradients and m_t, v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. As m_t and v_t are initialized as zero's vectors, the authors of ADAM observe that they are biased towards zero, especially during the initial time steps or when the decay rates are small. They counteract these

biases by computing bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.21}$$

They then use them to update the parameters:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.22}$$

The steps of ADAM are described in the following way:

- m_t is the first order momentum and is calculated as the convex combination of the previous first order momentum and the gradients. This can be compared to the momentum 2.18.
- v_t is the second order momentum and is calculated in a similar way as the first order momentum as a convex combination of the previous second order moment and the element wise multiplicated gradients.
- $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected momentums. These operations are done to decrease the influence of the bias arisen from the initializations of m_t and v_t. Since β_1, the influence of this operation, will decrease as the number of iterations increases then $\hat{m}_t$ converges to m_t and $\hat{v}_t$ converges to v_t.
- The parameters θ are then updated using the bias-corrected momentums and the step size λ, The term ϵ is used to prevent division with zero.

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and $\lambda = 10^{-8}$. They show empirically that ADAM works well in practice and compares favorably to other adaptive learning-method algorithms.

2.2.4 Regularization

The ability to learn is the key concept of neural networks. A substantial problem in machine learning is how to create an algorithm that performs well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known as regularization. In the context of modern neural netwoks, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias.

L2 regularization

L2 regularization is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the loss function. That is, for every weight w in the network, we add the term $\lambda\frac{1}{2}\boldsymbol{w^2}$ to the loss function:

$$L'(\boldsymbol{w}) = L(\boldsymbol{w}) + \lambda\frac{1}{2}\boldsymbol{w^2} \tag{2.23}$$

where E is the loss function and λ is the regularization strength. It is common to see the factor of $\frac{1}{2}$ in front because then the gradient of this term with respect to the parameter w is simply λw instead of $2\lambda w$. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. It has the appealing property of encouraging the network to use all of its inputs a little; rather that some of its inputs a lot.

L1 regularization

L1 regularization is another relatively common form of regularization, where for each weight w in the network, we add the term $\lambda|w|$ to the loss function:

$$L'(\boldsymbol{w}) = L(\boldsymbol{w}) + \lambda|w| \tag{2.24}$$

It is possible to combine the L1 regularization with the L2 regularization $\lambda_1 \mid w \mid +\lambda_2 w^2$, which is also referred to as Elastic net regularization [12]. The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization, i.e. very close to exactly zero. In other words, neurons with L1

regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the noisy inputs.

Dropout

A very effective, yet simple, way of regularization method for modern neural networks was introduced in 2014 [13]. In a sense it helps to view dropout as a form of ensemble learning. In ensemble learning we use a number of weaker classifiers, train them separately and then at test time we use them by averaging the responses of all ensemble members. Since each classifier has been trained separately, it has learned different aspects of the data and their mistakes are different. Combining them helps to produce a stronger classifier, which is less prone to overfitting. Random Forests or Gradient Boosting Tree's are typical ensemble classifiers. One ensemble variant is bagging, in which each member of the ensemble is trained with a different subsample of the input data, and thus has learned only a subset of the whole possible input feature space.

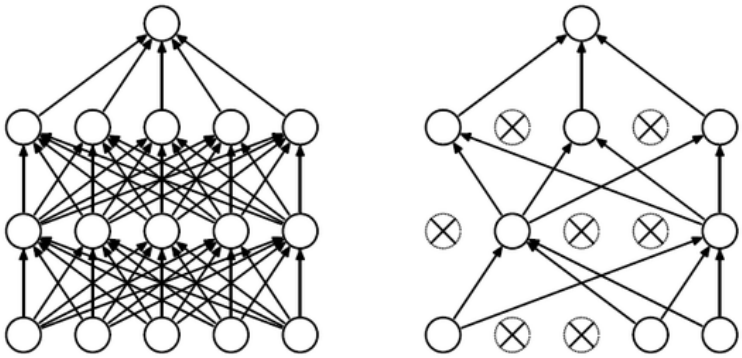


Figure 2.4: Dropout Regularizartion method.

Dropout can be seen as an extreme version of bagging. At each training step of a batch, the dropout procedure creates a different network by randomly removing some units, which is trained using backpropagation. Conceptually, then, the whole procedure is a kin to using an ensemble of many different networks, which were created at each step, with each one trained with a single sample. At test time the whole network uses all units but with scaled down weights for each of them.

2.3 Convolutional Neural Networks (CNNs)

Convolutional networks, introduced by Yann LeCun in 1989 [5], are a type of neural network specialized in processing data that has a known grid-like topology. Data with grid-like topology includes time-series, which can be thought of as a 1-D grid, taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. Convolutional neural networks have gained a special status over the last few years as an especially promising form of deep learning. The fundamental difference between fully connected and convolutional neural networks is the pattern of connections between consecutive layers. In the fully connected case, as the name might suggest, each unit is connected to all of the units in the previous layer In a convolutional layer of a neural network, on the other hand, each unit is connected to a, typically small, number of nearby units in the previous layer. Furthermore, all units are connected to the previous layer in the same way, with the exact same weights and structure. This leads to an operation known as convolution, giving the architecture its name.

It has been popular to describe neural networks in general, and specifically convolutional neural networks, as biologically inspired models of computation. At times, claims go as far as to state that these mimic the way the brain performs computations. Convolutional neural networks kind of follow that pattern. Each convolutional layer looks at an increasingly larger part of the image as we go deeper into the network. Most commonly, this will be followed by fully connected layers that in the biologically inspired analogy act as the higher levels of visual processing dealing with global information. Convolutional Neural Networks have introduced some great novelties like parameter sharing, local connectivity and sampling layers which will be described in the next setions.

2.3.1 Convolution

The name convolutional neural network indicates that the network employs a mathematical operation called convolution. It is defined as the integral of the product of two functions from which the one shifts within a domain, and it is typically denoted with an asterisk:

$$y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau \tag{2.25}$$

We can interpret the convolution operation as thinking of y as a smooth estimation of x which shifts in a domain t where h is a weighted average operation in a τ domain. Convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages. In convolutional network terminology, the first argument, x, is often referred to as the input, and the second arguement h is referred as the kernel. The output is sometimes referred as the feature map. When working with data on a computer we will have to discritize the operation, the integral of 2.25 will then be:

$$y(t) = x(t) * h(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)h(t-\tau) \tag{2.26}$$

For image data there is usually a use of convolutions over more than one axis at a time. For example, when using a two-dimensional image I as input and a two-dimensional kernel K, the convolution is denoted as:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.27}$$

Convolution is commutative, meaning it can equivalently be written as:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n K(i-m, j-n)I(m,n) \tag{2.28}$$

The commutative property of convolution arises because of the flipped kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. On the contrary, many neural network libraries implement a related function called the cross-correlation:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.29}$$

Many machine learning libraries implement cross-correlation but call it convolution. In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel in the appropriate place, so an algorithm based on convolution with kernel flipping will learn a kernel which is flipped relative to the kernel learned by an algorithm without the flipping.

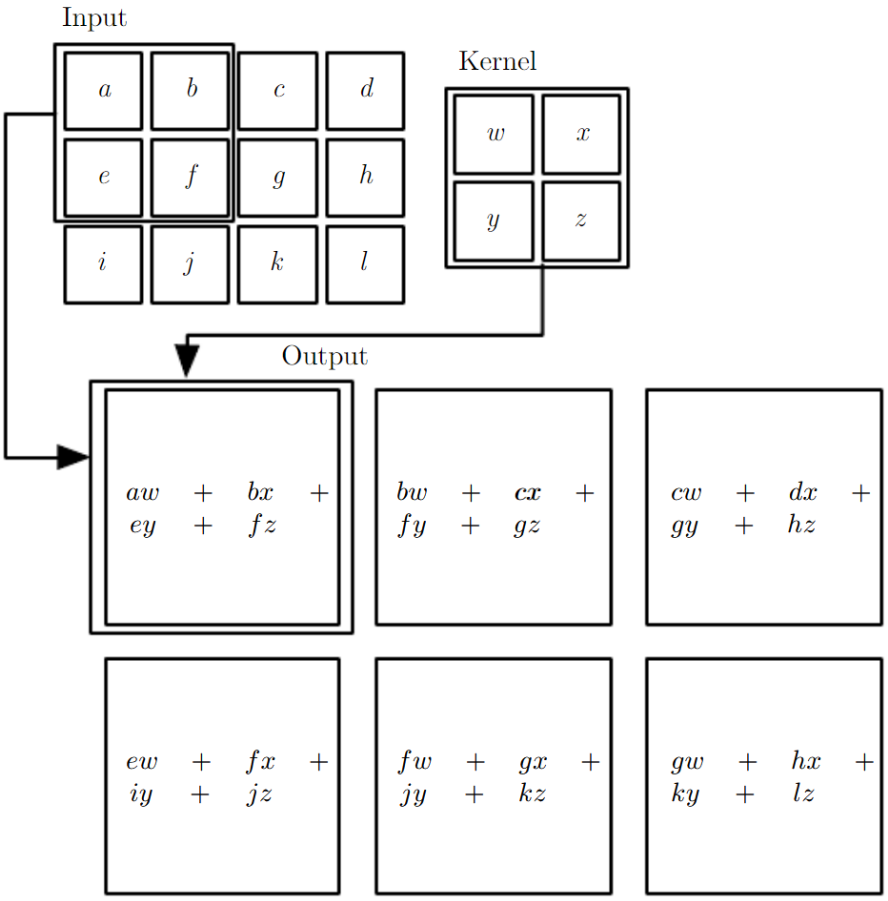


Figure 2.5: 2-D convolution. Figure reproduced from [1].

Discrete convolution can be viewed as multiplication by a matrix, but the matrix has several entries constrained to be equal to other entries. For a univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. In two dimensions, a block circulant matrix corresponds to convolution. In addition to these constraints that several elements are equal to each other, convolution usually corresponds to a very sparse matrix. An example of a discrete convolution is given in figure 2.5.

2.3.2 Local connectivity

Traditionally the fully-connected network architectures that were discussed in the previews sections use matrix multiplication by a matrix of parameters with a separate parameter, describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. When dealing with high-dimensional inputs such as images it is impractical to connect every

neuron to all neurons in the previous volume. On the contrary, Convolutional neural networks connect each neuron to only a local region of the input volume. This is accomplished by making the kernel smaller than the input. This means that it stores fewer parameters, which reduces the memory requirements of the model, computing the output requires fewer operations and improves its statistical efficiency. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron. The receptive field for a layer is the part of the input that a single pixel in the output of that layer depends on.

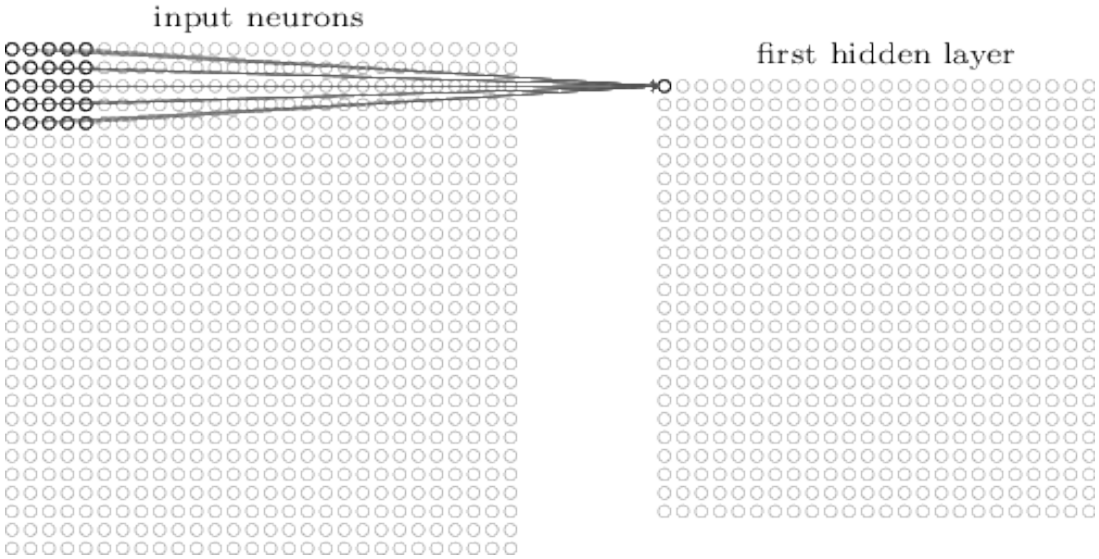


Figure 2.6: Receptive field of a Convolutional Neural Network.

In a Convolutional Neural Network, the receptive field of a layer is the spatial dimensions of the filter kernel. In figure 2.6 we present an example where every hidden layer neuron has a local receptive field of region 5x5 pixels. The receptive field for the entire network is the region of the input of the network that affects a single pixel in the network output. By making the network deeper, the receptive field is increased linearly. Due to the properties of the forward and backward pass in the loss function the gradient of the central pixels of the receptive field has a larger magnitude. The distribution of the impact of the receptive field is Gaussian and decays rather quickly from the center yielding an effective receptive field that is only a small part of the theoretical receptive field [14]. Moreover, in convolutional neural networks a hidden unit is connected to all input channels, for example in RGB images a hidden unit has connections with three different channels, while in gray-scale images it has connections with a single channel. Different hidden units are

connected to different patches of the input image, such that the image is covered with the receptive fields for every hidden unit. All the above describe the connectivity of each neuron in a Convolutional layer. There are three important hyperparameters for the implementation of a convolutional layer, the size of the output volume, stride and zero-padding.

- The depth of the output volume corresponds to the number of filters we would like to use. Each filter is learning to look for something different from the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color.

- The stride is the value based on which we slide the filter at every step of the convolution. When setting the stride at 1 then the filters move one pixel at a time. When the stride is 2 then the filters jump 2 pixels at a time as they slide around. This will produce smaller output volumes spatially.

- The procedure of Zero-padding is adding a layer of pixels with zero value all the way around the input image. Zero-padding preserves the information at the image borders. If we use convolutional layers without zero-padding, the size of the feature maps is reduced after each convolutional layer and the information at the image borders is trimmed and cannot be exploited.

2.3.3 Parameter sharing

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. As it was mentioned before, during a convolution operation a kernel slides through the whole input, thus each member of the kernel is used at every position of the input, which means that rather than learning a separate set of parameters for every location, we learn only one set, which is the kernel. This does not affect the runtime of forward propagation, which still is $O(kxn)$, but it does further reduce the storage requirements of the model to k parameters. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property, called equivariance to translation. This property means that when the input changes, the output changes in the same way, more specifically, a function f is equivarient to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, i.e shifts it, then the convolution function is equivariant to g. For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x, y) = I(x-1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I, then applied the transformation g to the output. Equivariance means that if we move an object in an image, its detected features will be moved by the same amount at the feature map. This suggests that if we have a kernel that detects, for example horizontal edges and convolve an image with this kernel, in the output feature map, edges will be detected in all possible positions. Thus, in each feature map specific features based on the kernel that is used are detected. Equivariance means that if we move an object in an image, its detected features will be moved by the same amount at the feature map. This suggests that if we have a kernel that detects, for example horizontal edges and convolve an image with this kernel, in the output feature map, edges will be detected in all possible positions. Thus, in each feature map specific features based on the kernel used are detected.

2.3.4 Subsampling Layer

A typical layer of a convolutional network consists of three stages. In the first stage, the layer performs several convolutions at the same time to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. The third stage is the pooling or subsampling function, which replaces the output of the network at a certain location with a summary statistic of the nearby outputs, i.e pooling is a way of reducing the dimension of the input by choosing the most responsive node of a given interest region. The two most usual pooling operations are max pooling and average pooling which operate as shown in figure 2.8.

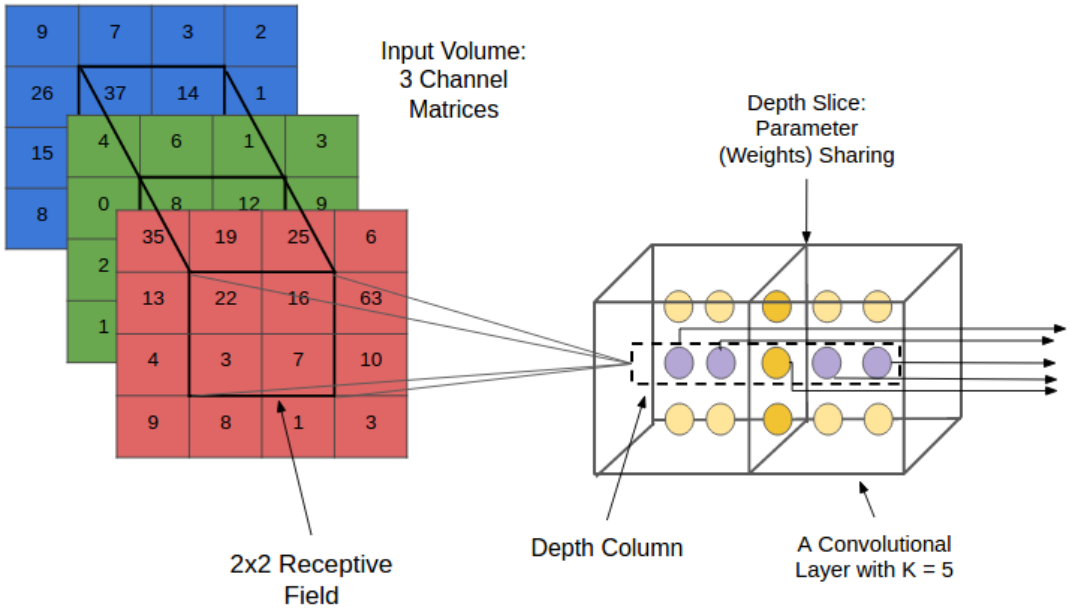


Figure 2.7: Parameter Sharing in Convolutional Neural Networks.

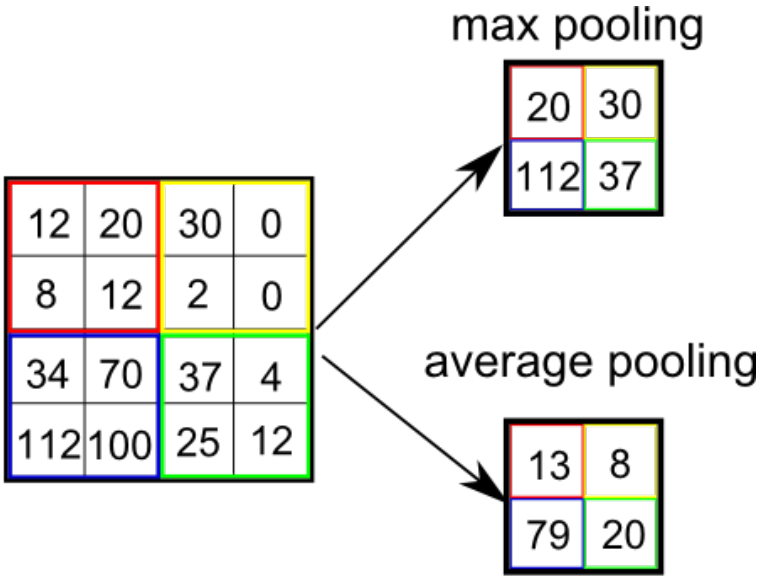


Figure 2.8: Max and Average pooling.

The max pooling operation (Zhouand Chellappa, 1988) reports small local neighborhoods of feature maps, and computes the maximum of the activations on each patch. The new pooled feature map of the max pooling layer is constructed by replacing the neighborhood pixel of the feautre map with their respective maximum. The max pooling operation is defined as:

$$h_{ijk}^{(n)}(x) = \underset{p,q}{max} h_{(i,j+p,k+q)}^{(n-1)}(x) \tag{2.30}$$

where p and q are indexes of a patch of the feature map $h_i^{(n-1)}$ of the previous layer, which is a convolutional layer and $h_{ijk}^{(n)}(x)$ is the corresponding pooled value for the i-th resulting feature map at position (j, k) of the current max pooling layer. Max pooling is performed in non overlapping regions which means that if we pool $l \times l$ regions, neighboring regions are l pixels apart which results in feature maps with reduced size.

Another pooling operation is average pooling. It is the same concept as max pooling though instead of the maximum it computes the average of every local neighborhood. The average pooling operation is defined as:

$$h_{ijk}^{(n)}(x) = \frac{1}{m^2} \sum_{p,q} h_{(i,j+p,k+q)}^{(n-1)}(x) \tag{2.31}$$

In all cases pooling helps with a significant reduce of the size of a feature map, additionally if one chooses the pooling regions to be contiguous areas in the image and only pools features are generated from the same hidden units, these pooling units will then be *translation invariant*. This means that the same pooled feature will be active even when the image undergoes small translations. Translation-invariant features are often desirable in many tasks. For example, if you were to take a classifier which needs to classify images of digits, and translate a digit left or right, you would want your classifier to still accurately classify it as the same digit regardless of its final position. An example is given in figure 2.9.

2.3.5 Architecture of a CNN

A convolutional network consists mainly of convolutional, pooling and fully connected layers. The model is composed of two main parts, the feature extraction part and the classification part. The feature extraction part consists of convolutional followed by pooling layers, though pooling layers are not necessary to be placed after each convolutional layer. In this part the network learns represantations of the given dataset, which provides discrimination across different classes. Those representations are learned through the learning process of the convolutional kernels with backpropagation.

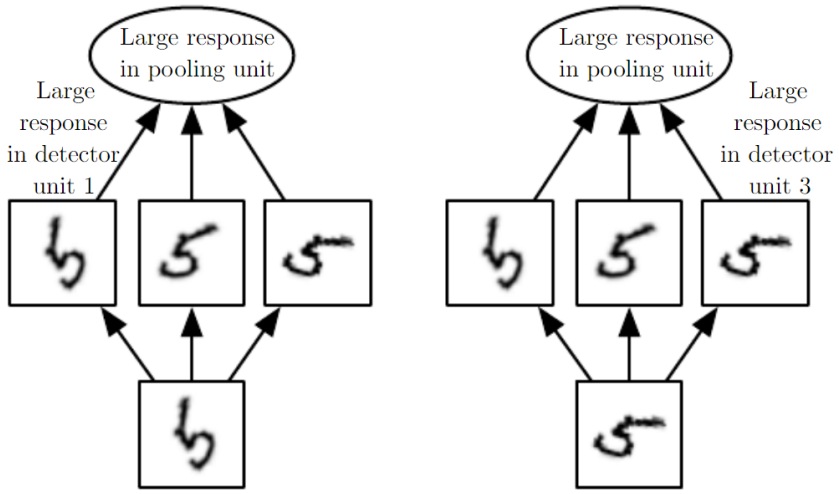


Figure 2.9: Example of learned invariances. A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation.

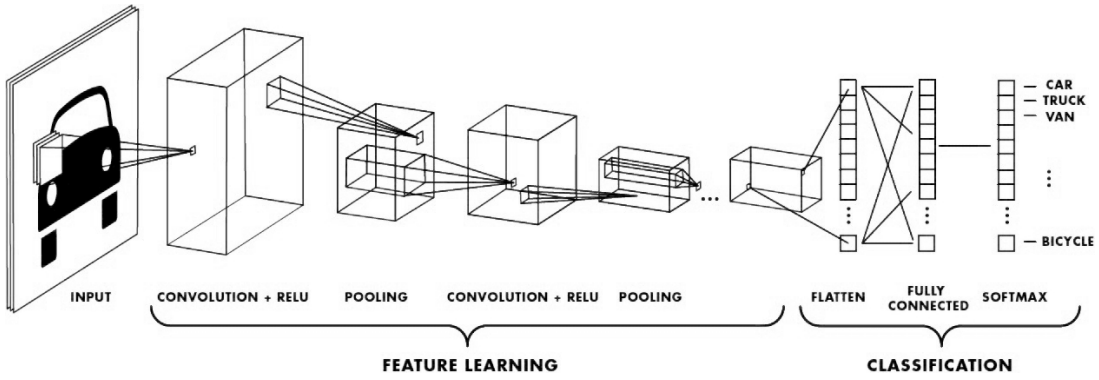


Figure 2.10: Example of a typical Convolutional Neural Network structure.

The classification part of the model is where the network uses the learned features from the previous part to classify the data. This part consists of a fully connected neural network which has as an input the vectorized output of the last convolutional's or pooling's layer. At the output of the fully connected layer a softmax activation layer which gives the normalized class probabilities as an output. A forward propagation in a Convolutional neural network consists of providing an image as input to the convolutional network and flowing the structure described above, in order to finally compute class predictions at the output layer of the network as illustrated in the figure 2.10. There are also alternative architectures [15] proposed where the deep convolutional networks are first trained using supervised objectives to learn good invariant

hidden latent representations and then feed the corresponding hidden variables of data samples into linear or kernel SVM's instead of a softmax layer.

Convolutional Neural Networks learn hierarchical feature representations, which means that they learn more complicated concepts through simpler features. The first convolutional layers learn simple features such as edges, the next layers combines these edges to construct more complicated features such as parts of an object and as we go on, they add more levels of abstraction, and eventually in the last convolutional layers highly abstract features are obtained. This happens because a convolutional layer computes each feature map as a weighted combination of the input channels. Hence, the convolutional kernels are learned in a way, such that the weighted combination of the input channels can provide meaningful higher level representations.

As for the convolutional layer's sizing patterns there are some hyperparameters for which common rules of thumb are followed in order to adjust them. First of all the input layer (that contains the image) should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512. The convolutional layers should be using small filters e.g. 3x3 or at most 5x5, using a stride of 1, and crucially, padding the input volume with zeros in such way that the convolutional layer does not alter the spatial dimensions of the input. The pool layers are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields and with a stride of 2.

CNN case studies

- LeNet-5 [16], illustrated in figure 2.11, which was developed by Yann LeCun in 1990s is a pioneering convolutional network architecture that was used to read zip codes, digits, etc.

- AlexNet [8] was the first work that popularized Convolutional Networks in Computer Vision, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up. The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other.

- GoogLeNet [17]; the ILSVRC 2014 winner. Its main contribution was the de-

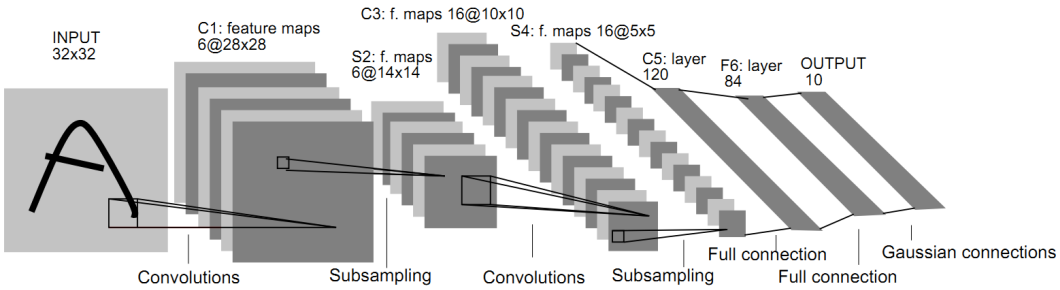


Figure 2.11: The pioneering convolutional architecture LeNet-5.

velopment of an Inception Module that dramatically reduced the number of parameters in the network.

- VGGNet [18] developed by Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution was in showing that the depth of the network is a critical component for good performance.

- Residual Network [19] developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special skip connections and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network. Residual Networks are currently the state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice.

Chapter 3

Functionally Weighted Convolutional Neural Networks (FWCNNs)

3.1 Introduction

One striking fact about Artificial Neural Networks is that they are known as universal approximators. i.e they are capable of approximating any measurable function to any desired degree of accuracy. More precisely, in [20] there is a proof which states that for any continuous function f on a compact set K, exists a feedforward neural network having only one hidden layer, which uniformly approximates f within an arbitrary $\epsilon > 0$ on K. This property gives Artificial Neural Networks no theoritical constraint for the success of accuracy, considering that they are applied at both classification and regression. Any lack of success of the network is due to inadequate learning, insufficient number of hidden units or the lack of a deterministic relationship between the input and the target.

There has been a vast literature of neural network architectures and approaches trying to solve problem of over-training a network which leads to poor generalization.

Empirical observation has shown that given two networks with similar performance on a training dataset, the one with the fewer parameters is likely to generalize better. Several techniques have been developed aiming to improve the networks generalization; for instance weight decay and weight bounding. These methods make use of an additional set, the validation set, i.e they partition the data into three subsets; the the training, the validation and the test subset. Training may end up in several candidate models i.e networks with different weight values. Then the validation set is used to pick the best performing model, and finally the test set will give the estimation for the size of the expected error on unseen data. Another framework for modeling, [21] is through sparse supervised learning models which utilize only a subset of the training data, discarding unnecessary samples based on certain criteria. Sparce models widely used are, among others, the Lasso [22], the Support Vector Machines (SVMs) [23] and the Revelance Vector Machines (RNMs) [24]. Sparce learning applies L_1 regularization, leading to penalized regression schemes, or sparse priors on the model parameters under the Bayesian framework [25]. Another popular regularization method within the modern neural networks is Dropout [13]. This approach employs a stohastic procedure for node removal during the training phase, reducing model parameters in order to avoid overfitting.

3.2 Functionally Weighted Neural Networks (FWNNs)

In this section we introduce a new type of neural network, the Functionally Weighted Neural Networks, with weights that depend on a continuous variable, instead of the traditional discrete index. This network employs a small set of adjustable parameters and at the same time employs a superior generalization performance as indicated by an ample set of numerical experiments. The idea of replacing the indexed parameters with continuous functions has been previously considered in [26], through a different setting and with different rather limited implementation settings. However since then, no further followups have been spotted in the literature. In order to give a loose definition behind the concept of our proposed method we may compare the definition of a traditional MLP layer which is given in 3.1.

$$f(x) = \sum \pi\phi(x;\theta) \tag{3.1}$$

Equation 3.1 describes the sum of the activations of every hidden unit at any hidden layer of an MLP. In comparison, at an FWNN we now define one *infinite* hidden unit at any given hidden layer, turning the sum of the hidden units into an integral for the new defined layer.

$$f(x) = \int \pi(s)\phi(x;\theta(s))ds \tag{3.2}$$

Radial basis functions (RBF) are known to be suitable for function approximation. An RBF network with K Gaussian activation nodes can be written as:

$$\begin{aligned} N_{RBF}(\boldsymbol{x};\theta) &= \pi_0 + \sum_{j=1}^{K} \pi_j \phi(\boldsymbol{x};\boldsymbol{\mu_j},\sigma_j) \\ &= \sum_{j=1}^{K} \pi_j exp\left(-\frac{|\boldsymbol{x}-\boldsymbol{\mu_j}|^2}{2\sigma_j^2}\right), \end{aligned} \tag{3.3}$$

where $\boldsymbol{x}, \boldsymbol{\mu_j} \in R^n$ and $\theta = \{\pi_j, \boldsymbol{\mu_j}, \sigma_j\}_{j=1}^{K}$ denotes collectively the network parameters to be determined via the training procedure. The total number of parameters is given by the expression:

$$N_{var}^{RB} = K(2+n)+1, \tag{3.4}$$

which grows linearly with the number of network nodes. Thus we define the Functionally weighted Neural Network for the RBF activations in correspondence to 3.4 as:

$$N_{FW}(\boldsymbol{x};\theta) = \int_{-1}^{1} \frac{ds}{\sqrt{1-s^2}} \pi(s) exp\left(-\frac{|x-\mu(s)|^2}{2\sigma(s)^2}\right), \tag{3.5}$$

where we applied the following transitions:

$$\pi_j \longrightarrow \pi(s) \tag{3.6}$$

$$\boldsymbol{\mu_j} \longrightarrow \boldsymbol{\mu(s)} \tag{3.7}$$

$$\sigma_j \longrightarrow \sigma(s) \tag{3.8}$$

$$\sum_{j=1}^{K} \longrightarrow \int_{-1}^{1} \frac{ds}{\sqrt{1-s^2}} \tag{3.9}$$

The weight model-functions $w(s), \boldsymbol{\mu(s)}, \sigma(s)$ are parametrized and these parameters are collectively denoted by θ. We model the parameters as polynomial forms, denoted as:

$$\pi(s) = \sum_{j=0}^{L_\pi} \pi_j s^j \tag{3.10}$$

$$\boldsymbol{\mu}(s) = \sum_{j=0}^{L_\mu} \boldsymbol{\mu}_j s^j \tag{3.11}$$

$$\sigma(s) = \sum_{j=0}^{L_\sigma} \sigma_j s^j \tag{3.12}$$

Note that $\boldsymbol{\mu_j} \forall 0, ..., L_\mu$ and $\boldsymbol{\mu}(s)$ are vectors in R^n. Therefore, the set of adjustable parameters becomes:

$$\theta = \{\{\pi_j\}_{j=0}^{L_\pi}, \{\mu_{ij}\}_{i=1,j=0}^{n,L_\mu}, \{\sigma_j\}_{j=0}^{L_\sigma}\} \tag{3.13}$$

with a total parameter given by:

$$\begin{aligned} N_{var}^{FW} &= (1 + L_\pi) + n(L_\mu + 1) + (L_\sigma + 1) \\ &= L_\pi + nL_\mu + L_\sigma + n + 2 \end{aligned} \tag{3.14}$$

We may interpretate the proposed scheme as a network with infinite number of nodes, since the integral may be expressed as a sum of an infinite number of terms:

$$\int_a^b f(s)ds = \lim_{m \to \infty} \frac{b-a}{m} \sum_{i=1}^{m} f\left(a + i\frac{b-a}{m}\right) \tag{3.15}$$

In order to evaluate the integrals we use the following accurate Guass-Chebyshev quadrature:

$$\int_{-1}^{1} \frac{ds}{\sqrt{1-s^2}} f(s) \approx \frac{\pi}{M} \sum_{i=1}^{M} f(s_i) \tag{3.16}$$

where s is given by:

$$s_i = cos\left(\frac{2i-1}{2M}\pi\right). \tag{3.17}$$

Where M is the number of integration points. One may be misled and interpret the above sum as an RBF network with M Gaussian nodes. This is not true since this sum is only a numerical approximation scheme for evaluating the integral. The Remann approximation scheme defined in equation 3.15 is a sum of an infinite number of terms.

The Functionally Weighted Neural Network can be also modeled with a logistic sigmoid as activaton function, in correspondence to 2.3 which is denoted as:

$$N_{FW}(\boldsymbol{x}; \theta) = \int_{-1}^{1} \frac{ds}{\sqrt{1-s^2}} \pi(s) \frac{1}{1 + e^{-\boldsymbol{w}(s)x}} \tag{3.18}$$

where we can intepret $\pi(s)$ as a weighted average for the infinite activation functions where in analogy to the RBF's weights, its polynomial form is denoted as:

$$\pi(s) = \sum_{j=0}^{L_\pi} \pi_j s^j \tag{3.19}$$

and the $a(x)$ is the preactivations of the network:

$$\boldsymbol{w}(s)x = \sum_{k=1}^{d} w_j(s)x_j + w_0(s) \tag{3.20}$$

where $w_k(s)$ is a polynomial form denoted as:

$$w_k(s) = \sum_{k=0}^{L_w} w_{kj} s^j \tag{3.21}$$

with j denoting the degree of the $\pi(s)$. Again M is **not** the number of the hidden units, we only make use of one infinite hidden unit where M is the numerical approximation for evaluating the integral.

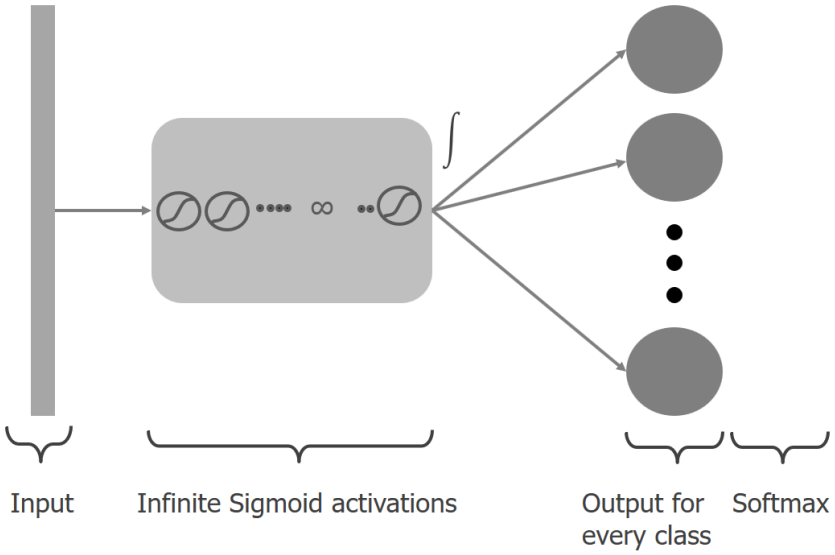


Figure 3.1: A typical FWNN architecture.

In figure 3.1 we illustrate an FWNN single hidden layer architecture for classification. In the figure we can observe from the amount of connections, that the proposed method requires only but a few weights for the hidden and output layers.

3.3 Functionally Weighted Convolutional Neural Networks (FWCNNs)

As we already stated convolutional layers are a state-of-the-art method for acquiring features from image data, though they demand a great deal of stacked layers in order to obtain satisfying results, thus our motivation for the FWCNN is to achieve the extraction of good features with the use of infinite size kernels. With an equivalent way of defining the Functionally Weighted Neural Network with a sigmoid activation function we study the same formulation for the FWCNN's, by having continuous 2-d kernels instead of the traditional discrete 2-d kernels that are used for the convolutional layers. Practicaly the only adjustment from FWNN is switching from matrix multiplications into convolutions operations, which changes the preactivation sum 3.20 into a convolution operation and modeling the vectorized weights from a 2-dimension kernel into one infinite kernel:

$$w_{(i,j)k}(s) = \sum_{k=0}^{L_w} w_{(i,j)kl} s^l \tag{3.22}$$

where we keep the weighted average of the activations $\pi(s)$ as it is, denoted at 3.19 and the model as it is denoted at 3.18.

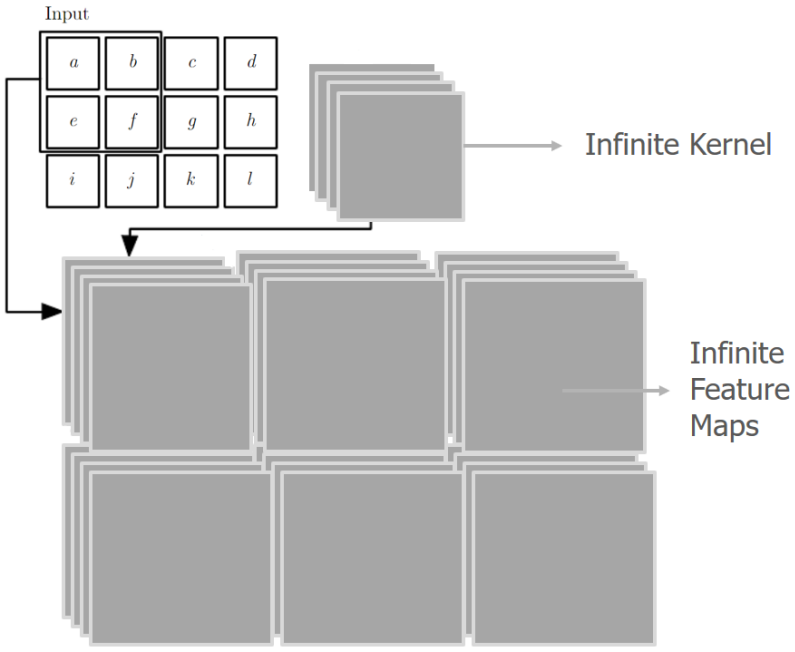


Figure 3.2: Convolution with an infinite modeled kernel.

In favor of strengthening the intuition behind the 2-dimension convolution of

functionally weighted convolutional neural networks; in figure 3.2 we illustrate the comparison between them and a traditional 2-dimension convolution as shown in figure 2.5. Notice we only use one infinite kernel on each layer of the proposed architecture.

The remarkable leverage of our method is taking advantage of the parameter sharing property from the traditional convolutional neural networks and it unifies it with the properties of an FWNN, having as a result a massive minimization in the number of parameters of the layer. Furthermore it combines the FWNN architecture at the output layer which gives us the ability to discard entirely the need for a traditional neural network at the classification part of the network. Note that the biggest portion of parameters of a CNN is at the classification part where an immense number of feature maps is being vectorized in order to feed the classifier. The total number of parameters on a FWCNN is given by:

$$N_{var}^{FWCNN} = n(L_\pi + 1) + (L_w + 1) + K_{size} + C_{size} + N + I \tag{3.23}$$

Where K is the size of the kernel used for the convolution, C is the number of the input channels (e.g 1 if the input is a grayscale image, 3 if the input is an RGB image), N is the number of classes and I is the dimension of the input image. Note that the I parameters concern the preactivation which feeds the output layer of our network meaning that after performing the convolution we can add a max pooling layer therefore we trim at half the number of parameters of I in consequence of reducing the dimension of the input.

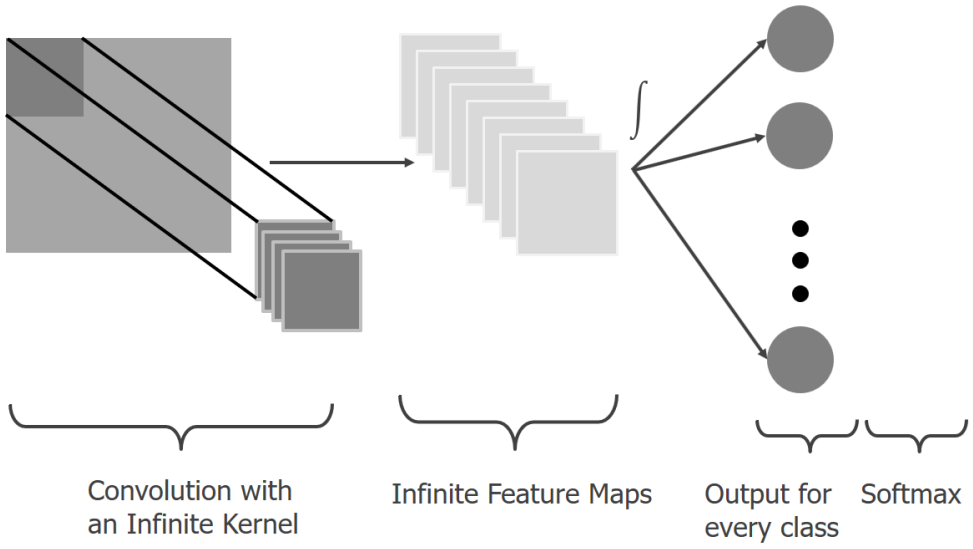


Figure 3.3: Typical FWCNN architecture.

As mentioned in 2.3.5, CNNs learn hierarchical feature representations, which means they learn more complicated concepts through simpler features; first convolutional layers learn simple features such as edges and as we go on to the next layers, they add more levels of abstraction and eventually in the last convolutional layers highly abstract features are obtained. Having the ambition to create an architecture with more than one layers for the Functionally Weighted Convolutional Neural Network topology in order to create an hierarchical feature representation we attempt the approach of convoluting with an infinite modeled kernel the infinite feature maps that were produced from the previous FWCNN layer.

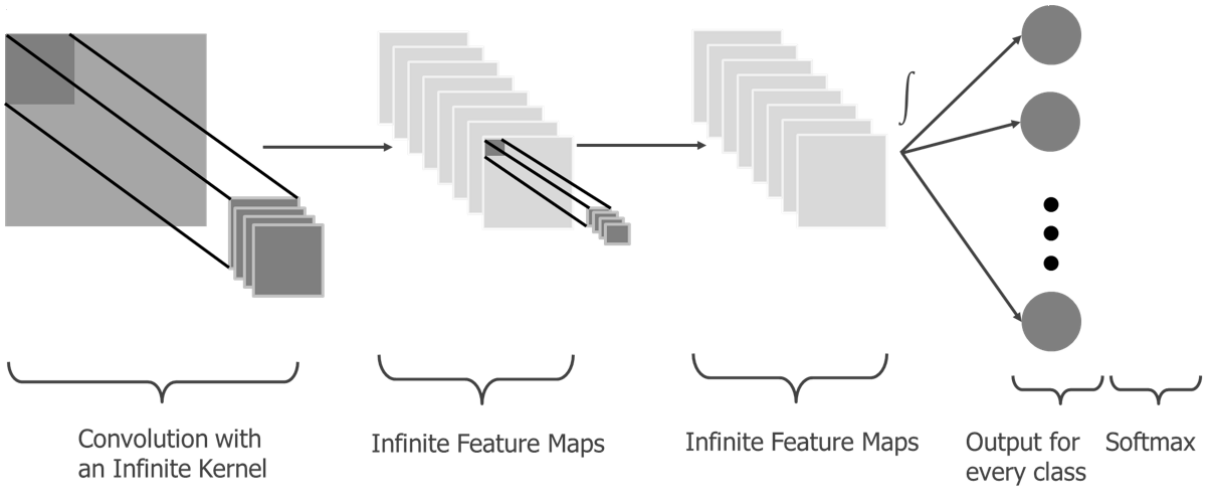


Figure 3.4: 2-Layer FWCNN architecture.

The implementation for this aproach was motivated by a typical Convolutional layer; where the number of the output feature maps is not decided by the size of the layers input. In a typical CNN layer for every output feature map, the input feature maps are convolved with a separated kernel and subsequently are summed up; in this sence we defined the infinite output number of feature maps of the FWCNN as infinite, which in our method is the approximation step number for the computation of the integral. In the next chapter we perform a variety of experiments for evaluating this approach and making the comparison of unknown parameter numbers between traditional CNNs and the proposed model.

Chapter 4

Experimental Results

4.1 Google Tensorflow Machine Learning Library

TensorFlow [27] is one of the most popular machine learning frameworks, created by Google, which operates at large scale and in heterogeneous environments. TensorFlow uses dataflow graphs to represent computation, shared state, and the operations that mutate that state. It maps the nodes of a dataflow graph across many machines in a cluster, and within a machine across multiple computational devices. Data in TensorFlow are represented as tensors. Tensors are geometric objects that describe linear relations between geometric vectors, scalars and other tensors. Elementary examples of such relations include the dot product, the cross product, and linear maps. Everything in TensorFlow is based on creating a computational graph. Nodes of the graph represent methematical operations while the graph edges repressent the multidimensional data arrays (tensors) communicated by them. This leads to a low-level programming model in which one can define the dataflow graph and then create a Tensorflow session to run parts of the graph across a set of local or remote devices. Dataflow is a common programming model for parallel computing and has several advantages that TensorFlow leverages.

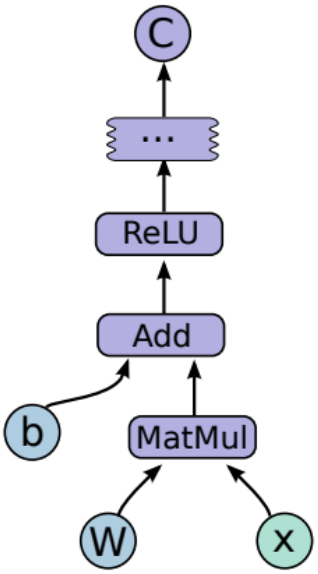


Figure 4.1: Computation graph of a simple Tensorflow operation.

- Parallelism; by using explicit edges to represent dependencies between operations, it is easy for the system to identify operations that can execute in parallel.
- Distributed execution. By using explicit edges to represent the values that flow between operations, it is possible for TensorFlow to partition a program across multiple devices attached to different machines. TensorFlow inserts the necessary communication and coordination between devices.
- Compilation; TensorFlow's XLA compiler can use the information of a dataflow graph to generate faster code by fusing together adjacent operations.

4.2 Description of datasets

We evaluated our methods on two popular benchmark datasets, MNIST and CIFAR-10.

MNIST dataset [28] has been widely used as a benchmark for testing classification algorithms in handwritten digit recognition systems. MNIST is an abbreviation for Mixed National Institute of Standards and Technology database. This database is created by mixing the original samples of NIST's database. The database has two parts; training samples that were taken from American Census Bureau employees and the test samples that were taken from American high school students. The origi-

nal NIST's database is too hard, therefore the MNIST database was constructed from NIST's Special Database 3 and Special Database 1 which contain binary images of handwritten digits. The samples that were taken from American Census Bureau employees, training database, was very cleaner than the samples that were taken from American high school students, test database. By combining 30000 samples from first dataset and 30000 samples from second dataset, the Mixed NIST training set was created. 60000 test samples were collected to constitute the test dataset, but only 10000 of patterns are now available on MNIST webpage. The first 5000 examples of the test set are taken from the original NIST training set and the last 5000 are taken from the original NIST test set. The MNIST is widely used for training and testing in the field of machine learning.

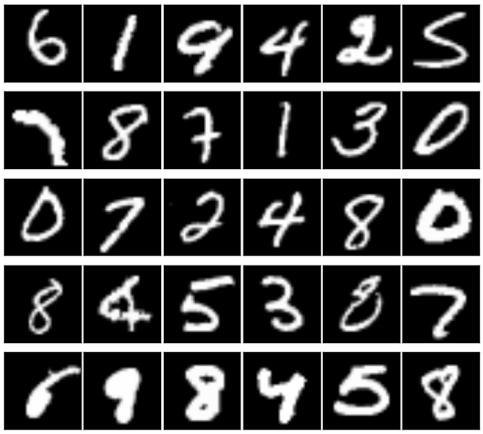


Figure 4.2: Sample of images from the MNIST dataset.

CIFAR-10 [29] is another established computer-vision dataset used for object recognition on which we evaluated our method. It is a labeled subset of the 80 million tiny image dataset. It is called CIFAR-10 dataset, after the Canadian Institute for Advanced Research, which funded the project and was collected by Alex Krizhevsky, Vinod Nair and Geoffrey Hinton. It consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. There are 50000 training images and 10000 test images.

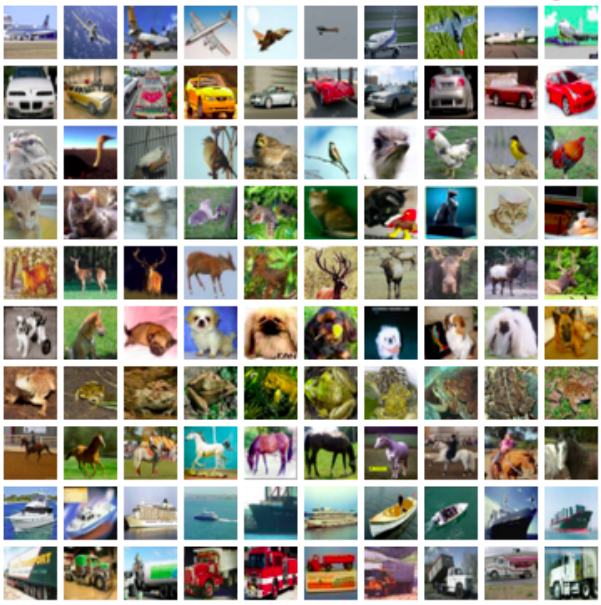


Figure 4.3: Sample of images from the CIFAR-10 dataset.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. The classes are *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship* and *truck* which are completely mutually exclusive.

4.3 Implementation details

To evaluate our methods we performed a variety of experiments, on different architectures and hyperparameters, in order to choose the best performing network configurations for the MNIST and CIFAR-10 datasets. The evaluation metric of our method is classification accuracy which we are trying to maximize for every class. Neural Networks have several sensitive hyperparameters that affect the performance of the network which can lead to undefitting and overfitting issues. Thus we perform hyperparameter tuning for the selection of the fittest values. There are two popular methods for choosing hyperparameters, grid search and random search. In grid search, sets of values are defined for each hyperparameter and all possible combina-

tions of hyperparameters values are exploited. Random search [30] is another way for parameter tuning for deep learning models, which proposes to sample independently each hyperparameter from a different hyperparameter distribution. For cross validation we split the training set in training and validation set, where we train our models with the trainining set and select the best hyperparameters based on the performance on the validation set. We set the validation set as the 20% of the training set for both the MNIST and CIFAR-10 datasets. For every experiment we train our models using the ADAM [11] optimization algorithm with setting the hyperparameters at the authors proposed values of 0.9 for β_1, 0.999 for β_2, but we applied exponential decay to the learning rate λ. We initialized λ at 0.1 and we applied decay with a base of 80% every some steps. The steps value for the decay of λ differ according to the dataset and the complexity of the model, though it is set in a range of $[800, 2000]$. We used Rectified Linear Units as an activation function for every experiment we conducted.

The training was done using a quad-core Intel i5-4590 CPU and an NVIDIA Titan X Pascal GPU. The implementation was written in Python version 2.7.2 with the Tensorflow machine learning library version 1.2.0.

4.3.1 Evaluation on the MNIST dataset

Functionally Weighted Neural Network on the MNIST dataset

We tested various architectures for the evaluation of our method on the MNIST dataset. Starting with a single layer FWNN we performed a series of experiments for different L_π and L_w degrees. The data was processed in batches of 128 samples and trained for 50 epochs.

We tested different polynomial degrees of L_w and L_π for a Functionally Weighted Neural Network, as presented in table 4.1, measuring the classification accuracy in the test set and as expected, we obtain the best results for the highest poloynomial degrees.

Table 4.1: FWNN experiments on various polynomial degrees results.

Polynomial Degree $L_\pi = L_w$	Test Accuracy	# of trainable Parameters
3	0.766	2385
4	0.866	3180
5	0.878	3975
6	0.852	4770
7	0.901	5565
8	0.888	6360
9	0.903	7155
...	...	...
21	**0.924**	**16695**

For a setting of values from a range $[9...14]$ we did not see any important improvement on the classification accuracy. We tested a range of degrees coming to the *extreme* value of 21 which gave the best performance for the given dataset.

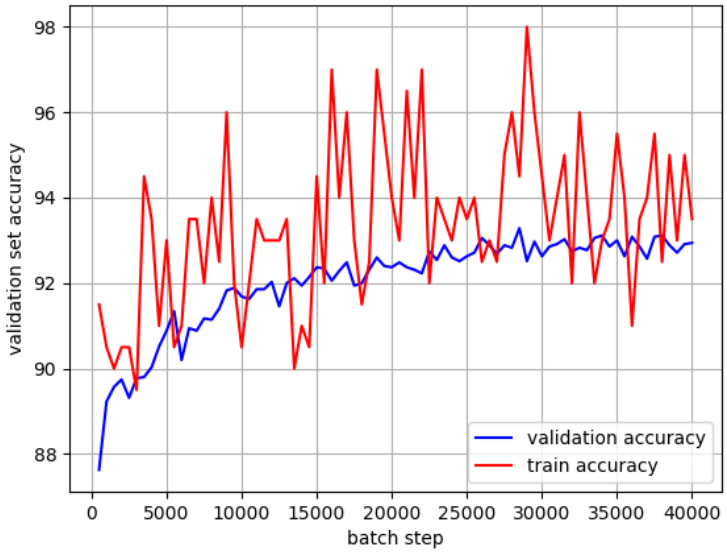
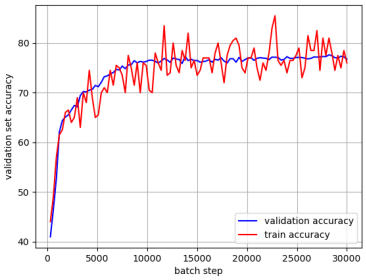
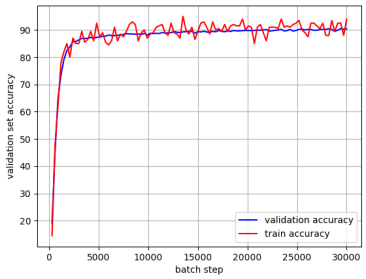


Figure 4.4: Best performing FWNN with polynomial $L_\pi = L_w$ degree of **21** on MNIST.

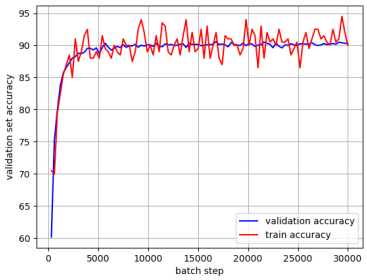
At a degree of 21 we obtain a test accuracy of **0.924** with a total number of **16695** trainable parameters. We performed the same experiments adding L_2 regularization to the network. Through exploring different values, we achieved the best results with a regularization λ value of 0.00001. We illustrate a fraction of the experiments at figures 4.5.



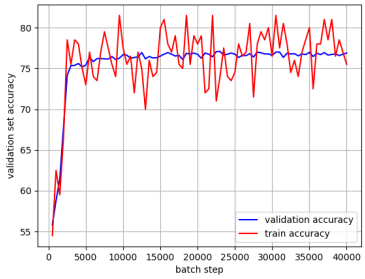
(a) $L_\pi = L_w = 3$



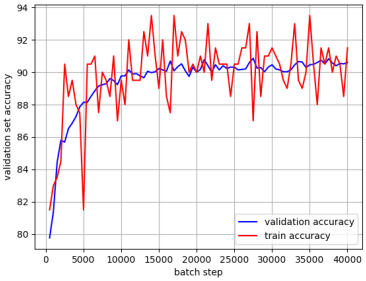
(b) $L_\pi = L_w = 7$



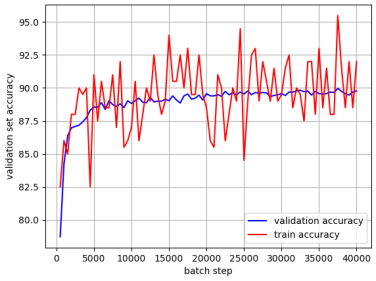
(c) $L_\pi = L_w = 9$



(d) $L_\pi = L_w = 3$



(e) $L_\pi = L_w = 7$



(f) $L_\pi = L_w = 9$

Figure 4.5: The figures (a), (b) and (c) show the implementation results without any regularization while the figures (d), (e) and(f) show the networks performance with regularization.

We achieved slightly better results when applied regularization at the weights of the network, though the differences were small having a standard deviation of 0.05 in accuracy. The reason is the small number of trainable parameters, judging from the improved results when applying regularization in the next experiments shown, where the number of trainable parameters exceed the number of 100000.

Functionally Weighted Convolutional Neural Network on the MNIST dataset

We performed a variety of experiments to evaluate the Functionally Weighted Convolutional Neural Network topology, starting again with different polynomial degrees for L_w and L_π. We trained the network for 40 epochs with and without weight regularization.

Table 4.2: FWCNN testing accuracy results on various polynomial degrees.

Polynomial degree L_π	Test Accuracy	# of trainable Parameters
4	0.975	7908
5	**0.978**	**9885**
6	0.973	11862
7	0.973	13839
...	...	...
10	0.972	19770
11	0.977	21747
12	0.972	23724

We achieved best performance at $L_w = L_\pi = 5$, whose loss function values at every step are illustrated in figure 4.7. Again the results after having the same experiments with applying L_2 regularization were not worth mentioning. The accuracy result is much superior than the FWNN results considering the fact that we managed to produce it with 51.2% less parameters. The unforseen outcome of the results though, was the case that the polynomial degree didnt seem to heavily affect the result, thus we could not improve thickly the testing accuracy no matter the raise of the polynomial degrees, as illustrated in 4.6.

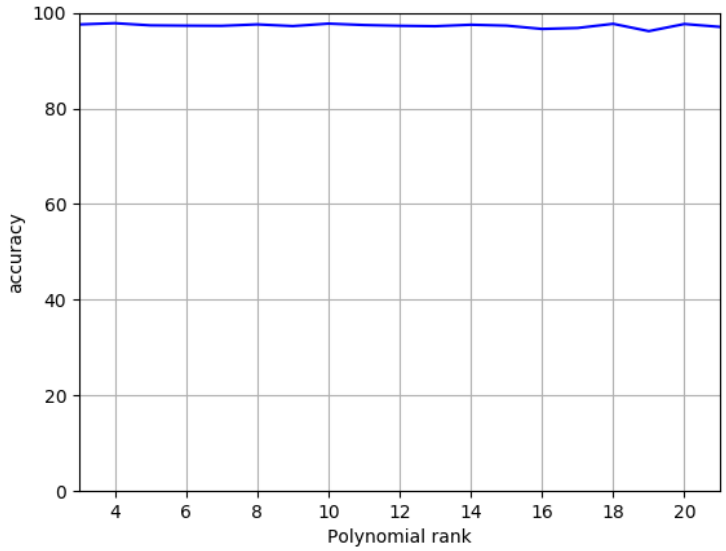


Figure 4.6: FWCNN test accuracy results on various polynomial degrees.

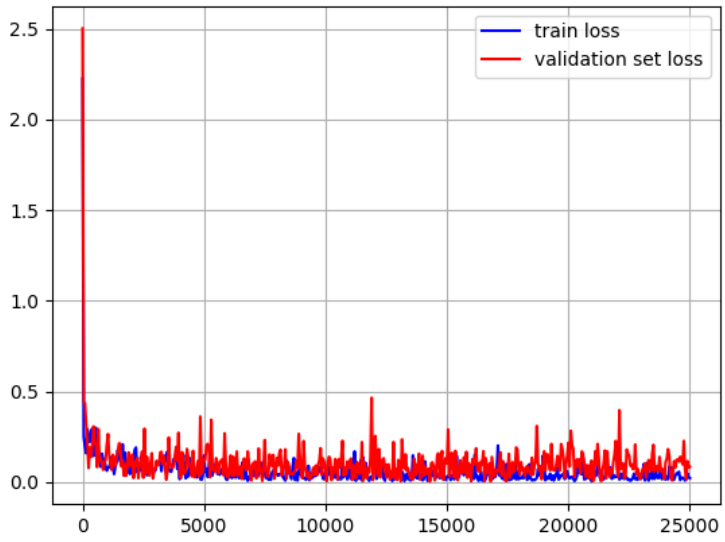
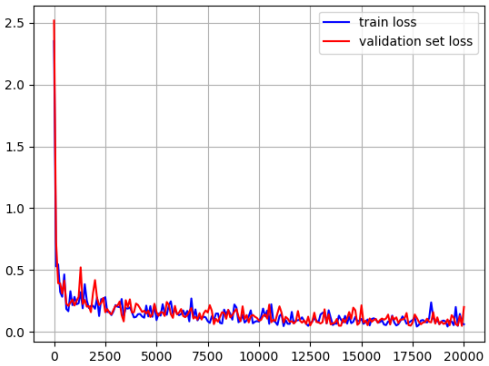


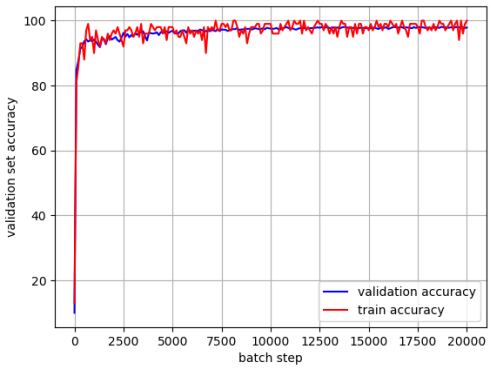
Figure 4.7: Loss function values at every step for best performing FWCNN configuration.

2-layer Functionally Weighted Convolutional Neural Network on the MNIST dataset

As described in 3.3 we studied the approach of stacking more than one FWCNN layers on top of each other anticipating to get the combination of better features at each layer and conducted experiments on the MNIST dataset. As a result, we managed to overcome the 97% testing accuracy of the 1-layer implementation, with just a marginal increase in the number of parameters.



(a) $L_\pi = L_w = 4$



(b) $L_\pi = L_w = 4$

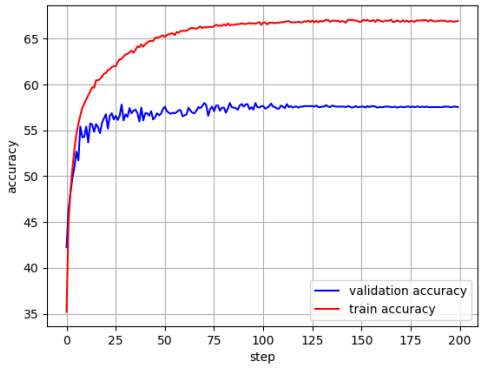
Figure 4.8: Loss function values at every step (a), Validation and Testing accuracies (b) for best performing 2-layers FWCNN configuration on MNIST.

The best results we achieved for the 2-layer was at a polynomial degree of 10,

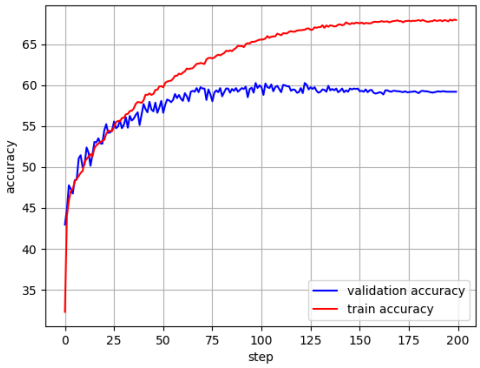
with a test accuracy of **0.985** and a total number of **23080** trainable parameters.

4.3.2 Evaluation on the CIFAR-10 dataset

Comparing to the MNIST dataset, CIFAR-10 is a far harder problem to solve, even for state-of-the-art classifiers. Thus, to obtain fair results, the implementations of our method we tested, were more complex. We first performed experiments with an 1-layer implementation for various L_π and L_w polynomial degrees. We got the best performance in 200 epochs of training, for $L_\pi = 50$, $L_w = 30$ and a regularization scale of $\lambda = 0.00005$, with a total of 130700 trainable parameters. Note that the number of parameters on these configurations is excessively higher than the ones that were tested on the MNIST dataset, thus regularization of the weights boosts the method's test accuracy. Note that we applied regularization only on L_π weights, which is the larger value. On the 1-layer implementation we succeed a 59% test accuracy.



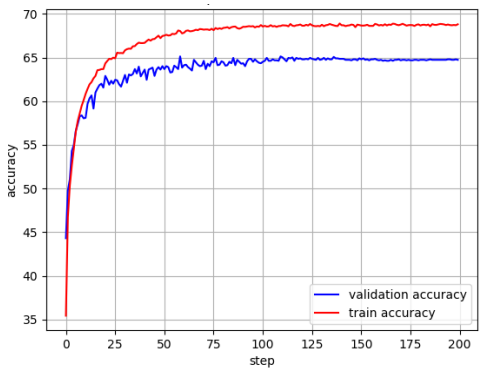
(a) No weight regularization.



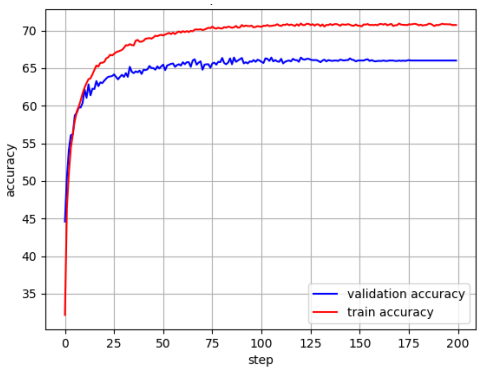
(b) L2 regularization with a $\lambda = \lambda = 0.00005$ on L_π weights.

Figure 4.9: Validation and Testing accuracies comparison, with (a) and without (b) weight regularization for best performing 1-layer configuration FWCNN configuration on CIFAR-10.

We experimented with various architectures, with different size of kernels, number of FWCNN layers and implemented max-pooling layers in between which drastically downsized the number of parameters; trying to achieve a fair proportion of the best test accuracy and architecture simplicity. Results of 2 and 3-layer are illustrated in figure 4.10.



(a) 2-layer FWCNN, $L_\pi = 50, L_w = 30$ with a regularization $\lambda = 0.00008$ and 2 max-pooling layers.



(b) 3-layer FWCNN, $L_\pi = 50, L_w = 10$ with a regularization $\lambda = 0.00008$ and 2 max-pooling layers.

Figure 4.10: Results of 2 and 3-layer FWCNN architecures on CIFAR-10.

The most successful configuration was a 6-layer architecture; with 2 max-pooling layers and polynomial degrees of $L_\pi = 50, L_w = 20$, at 232900 parameters, we achieved a 71.3% accuracy.

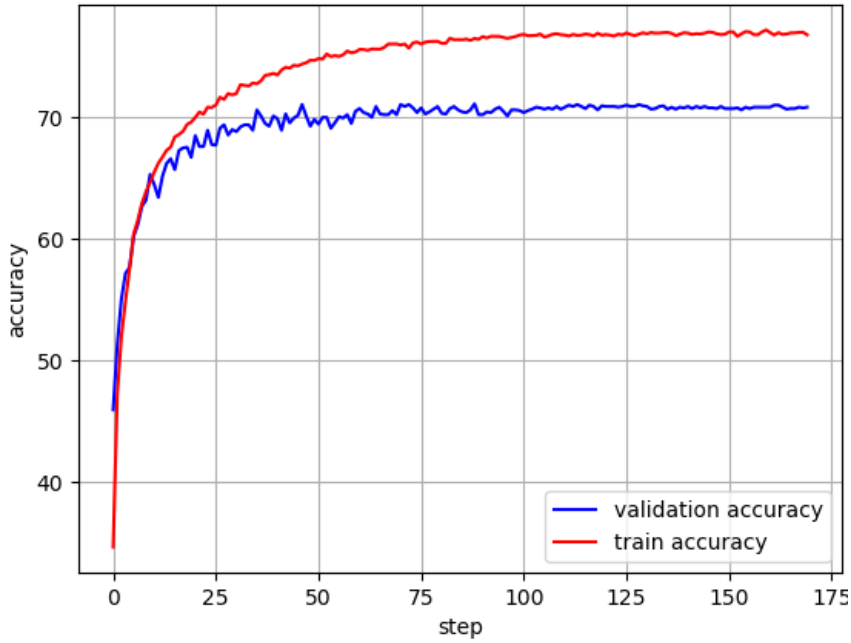


Figure 4.11: 6-layer FWCNN architecture.

Combination of FWNN and CNNs on the CIFAR-10 dataset

We also tried another approach by substituting the fully connected layers of a traditional Convolution neural network implementation with a functionally weighted

neural network. We tested a 2-layer convolutional layer which produced 64 feature maps at each layer, with 5x5 kernels. In the fully connected layer we used 2-layers with 256 and 128 hidden units, in comparison with the functionally weighted were we used 30 and 10 polynomial degree for the $L\pi$ and L_w respectively. As seen in figure 4.12 results are really promising.

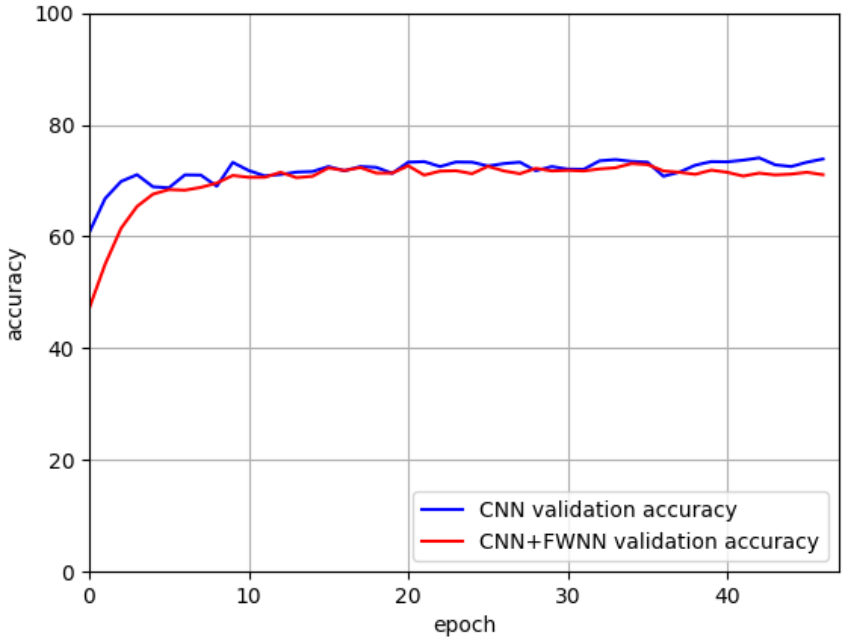


Figure 4.12: CNN with FWCNN classifier comparison with a traditional CNN.

Chapter 5

Conclusion

In this thesis, we studied deep learning techniques on neural networks with a focus on convolutional neural networks and introduced a new convolutional model, the functionally weighted convolutional network. We presented the dominance of the model considering the immense decrease in the number of unknown parameters and presented the flexibility of the model in constructing different architectures merged with the state-the-art methods.

In the experimental evaluation we conducted a variety of experiments on popular datasets. We conducted an extensive search on a variety of hyperparameters like learning rates, number of layers etc. with a primary focus on the hyperparameters of the degrees of the polynomial modeled weights. Furthermore we investigated the possibility of using a FWNN as a classifier on modern CNN architectures that are used for feature extraction. Although our model's results are comparable alone with simple CNN architectures we hold a strong belief into fine tuning our model for improved results.

Given the encouraging results obtained from the experiments there are several research directions to be followed in future work. At first it would be interesting to test an implementation in which we will try to produce features with more than one infinite kernel. Another research direction is to consider alternative approaches for the activation functions and how we could maybe introduce radial basis functions to the operation of convolution and finding an approach for parameter sharing between RBFs kernels centers and widths.

It would also be important to test our model on different datasets and problems; like text or signal classification. It is also interesting to conduct a more detailed analysis on the produced infinite feature maps. Visualizing the weights would possibly show us the regularization strength of the network; knowing that noisy patterns on filters can be an indicator of a network that hasn't been trained for long enough, or possibly has a very low regularization strength. Finally, another important visualization technique is also important; the visualization of the feature maps in order to get an understanding of what the neuron is looking for in its receptive field.

Bibliography

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

[2] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[3] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*, pp. 267–285, Springer, 1982.

[4] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[5] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[6] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[7] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted boltzmann machines for collaborative filtering," in *Proceedings of the 24th international conference on Machine learning*, pp. 791–798, ACM, 2007.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[9] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[10] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, pp. 177–186, Springer, 2010.

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

[12] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

[13] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[14] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 4898–4906, 2016.

[15] Y. Tang, "Deep learning using support vector machines," *CoRR*, vol. abs/1306.0239, 2013.

[16] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, *et al.*, "Learning algorithms for classification: A comparison on handwritten digit recognition," *Neural networks: the statistical mechanics perspective*, vol. 261, p. 276, 1995.

[17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

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

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

[20] K. Hornik Maxwell and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[21] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[22] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[23] D. Meyer, F. Leisch, and K. Hornik, "The support vector machine under test," *Neurocomputing*, vol. 55, pp. 169 – 186, 2003.

[24] M. Tipping, "Sparse bayesian learning and the relevance vector machine," *The Journal of Machine Learning Research*, vol. 1, pp. 211–244, 2001.

[25] M. W. Seeger, "Bayesian inference and optimal design for the sparse linear model," *The Journal of Machine Learning Research*, vol. 9, pp. 759–813, 2008.

[26] N. Le Roux and Y. Bengio, "Continuous neural networks," in *Artificial Intelligence and Statistics*, pp. 404–411, 2007.

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[28] 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.

[29] A. Krizhevsky, "Learning multiple layers of features from tiny images," 05 2012.

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

Short Biography

Dimitris Triantis was born in Preveza, Greece in 1989. He received his BSc degree from the Department of Computer Science & Engineering of University of Ioannina in 2015. In 2015 he became an MSc student at the same institution under the supervision of Konstantinos Blekas. In 2016 he worked as a software developer for Geotest S.A. His research interests lay in the areas of robotics, computer vision and machine learning, with a focus on deep learning.