VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

APPLICATION OF MEAN NORMALIZED STOCHASTIC GRADIENT DESCENT FOR SPEECH RECOGNITION

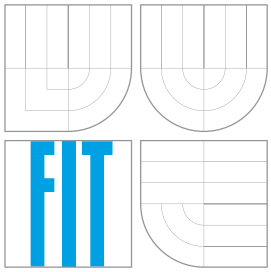
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN KLUSÁČEK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

APLIKACE METODY MEAN NORMALIZED STOCHASTIC GRADIENT DESCENT PRO ROZPOZNÁVÁNÍ ŘEČI

APPLICATION OF MEAN NORMALIZED STOCHASTIC GRADIENT DESCENT FOR SPEECH RECOGNITION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN KLUSÁČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN PEŠÁN
SUPERVISOR

BRNO 2015

Abstrakt

Umělé neuronové sítě jsou v posledních letech na vzestupu. Jednou z možných optimalizačních technik je mean-normalized stochastic gradient descent, který navrhli Wiesler a spol. [16]. Tato práce dále vysvětluje a zkoumá tuto metodu na problému klasifikace fonémů.
Ne všechny závěry Wieslera a spol. byly potvrzeny. Mean-normalized SGD je vhodné použít pouze pokud je síť dostatečně velká, nepříliš hluboká a pracuje-li se sigmoidou jako nelineárním prvkem. V ostatních případech mean-normalized SGD mírně zhoršuje výkon neuronové sítě. Proto nemůže být doporučena jako obecná optimalizační technika.

Abstract

The artificial neural networks are on the rise in recent years. One possible optimization technique is mean-normalized stochastic gradient descent recently proposes by Wiesler et al. [16]. This work further explains and examines this method on phoneme classification task.
Not all findings of Wiesler et al. can be confirmed. The mean-normalized SGD is helpful only if the network is large enough (but not too deep) and if the sigmoid non-linear function is used. Otherwise, the mean-normalized SGD slightly impairs the network performance and therefore cannot be recommended as a general optimization technique.

Klíčová slova

Neuronové sítě, strojové učení, rozpoznávání řeči, stochastic gradient descent.

Keywords

Neural networks, machine learning, speech recognition, deep learning, stochastic gradient descent.

Citace

Jan Klusáček: Application of Mean Normalized Stochastic Gradient Descent for Speech Recognition, bakalářská práce, Brno, FIT VUT v Brně, 2015

Application of Mean Normalized Stochastic Gradient Descent for Speech Recognition

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Pešána a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Klusáček
May 20, 2015

Poděkování

Děkuji svému vedoucímu Ing. Janu Pešánovi a Ing. Lukáši Burgetovi, Ph.D. za to, že mě nadchli pro strojové učení a že mi vždy uměli dobře poradit.

Contents

Chapter 1

Introduction

Artificial neural networks are experiencing a renaissance in recent years. Due to the rise of computational power, the use of deep neural networks is possible and they have become a crucial part of many recognition and classification systems, particularly in voice and image processing. The training of such networks is still a very difficult optimization problem. One of the most commonly used optimization techniques is stochastic gradient descent (SGD) [1]. Unlike batch gradient descent, the SGD is able to process large datasets quickly and often converges to a better local minimum than batch gradient descent. It is also possible to paralelize the computation on GPUs with the use of mini-batch.

Despite all the recent advancements, the training of a deep neural network is still time-consuming and computationally costly. There is a constant effort to improve the results and/or reduce the computation time. One way to improve the performance is to make use of the second-order information. Several successful techniques based on the approximation of Hessian matrix of error function on the mini-batch have been proposed [10, 15, 8]. Recently, the group of Simon Wiesler of RWTH Aachen University proposed a mean-normalized stochastic gradient descent. With this method, they were able to reduce the training time as the method converges faster than standard SGD. They were also able to reduce the size of the network by inserting a linear bottleneck [12, 18] and then train this network from scratch using mean-normalized SGD.

In this work, I focus on the proposed mean-normalized stochastic gradient descent in greater detail. I implement the mean-normalized method to an existing speech recognition system of Speech@FIT, explain step-by-step the derivation of the algorithm, try to replicate the results of Wiesler et al. and run some other experiments. The aim of this work is to compare the standard and mean-normalized stochastic gradient descent and decide whether and under what conditions is the proposed method better than the traditional SGD.

Chapter 2

Neural networks

Neural network is one of the oldest model for machine learning. They were first proposed in the 1950's and have been experiencing a renaissance in the last decade due to the advancement of computational power and the use of GPUs. Though they are more often referenced as deep learning, the basic algorithms stays the same. In particular, neural networks proved their value in speech recognition and image processing. They learn fast, generalize well and can be trained stochastically on the run.

The basic unit of every artificial neural network is an artificial neuron, inspired by the real neuron in a human nervous system. It takes several inputs, compute their weighted sum and if the sum exceeds some threshold it produces positive output. Usually there is no threshold but some differentiable function like sigmoid or tanh is used (figure 2.1). Single neuron (e.g. perceptron) is the simplest neural network possible and it can linearly divide the input space. Output y of the neuron can be computed as

$$y = \mathcal{F}(\sum_{i=0}^{n} w_i x_i),$$

where x_i is i^{th} of n input values, w_i is corresponding weight and $\mathcal{F}$ the non-linear function.

In feed-forward neural networks, the neurons are arranged in several layers. Each layer consists of many neurons and works with vector values on input/output. Output of each

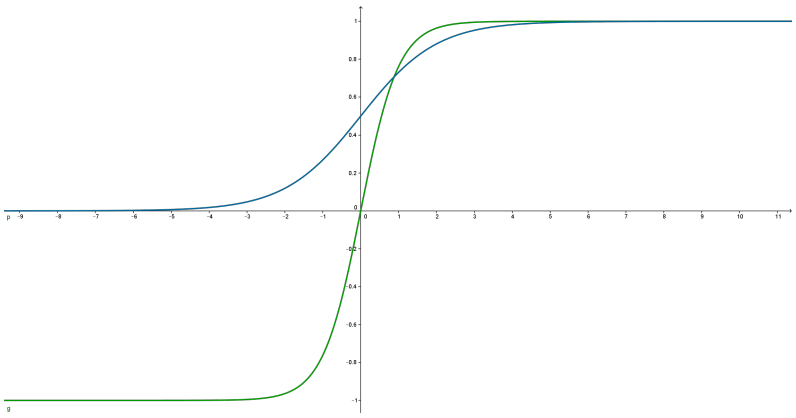


Figure 2.1: Most common non-linear functions, hyperbolic tangent (green) and sigmoid(blue).

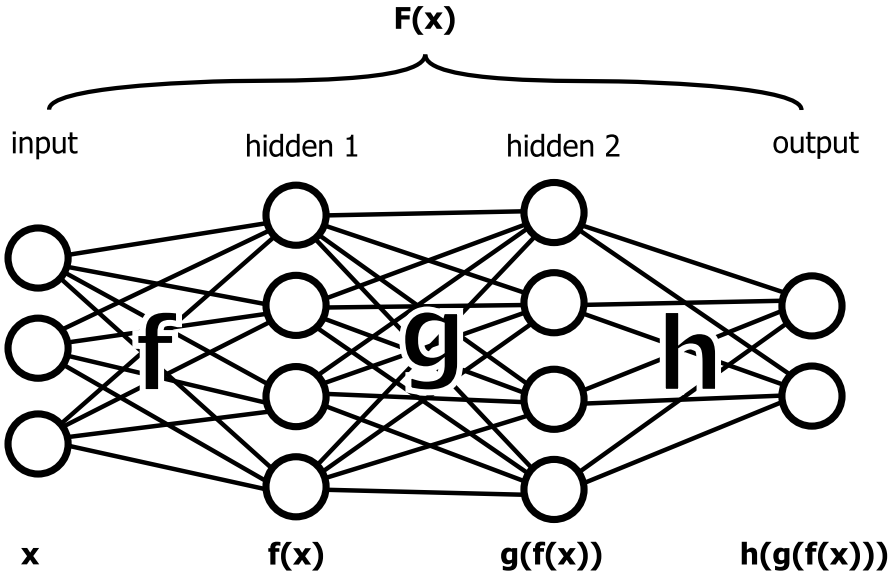


Figure 2.2: Scheme of feed-forward neural network viewed as a chain of several non-linear transformations.

layer is simultaneously the input to the following layer. Each neuron in layer i computes weighted sum of values produced by layer $i-1$. The function of one layer can be written as

$$f(x) = \mathcal{F}(W^T x + a) \tag{2.1}$$

Where $\vec{x}$ is the layer input vector, $\mathbf{W}$ is the matrix of weights, $\vec{a}$ is the bias vector which adjust the threshold of each neuron and $\mathcal{F}$ is the elementwise non-linear function[1]. The whole network can be seen as chain of several such functions as is shown in figure 2.2.

The neural network is an universal model which can (given enough hidden layers) approximate arbitrary non-linear function. Finding proper parameters of the network (weights and biases) so that it computes the desired outputs is a complicated optimization task. The most common optimization algorithm used in neural networks is the gradient descent.

[1]The symbols a, x always reference vectors but to clear up the equations, they will not be typed with arrow or in bold. The same stands for weight matrix W.

2.1 Gradient descent

Gradient descent is a general optimization method. In the case of neural networks, an error function $\mathrm{F_E}$ is optimized. This function determines how far off the network output is from the correct solution. In classification tasks, the most common is the cross-entropy function

$$F_E = -\log(F_{(W,a)}(x)[t]), \tag{2.2}$$

where $F_{(W,a)}(x)[t]$ is the probability for the target class t predicted by the network with parameters (W, a) based on input x. The aim of gradient descent is to minimize this error function. The gradient of the error function with respect to all the weights is computed using the backpropagation algorithm. The gradient is then multiplied by some learning rate η and subtracted from current weights. Because the error function is smooth, the gradient descent algorithm can always find a local minimum[3]. The update rules for weights and biases can be written are

$$\widehat{W} := W - \eta \nabla_W F_E, \tag{2.3}$$

$$\widehat{a} := a - \eta \nabla_a F_E, \tag{2.4}$$

where $\nabla_W F_E$ and $\nabla_a F_E$ are the gradients of error function F_E with respect to weights W and biases a respectively. The gradient can be computed either on whole training dataset (batch gradient descent) or stochastically for each data sample. Because the batch gradient descent is slow and the gradients of single data sample might be noisy[17], an intermediate way of mini-batch is often used [7]. This method allows more efficient calculation on CPU or GPU than batch method and is less noisy than pure stochastic learning. Only this method will be considered in this work from now on.

While this basic algorithm is well-known and understood, there is a constant effort to improve it via numerous optimizations and heuristics. One way is to make use of the information about the second-order derivatives. Several techniques based on the approximation of Hessian matrix of error function on the mini-batch have been proposed, some of them proved to be successful [10, 15, 8]. The method on which this work is focused – mean-normalized stochastic gradient descent recently proposed by Wiesler et al. [16] – is also such approximation, even though it does not try to approximate the Hessian directly.

Chapter 3

Mean-normalized stochastic gradient descent

3.1 Motivation

In most classification tasks with discriminative models the input data are normalized to ensure that, in all dimensions, they have the same mean value and standard deviation. Neural networks could theoretically learn to scale the features themselves but it was shown that normalizing the inputs accelerates the network training [7]. Inputs shifted from zero bias the updates and slow down the learning. Normalizing the variance helps eliminate the differences of learning speed of particular weights. Alternative explanation provided by [9] operates with hyperplanes defined by columns of the weight vector. When bias is set to zero, each hyperplane goes through the origin. Hence, when the input is mean-normalized, there is a higher probability, that the hyperplane cut the dataset in half. This also results in speeding up the training. One way or another, normalization of the inputs is helpful and commonly used.

3.2 Idea of the algoritm

For feedforward neural networks, the normalization described above works only for the first layer. The normalization is lost in deeper layers, because of the transformation and application of non-linear functions. In figure 2.2, the inputs for $f()$ are normalized, but not for $g()$ or $h()$. The shift of mean values away from zero can be clearly seen when using the sigmoid function. The outputs of each layer (which are the inputs to the following layer) are always positive and thus the means are non-zero. According to [7], this can be avoided by carefully choosing the non-linear function[1], initial weights and the standard deviation of normalized inputs.

Wiesler et al[wiesler] takes different approach to the problem. The basic idea is to normalize the calculated values in every layer. All the weights of the neural network would then benefit from the normalization, without the need of perfectly tuning the network. However, this would also make the backpropagation of error more difficult. Wiesler et al. have found an elegant way which avoids explicit normalization of the weights. They consider a hypothetical model which normalizes outcomes of every layer and a real one, which does not. There is a mapping ϕ of parameters (weights and biases) between the

[1] [7] recommends using tanh instead of sigmoid.

hypothetical model with mean-normalization and the actual one. The forward run of the network remain intact. The update is carried out as follows:

1. *Map parameters to space of mean-normalized model by* ϕ
2. *Update those parameters using stochastic gradient descent*
3. *Map parameters back to original space by* ϕ^{-1}

The gradients needed for the update in step 2 are also affected by the transformation between the models, but they still can be calculated using gradient values form standard backpropagation and (continuously updated) mean values of hidden layers outcomes. How exactly this is done is shown in section 3.3. This method is reasonably fast, as it does not burden neither the forward nor the backward run with additional computations. All the mapping, updating and mapping back happens at once for every batch. It is necessary, however, to keep track of the outcomes of hidden layers and to continuously update its mean values.

3.3 Derivation of the algorithm

In this section, the algorithm of Wiesler at al. is further explained step by step. It describes how the update rule for mean-normalized SGD is derived.

Consider only weights between two hidden layers L_1, L_2 with dimensions D_1 and D_2 respectively. The $W \in \mathbb{R}^{D_1 \times D_2}$ denotes the weight matrix between L_1 and L_2 and $a \in \mathbb{R}^{D_2}$ the bias vector. The objective function can be written as:

$$F : \mathbb{R}^{(D_1+1)\times D_2} \mapsto \mathbb{R}, \quad (W, a) \mapsto \mathcal{G}(W^T z + a) \tag{3.1}$$

It maps the whole weight matrix to a real value which we try to minimize (the error function). First, the input vector z is multiplied with the weight matrix and then the bias vector a is added. The function $\mathcal{G}$ combines the non-linear function, the transformations of all following layers and the error function. Now suppose that input to each layer was mean-normalized, i.e. the mean value b would be subtracted from input z ($\widetilde{z} = z - b$).To compensate for this, the output of the layer shall be computed as $W^T(\widetilde{z} + b) + a$ and the objective function ($\widetilde{F}$) of mean-normalized model would be therefore:

$$\widetilde{F} : \mathbb{R}^{(D_1+1)\times D_2} \mapsto \mathbb{R}, \quad (W, a) \mapsto \mathcal{G}(W^T z + W^T b + a) \tag{3.2}$$

Note that bias vector of such model would be different. A function ϕ maps the parameters of the original model to those of mean-normalized one. The inverse function ϕ^{-1} perform the same in opposite direction. They can be derived from the equality of the objective functions:

$$\begin{aligned} F(W, a) &= \widetilde{F}(\phi(W, a)) \\ \phi(W, a) &:= (W, a - W^T b) \end{aligned} \tag{3.3}$$

$$\widetilde{F}(W, a) = F(\phi^{-1}(W, a)) \tag{3.4}$$

$$\phi^{-1}(W, a) := (W, a + W^T b) \tag{3.5}$$

The central idea of the algorithm is to transform the parameters of the original model with ϕ, then update those transformed parameters using SGD and finally transform them back to the original space with ϕ^{-1}.

$$(\hat{W}, \hat{a}) = \phi^{-1}\Big(\phi(W, a) + \eta \cdot \nabla \widetilde{F}\big(\phi(W, a)\big)\Big) \tag{3.6}$$

This update rule requires the gradient of $\widetilde{F}$, which can be obtained by chain rule. Breaking the function ϕ^{-1} in two ($\alpha()$ for transforming bias and $\omega()$ for weights) will simplify the calculations (note that ω is actually an identity function).

$$\begin{gathered}\phi^{-1}(W, a) = (\omega(W, a), \alpha(W, a)) \\ \omega(W, a) = W, \quad \alpha(W, a) = W^T b + a\end{gathered} \tag{3.7}$$

First, lets calculate the gradient of $\widetilde{F}$ with respect to weights W_{ij}. To avoid using three and more dimensional tensors, it is convenient to transform the weight matrix W_{ij} (size $D_1 \times D_2$) to a one-dimensional vector $\overline{W_k}$ of lenght $D_1 D_2$ with $k = i \cdot D_1 + j$. Starting with equation (3.5) we get:

$$\begin{aligned}\nabla_W \widetilde{F}(\overline{W}, a) &= \frac{\partial F\left(\omega(\overline{W}, a), \alpha(W, a)\right)}{\partial \overline{W}} \\ &= \frac{\partial F\left(\omega(\overline{W}, a), \alpha(W, a)\right)}{\partial \omega(\overline{W}, a)} \cdot \frac{\partial \omega(\overline{W}, a)}{\partial \overline{W}} + \frac{\partial F\left(\omega(\overline{W}, a), \alpha(W, a)\right)}{\partial \alpha(W, a)} \cdot \frac{\partial \alpha(W, a)}{\partial \overline{W}} \\ &= \nabla_W F(\phi^{-1}(W, a)) \cdot I + \nabla_a F(\phi^{-1}(W, a)) \cdot J\end{aligned} \tag{3.8}$$

The first addend is quite clear. The I is the Jacobian of $\omega(\overline{W}, a)$ with respect to $\overline{W}$. In this case it is equal to the identity matrix, as ω does not perform any operation on W. The term on the left can be expressed as a simple gradient of $\nabla_W F$, which will be useful later on.

The second addend is more complicated. The J is the Jacobian of $\alpha(\overline{W}, a)$ with respect to $\overline{W}$. It has the shape $D_2 \times D_1 D_2$ and has the following form:

$$J = \begin{pmatrix} b_1 & 0 & \cdots & 0 & b_2 & 0 \\ 0 & b_1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 & 0 \\ 0 & 0 & \cdots & b_1 & 0 & b_{D_1} \end{pmatrix}$$

Multiplying J with the vector $\nabla_a F(\phi^{-1}(W, a))$ results in vector with shape $1 \times D_1 D_2$. It can be now rearranged back into $D_1 \times D_2$ matrix

$$\begin{pmatrix} b_1 \cdot \partial_{a_1} F(\phi^{-1}(W, a)) & \cdots & b_1 \cdot \partial_{a_j} F(\phi^{-1}(W, a)) \\ \vdots & \ddots & \vdots \\ b_n \cdot \partial_{a_1} F(\phi^{-1}(W, a)) & \cdots & b_n \cdot \partial_{a_j} F(\phi^{-1}(W, a)) \end{pmatrix} = b \cdot \nabla_a^T F(\phi^{-1}(W, a))$$

, which is an outer product of two vectors – b and $\nabla_a F(\phi^{-1}(W, a))$. The final formula for gradient $\nabla_W \widetilde{F}$ is therefore:

$$\nabla_W \widetilde{F}(W, a) = \nabla_W F(\phi^{-1}(W, a)) + b \cdot \nabla_a^T F(\phi^{-1}(W, a)) \tag{3.9}$$

Gradient of $\widetilde{F}$ with respect to a is obtained in similar manner:

$$\nabla_a \widetilde{F}(W,a) = \frac{\partial F(\omega(W,a),\alpha(W,a))}{\partial a} = \frac{\partial F(\omega(W,a),\alpha(W,a))}{\partial \alpha(W,a)} \cdot \frac{\partial \alpha(W,a)}{\partial a}$$
$$= \nabla_a F(\phi^{-1}(W,a)) \cdot I = \nabla_a F(\phi^{-1}(W,a)) \tag{3.10}$$

Now we have the gradient of $\widetilde{F}$ and can finally get to the update rule of mean-normalized SGD. Inserting into (3.6) gives:

$$(\hat{W},\hat{a}) = \phi^{-1}\Big(\phi(W,a) - \eta \cdot \nabla \widetilde{F}(\phi(W,a))\Big) \tag{3.11}$$
$$= \phi^{-1}\Big((W, a - W^T b) - \eta \cdot \nabla \widetilde{F}(W, a - W^T b)\Big)$$
$$= \phi^{-1}\Big(W - \eta \nabla_W \widetilde{F}(W, a - W^T b), \quad a - W^T b - \eta \nabla_a \widetilde{F}(W, a - W^T b)\Big)$$
$$\hat{W} = W - \eta \cdot \big(\nabla_W F(W,a) + b \cdot \nabla_a F(W,a)^T\big) \tag{3.12}$$
$$\hat{a} = a - W^T b - \eta \nabla_a F(W,a) + \hat{W}^T b \tag{3.13}$$
$$= a - \eta\Big(\nabla_W F(W,a)^T b + \nabla_a F(W,a) + (b \cdot \nabla_a F(W,a)^T)^T b\Big)$$
$$= a - \eta \cdot \big(\nabla_W F(W,a)^T \cdot b + (1 + b^T b)\nabla_a F(W,a)\big)$$

The weights W and biases a are taken directly from the model. The gradient $\nabla F(W,a)$ is computed using the backpropagation algorithm. The mean values of inputs to each layer needs to be continuously traced:

$$b_n = \psi \mathbb{E}(y|\mathcal{B}_n) + (1-\psi) b_{n-1} \tag{3.14}$$

Here, $\mathbb{E}(y|\mathcal{B}_n$ is the mean on n^{th} mini-batch $\mathcal{B}_n$ and $0 < \psi < 1$ is a smoothing factor. In this work $\psi = \frac{1}{n}$ but supposedly it could be a constant as well. That would result in favouring more recent values over the old ones.

This algorithm operates only with mean-normalization and ignores the variance normalization. Wiener et al. says that variance-normalization did not result in any improvements and are be omitted in this work as well.

3.4 Convergence proof

The algorithm can be viewed as a general second-order approximation. A joint update matrix for weight and biases B, i. e. the approximation of Hessian approximation can be constructed.

Wiesler et al. provided a convergence of this algorithm based on previous results [11, 13]. For this proof to hold, there must be several constraints satisfied. Most importantly, the eigenvalues of B must be bounded both from below and above. Considering finite weights (i.e. L_2 regularization constraint) this holds as long as the mean-values b are finite as well. For the first layer, it is safe to assume that network inputs are bounded (usually real-world data); inputs to later layers are given by some sigmoid-like non-linear function and thus are bounded as well. Other constraints given by [13] are also easily met.

3.5 Expected results

Wiesler et al have found that mean-normalized SGD decreases the error on training dataset faster than the standard SGD. This was particularly the case of models with very large

number of parameters – *"... more than twice as many parameters as the number of training samples..."*. But such models can be trained to arbitrary precision and overfitting must be avoided by some regularization method.

One way to improve the ability to generalize is the reduction of parameters. Wiesler et al. focused on the technique of linear bottleneck [12]. This method can reduce the size of the network while preserving the learning capabilities. Training a network with linear bottleneck with standard gradient descent and no pretraining is virtually impossible [18]. With mean-normalized stochastic gradient descent, training such network from scratch should be possible.

Chapter 4

Experiments

The experiments are the most important part of this work. Their purpose is to test the mean-normalized SGD algorithm in a number of ways. I try to replicate the results of Wiesler et al. and also run other experiments with different topology and linear bottleneck. The main goal of the experiments is to find out whether and under what conditions is the mean-normalized method better than the classical one.

First, the task and experimental setup is be described. After that I try to find optimal parameters for the basic setting. These results also serve as a baseline for further experiments with the network topology. There I examine larger and deeper networks and with linear bottleneck.

After description of each experiment, the results are immediately shown and described. Usually, there are similar sets of experiment for models with sigmoid and hyperbolic tangent. Every experiment examines both mean-normalized and standard stochastic gradient descent. Differences between performances of these methods are pointed out and discussed in the end of every section. All the results are summarized in the final chapter.

The network performance is be measured by accuracy on cross-validation dataset (percentage of correctly classified samples) and by the cross-entropy of these samples (the objective used for network training). The cross-entropy is also referenced simply as "error". The accuracy and error on the training set are monitored as well, but are not always mentioned.

4.1 Task description

The experimental setup is taken over from the speech recognition system of Speech@FIT group of BUT, further described in [14]. The neural network performs a phoneme classification task. The inputs are derived from 40 log Mel filter bank outputs. For every frequency band, temporal context of 31 frames is projected into 16 Hamming window weighted DCT bases. The process of feature extraction is shown in figure 4.1 The idea is to focus on the major changes in the neighbourhood of the current frame. Also, fast changes are smoothed out and the resulting coefficients are partially decorrelated. Output is a 135-dimensional vector where values represent posterior probability of a phoneme (3 states for each of 45 phonemes).

All experiments are realized on proprietary database of spoken English. From total 8 hours, one is taken for cross-validation and another for a test set. The mini-batch size is set to 512. The inputs are both mean and variance normalized before training.

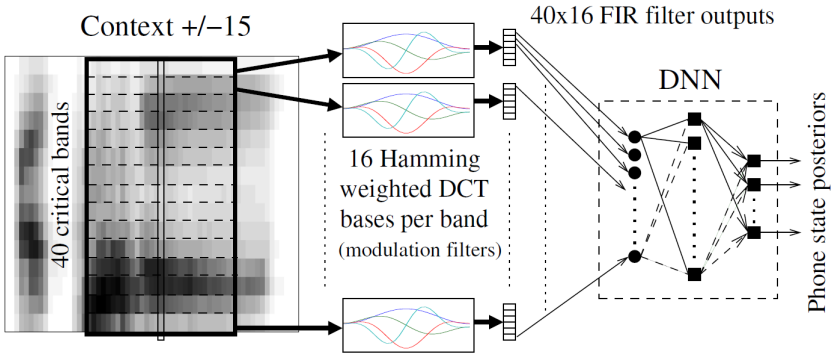


Figure 4.1: Feature extraction using Mel filter bank outputs and weighted Hamming window DCT bases.

4.2 Tanh

One of the most important parameter in neural networks is the choice of non-linear function $f()$ which is applied in every neuron of hidden layer.

$$y = f(W^T x + a)$$

For classification tasks, the nonlinearity of output layer is usually the softmax function so that the output can be interpreted as posterior probability. For the hidden layers, most common functions are hyperbolic tangent and the sigmoid function [2]. I examine models with either function, starting with tanh.

The use of tanh is recommended by [7]. It uses tanh() and the topology is $(358 \times 1024 \times 1024 \times 1024 \times 135)$. Weights are drawn from distribution $\mathcal{N}(\mu = 0, \sigma = \sqrt{\frac{2}{n_i + n_{i+1}}})$ and biases set to zero. This initialization is recommended by [5] and helps the error to propagate easily even in a very deep network.

The aim of this set of experiments is to find the optimal settings for learning rate and its decay.

4.2.1 Initial learning rate

The initial value of learning rate is something, that is often estimated empirically. With too small learning rate, the gradient descent is very slow. On the other hand, big learning rate often misses the local minimum completely and does not converge at all. The best learning rate is a compromise between these two and lies somewhere in between [4]. Even with this optimal value, it is good practice to start decreasing the learning rate at some point. Different strategies can be used (see section 4.2.2, by default the Newbob/CV is used.

The aim of this set of experiments is to find the most suitable initial value of learning rate. Values of different order of magnitude between 0.01 and $0,00002$ are tried.

Results

From the table 4.1, it can be seen that the most suitable values lie in $< 0,001; 0,0001 >$. The best one is 0.0002 (marked with bold) and it is used in other experiments with hyperbolic

tangent from now on. Other values show typical malfunctions, whether they are too big or too small.

LR value	Standard			Mean-normalized		
	Epoch 4	Epoch 8	Epoch 12	Epoch 4	Epoch 8	Epoch 12
0.01	20.3	3.00	2.82	23.7	3.08	2.84
0.005	6.75	2.73	2.56	7.55	7.11	2.64
0.002	3.13	2.47	2.20	3.32	2.39	2.25
0.001	2.06	1.95	1.94	2.08	1.97	1.96
0.0005	2.17	1.96	1.94	2.20	1.97	1.95
0.0002	**2.06**	**1.96**	**1.92**	**2.07**	**1.97**	**1.93**
0.0001	2.12	2.02	1.95	2.13	2.03	1.95
0.00005	2.25	2.09	2.03	2.25	2.10	2.03
0.00002	2.46	2.29	2.19	2.45	2.29	2.19

Table 4.1: Tanh: Cross-validation errors for different initial learning rate values.

4.2.2 Learning rate decay strategy

Wiesler et al. do not use the Newbob/CV strategy in their experiments, because it decays the learning rate too quickly. Instead, they propose the Newbob+ strategy. The learning rate starts halving not after one, but after two subsequent epochs with error raise. They also propose not to follow the error on cross-validation set but rather the error on the training data. This would generally lead to overfitting and some regularization method should be used to avoid that.

Results

I have tried four strategies: Newbob and Newbob+ for both CV and training data set. From experiments guided by training error, the weights which performed best on cross-validation set are considered final. The best values of errors on cross-validation and training set (not necessarily from the same epoch) are shown in table 4.2.

As expected, the use of Newbob/TR or Newbob+/TR leads to massive overfitting. In fact, the training error is decreasing steadily throughout the training and the learning rate does not start decreasing at all. The best results on cross-validation set retrieved by early stopping are similar to those of Newbobo/CV just before the halving started and thus not very good. Using Newbob+ instead of Newbob had very little effect as well. It presumably help decrease the error on training set, but the CV error is not affected.

4.2.3 Result summary

I have found an optimal learning rate and its decay strategy for neural network with hyperbolic tangent. The optimal setting is the same both for standard and mean-normalized stochastic gradient descent.

The best results from model with tanh non-linear function are obtained when the initial learning rate is set to 0.0002, its decay strategy to Newbob/CV, the weights initialized from a random distribution $\mathcal{N}(\mu = 0, \sigma = \sqrt{\frac{2}{n_i+n_{i+1}}})$ recommended by [5] and the biases set to zero. The graphs of error and accuracy on cross-validation set are shown in figure 4.2.

LR decay strategy	Standard		Mean-normalized	
	CV error	Training error	CV error	Training error
Newbob/CV	1.92	1.41	1.93	1.43
Newbob+/CV	1.93	1.31	1.94	1.31
Newbob/TR	2.04	0.91	2.06	0.93
Newbob+/TR	2.04	0.91	2.06	0.93

Table 4.2: Tanh: Error on cross-validation and training set for different LR decay strategies.

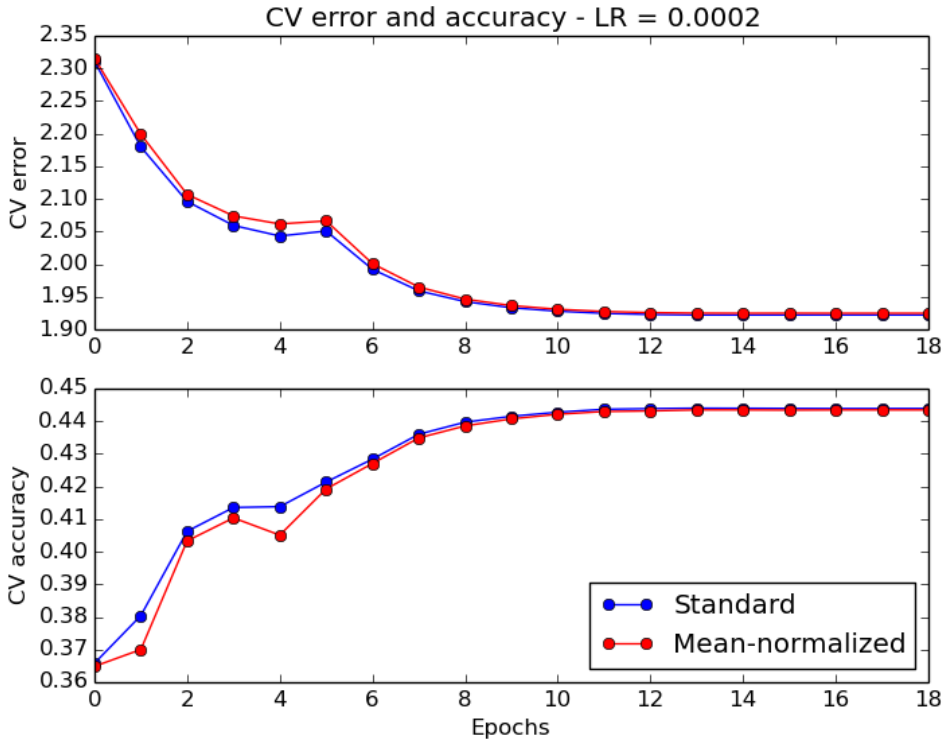


Figure 4.2: Tanh: Cross-validation error and accuracy of best model with tanh.

The differences between standard and mean-normalized SGD are not very distinctive in any of the experiment. For most learning rates, the standard approach works slightly better than the mean-normalized one. This difference is not very significant though. Usually it does not exceed 1 %. With the error on training set as the optimizing criterion, the standard method is giving better results as well. It seems that mean-normalized method is not helpful at all while using the hyperbolic tangent. Possible cause of this may be that all the values computed inside the network are already normalized. As the tanh output values lay in $(-1, 1)$, the mean of these values is presumably close to zero.

Using mean-normalized method along with tanh slightly worsens the performance (by cca 1 %) and cannot be recommended.

LR value	Standard			Mean-normalized		
	Epoch 4	Epoch 8	Epoch 12	Epoch 4	Epoch 8	Epoch 12
0.1	2.18	2.09	2.06	4710	176	5.25
0.05	2.26	2.07	2.03	2.37	2.07	2.02
0.02	2.10	1.98	1.94	2.09	1.96	1.91
0.01	**2.02**	**1.89**	**1.86**	2.05	2.00	1.92
0.008	2.03	1.88	1.86	2.04	1.98	1.90
0.005	2.01	1.91	1.88	**2.05**	**1.92**	**1.88**
0.002	2.08	1.98	1.90	2.12	2.01	1.91
0.001	2.27	2.04	1.99	2.27	2.08	2.02
0.0005	2.92	2.24	2.09	2.63	2.21	2.11
0.0002	4.38	3.40	2.61	4.34	2.96	2.44

Table 4.3: Sigmoid: Cross-validation errors for different initial learning rate values.

4.3 Sigmoid

The sigmoid function has similar shape as the hyperbolic tangent, but it always gives positive results from interval $(0, 1)$. The sigmoid is used in the QuickNet software [6] and Wiesler et al. are using it as well. Weights are initialized from random distribution $\mathcal{N}(\mu = 0, \sigma = 0.1)$ and biases similarly from $\mathcal{N}(\mu = -4, \sigma = 0.1)$. When the biases are initially set this low, with the neurons saturated, only the useful portion of them becomes active which results in speeding up the learning. This setting has been previously proven successful on the exact same task [14]. Once again, I am searching for optimal learning rate.

The topology is the same as in the tanh model in order for the models to be comparable.

4.3.1 Initial learning rate

Similarly to tanh, a set of experiments with the aim to find an optimal initial learning rate is performed.

Results

The results are shown in table 4.3. The error on cross-validation set is smallest when the initial learning rate is set to 0.01 and 0.005 for standard and mean-normalized model respectively. The mean-normalized model may seem worse, but it has smaller error on training set. For example, the final error on training set for lr = 0.01 is 1.27 for the standard model but only 1.19 for the mean-normalized one.

In most of later experiments, initial learning rate is set to 0.008, as it lies between the best values for standard and mean-normalized models and gives good results for both of them.

The learning rate for sigmoid models is orders of magnitude larger than the one for tanh models. This is due to different steepness (see figure 2.1 for comparison).

4.3.2 Learning rate decay strategy

As with the tanh, Newbob and Newbob+ for both CV and training error have been tried.

LR decay strategy	Standard		Mean-normalized	
	CV error	Training error	CV error	Training error
Newbob/CV	1.86	1.32	1.88	1.22
Newbob+/CV	1.86	1.32	1.89	1.23
Newbob/TR	2.01	0.75	2.02	0.91
Newbob+/TR	2.02	0.75	2.01	0.91

Table 4.4: Sigmoid: Error on cross-validation and training set for different LR decay strategies.

LR value	Standard			Mean-normalized		
	Epoch 4	Epoch 8	Epoch 12	Epoch 4	Epoch 8	Epoch 12
0.01	4.78	3.14	3.07	14.2	4.10	3.88
0.005	2.27	2.06	1.97	13.4	4.09	3.92
0.002	**2.07**	**1.97**	**1.90**	13.24	3.28	3.23
0.001	2.13	2.01	1.93	4.91	3.14	2.26
0.0005	2.22	2.07	2.02	**2.41**	**2.16**	**2.08**
0.0002	2.39	2.22	2.13	2.46	2.26	2.20

Table 4.5: Sigmoid, zero biases: Cross-validation errors for different initial learning rate values.

Results

The smallest cross-validation and training errors are shown in table 4.4. Similarly to tanh, the use of Newbob(+)/TR leads to massive overfitting and the results on cross-validation set are worse than those of Newbob(+)/CV. The training error on Newbob(+)/TR is smaller without mean-normalization which suggests that the algorithm does not improve the approximation. There is no distinctive difference between Newbob and Newbob+ either.

The second error raise occurred always right after the first one. Thus the only effect of Newbob+ is slowing down the training.

4.3.3 Bias initialization

In previous experiments, the initial biases are drawn from the $\mathcal{N}(\mu = -4, \sigma = 0.1)$ distribution. This initialization was previously successfully used on nearly identical setting [14]. Wiesler et al. do not use this method and that is why I have examined the models with zero biases as well.

Results

The results for models with zero initial biases are shown in table 4.5. The optimal learning rates (marked bold) are smaller than those for model with negative biases. Again, the mean-normalized model needs smaller learning rate than the standard one. Neither of the models performed better with zero initial biases.

4.3.4 Result summary

The experiments on model with sigmoid as the non linear function are similar to those with tanh. Their aim is to find best suitable parameters and to compare standard and

mean-normalized stochastic gradient descent.

The best results from model with sigmoid non-linear function are obtained when the initial learning rate is set to 0.01 (standard SGD) or 0.005 (mean-normalized SGD), its decay strategy to Newbob/CV, the weights initialized from random distribution $\mathcal{N}(\mu = 0, \sigma = 0.1)$ and the biases from $\mathcal{U}(-4.1, -3.9)$. The graphs of error and accuracy on cross-validation set are shown in figure 4.3. These results are lightly better than those of tanh.

Using different learning rates, no clear relation can be seen between the standard and mean-normalized model. Usually, the standard gradient descent works slightly better, but sometimes not. The differences between resultant errors are around 1 %. This is similar to the observation on tanh.

The use of Newbob(+)/TR proposed in [16] did not prove helpful as well. Both standard and mean-normalized method are affected by massive overfitting. The standard method yields smaller error on training set. When the biases are initialized to zeros, the optimal learning rates are smaller. The results of such models, however, are not as good as with negative initial biases. The performance of mean-normalized models is in either case slightly worse than that of standard models.

I have not found enough evidence to support the claim that the mean-normalize stochastic gradient descent is better in any way. In some cases, it is true but definitely not always. The models with sigmoid gives better results than those with tanh.

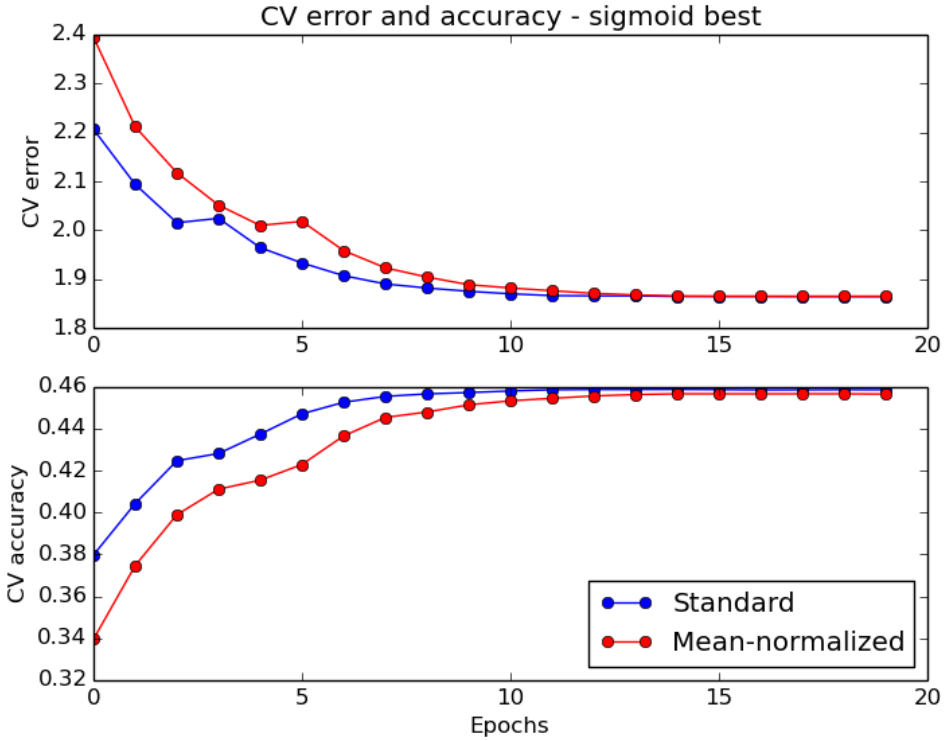


Figure 4.3: Sigmoid: Cross-validation error and accuracy of the best model.

4.4 Linear bottleneck

Linear bottleneck is an extra hidden layer inserted between existing two. Neurons in this layer have no built-in non-linearity (=linear) and the layer is smaller than the others (=bottleneck). Inserting a linear bottleneck is equivalent to factorizing the weight matrix into product of two smaller matrices. This allows reducing the number of parameters while keeping the learning capability. This can be useful when size of the network is in question. The bottleneck can be inserted into the network after training (by factorizing existing weights [12]) or before [18]. Training a network with bottleneck from scratch should be harder, especially with traditional methods [18]. According to Wiesler et al., the mean-normalized method should enable it without any pretraining.

However, the setting of Wiesler et al. [16] setting was somewhat different. They had huge output, more than 4000 context-dependent states of following HMM. Inserting linear bottleneck between the last hidden layer and output helped to reduce the number of weights and subsequently the size of the whole model by a ratio of $\frac{1}{8}$ without any loss in network performance.

This is impossible on the task of phoneme state classification, as the output is much smaller (only 135 classes) and the last weight layer does not contribute much to the size of the whole network. Biggest weight matrices are between inner hidden layers and therefore it is reasonable to insert the bottleneck there.

In the experiments, models with one, two and three bottlenecks are examined for a whole range of bottleneck sizes ($1/2$, $1/4$, $1/8$, $1/16$ and $1/32$ of hidden layer size). There are three full hidden layers of size 1024. The first two bottlenecks are inserted in between them, the third bottleneck between input and first hidden layer. All cases are compared to neural networks with similar number of weights but no bottleneck. This should help distinguish between the influence of having less weights and the bottleneck itself.

Same set of experiments was performed on both models with tanh and sigmoid function.

4.4.1 Tanh

With the hyperbolic tangent, the effect of linear bottleneck is clearly visible. First two bottlenecks (each inserted between two hidden layers) did not worsen the network performance. There is even a small improvement (marked in bold in table 4.6). Putting bottleneck between input and first hidden layer is always quite harmful.

Similarly to other models with tanh, no major difference between standard and mean-normalized model have been observed. Both models behaved similarly in every case.

Inserting linear bottleneck successfully reduced number of weights while preserving the network learning and generalization ability. That is something that cannot be done by simply reducing the number of neurons in hidden layers. When reducing the size this way, the cross-validation error gradually rises (see tables 4.9 and 4.10). Note the difference between models of the same relative size with and without bottleneck.

4.4.2 Sigmoid

The models with sigmoid and simple linear bottleneck were at first impossible to train. To enable the training, the linear bottleneck must have been modified to have similar slope as the non-linear function in surrounding hidden layers. I have used $y = 0.25x$, because $\text{sigmoid}'(0) = 0.25$. Also, the initial learning rate need to be decreased by an order of magnitude.

b-neck size	No. of b-neck	Relative size	Standard E_{CV}	Mean-normalized E_{CV}
–	0	100 %	1.92	1.93
512	1	100 %	1.92	1.93
	2	100 %	1.92	1.92
	3	110 %	1.96	1.97
256	1	80 %	1.92	1.92
	2	**60 %**	**1.91**	**1.92**
	3	58 %	2.00	2.01
128	1	70 %	1.92	1.93
	2	**40 %**	**1.91**	**1.92**
	3	33 %	2.06	2.08
64	1	65 %	1.92	1.93
	2	30 %	1.93	1.93
	3	18 %	2.22	2.20

Table 4.6: Tanh: Cross-validation error for different settings with linear bottleneck.

I have tried inserting one or two bottlenecks of size 1/2, 1/4 and 1/8 into models with negative or zero initial biases. The results are shown in tables 4.7 and 4.8.

b-neck size	No. of b-neck	Relative size	Standard E_{CV}	Mean-normalized E_{CV}
–	0	100 %	1.86	1.88
512	1	100 %	1.90	1.94
	2	100 %	1.93	1.94
256	1	80 %	1.91	1.91
	2	60 %	2.05	1.97
128	1	70 %	1.94	1.99
	2	40 %	2.02	2.05

Table 4.7: Sigmoid, negative biases: Cross-validation error for different settings with linear bottleneck.

The training of neural network with linear bottlenecks and sigmoid is indeed harder. After tuning the linear function in the bottleneck and adjusting the learning rates, I was able to train the network. I have not observed the benefits of the bottleneck (reducing the number of parameters while preserving the learning ability) while using sigmoid as non-linear function. The use of mean-normalized SGD did not have any positive effect on the network performance either.

To conclude, inserting a linear bottleneck may be useful and it can lead to smaller and better networks. To allow this, a network should be carefully tuned and/or pretrained. Otherwise the bottleneck would worsen the network performance and it may make the whole training process more difficult or even impossible.

b-neck size	No. of b-neck	Relative size	Standard E_{CV}	Mean-normalized E_{CV}
–	0	100 %	1.90	2.08
512	1	100 %	1.97	2.10
	2	100 %	2.03	2.18
256	1	80 %	1.92	1.97
	2	60 %	2.10	2.34
128	1	70 %	2.12	2.14
	2	40 %	2.11	2.28

Table 4.8: Sigmoid, zero biases: Cross-validation error for different settings with linear bottleneck.

4.5 Hidden layer size

The previous section was focusing on decreasing the number of parameters of a network by inserting a linear bottleneck. It was also shown that simple reduction of hidden layer size worsens the network performance. This section further examines the effect of changing (both increasing and decreasing) the hidden layer size. In theory, with small size the network loses its ability to fit the training data perfectly and it would generalize better. Larger networks are better at approximation but needs more training data in order to generalize [2]. In each experiment, there are three hidden layers of the same size. The size varies from 256 to 2048. Other parameters are equal to those from section 4.2.3 and 4.3.4 for tanh and sigmoid respectively.

4.5.1 Results

For hyperbolic tangent, the results are somehow expectable and tendency can be clearly seen (see table 4.9). The error on cross-validation set is bigger in smaller networks. The differences between standard and mean-normalized gradient descent are the same as in all the previous experiments with hyperbolic tangent.

Hidden layer size	Relative size	Standard E_{CV}	Mean-normalized E_{CV}
256	10 %	2.04	2.04
384	19 %	1.99	1.99
512	30 %	1.97	2.00
640	43 %	1.96	1.96
768	60 %	1.94	1.95
896	78 %	1.93	1.94
1024	100 %	1.92	1.93
1152	124 %	1.91	1.92
1280	150 %	1.91	1.91
1408	180 %	1.90	1.91
1536	210 %	1.90	1.91
1664	245 %	1.89	1.90
1792	280 %	1.89	1.90
1920	320 %	1.89	1.90
2048	360 %	1.89	1.93

Table 4.9: Tanh: Cross-validation error for models with different hidden layer size.

Hidden layer size	Standard E_{CV}	Mean-normalized E_{CV}	rel. improvement
256	1.98	2.01	-1.35 %
384	1.93	1.95	-1.03 %
512	1.91	1.92	-0.57 %
640	1.89	1.91	-1.15 %
768	1.87	1.89	-0.95 %
896	1.90	1.87	1.60 %
1024	1.86	1.90	-1,92 %
1152	1.88	1.86	0.91 %
1280	1.88	1.85	1.63 %
1408	1.88	1.85	1.84 %
1536	1.90	1.86	1.94 %
1664	1.87	1.84	1.41 %
1792	1.87	1.87	0.17 %
1920	1.87	1.84	1.54 %
2048	1.88	1.84	2.26 %

Table 4.10: Sigmoid: Error on cross-validation dataset for models with different size and relative improvement of mean-normalized stochastic gradient descent.

With the sigmoid, the expected tendency is observed as well. However, in larger networks, there is a considerable improvement when using the mean-normalized stochastic gradient descent. This enhancement gets bigger with the size of the network (see table 4.10). The best results are obtained from topology with three hidden layers of size 2048 and are show in figure 4.4. This is in agreement with the result of Wiesler et al. They have found the mean-normalized SGD particularly useful when training networks with very large number of parameters.

4.6 Deep networks

This set of experiments is designed to compare performance of standard and mean-normalized method on different network topologies. The number of hidden layers is chosen arbitrary (1, 2, 3, 6, 12, 18 and 24), but sizes of the layers are calculated so that the total number of weights is always roughly the same as in the reference model.

4.6.1 Results

For shallow networks with hyperbolic tangent, the standard and mean-normalized method behave again in a similar way. With one and two hidden layers the results are quite poor. The best results are obtained from the model with 6 hidden layers. This topology even gives better results than the reference model ($E_6 = 1.912 < E_3 = 1.923$), see figure 4.5.

When sigmoid is used, the best results are given by the basic setting of three hidden layers. Reducing the number of hidden layers caused degradation of the results.

Deep deeper networks (12, 18 and 24) are gradually harder to train and do not converge to better solutions. It is still possible to train them with hyperbolic tangent. The sigmoid, however, fails to train networks deeper than five hidden layers even when using mean-normalized stochastic gradient descent.

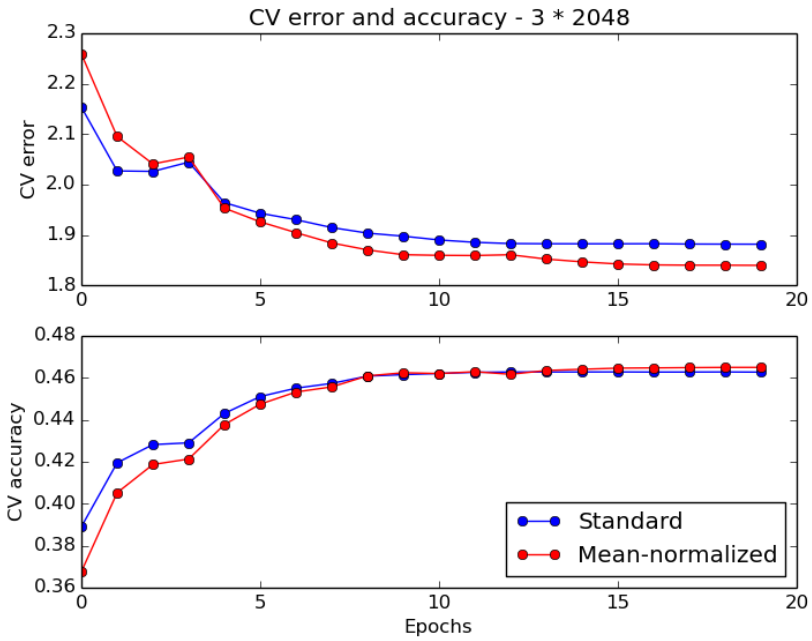


Figure 4.4: Sigmoid: Cross-validation error and accuracy of a large network with three hidden layers each with 2048 neurons ($2048 \times 2048 \times 2048$).

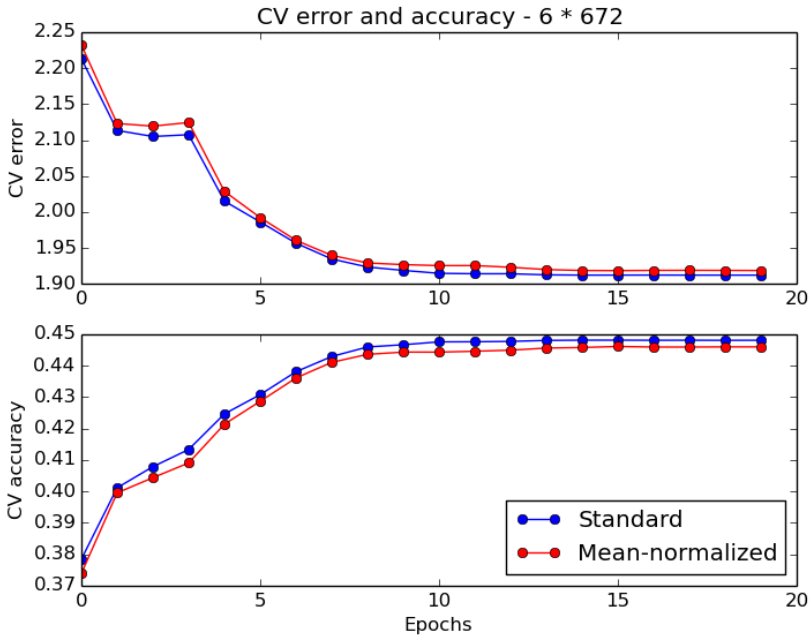


Figure 4.5: Tanh: Cross-validation error and accuracy of a deep network with 6 hidden layers each with 672 neurons.

Chapter 5

Conclusion

In this work, the mean-normalized stochastic gradient descent recently proposed by Wiesler et al. from RWHT Aachen University is examined in detail. Fist, the motivation for this method is explained in section 3.1. Then a detailed step-by-step derivation of the algorithm is provided in section 3.3. The most important part of this work is the experiments on phoneme classification task described in section 4. Their aim is to test the mean-normalized stochastic gradient descent in a number of different settings, try to replicate the results of Wiesler et al. and decide whether the method is better and under what circumstances.

With the hyperbolic tangent as a non-linear function the mean-normalized SGD do not yield improvements of any kind. Usually, both the accuracy of classification and frame error are worse by 1–2 %. It is possible to train network with a linear bottleneck and without the loss in accuracy while reducing the network size by 60 %. I was able to train arbitrary deep networks without difficulty, but adding more layers did not help to improve the performance significantly. Possible explanation why the mean-normalized stochastic gradient descent does not work well with hyperbolic tangent is that the values produced by the tanh() are from interval $(-1, 1)$ and therefore their mean value is already near zero.

Models with sigmoid produce better results than models with tanh in the basic setting. The mean-normalized SGD do not lead to any improvements and it needs smaller learning rate in order to produce the best possible results. It is possible to train neural networks with linear bottleneck but only after further decreasing of the initial learning rate and adjusting the linear function as described in section 4.4.2. The use of mean-normalized SGD has no positive inputs on training networks with linear bottleneck. In the contrary, such networks require further decreasing of the learning rate (to even start training) which leads to slower learning and worse results. With the sigmoid, I was unable to train networks deeper than 4 layers regardless of the mean-normalization.

I can confirm the observation of Wiesler et al. that the mean-normalized SGD is particularly useful when training larger networks. After increasing the number of neurons in hidden layers, the mean-normalized model produces better results than the standard one. The use of nearly 3.5× larger network lead to a small improvement by 2 %.

The mean-normalized SGD could have some positive impact, but only under specific circumstances. The network has to be large, not too deep and the sigmoid function must be used. When the network is smaller, deeper or uses the hyperbolic tangent along with mean-normalization, the results are often worse. Therefore, the mean-normalized stochastic gradient descent cannot be recommended as a general optimization technique.

Bibliography

[1] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4):185–196, 1993.

[2] Christopher M Bishop et al. *Neural networks for pattern recognition*. Clarendon press Oxford, 1995. ISBN-13: 978-0198538646.

[3] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

[4] Christian Darken and John Moody. Towards faster stochastic gradient search. In *NIPs*, pages 1009–1016, 1991.

[5] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[6] David Johnson, D Ellis, C Oei, C Wooters, P Faerber, N Morgan, and K Asanovic. Icsi quicknet software package. *h ttp://www. icsi. berkeley. edu/Speech/qn. html*, 2004.

[7] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[8] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

[9] Usenet newsgroup comp.ai.neural nets. Faq. online, 2012. ftp://ftp.sas.com/pub/neural/FAQ.html.

[10] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

[11] Herbert Robbins and David Siegmund. A convergence theorem for non negative almost supermartingales and some applications. In *Herbert Robbins Selected Papers*, pages 111–135. Springer, 1985.

[12] Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6655–6659. IEEE, 2013.

[13] Peter Sunehag, Jochen Trumpf, SVN Vishwanathan, and Nicol Schraudolph. Variable metric stochastic approximation theory. *arXiv preprint arXiv:0908.3529*, 2009.

[14] Karel Veselỳ, Lukáš Burget, and František Grézl. Parallel training of neural networks for speech recognition. In *Text, Speech and Dialogue*, pages 439–446. Springer, 2010.

[15] Oriol Vinyals and Daniel Povey. Krylov subspace descent for deep learning. *arXiv preprint arXiv:1111.4259*, 2011.

[16] Simon Wiesler, Alexander Richard, Ralf Schluter, and Hermann Ney. Mean-normalized stochastic gradient for large-scale deep learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 180–184. IEEE, 2014.

[17] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

[18] Jian Xue, Jinyu Li, and Yifan Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *INTERSPEECH*, pages 2365–2369, 2013.