Graz University of Technology

442.022 - Nonlinear Signal Processing

Assignment 2

Author: Sophie Steger, Felix Stollberger
Date: Graz, June 17, 2021

Contents

1 Fading Memory and BIBO Stability

1.1 Problem 1

Consider a linear, causal and time-invariant system with impulse response h[n]. Further, we consider only bounded input signals x[n]. **Hint:** In case you need to refresh your memory about these terms, e.g. Oppenheim and Schafer 2009 may answer your questions.

1.1.1 Task a)

Show that such a system is a fading memory system if and only if the system is Bounded Input Bounded Output (BIBO) stable. **Hint:** A linear system is BIBO stable, if and only if it's impulse response is absolutely summable.

DEFINITIONS [5]

A discrete-time system is a transformation or operator $T\{\cdot\}$ that maps the input sequence $x[n]$ into the output sequence $y[n]$: $y[n] = T\{x[n]\}$

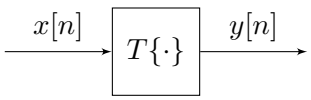


Figure 1.1: Block diagram of a discrete-time system

If the system $T\{\cdot\}$ is linear and time-invariant, the system can be characterized by its impulse response $h_k[n] = T\{\delta[n-k]\}$ where $\delta[n-k]$ is the Kronecker delta with an impulse at time step $n = k$.
In this assignment, the system is linear, time-invariant and causal. In this case, the system response can be calculated as follows:

$$
\begin{aligned}
y[n] &= T\left\{\sum_{k=-\infty}^{\infty} x[k]\delta[n-k]\right\} \\
&= \sum_{k=-\infty}^{\infty} x[k]\, T\{\delta[n-k]\} && \text{..... principle of superposition (linearity)} \\
&= \sum_{k=-\infty}^{\infty} x[k]h_k[n] \\
&= \sum_{k=-\infty}^{\infty} x[k]h[n-k] && \text{..... time-invariance} \\
&= \sum_{m=-\infty}^{\infty} x[n-m]h[m] && \text{..... substitution of variables: } m = n-k \\
&= \sum_{m=0}^{\infty} x[n-m]h[m] && \text{..... causality: } h[m] = 0,\ m < 0
\end{aligned}
$$

Definition: BIBO stability [5]

A system is called BIBO stable if and only if every bounded input sequence produces a bounded output sequence. The input $x[n]$ is bounded, therefore an upper bound B_x exists such that

$$x[n] \leq B_x < \infty \quad \forall n$$

If the system is BIBO stable, for every input there exists an upper bound B_y for the output such that

$$y[n] \leq B_y < \infty \quad \forall n$$

Linear, time-invariant systems are stable if and only if the impulse response $h[n]$ is absolutely summable:

$$|y[n]| = \left| \sum_{k=-\infty}^{\infty} h[k]x[n-k] \right| \leq \sum_{k=-\infty}^{\infty} |h[k]||x[n-k]| \leq B_x \overbrace{\sum_{k=-\infty}^{\infty} |h[k]|}^{S} = B_x \cdot S$$

Therefore, the output sequence $y[n]$ is bounded if $S < \infty$. For completeness, one needs to show that if $S = \infty$, a bounded signal can be found that results in an unbounded output. This is left as an exercise for the reader. ☺

Definition: Fading Memory (FM) System [4]

A fading memory function $w[n]$ fulfills the following properties:

$$w[n] \in (0, 1], \quad \lim_{n \to \infty} w[n] = 0$$

Furthermore, we assume a causal and time-invariant operator and thus set the reference time $t_0 = 0$ w.l.o.g.

Then a system $T\{\cdot\}$ is denoted as fading memory (FM) if

$\forall \epsilon > 0 \quad \exists \delta = \delta(\epsilon)$ such that $\forall\ x_1[n],\ x_2[n]$ that satisfy

$$\sup_{n \leq 0} |x_1[n] - x_2[n]|\ w[-n] < \delta = \delta(\epsilon)$$

it follows that $|y_1[0] - y_2[0]| < \epsilon$

Solution

First, we show that a system is fading memory if it is BIBO stable. Since all input signals $x[n]$ are bounded, there exists an upper bound B_x also for the difference between two input signals $|x_1[n] - x_2[n]| \leq B_x < \frac{\delta}{w[-n]} \quad \forall n$. If the system is BIBO, the impulse response is absolutely summable $\sum_{k=-\infty}^{\infty} |h[k]| = S < \infty$.

$$|y_1[0] - y_2[0]| = \left| \sum_{k=-\infty}^{\infty} h[k]x_1[-k] - \sum_{k=-\infty}^{\infty} h[k]x_2[-k] \right| \tag{1.1}$$

$$= \left| \sum_{k=-\infty}^{\infty} h[k](x_1[-k] - x_2[-k]) \right| \tag{1.2}$$

$$\leq \sum_{k=-\infty}^{\infty} |h[k]|\,|(x_1[-k] - x_2[-k])| \tag{1.3}$$

$$\leq B_x \sum_{k=-\infty}^{\infty} |h[k]| \tag{1.4}$$

$$= B_x \cdot S < \epsilon < \infty \tag{1.5}$$

Finally, we need to show that a system is BIBO stable if it is fading memory. If a system is fading memory, the following property is fulfilled:

If $\sup_{n\leq 0} |x_1[n] - x_2[n]| < \frac{\delta}{w[-n]}$, then $|y_1[0] - y_2[0]| < \epsilon < \infty$.

Inserting this inequality into Equation 1.4, we get

$$|y_1[0] - y_2[0]| \leq \sum_{k=-\infty}^{\infty} |h[k]|\,|(x_1[-k] - x_2[-k])| \tag{1.6}$$

$$< \delta \sum_{k=-\infty}^{\infty} \frac{|h[k]|}{w[k]} \tag{1.7}$$

W.l.o.g. we assume that $w[n] = 1\ \forall n$, then

$$|y_1[0] - y_2[0]| < \delta \sum_{k=-\infty}^{\infty} |h[k]| < \epsilon \quad \text{..... } \textit{fading-memory assumption} \tag{1.8}$$

Thus $\sum_{k=-\infty}^{\infty} |h[k]| = S < \infty$ and the system is BIBO stable.

1.1.2 Task b)

Assume h is a finite impulse response (FIR) filter with 10 coeffcients. Is the window

$$w[n] = \begin{cases} 1, & 0 \leq n < 14 \\ 0, & \text{otherwise} \end{cases} \tag{1.9}$$

a valid memory fading function? Why/why not?

Solution

The FIR filter has 10 coefficients, therefore the impulse response is absolutely summable $\sum_{n=0}^{9} h[n] = S < \infty$ and the system is fading memory regardless of the window function $w[n]$.

If $\sup_{n\leq 0} |x_1[n] - x_2[n]| < \frac{\delta}{w[-n]}$, then

$$|y_1[0] - y_2[0]| < \delta \sum_{k=-\infty}^{\infty} \frac{|h[k]|}{w[k]} \tag{1.10}$$

$$= \delta \sum_{k=0}^{9} \frac{|h[k]|}{w[k]} \tag{1.11}$$

$$= \delta \sum_{k=0}^{9} |h[k]| < \epsilon < \infty \tag{1.12}$$

1.1.3 Task c)

Is w[n] a valid memory fading function for an infinite impulse response (IIR) filter? Why/why not?

Solution

If $\sup_{n\leq 0} |x_1[n] - x_2[n]| < \frac{\delta}{w[-n]}$, then

$$|y_1[0] - y_2[0]| \leq \sum_{k=-\infty}^{\infty} |h[k]|\,|(x_1[-k] - x_2[-k])| \tag{1.13}$$

$$= \sum_{k=0}^{13} \underbrace{|(x_1[-k] - x_2[-k])|}_{<\delta} |h[k]| + \sum_{k=14}^{\infty} \underbrace{|(x_1[-k] - x_2[-k])|}_{<\infty} |h[k]| \tag{1.14}$$

But regardless of the window function $w[n]$, the input signal difference is bounded by a finite value as we consider only bounded input signals, therefore $\sup_{n\leq 0} |x_1[n] - x_2[n]| \leq B_x$ and

$$|y_1[0] - y_2[0]| \leq \sum_{k=-\infty}^{\infty} |h[k]|\,|(x_1[-k] - x_2[-k])| \tag{1.15}$$

$$\leq B_x \sum_{k=-\infty}^{\infty} |h[k]| \overset{?}{<} \epsilon \tag{1.16}$$

If h is absolutely summable, so the IIR system is BIBO stable, $w[n]$ is a valid window function and the system is fading memory. Otherwise no upper bound ϵ can be found.

2 Speech Recognition

2.1 Problem 2

This task deals with speech recognition using convolutional neural networks (CNN) to identify several keywords from approximately one second long audio recordings. The model development and evaluation was done on a subset of the Warden 2018 Speech Commands dataset [6], which contained five different words: *four, go, no, stop* and *yes*. The implementation was done using PyTorch, and the provided code skeleton.

2.1.1 Task a) - Implementing the model

Following from the problem description, a one-dimensional convolution layer, followed by a fully-connected hidden layer and an output layer with a log-softmax activation function should be used as basic structure for the speech recognition model. Beside this, the architecture was not specified and could be freely chosen by us. Before describing the details of our model, it is worth while to consider the shape and dimensionality of the input data for the model. This data is provided by the data loader, which was already implemented in the given skeleton.

The speech recognition model works not directly with the given audio recordings, but rather with the 40 log MEL Frequency Cepstral Coefficents (MFCCs), which where computed using `mfcc_transform()` for each given audio sequence. The output of this transform is a two dimensional signal of the form $[X, Y]$ where $X = 40$ is the number of features in the MFCC spectrum. The size of the other dimension depends on the signal length, and is typically in the range of 40 to 80. Model training is performed by processing batches of MFCCs in the network. These batches are created in the data loader and could have an arbitrary integer size, smaller than the total number of files in the training set. In our case, we observed an increased model performance when training the NN with smaller batches of size 25 to 50. Therefore, the dimensionality of the data feed to the model can be expressed as:

Dimensionality of the input data for the 1D-CNN model

$$\text{data.shape} = [B, 1, F, Y]$$

B... batch size, $F = 40$ ... number of features, Y... signal length

As the data loader is also used for the 1D-CNN, as well as the 2D-CNN, which requires a four dimensional input, the single second dimension is not removed at this point. Instead, it is removed internally in the 1D-CNN by applying `torch.squeeze(data)`.

The final implementation of our model consists of a 1D-CNN with 40 in channels (equal to the number of features) and 20 convolution kernels with a size of 12. The output of the convolution layer (output.size = [B, 20, Y]) is evaluated with a ReLu-activation function, and averaged across the third dimension to eliminate problems with variable signal length. The result is then passed to a MLP with 20 input channels (equal to the number of convolution kernels), and 1024 hidden neurons. This output was again evaluated with a ReLu-activation function and passed to a linear log-softmax output layer with 1024 in channels, and 5 output channels.

In the training framework, the learning rate of the optimizer was increased to 0.001, while decreasing the batch size from 256 to 25. With this setup, the model has 36289 trainable parameters and could archive an accuracy up to 96% after a few training epochs. For the submission of the assignment, the number of epochs was reduced to three, to limit the computation time while still maintaining a sufficient accuracy. This model takes around 48s training time on an Intel i7 hexacore processor, and archives typically an accuracy of 92-94%.

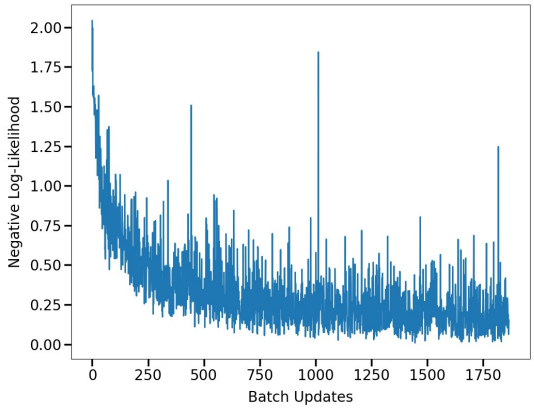


Figure 2.1: Training loss as a function of batch iterations. Overall performance 94% after 3 epochs.

Figure 2.1 shows the training loss of the model as a function of the batch iterations. At the beginning, a very steep decrease of the training loss is visible, followed by a linear decrease with minimal slope. Although the model performance would slightly increase with more batch iterations, the relative performance change is not worth the increased computational effort for our application.

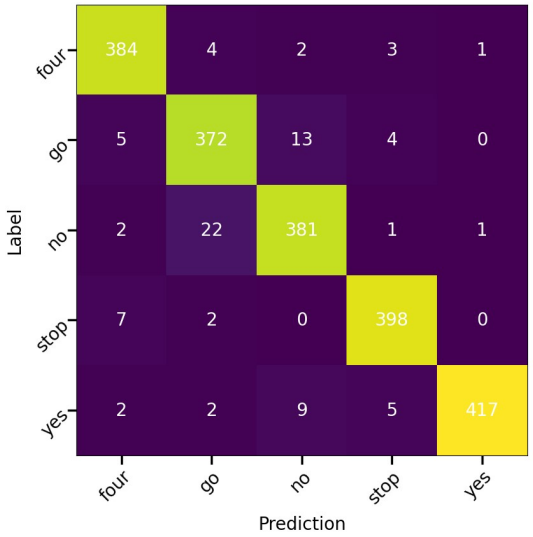


Figure 2.2: Confusion matrix of a model with 96% accuracy.

For a better evaluation of the model performance, a confusion matrix (see Figure 2.2) was created in addition to the absolute accuracy. This matrix shows the predicted class of each sample from the test set in the columns, versus its actual label in the rows. For example, 13 samples labeled with "go" where falsely predicted as "no" by the model, whereas 22 samples with the label "no" where predicted as "go".

The optimization of the model parameters was carried out manually. The number of vari-

able parameters, and the computation time for model evaluation prevented us from using a systematical approach. Therefore, this model could be definitely further optimized to reduce the number of trainable parameters, the computation time or the final accuracy. Nevertheless, this parameter set outperformed any other sets we have evaluated.

2.1.2 Task b) - Limiting internal connections

For this task, the parameter `groups` in `cov1D()` should be set equal to the number of features (40). When the parameter groups=X is set, the input channels are segregated into X groups, with individual convolution layers. The output of those (sub-)layers are than combined to the final output channel of the layer.

Grouping filter kernels is a relatively new topic and was first described by Krizhevsky et al. [3] in 2012, as they had to train a large CNN on multiple GPUs for image recognition. Surprisingly, the authors observed a specialization of the individual kernels on different features (color and texture), persisting even over different random initializations. Furthermore, the network performance was slightly better compared to conventional architectures, with 57% less connection weights. [1] Further work on this topic had shown that limiting the internal connectivity of the networks has generally a beneficial effect on the overall performance. As observed by Krizhevsky et al. [3], individual filters could than specify on special features of the input signal. This reduced connectivity reduces also the computational complexity and overall model size.

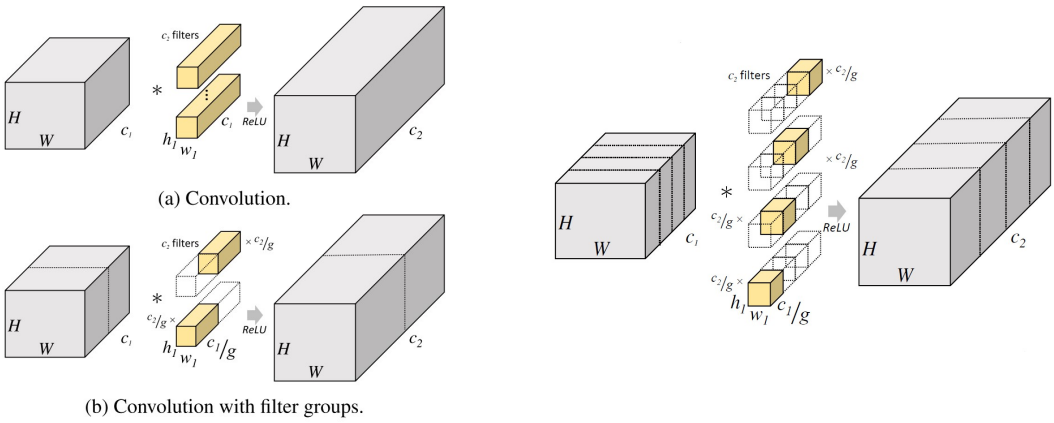


Figure 2.3: Schematic drawing of filter grouping in a 2D-CNN (From [1])

Figure 2.3 shows a basic sketch of a 2D CNN, with-, and without filter groups [1]. The later case could be considered as several individual CNNs side by side, with concatenated output of all channels.

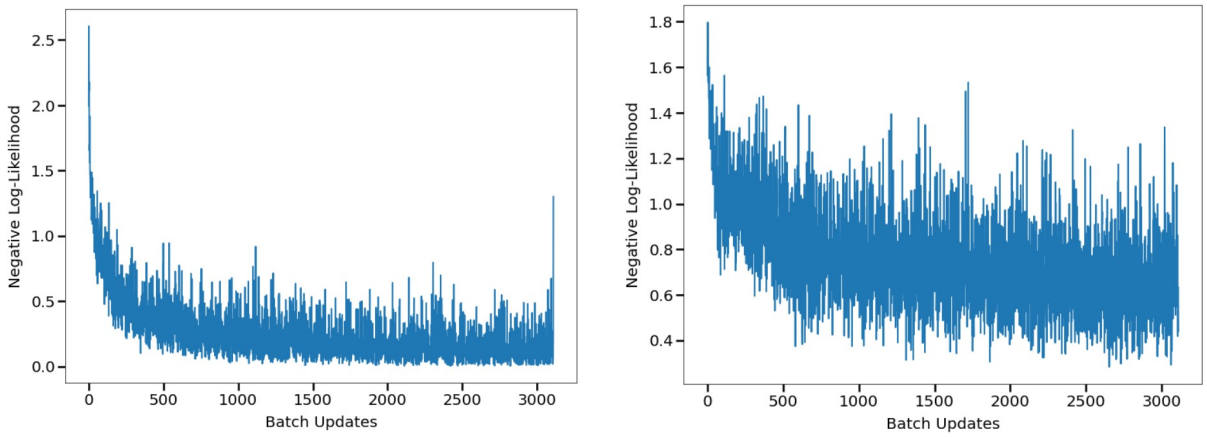
In our case, the number of groups had to be equal to the number of features, which was also the number of input channels. This corresponds to a network were the smallest group size is reached, and each input channel is separately convolved with its own convolution kernel of size $[\frac{K}{F}, M]$. Here K denotes the number of kernels, F the number of features and M the memory depth of the kernels. For the chosen parameters of our network, the size of the individual kernels was $[1, 40]$.

A problem when implementing the grouped CNN was that the number of kernels had to be divisible by `groups`. Our optimal model from task a) had only 20 kernels, which violated this

[1] The basic idea behind grouping could be translated to 1D-CNNs (as in our case) without loss of generality.

condition. Therefore, we increased the number of kernels to $K = 40$, and expanded the memory depth M form 12 to 20. Hence, the results were no longer comparable to the original model described in task a. To overcome this issue, the original model was also evaluated with this parameter set.

Setting `groups = num_features` causes an overall reduction of the number of trainable parameters from 79229 to 48029, which corresponds to 39.3%. In contrast to that, the accuracy of the model decreased from around 94% to 72% with the same training environment as in task a). A comparison of the learning rates can be seen in Figure 2.4.



(a) Original model (b) Grouped input

Figure 2.4: Training loss of the original model in comparison to the model with grouped input.

As visible in the figures, the training loss of the grouped convolution decreases much slower compared to the original system. It seems as if the model was not able to adapt well to the input data, as some training loss remained even after several thousand batch updates. Increasing the number of epochs had also no effect on the model performance, as a significant training loss remained. As a consequence, the accuracy of the model hardly exceeded 74-76%, which is way lower than the original model. Even the adapted model with increased kernel number outperformed this model by a more than 18% higher accuracy. Therefore, grouping in this form is not a favorable approach for our problem. A reason for this could be that the size of the input groups is to small, and information contained in the connection of an input channel to its surrounding channels is not considered by this individual filtering approach. To overcome this problem, the group size could be increased, such that several input channels form a group (eg. using 4 or 8 groups). Such group sizes where compatible with our original model, and showed similar results as in task a), while decreasing the overall number of trainable parameters.

2.1.3 Task c) - Two dimensional CNN

Two-dimensional CNNs are very common layers in NN, and are often used for image-, or video recognition. For this task, we had to implement such a 2D-CNN, and investigate the performance on our given data set.

Changing from a one-dimensional convolution layer to a two-dimensional layer required some adaptions on the internal structure of our model. The size of the filter kernel was given by the problem description, and should have one dimension equal to the number of features. For the second dimension, we have chosen the variable parameter `mem_depth`, which was set to 12 for the evaluation of the model. The size of the fully connected hidden layer had to be adapted, as

the dimensionality, and size of the output of the convolution layer changed. Therefore, a quick discussion of the dimensions of the signals and layers is reasonable.

The structure of the data-loader allowed us to use the same dimension of the input data for the 2D-CNN as for the 1D-CNN, described in task a). The only difference now is that the single, second dimension is not removed in the network, as it corresponds to the single input channel of our 2D-CNN. The implementation of the CNN used $K = 5$ kernels with a size of $[40, 5]$, which where convolved with the input data of size $[B, 1, F, Y]$. [2]
This convolution resulted in an output data size of $[B, K, 1, W_{out}]$, where W_{out} was dependent on the variable signal length Y. This dependency was again removed by taking the average value across this dimension. In addition to that, `torch.flatten(x, start_dim=1)` was applied on the averaged result to archive a two dimensional signal which could be passed to the fully connected, hidden MLP with 1024 hidden neurons. `torch.flatten` reduces hereby the dimension of the signal, and projects the third-, and fourth dimension into the second one. This corresponds to squeezing the matrix from four to two dimensions, as third and fourth dimension where both one.

Using this hyperparameter set resulted in a model with 36289 trainable parameters - the same amount as in the 1D-CNN. In the two-dimensional implementation the computed MFCC spectrum of the signal could be directly processed by the network as 2-dimensional input data. In contrast to that, the one-dimensional network treated every feature of the MFCC as individual channel. However, the way that we set the parameters of the 2D-CNN we could achieve the exact same behavior as the 1D-CNN (also by setting `padding = 0`). Therefore, the model achieved the same accuracy as denoted in task a. Note that a change in the parameters of the 2D-CNN also changes its behavior and it can also differ from the 1D-CNN.

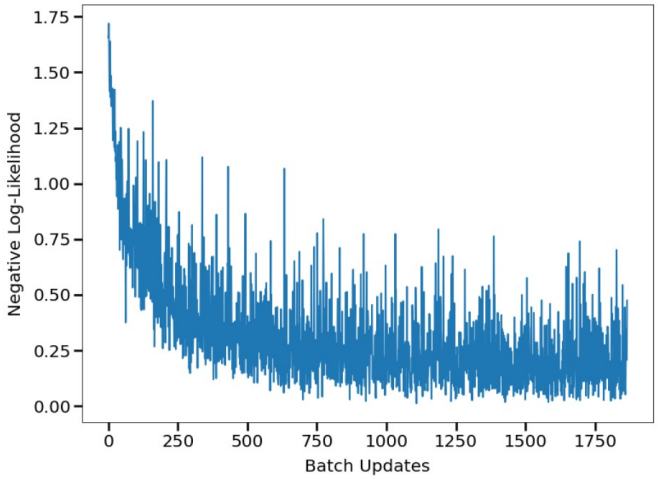


Figure 2.5: Training loss as a function of batch iterations for the 2D-CNN. Overall performance 94% after 3 epochs.

2.1.4 Task d) - BatchNorm layer

Theoretical Background:

All the following information about BatchNorm layers is taken from the paper "*Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*" [2].

[2] The same nomenclature was used as in task a)

Deep Neural Networks (DNN) are used for many different applications, e.g. in our case for automatic speech recognition. The goal is to optimize some parameters Θ according to a chosen cost function $\mathcal{C}(\cdot)$

$$\Theta = \arg\min_{\Theta} \frac{1}{N} \sum_{i=1}^{N} \mathcal{C}(x_i, \Theta)$$

The parameters are typically trained using a stochastic gradient descent (SGD). There, the gradient of the loss function is not computed for the whole training set but for a smaller subset called mini-batch. This is just an estimate of the gradient but proven to be highly effective.

However, a drawback of the SGD is that the quality of the training is dependent on the hyperparameters and the initial values of the model, which have to be chosen appropriately. Especially when working with DNNs consisting of many layers, the parameters of the initial layers strongly influence the behaviour of succeeding layers. Therefore, it becomes difficult to predict how small changes affect the behaviour of the network and individual layers.

The input data to a neural network has a certain distribution. When the data then passes a layer of the DNN with a nonlinear activation function, the distribution of the data changes - this effect is denoted in the paper as *covariate shift*. It has been known that whitening the input of the DNN (transformation to achieve zero mean and unit variance) leads to faster convergence. Now it also has been shown that whitening of inputs to individual layers of the DNN can improve the learning process immensely.
Let's consider a simple example why the distribution of the input data affects the learning process: Let $g : x \to y$ be a nonlinear activation function $g(x) = \frac{1}{1+\exp(-x)}$. With $|x|$ increasing, the derivative $g\prime(x)$ tends to zero and the model will train slower.

The authors of the paper proposed now a method to normalize the distribution (i.e. normalize the mean and variance) of the data to internal layers of the DNN, called *Batch Normalization*. In experiments they showed that this method achieves faster convergence while training the model. Batch Normalization allows higher learning rates without divergence and minimizes the risk of data being stuck in the saturation region of a nonlinearity as mentioned before.

Full whitening of data leads to zero mean, unit variance distribution, and decorrelation of each dimension. For this we need to compute the covariance matrix $\mathrm{Cov}[x] = \mathbb{E}_{x\in\mathcal{X}}[xx^T] - \mathbb{E}[x]\mathbb{E}[x]^T$ as well as its inverse square root in order to whiten the data according to $\mathrm{Cov}[x]^{-\frac{1}{2}}(x - \mathbb{E}[x])$. Additionally, when using backpropagation to train the network parameters, we not only need the normalization function, but also its derivative with respect to a given training example x and all examples $\mathcal{X}$ [3]. These operations are computationally highly expensive and the normalization function is required to be differentiable everywhere.

Instead, we disregard the decorrelation and perform whitening on each dimension separably. Let $x = (x^{(1)} \dots x^{(d)})$ be a d-dimensional input. Each dimension is normalized with

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}] + \epsilon}}$$

where ϵ is added for numerical stability.
The expectation and variance should be computed over the whole training set. However, when working with a stochastic gradient descent, these values are computed over each mini-batch.

[3] for more detailed explanations see [2]

Additional parameters, which are trained along the model parameters, are introduced to scale and shift the normalized data

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}.$$

Experiments:

In our application for speech recognition we used a convolutional layer, a fully connected hidden layer - both with relu activation function - and an output layer. The output layer is followed by a softmax activation function as our application is a non-binary classification task. In this task we added a batch normalization layer after the convolutional layer. It can be observed that the network achieves higher test accuracy in fewer epochs with the same hyperparameters as in task a). Due to the Batch Normalization we could also change the hyperparameters of the network and increase the learning rate for faster training.
Overall, we can observe that even for a quite simple network as ours - one convolutional layer and one fully connected hidden layer - the batch normalization can greatly improved the overall performance of the model and the training time. Although our model in task a) achieved quite high accuracy in few epochs, the model in task d) achieved the highest accuracy in less epochs. Figure 2.6 shows the the training loss as a function of the batch iterations. In comparison to

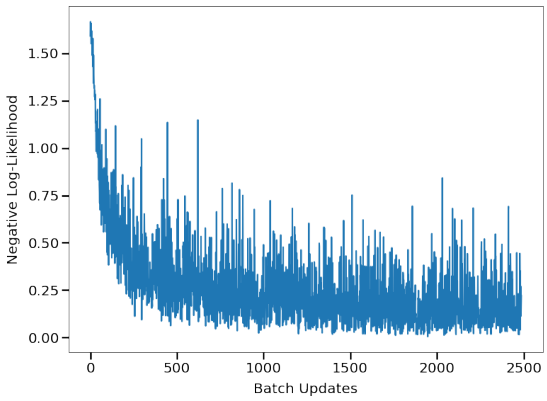


Figure 2.6: Training loss as a function of batch iterations. Overall performance 93% after 1 epoch.

Figure 2.1 we can observe, that the negative log-likelihood of our new model decreases faster. With the exact same hyperparameters as in task a) we can achieve an accuracy of 92-95% already after only one epoch.

2.1.5 Task e) - Max pooling layer

Theoretical Background [8]:

Pooling layers are often used in combination with convolutional neural networks. They reduce dimensionality of the input data. Max pooling layer select the maximum value of the data within the selected subsections of the data and discards the rest of the data points. Figure 2.7 shows the idea behind this concept for a 2d-pooling layer. The same principle can be applied to a 1d-pooling layer that we used in this task.

Max pooling layer can reduce computational complexity and thus allow to build deeper layers. Furthermore, it can be used to prevent overfitting to some degree.

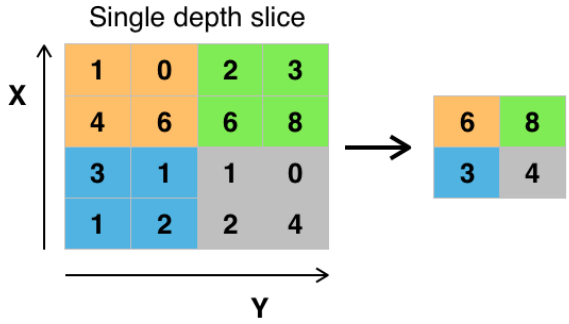


Figure 2.7: Max pooling with a 2x2 kernel size. (From [7])

Experiments:

In this task we added a max pooling layer after the convolutional layer to the network specified in task a). For the max pool layer we used a kernel size of 30. Figure 2.8 again shows the the training loss as a function of the batch iterations. Without changing any hyperparameters of the model in task a), we could achieve an accuracy of around 93% after one epoch and 97% after 4 epochs.

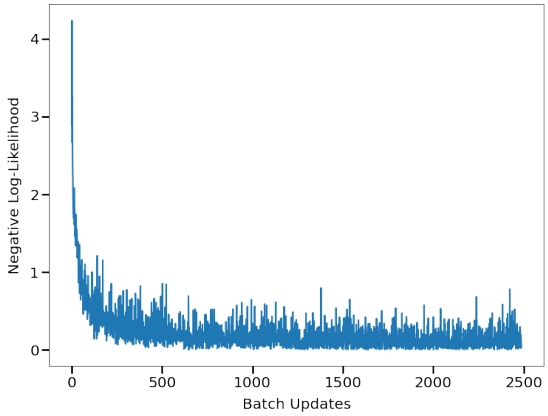


Figure 2.8: Training loss as a function of batch iterations. Overall performance 93% after 1 epoch.

List of Figures

Bibliography

[1] Ioannou, Y., Robertson, D., Cipolla, R., and Criminisi, A. (2017). Deep roots: Improving CNN efficiency with hierarchical filter groups. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-January:5977–5986.

[2] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[3] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

[4] Kubin, G. and Vogel, C. (2019). Lecture slides, nonlinear signal processing.

[5] Oppenheim, A. V. and Schafer, R. W. (1989). *Discrete-time Signal Processing* -. Prentice Hall, London.

[6] Warden, P. (2018). Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR*, abs/1804.03209.

[7] Wikipedia (2021a). Aphex34, public domain, via wikimedia commons. Available at `https://commons.wikimedia.org/wiki/File:Max_pooling.png` (2021/06/17).

[8] Wikipedia (2021b). Convolutional neural networks. Available at `https://en.wikipedia.org/wiki/Convolutional_neural_network` (2021/06/17).