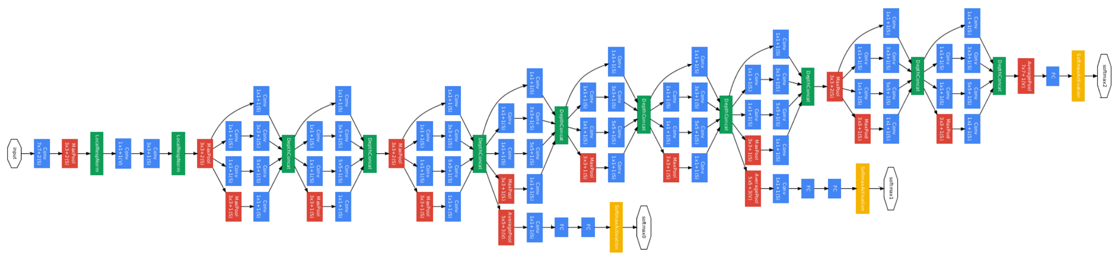
Deep Learning

Laboratory Course Manual

2020

Instructors : Jan van Gemert, David M.J. Tax
Attila Lengyel, Osman S. Kayhan, Seyran Khademi, Xin Liu, Ziqi Wang



Contents

Assignments:

Week1: Chapters 1 & 2
Week2: Chapters 3 & 4
Week3: Implement Chapter 5 (obligatory), Chapters 6 & 7 (optional)

Chapter 1

Introduction

The Deep Learning lab will cover the practical aspect of concepts introduced during the lectures. The assignments of the laboratory are not graded and these lab sessions are not mandatory. In the 3rd quarter, the lab sessions will provide a gentle introduction to Pytorch. The aim is to provide all students with the tools to work on a deep learning project of their own design in the 4th quarter. If anything is unclear regarding the exercises, please discuss with your peers or contact any of the teaching assistants during the lab sessions.

Objectives When you have done the exercises for this week, you

- have a working installation of Python, Numpy, Scipy and PyTorch.
- understand the basics of working with the above

To proceed with the lab, you are free to use the tools that you are most comfortable with. You can work with any software library, editor etc., of your choosing. However, there is no guarantee that the TAs are familiar with your tools.

1.1 Setting up the environment

- Try to use Google Colab which comes with a limited amount of free GPU compute time but requires a Google account. This link will walk you through installing PyTorch on colab.
- If you prefer to use Miniconda for the practical sessions, follow the instructions at this link to install Miniconda on your system.
- Create a virtual environment and setup Pytorch in it with the following commands
  - `conda create -n` *`myenv`*
  - `source activate` *`myenv`*
  - `conda install -c pytorch pytorch torchvision`

  For windows users, replace the last command with the following

    – `conda install pytorch-cpu -c pytorch`

- IPython notebooks present a great way to tinker with the code you create. You can run a notebook by `jupyter notebook`

1.2 Using Numpy

Numpy makes working with multidimensional arrays easy. Given that it is at the core of many software libraries, it might be useful to know the basics of how it works.

- If your unfamiliar with numpy, try to work your way through the list of exercises in this link.

Refer to the Numpy documentation to learn about any new commands.

1.3 Working with PyTorch

PyTorch is an imperative and dynamic framework for creating deep learning models. The following exercises provide a gentle introduction to working with PyTorch.

- The script in this link will introduce the basic concepts in PyTorch
- Autograd is one of the great features of PyTorch. It saves us the trouble of defining the backward pass through a network. This script link deals with autograd.

1.4 Creating a FeedForward Network

A simple feed forward network is one where the weighted input summed to the bias and has an 'activation function' applied to it. The output of each layer is the input to each successive layer. A final prediction is generated which is compared to the ground truth with a loss function. The Chain rule is used to propagate the errors backward and update the model weights.

- Using link try to re-implement a simple neural network first in numpy, then with PyTorch.

Chapter 2

Back propagation

Objectives When you do the exercises for this week, you

- code the back-propagation method from scratch in Numpy and PyTorch.
- understand and use Autograd module in PyTorch.
- learn how to use pytorch.nn module.

Back propagation Here you will understand back-propagation deeper and implement it with different ways.

- **Backprop with pen and paper**
  Please try to complete backward pass. (Fig. 2.1).

- **Numpy and Pytorch**
  Please refer to the file ”numpy_backprop.py” and ”pytorch_backprop.py” in BrightSpace (under LAB module in week 2).

- **Autograd in pytorch**
  Instead of doing gradient calculation for backpropagation by hand, pytorch can handle it easily with the module **Autograd**. We can use automatic differentiation to automate the computation of backward passes in neural networks. When using autograd, the forward pass of your network will define a computational graph; nodes in the graph will be Tensors, and edges will be functions that produce output Tensors from input Tensors. Backpropagating through this graph then allows you to easily compute gradients. What you need to do is just define which parameters (tensors) need to be optimized. Please check the file ”pytorch_autograd.py” in Brightspace (under LAB module in week 2).

$$\begin{tikzcd}
x \arrow[dr] & & t \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & y \arrow[r] & \mathcal{L} \arrow[r] & \mathcal{L}_{reg} \\
w \arrow[ur] \arrow[rrr] & & & \mathcal{R} \arrow[ur] &
\end{tikzcd}$$

Q: Do the backward pass:

$$\overline{\mathcal{L}_{reg}} =$$

$$\overline{\mathcal{R}} =$$

$$\overline{\mathcal{L}} =$$

$$\overline{y} =$$

$$\overline{z} =$$

$$\overline{w} =$$

$$\overline{b} =$$

Forward pass:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{reg} = \mathcal{L} + \lambda\mathcal{R}$$

Figure 2.1: Simple back-propagation example

- **Pytorch.nn**
  It is simple to design a neural network by using the "torch.nn" package. The nn package contains the necessary layers, e.g. torch.nn.ReLU(), and some popular loss functions such as torch.nn.CrossEntropyLoss(). For more information and example of "torch.nn", please go to the lab documents "pytorch_nn.py"in Brightspace (under LAB module in week 2). If you want to have broader knowledge about pytorch.nn please check the link.

Chapter 3

Convolutional Neural Networks

Objectives When you have done the exercises for this week, you will have learned

- applying convolution operation manually,
- how to implement CNNs with PyTorch.

3.1 The Convolution Operation

- Convolution operation with pen and paper.
  Please try to complete the convolution operationon the image size of 7×7 patch and filter size is 3×3 (Fig. 3.1). Take the stride size as 1 pixel.

- Why does the feature map size shrink?

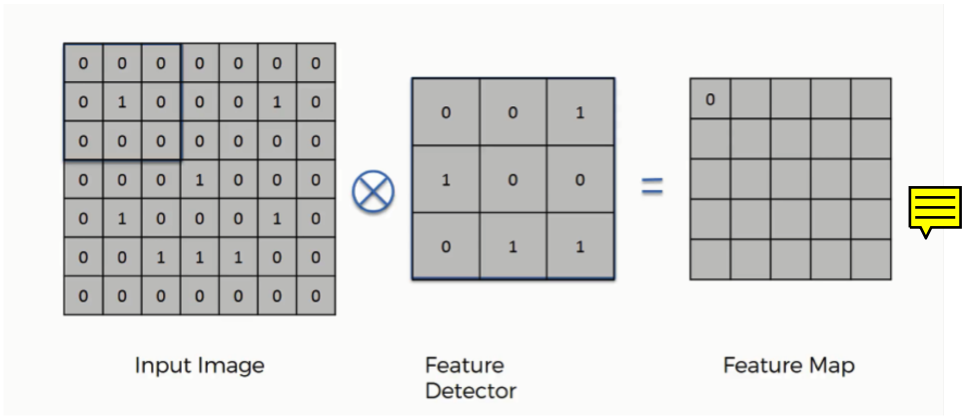


Figure 3.1: Applying convolution operation on 7×7 input image with 3 x 3 filter ("kernel"), and using one-pixel stride.

3.2 Reading (optional)

If you want to understand this chapter better, we definitely recommend you to read the note after lab.

3.3 Convolutional Neural Networks

3.3.1 Tutorial

In this section, please refer to the ipython documents in Brightspace. We introduced the whole process of training a CNN including the following modules:
- Get the data
- Dataloader
- Define the network
- Loss Function
- Optimizer
- Training
- Testing

We added explanation for each module in the ipython file. In the end of the file, we also proposed several questions you can think about and play with the code to get the answer.

3.3.2 More Experiments (optional)

To get more experience with classification problems you can repeat the experiments in the ipython file using other grayscale datasets such as FashionMNIST

3.4 Answer for convolution operation

Result is shown in Figure 3.2.

- An image matrix (volume) of dimension $(W_1 \times H_1 \times D_1)$
- Filter $(W_2 \times H_2 \times D_1 \times K)$, where K is filter number.
- Stride (S) and padding (P)
- Outputs a volume dimension $(W \times H \times D)$
  Here $W = (W_1 - W_2 + 2P)/S + 1$, $H = (H_1 - H_2 + 2P)/S + 1$ and $D = K$

For example Fig. 3.1, input image $(7 \times 7 \times 1)$; Filter $(3 \times 3 \times 1 \times 1)$; Filter number (K $= 1$); Stride $(S = 1)$; Padding $(P = 0)$, then $W = (7 - 3 + 0)/1 + 1 = 5$, $H = (7 - 3 + 0)/1 + 1 = 5$ and $D = 1$. For more complex example, please visit link.

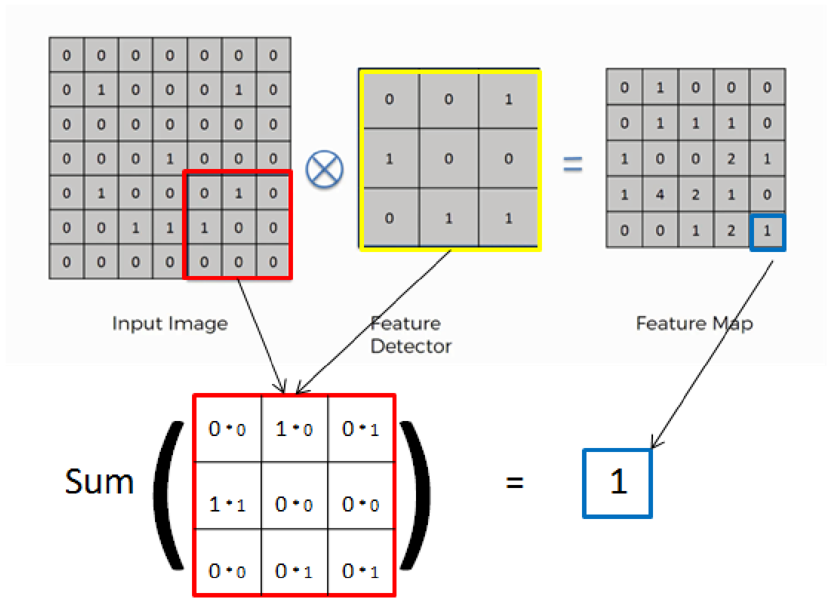


Figure 3.2: The result of convolution operation on 7×7 input image with 3 x 3 filters.

Chapter 4

Optimization

Objectives This weeks exercises will deal with

- visualising model parameters/performance,
- initialisation strategies,
- optimisation methods given by torch.optim,
- learn how to save and load your trained models.

While training models, there might be a need to visualize the weights and loss to analyze anomalies in the performance.

4.1 Visualisation (optional)

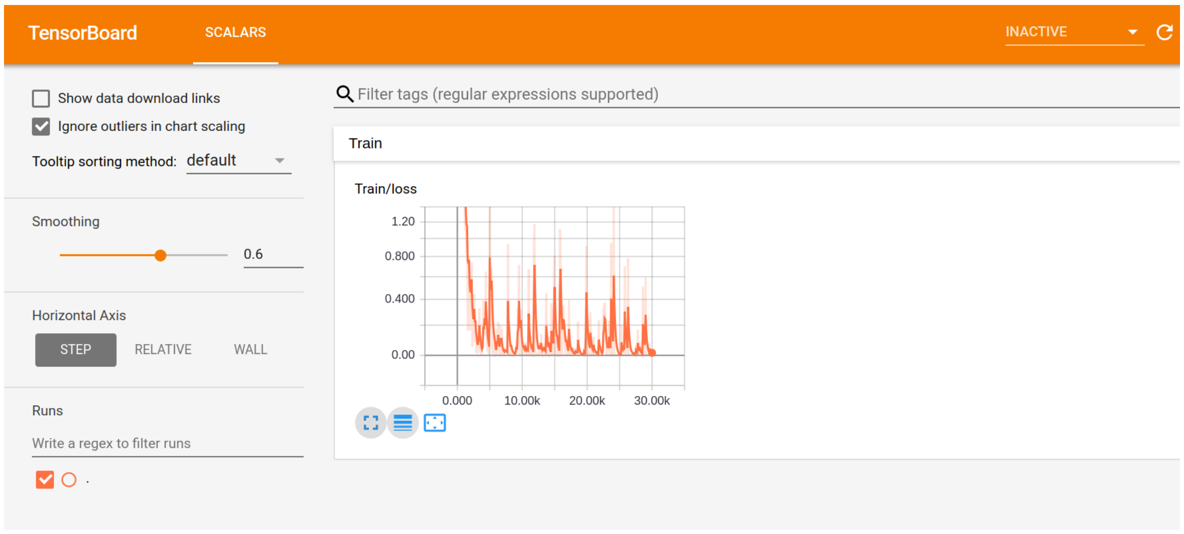
During training, it is important to visualise training progress of a neural network such as training curve, PR (Precision-Recall) curve, architecture, images etc. For this purpose, we will use TensorboardX tool. Please follow the instructions in the link for installation. However, tensorboardX is based on Tensorflow-tensorboard. That is why, you need to install tensorflow in order to visualise the network progress. There are already some other packages letting users to log the events without tensorflow; however they only provides basic functionalities.

Last week, you trained a network for MNIST dataset. This week, you will try to visualise the training curve during training. Please download the `MNIST_training_curve.ipynb` code from BrightSpace and check the training curve. When you want to see the training curve, run

```
tensorboard --logdir runs
```

in the terminal and click the link. The training curve will be seen in your browser (in Figure 4.1).

If you want to try more functionalities of tensorboardX, check the document. You can find many demos in this link.



☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis: STEP

Figure 4.1: Training curve visualisation of MNIST training for 2 epochs on tensorboard tool.

4.2 Initialisation

Training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Please download the initialisation example `init_example.py` code from BrightSpace and visit link to see more initialisation methods.

Check how the performance change for your designed classifier on MNIST dataset, by using different initialization.

4.3 Optimization Techniques

The idea behind this section, is to 'tinker' with various optimization methods and gain an appreciation for the role the hyperparameters play. Using the visualization methods mentioned above and the CNN example from last week, tweak the parameters as given below (or as you see fit) and analyze their effects on the gradient, loss and performance across few epochs. Try training the model for 10 epochs or so for each experiment otherwise it will take too long to get through the exercises.

The optim package of pytorch provides implementations of commonly used optimization algorithms. Optimization techniques such as mini-batch gradient descent are usually used to update weights in deep learning models. To enable faster convergence and improve flexibility, additions such as momentum terms and algorithms such as RMSProp, Adam were introduced. In this section, we will look at a few optimization algorithms.

- Momentum pushes SGD in the right direction, helping it negotiate 'ravines' (areas where the surface curves much more steeply in one dimension than in another).
  - Try running the script from last week with no momentum (set to 0). It should take longer to converge than when you add a momentum term (typically $\gamma = 0.5$ to 0.9).
- RMSprop proposes decaying the learning rate by dividing it with the square root of running average of the squared gradients. Try setting α to different values.
- Adam uses averages of past gradients and past squared gradients in the weight update. The beta terms in Adam prevent bias toward zero (since those are the initial values for the averages). β_1 is typically taken to be 0.9 and β_2 is usually 0.999. Try running different values of β.

4.4 CIFAR10 Experiment

You have learnt how to initialise the network parameters and use different optimisation methods and you have compared them. Here, you will choose your best way of training your network with CIFAR10 dataset. You have already gone deeper and deeper in the deep learning topic. That is why, you are free to choose your network architecture and hyperparameters (be careful about overfitting).

4.4.1 Optional experiment

If you want to go further, maybe you can try to outperform the networks in the paper-5. You can use also `confusion matrix` to see the performance of each classes and where the network is confused.

4.5 Model Saving and Loading

There are two main approaches for serializing and restoring a model. Please visit the link to check how to save and load your model.

Chapter 5

Regularization

Objectives When you do the exercises for this week, you become familiar with regularization methods such as,

- Weight decay
- Dropout
- Early stopping
- Batch Normalization (optional)

5.1 Weight decay

Weight regularization introduces a penalty on the weights of the model. The penalty term can be passed as a parameter to the Pytorch optimizer. Given below (Fig. 5.1) is a graph depicting performance of models with varying weight penalties and given a random input. Train models on CIFAR-10 with varying values of weight decay (say 0.0001,0.001 and 0.01) and plot the train and test accuracy with various values.

5.2 Dropout

Dropout has proven to be an effective technique for regularization and preventing the co-adaptation of neurons. Please visit link to implement dropout layer on your designed model, and train your model on MNIST to check how it affects your model when: 1) with and without dropout; 2) choose different dropout probabilities such as $p = \{0.1, 0.5, 0.9\}$. Plot the learning curve by yourself and compare with the curve given by the original paper as shown in Fig. 5.2.

5.3 Early Stopping

`Early stopping` is an effective form of regularization which stops training when the model starts to overfit on the training set. You need to sacrifice some of your data to create a

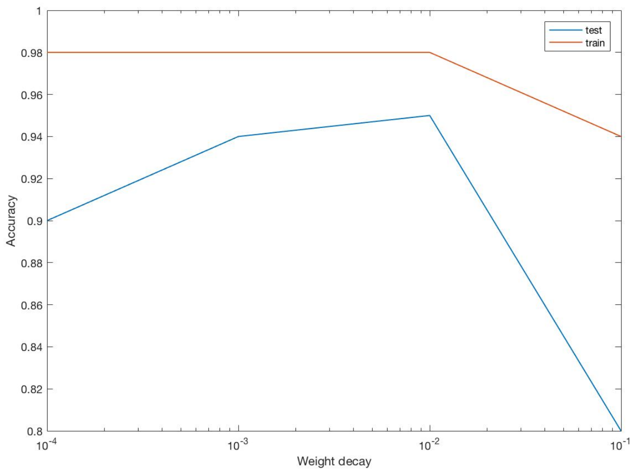


Figure 5.1: Effect of varying weight penalties.

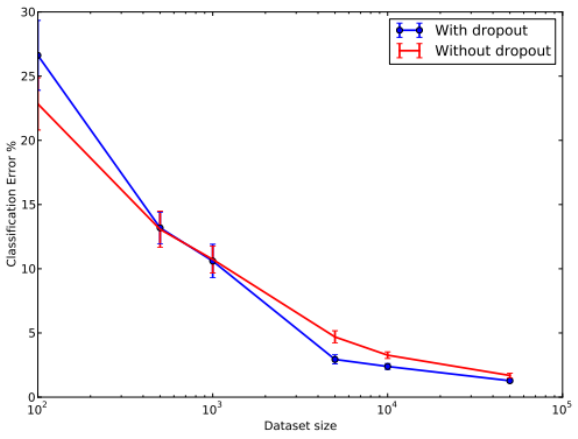
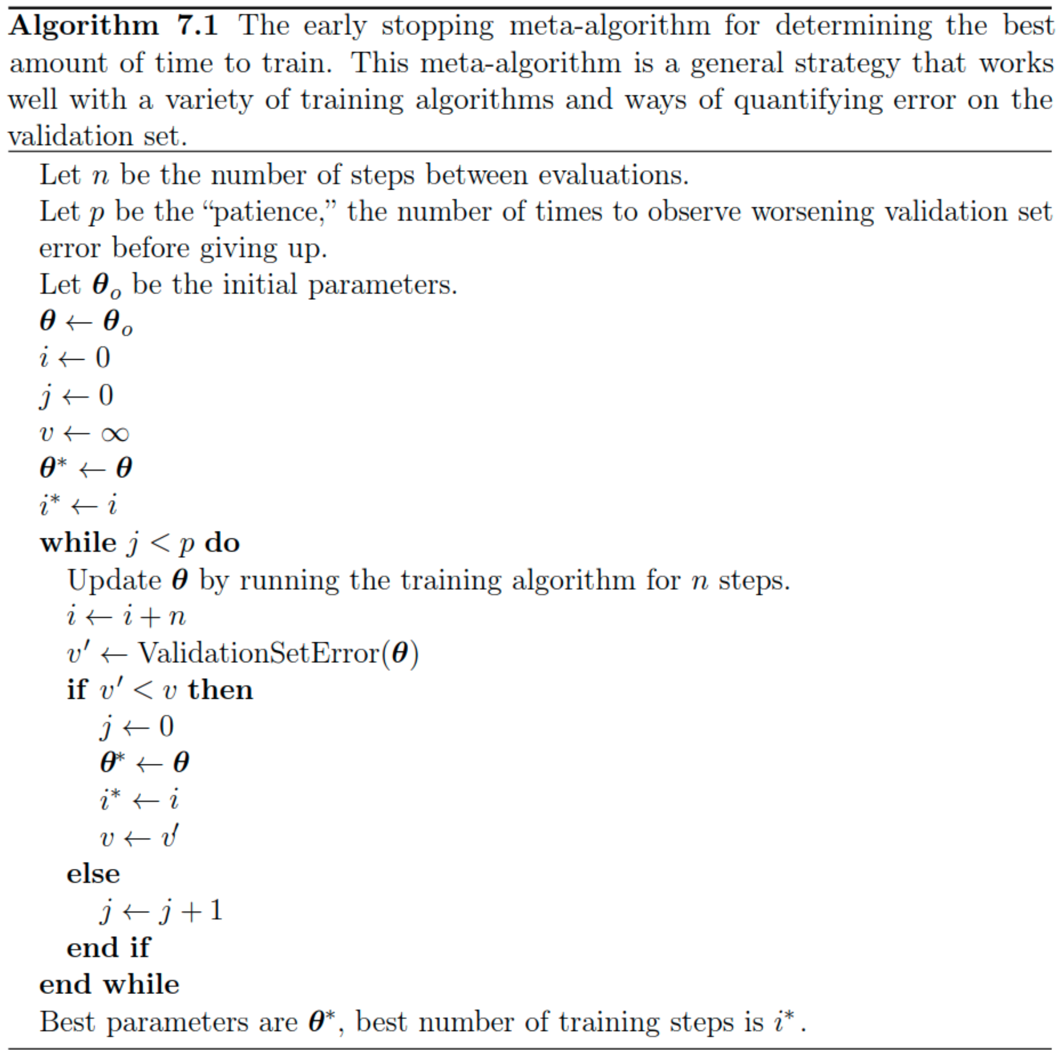


Figure 5.2: Effect of varying data set size.

validation set for early stopping. In this assignment, you will write your own early stopping method. You can use Algorithm 7.1 from your book for this purpose. It is important to save the best model in order to continue training or testing from that point instead of using the latest model. You can use the model save/load methods from last week.



Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

```
Let $n$ be the number of steps between evaluations.
Let $p$ be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ do
  Update $\boldsymbol{\theta}$ by running the training algorithm for $n$ steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ then
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is $i^*$.
```

Figure 5.3: Algorithm for early stopping.

5.4 Optional : Batch Normalization

Batch Normalization (paper)(deep learning book section 8.7.1) proposes layer-wise normalization of inputs. The original paper claims that it combats a phenomenon introduced in the same paper, called 'internal covariate shift'. But a recent paper argues that batch norm 'reparametrizes the underlying optimization problem to make it more stable'. Implement batch norm with the in-built pytorch function (link) and analyze how it's weights evolve. For additional reference, this blog has a nice explanation.

Chapter 6

Recurrent neural networks

Objectives This week, the lab deals with recurrent neural networks and variants such as

- LSTM
- GRU

Recurrent neural networks can process sequences of units more effectively than their feed-forward counterparts because of their ability to use information from previous iterations. However, RNNs struggle to learn long term dependencies, which is when LSTMs come in handy. This blog post has a great description of how RNNs and its variants work.

There are two tutorials for this week - classifying names based on country of origin and generating names based on the same. Try to experiment with different model depths and model composition (use LSTM blocks instead of RNN). Visualize how the gradients evolve over epochs for each case (optional) and analyze how each of the different models perform.

6.1 Classifying names with an RNN

Follow this Pytorch tutorial and try to experiment with the model by adding more layers or changing the model to inlcude `nn.LSTM` and `nn.GRU` (optional).

6.2 Generating names with an RNN (optional)

Follow this Pytorch tutorial for the task of generating names based on a given country. Once again try to experiment with other cells such as `nn.LSTM` and `nn.GRU` (optional).

Chapter 7

Unsupervised Learning

Objectives This week, the lab deals with unsupervised learning with

- Auto-encoder,
- Convolutional Auto-encoder,
- Denoising Auto-encoder.

7.1 Auto-encoder

We have already given you a simple auto-encoder example for the MNIST dataset. Please download the `autoencoder.ipynb` code from Bright Space. Fill the hyperparameters and try to train your first auto-encoder.

7.2 Convolutional Auto-encoder

Here it is expected from you to design a convolutional auto-encoder. Train your auto-encoder and compare your results with a simple one.

7.3 Denoising Auto-encoder (optional)

We have learnt that auto-encoders can be useful to suppress noise. Given a noise-free input x, you can add random noise ϵ to get a noisy sample $\tilde{x} = x + \epsilon$ and then minimize the reconstruction loss on $D(E(\tilde{x})) = x'$, where $E(\cdot)$ and $D(\cdot)$ are the encoder and decoder, respectively. Please design an experiment in order to achieve this goal.