Assignment 2

DS405B - Practical Deep Learning from Visual Data

Dr. Aseem Behl
School of Business and Economics
University of Tübingen
Wednesday, 8. June 2022

Introduction

The goal of this assignment is to practice designing, training and tuning of fully connected neural networks using PyTorch. For the purpose of this exercise, we will consider the image classification task. In the lecture and tutorial, we employed the widely used MNIST image classification dataset for classifying images to respective digit category. To make our task a bit more interesting, we will focus on a qualitatively similar, but comparatively complex Fashion-MNIST dataset, which was released in 2017.

As illustrated in the above figure, Fashion-MNIST consists of images from 10 categories, each represented by 55000 images in the training dataset, 5000 in the validation set and by 1000 in the test dataset.

In this assignment, you are provided with a baseline model in the companion notebook. Starting from a baseline fully connected multi-layer perceptron architecture with 1 hidden layer of 128 hidden features, you will design an improved deep net architecture to classify images into 10 categories. Next, you will use the accuracy on the Fashion MNIST validation dataset, to select the best hyper-parameters for your model. Finally, you will evaluate the performance of your network on the Fashion MNIST test dataset.

Designing an Improved Model

Our main goal is to improve the baseline model to maximise the performance network on the validation dataset. Towards this goal we will implement the following subtasks.

1. **Data Preprocessing** - So far, in the baseline method, we used the raw data, which are not necessarily normalized. It is usually a good idea to normalize the input because it allows to make training faster (because normalization equalizes the relative importance of the dimensions of the input). It turns out that FashionMNIST is already kind of normalized so that what we are going to do in this section will not have a big impact on your training. Anyhow, there are various ways to normalize the data. One normalization technique is called standardization. Here, you compute the mean of the training vectors and their variance and normalize every input vector (even the test set) with these data. Given a set of training images $X_i \in \mathbb{R}^{28\times28}, i \in [0,N-1]$, and a vector $X \in \mathbb{R}^{28\times28}$, you feed the network with $\hat{X} \in \mathbb{R}^{28\times28}$ given by

$$X_\mu = \frac{1}{N}\sum_{i=0}^{N-1} X_i$$

$$X_\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left(X_i - X_\mu\right)^T\left(X_i - X_\mu\right)} + 10^{-5}$$

$$\hat{X} = \left(X - X_\mu\right)/X_\sigma$$

   We introduce normalization in our Pytorch model by applying an additional torchvision.transforms.Normalize transform to our datasets. You must first compute the mean and standard deviation images on the training dataset images. Note that the mean and standard

deviation statistics for normalization are to be computed on the training set and the same statistics should be used on the validation and test sets. Can you think of what is the reason behind using the training statistics for test image normalisation?

2. **Data Augmentation** -

   One process which can lead to better generalisation performance is Dataset Augmentation. The basic idea is to apply transformations to your input images. For example, slightly rotating, translating an image of, say, a pullover is still an image of a pullover as shown below :

   Now, the idea is to produce a stream of augmented training samples generated from your original set of training samples. Transformation of images can be performed by composing and calling functions in transforms in PyTorch. Implement at least three dataset augmentation approaches: random horizontal flips, random rotation and random resize+cropping. Feel free to also try any additional transforms of your choice from the available transforms in PyTorch.

3. **Network Capacity** - Adapt the model architecture code to take number of hidden layers and size of each hidden layer as an input parameter and modify the neural network architecture accordingly. It might be helpful here to review the section on Neural Network capacity from the lecture notes by Andrej Karpathy.

4. **Activation Functions** - Adapt the model architecture code to accept the type of non-linear activation function (for example, ReLU, tanh or

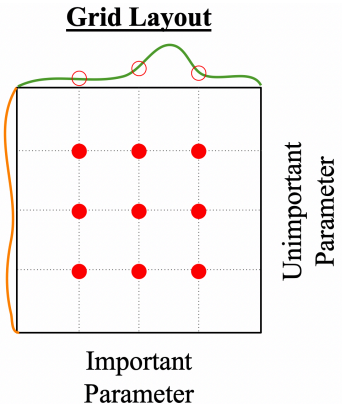
sigmoid) as input. It might be helpful to review the section on activation functions from the lecture notes by Andrej Karpathy.

5. **Model Regularisation** - Include Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014) in your model to prevent overfitting. As discussed in the lecture, dropout consists in setting to 0 the activations of a certain fraction of the units in a layer. In PyTorch, we simply need to introduce nn.Dropout layers specifying the rate at which to drop (i.e. zero) units.

Searching for the best Hyper-parameters

After implementing each of the above sub-tasks, one also needs to select the appropriate values of the corresponding hyper-parameter. For example, for model regularisation using dropout, we need to also select the optimal value of the dropout probability. We need to do the same for other hyper-parameters like learning rate, size and number of layers, activation function etc. Here, the optimal hyper-parameters are the values of the hyper-parameters that maximise the performance on the validation dataset.

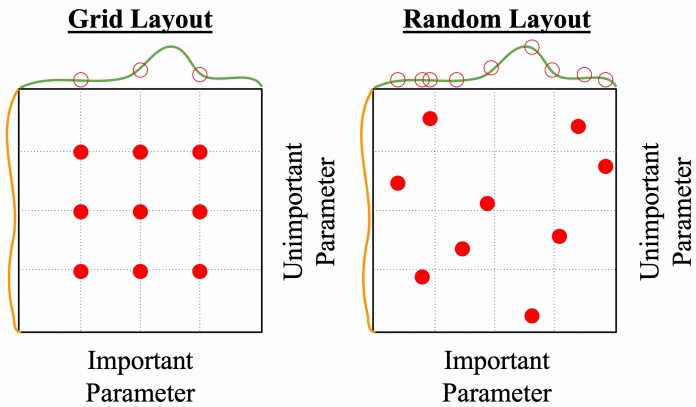
Grid Search



To select the optimal hyper-parameters, we commonly see people do is this notion of a grid search. So here what we're going to do is select some some set of hyper parameters that we care to tune and then for each of those hyper parameters. we'll select some set of values that we want to evaluate for that hyper parameter. For example we might want to evaluate different learning rates that are spaced out in a log linear way

(`1x10-4, 1x10-3, 1x10-2, 1x10-1`) and also test out for different values of dropout strengths (`0, 0.2, 0.4, 0.6`). Next, given four values of the dropout probabilities and four values of the learning rate gives rise to 16 combinations and we just try them all and see which one performs best on the validation set. This is a fairly reasonable strategy that people sometimes do in practice but the problem is that this strategy is exponential in the number of hyperparameters that you want to tune so this very quickly gets infeasible very quickly.

Random Search



Another strategy that sometimes people employ instead is random search rather than grid search and the procedure is pretty much the same. We're going to select some set of hyper parameters that we want to tune and now rather than selecting values that we want to try for each of those hyper parameters instead we're going to select some ranges of those hyper parameters along which we want to search. For example, Learning Rate in the range [1x10-4, 1x10-1] and Dropout strength in the range [0, 0.6]. It turns out that random search is a more efficient for hyper-parameter optimization than search on a grid.

In practice, however, searching through high dimensional hyperparameter spaces to find the most performant model can get unwieldy very fast. The **Weights & Biases API** provides Hyperparameter sweeps, an API to automatically searching through combinations of hyperparameter values (e.g. learning rate, batch size, number of hidden layers) to find the most optimal values and pick the most accurate model in an organized and efficient way. The API is very straightforward to use, however, it would be helpful to review this tutorial from Weights & Biases to learn how to use their API.
Following are different values to try for the hyperparameters:

Hyper-parameters:

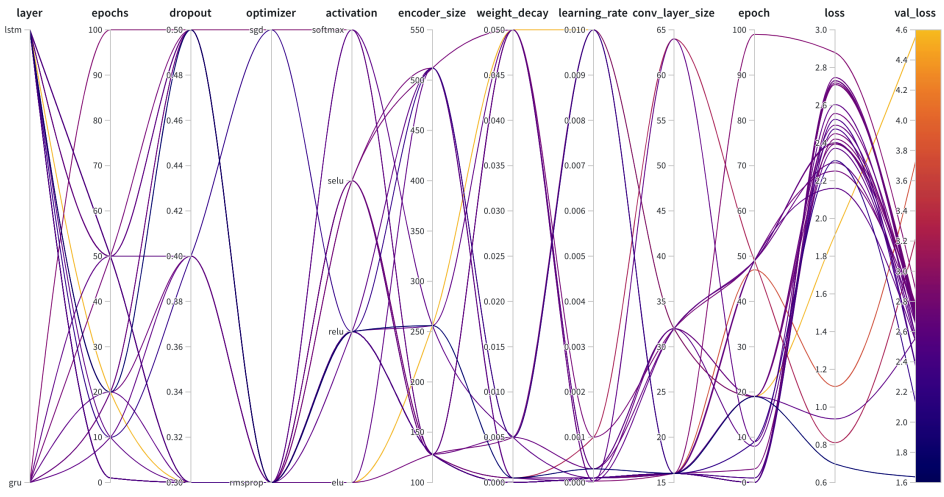
`dataNormalisation:` True or False (For data prepossessing)
`horizontalFlip:` True or False (For data augmentation)
`resizedCrop:` True or False (For data augmentation)
`rotation:` True or False (For data augmentation)
`numberHiddenLayers`: 1, 2, 4 (For model capacity)
`hiddenLayerSize`: 64, 256, 1024 (For model capacity)
`activationFunction`: sigmoid, tanh, ReLU (For activation capacity)
`dropoutProbability`: 0 to 0.8 (For regularisation)
`numberEpochs`: 5, 10 (For training)
`learningRate`: 1e-1 to 1e-4 (For training)
`batchSize`: 8, 64, 128 (For training)

As you can see that this leads to an exponential number of combinations. You will have to think about strategies to do this hyperparameter search efficiently. For example, in practice, it can be helpful to first search in coarse ranges, and then depending on where the best results are turning up, narrow the range. Also, it can be helpful to perform the initial coarse search while

only training for 1 epoch or even less, because many hyperparameter settings can lead the model to not learn at all, or immediately explode with infinite cost. You can exclude such bad settings from the next stage. The second stage could then perform a narrower search with 5 epochs, and the last stage could perform a detailed search in the final range for more epochs. Check out the options provided by `wandb.sweep` and write down what strategy you chose and why. For more details and tips for hyper-parameter tuning, it might be helpful to review lecture 6 slides from our course and also the section on Hyperparameter optimization from lecture notes by Andrej Karpathy.

There are other techniques to improve your model besides the ones discussed in the lecture. Feel free to be creative and try other ideas to improve your model. Keep in mind the computational resource constraints on Google Colab and setup your experiments judiciously. Starting with the assignment early on is also a good strategy in order to maximise GPU time.

Please also provide a short analysis of the results of your tuning experiments by creating a report using Weights and Biases API.
Based on the different experiments that you have run, can you make some inferences about which configurations of hyper-parameters worked and



which did not. Here again, ***wandb*** can be helpful. It automatically generates an interactive "Parallel co-ordinates plot" and an interactive "Hyperparameter Importance Plot" as shown below. Include the interactive plots in the accompanying report and write down your observations.

Final Evaluation of the Model

Finally, evaluate and **report the accuracy of your model on the test set** after fixing the various hyper-parameters to the optimal values. Note that you should use the test set only for the final evaluation of your model and not for tuning the hyper-parameters. Furthermore, randomly ample and display 10 images from the test set and predict the category of each image using your model.

This assignment is worth **20 points** and is due on **Friday, June 24, at 08:00 AM CET**. The best, second best and the third best performing method on the test set will receive 2, 1 and 0.5 additional bonus points respectively. Bonus points can be used to make up for lost points in previous or future assignments. In order to receive full credit for the assignment, please be prepared to present your solutions to selected exercises from the assignment or other similar exercises in the lecture on **Wednesday, June 29, 2021**.

Submission Steps

1. Download the starter notebook assignment_2_2022.ipynb and the dataset from ILIAS.

2. Please upload the notebook and dataset to DS405B folder on your Google Drive.

3. Then, open the notebook file with Google Colab by right-clicking the *.ipynb file.

4. You can save your work in Google Drive (In Colab click “File” -> “Save”) and resume later.

5. **Run all cells, and do not clear out the outputs, before submitting.**

6. Download the saved notebooks and the wandb report to your computer and upload it as submission to ILIAS.