Deep Learning Workshop, 27.03 – 30.03.2023

Assignment 01: Fully connected Feedforward Networks (FNN)

The main goal of this first programming session is to:

a) Implement a (fully connected) Feedforward Network (FNN) to recognize hand-written numbers of the MNIST dataset[1] which you have already explored in Assignment 00 at home.

b) Develop and embed the training process into a more generic structure to benefit in later assignments.

Although this might seem at first to be a quite challenging task, *don't worry*! It will be broken down into manageable subtasks. If we use one of the four blocks of the aforementioned code structure, we add the abbreviation

[N] Network (in this case the `FNN`)

[DL] Data Loader (in this case the `DataLoaderFNN`)

[NT] `NetworkTrainer`

[NU] `NetworkUser`

in front of a subtask to emphasize which parts of the generic coding structure we are implementing/using. *Note:* Whenever we train a network, the estimated training time is given in brackets.[2]

1) Data exploration: Visualizing what the machine will see.
See Take Home assignment. Your results should look similar to the two images in Figure 1.

2) Implementing a Deep Neural Network: Getting fully connected.
In this task we will work with the file `fnn.py`. Open the file and take a look at the `FNN` class. You should see two empty templates for the constructor of the class and the `forward` method.

a) We start by implementing a generic method to construct a single dense layer, consisting of a linear layer in PyTorch (defined by its input and output shape) and an (optional) activation function. We might want to not use an activation function in the last layer, so your method should be able to take care of that.
Hint: The command `torch.nn.Sequential`[3] might be helpful.

b) [N] Next, we implement the constructor of our `fnn.py` class and build a network with *two hidden layers*. The following hints might be helpful:

- Think back to the take home assignment: what size have the images?

[1] The dataset is a famous benchmark dataset for all types of machine learning models. The provided code will automatically download the dataset for you. You can find more information under http://yann.lecun.com/exdb/mnist/.
[2] The networks were trained on a Mac M1 Pro with 16 GB of RAM.
[3] See https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html.

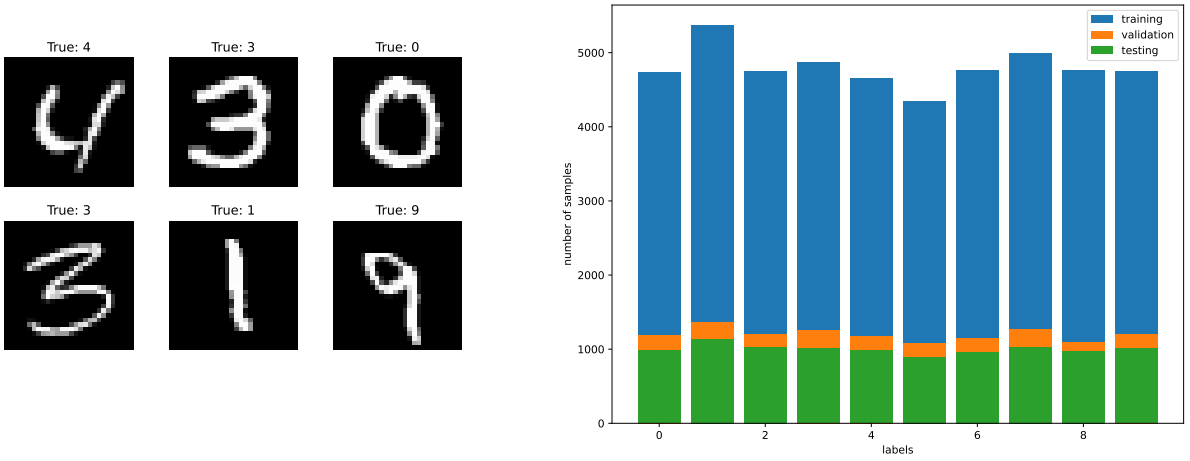

Figure 1: Results of Assignment 00: Visualization of six data points (left) and the data distribution (right).

- Assume we perform a reshape on the images to transform them into vectors (see below), what shape will such a vector have?
- For the second hidden layer just use the `_dense_layer` function again with the inputs `_dense_layer(hidden_shape, hidden_shape, activation_func)`. One of the Bonus tasks will clarify the question of how we can also make the number of layers generic.

c) [N] Now, we implement the `forward` method of our `FNN` class which takes a `torch.tensor` as an input, executes a forward propagation step and returns the predicted label. Why do we not need to implement a backpropagation step? [Pas+17, cf.]

d) [DL] Go to the beginning of the `DataLoaderFNN` class and uncomment the following command

```
data_transform = transforms.Compose(
    [
        transforms.ToTensor(),
        # reshape your images to vectors
        transforms.Lambda(
            lambda x: x.reshape(28 * 28)
        ),
    ]
)
```

This will transform our data from images to tensors in a vector format as you have already seen in the take home assignment.

3) Working with a generic structure: Can we send our networks to the gym?
In this task we will work on the file `./utils/network_trainer.py` and complete the already prefilled template. The `NetworkTrainer` class provides a generic training process that we will use for the rest of the course and acts as a gym for our networks. Thus, the model inside the network trainer does not have be an instance of our specific `FNN` class of the previous exercise, but every model is inherited by the class `torch.nn.Module` and consequently has the same underlying structure.

a) [NT] For each epoch, define two variables called `train_loss` and `validation_loss`, initialize both with the value zero.

b) [NT] For each training data batch, set the gradients of the optimizer to zero in order to reduce the memory footprint and ensure that we do not track any unnecessary information. Perform a forward pass of the model on the `input_data`, compute the loss by using the provided `criterion`, perform a backward propagation step on the loss and add an optimization step.

c) [NT] Add the loss of this batch onto the variable `train_loss` and compute the average loss for this training epoch after the batch iteration is finished.
Hint: The variable `len_train_data` might be helpful.

d) [NT] Perform similar steps on the validation dataset. Think closely about the difference between the training and the evaluation process. How can we make use of that to save time and memory?

4) **Training and improving initial network: How strong can we get?**
Finally, we switch back to the `01_main.py` file and train our network. The `01_main.py` file provides an additional code snippet (see `./utils/network_user.py`) which basically decouples the training process of your network from the evaluation of the test dataset by saving the network's parameters and loading them again. Don't worry about this class: we implemented this to make your life easier.

a) [NT] We are dealing with a classification task: What is the canonical activation function for the last layer and what might be a good choice as a loss function? Add both to your code and pass them into the `trainer`.
Hint: Based on the theoretical introduction, it should be clear that a **Softmax** activation function compared with the **Negative Log-Likelihood** (NLL) is the right choice. Can you explain why the CrossEntropyLoss fulfills our needs?

b) [NT] Define the parameters of your network as a dictionary and call the `trainer.train_network()` function. Start with a small step size, e.g. $\alpha = 0.001$, and use **Adam**[4] as your Optimizer. Choose a reasonable hidden layer size. Why is ReLU a good choice for activation? *Don't run your code yet.*

c) [NU] ($\sim 40\,\text{s}$, 12 epochs) Uncomment the code snipped where the `NetworkUser` class is used and run your code.

- How does your network perform on the training and validation data?
- How well does the network generalize to unseen data?

d) Use the `NetworkUser` class to manually iterate over some instances of the test data set. Use your visualization function of the Assignment 00 to compare the networks predicted labels with the real labels, what do you observe?
Hint: As we reshaped our input data inside the `DataLoader` class, you might need to reshape the tensors again in order to make your plotting method work.
Hint: If you already trained the network, you can uncomment the training part and just use the `NetworkUser` class.

e) How do your plots differ from the ones of your neighbors?

What have we done so far and what options do we have to proceed further? In Task one to four we followed the basic steps of any Machine Learning project. We started by analyzing our dataset and the distribution of training, validation and testing data samples. In task two we have chosen a model type and a specific architecture of that model type to solve the problem. Task three generalized the training and validation steps each model has to run through in a general machine learning pipeline. As we already mentioned, this structure will simplify the training in the upcoming assignments and can serve you as a starting point for future Deep Learning projects. In task four we trained our network and evaluated it's performance and ability to generalize on new (so far unseen) data points.

[4] See https://pytorch.org/docs/stable/generated/torch.optim.Adam.html and [KB14]

You now have several options to dive deeper into details of each one of the aforementioned steps by working on the bonus examples. We provide tasks to further investigate how we can measure the accuracy of our network, see how we can shift the data distribution to accelerate the training or you can work on making our original network even more generic. We recommend to start with Bonus, a) and afterwards you have the free choice.

5) BONUS: Networks on steroids.

Disclaimer: The bonus tasks are designed to give you additional insights supplementary to the material above. We will provide proposed solutions for some of the exercises. Tasks without a solution are marked with a (∗) and serve as a starting point, if you want to dive deeper into the topic.

We suggest to skim all the exercises first, spend some time to think about the discussion questions and afterwards pick a task that you want to implement.

a) [NT] One of the network trainer's parameters is a function called `sum_correct_preds` which is called after every batch iteration in the training and validation cycle. Internally, the function is always called with the predicted labels and the target labels and is used to manually compute the accuracy of a model. Switch to the file `01_main.py`: Define a custom `sum_correct_preds` function to compute the accuracy of the network in percentage points.

b) A common way to improve, stabilize and accelerate the training process is to normalize your dataset. In this subtask we will explore two different methods to perform data normalization.

***Full data normalization** [DL] By using `torchvision.transforms.Normalize`[5] in our `DataLoaderFNN` class as an additional transformer, we can normalize our dataset as a whole. Compute the channel-wise mean and standard deviation of the MNIST-dataset and normalize your data accordingly.

Recommended: (Mini) Batch Normalization [N] (∼ 50 s, 12 epochs) In their paper "Batch Normalization: Accelerating Deep Network Training by Reducing internal covariate shift" [IS15] the authors suggest to use so called **Mini Batch Normalization** layers after a linear layer and *in front* of the activation function. Extend your generic dense layer from one of the previous exercises and use `BatchNorm1d`[6] to implement the paper's idea. The basis intuition about batch normalization is that we overcome the internal covariate shift by combining the standard normalization step (zero mean and unit variance) with an additional (learnable) scale and shift. By applying this concept *layer by layer*, we put the layer's inputs at similar scale and reduce the risk of the unstable gradient problem (that is why the authors of the original paper suggested to use batch normalization in front of the activation function). Thus, our training process becomes more stable, we eliminate the need for other regularization methods and in many cases we obtain a faster convergence as we can use higher learning rates. Think closely about the formula

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\mathbb{V}\mathrm{ar}[x] + \varepsilon}} \cdot \gamma + \beta,$$

with γ and β learnable (see link below). Add Batch Normalization layers into your generic `_dense_layer` function inside the `fnn.py` File.

- Can you omit some parts of a linear layer in front of a batch normalization layer to reduce the number of parameters?
- How does batch normalization affect your training process?

[5] See https://pytorch.org/vision/main/generated/torchvision.transforms.Normalize.html

[6] See https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html#torch.nn.BatchNorm1d.

- *In practice, batch normalization layers are sometimes also used *after* the action function, how does this affect your training process?

c) [N] Can you implement a way to generically change the initialization process of the network's weights and biases for a general network.

d) [NT] A common technique to prevent overfitting in DNNs is to add an L1 or L2 penalty term to the loss function. **L1** regularization, also known as **Lasso regularization**, adds the absolute value of the network's parameters to the loss function, while **L2** regularization, also known as **Ridge regularization**, adds the square of the network's parameters to the loss function.

Both, L1 and L2 regularization, work by shrinking the weights (and biases) of the model towards zero, which helps to reduce the complexity of the model and prevent overfitting by reducing the capacity of the model to fit the training data, i.e. "the loss has a smoother topography and there is relatively less gain from fitting individual samples" [SAV20, p. 220].

Check the documentation of Adam and add an L2 regularization penalty to the loss function, i.e. modify the `01_main.py` file.

L2 regularization can be also implemented via:

```
l2_lambda = 0.001
l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
loss = loss + l2_lambda * l2_norm
```

Modify the code snippet above to obtain L1 regularization, implement a function called `regularization` in the `01_main.py` file and pass it into the `network_trainer.py` via the `regularization` parameter. Why could it be more attractive to use L1 regularization instead of L2 regularization? How does your regularization strategy affect your training results?

*e) [NT] How does your network performance change if you modify the

- number of neurons of your hidden layer
- number of hidden layers
- activation function for the hidden layer(s)

individually or all at once?

*f) [N] Our original network consists of two hidden layers. How do you need to change the constructor of our `FNN` class to make the number of layers generic? Think about a way to efficiently change the sizes of the layers.
Hint: An additional class called `Layer` as well as a `LayerFactory` method could be helpful.

*g) What happens if you change your last layer to a plain linear layer again (without an activation function) and use the Mean-squared-error (MSE)[7] as your loss function (i.e. we treat our problem as a regression problem)? What behavior do you expect? Is your model still able to learn?

*h) Create a hand-written number by yourself (e.g. using paint or gimp). Can your model recognize your handwriting?

*i) [DL] Can you think of ways to artificially enlarge the data set (in general this is called data augmentation)? Why could this be a good idea (in general)?

[7] See: https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html

References

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[Pas+17] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[SAV20] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep learning with PyTorch*. Manning Publications, 2020.