

ADL - Assignment 1

Jarl-Sebastian Sørensen, Magnus Lindberg Christensen, Ronja Stern

May 6, 2024

1 Deep versus wide networks

In the following exercise we will compare wide versus deep networks.

1.1 Description of notebook

The notebook consists of six cells the contents of which we explain in the following.

In the first cell, all the needed packages are imported.

In the second cell, the device to run the notebook is selected and printed in addition the torch version is printed.

In the third cell, a class to create an XOR dataset is provided. The class is initiated with the wanted number of data points and the standard deviation of the noise. The data points and corresponding labels are then generated using the `generate_continuous_xor(self)` function. A few additional helper functions are provided.

In the fourth cell, the function `visualize_binary_samples()` is used to visualize the XOR dataset.

In the fifth cell, the Neural Network is implemented and a training and evaluation loop are provided. In the class `SimpleClassifier(nn.Module)` the Neural Network is defined. According to the exercise sheet the Neural Network can be initialized with a varying depth and width of hidden layers. In the training loop (`train_model`) the data gets passed through the model after which the loss is calculated. Then the loss is backpropagated and the parameters are updated accordingly. This happens for a specified number of epochs. In the evaluation loop (`eval_model`) the data also gets passed through the model to get the predictions. Instead of doing backpropagation and updating the weights the accuracy of the model is computed which is given by the number of true predictions divided by total predictions.

In the sixth cell, the model gets trained and evaluated with depths between 0 and widths between 1 and 3.

Width	Depth	Mean	Standard Deviation	Number Parameters
1	0	0.7639999985694885	0.0	5
1	1	0.6793333888053894	0.14519259333610535	7
1	2	0.7099999785423279	0.0	9
1	3	0.5633333325386047	0.12633639574050903	11
2	0	0.625333309173584	0.08198104053735733	9
2	1	1.0	0.0	15
2	2	0.543999969959259	0.09899495542049408	21
2	3	0.9273333549499512	0.10276618599891663	27
3	0	0.9979999661445618	5.960464477539063e-08	13
3	1	1.0	0.0	25
3	2	1.0	0.0	37
3	3	0.9979999661445618	5.960464477539063e-08	49

Table 1: Mean, Standard Deviation, and Number of Parameters for different Widths and Depths of the hidden layers.

1.2 Combinations of Depth and Width

In the following [Table 1](#), we report the number of parameters and the mean and standard deviation of the accuracies on the evaluation data over varying widths and depths of the hidden layers. I chose to calculate the mean and standard deviation on the accuracy instead of the loss function because the accuracy better reflects the model’s actual performance instead of just the loss.

1.3 Interpretation and Parameters

Increasing the Depth seems to increase the overall accuracy of the classification. An increased depth allows the network to capture more complex patterns in the data, potentially better capturing the nuances of the XOR function. Increasing the Width seems to increase the overall accuracy as well as increase with allows to capture more diverse features. However, increasing both the width and depth too much can also lead to overfitting leading to a decline in accuracy. This is what may be happening in the case of a Width of one and a depth of three.

The number of parameters for each of the networks are given in [Table 1](#). The training parameters used are:

- The number of epochs is the number of times the training data is passed to the model. More epochs can increase the accuracy but too many can also lead to overfitting.
- The batch size is the number of samples used to compute the gradient and update the model’s weights in each iteration (batch) of training. Larger batch sizes can lead to faster training but may require more memory and computational resources. Smaller batch sizes can provide more noise in the updates, potentially helping the model escape local minima, but they might also slow down training.

- The learning rate adjusts the step size taken during gradient descent. A larger learning rate can increase the speed of the convergence. However, a too-large learning rate may lead to the model diverging.
- The optimizer used in this case is Stochastic Gradient Descent it computes the gradient on random batches of the samples. Changing the optimizer can potentially increase or decrease performance this depends on the model and data.

2 Convolutional Neural Network

In the following exercise we will compare the structure and efficiency of a feed-forward network and convolutional neural network when classifying hand written digits.

2.1 Feed Forward Network description

The *feedForwardMNIST.ipynb* file consists of 9 different parts. The first part checks whether the user is running in Google Colab, and behaves accordingly.

Import: This part imports the utilities PyTorch packages from the library.

Set global device: Sets the device to the designated GPU if it exists, otherwise the device is set to use the CPU. These lines has been altered to also check *torch.backends.mps.is_available()*, which is the designated GPU for apple silicon MacBooks.

Functions:

- *training_loop* trains the model. Each epoch sets the loss to 0 and does the following: First it defines the input and label data to the device, whereafter it creates a new tensor with *inputs.view(inputs.shape[0], -1)* containing all samples which has then been flattened. The gradients are then set to 0 with *optimizer.zero_grad()* to avoid gradient accumulation through backpropagation. Then, the *loss* is calculated by comparing the predicted output and ground truth label. *loss.tensor.backward()* are then used to calculate the gradients, and thereafter update paramaters using *optimizer.step()*. Lastly, it prints the loss at certain times for each epoch.
- *evaluate_loop* evaluates the model on all of the batches. It first defines two variables *correct* and *total* to 0. It then runs the evaluation with *torch.no_grad()* to increase performance, as the pytorch *autograd* engine isn't used at this step. Hereafter input and label data are set as in the *training_loop*. Then, it makes the prediction and returns the maximum value of the the output data. The total is then defined as the size of the labels (all samples), and the correct labels is where the prediction equals the data labels. Further, it does the same thing but on a class specific level. Afterwards, it creates two lists (*class_correct* and *class_total*) containing 10 elements, one for each digit/class. Once again, it finds the max and defines the class where the prediction is equal to the label. This is squeezed to remove one-dimensional shapes, whereafter it updates the class lists, correct and total, and returns the evaluations.

Main program: This is the main program. First it defines transformations that the data should follow when loaded. Here, they are being converted to tensors and normalized. It then loads the dataset containing the data and its

labels, with the transformations applied. Hereafter, it wraps an iterable around the data by using the Dataloader, batching the data to sizes of 64. Further, the data for training are being shuffled in the dataloader as well.

Next, the network is specified containing 6 layers, alternating between a linear layer and the ReLU activation function. The last and sixth layer is the output activation function, defined as a *Logarithmic soft max* function.

Lastly, the scripts combine everything by defining the epochs, learning rate, loss function, and optimizer, whereafter the script train the model.

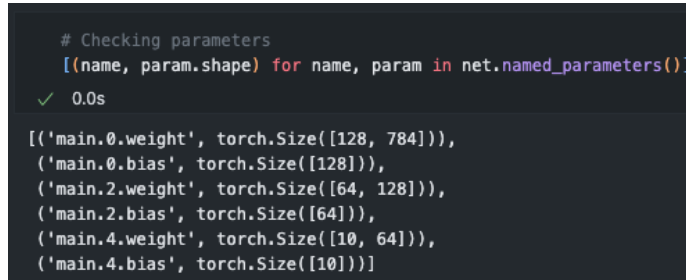
All accuracies for each digit can be seen in column *Feed Forward* in [Table 2](#). Generally the classifications are between 89% and 97% accuracies, and the model performs the best with lower and higher digits.

Parameters: The model uses 109386 parameters. This can be inferred from calculation with the parameters seen in [Figure 1](#). The total parameters can be found by following [Equation 1](#), which focus on non-activation layers.

$$total_parames = \sum_{i=0}^{total_layers} in_connections[i] \cdot out_connections[i] + bias[i] \quad (1)$$

The calculation is done by done by summing up the product of the number of input neurons and output neurons, with their respective biases, leading to the following result in [Equation 2](#).

$$params = (784 \cdot 128) + 128 + (128 \cdot 64) + 64 + (64 \cdot 10) + 10 = 109386 \quad (2)$$



```
# Checking parameters
[(name, param.shape) for name, param in net.named_parameters()]
✓ 0.0s
[('main.0.weight', torch.Size([128, 784])),
 ('main.0.bias', torch.Size([128])),
 ('main.2.weight', torch.Size([64, 128])),
 ('main.2.bias', torch.Size([64])),
 ('main.4.weight', torch.Size([10, 64])),
 ('main.4.bias', torch.Size([10]))]
```

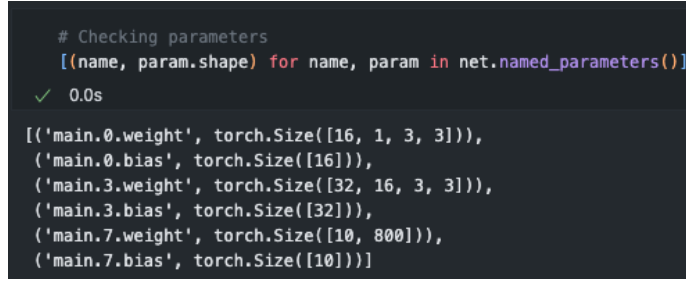
Figure 1: Feed Forward Network model parameters

2.2 Convolutional Neural Network description

When defining the CNN network, the main change was the network layers. This generally consists of convolution and pooling layers. However, when this is done, a flattening layer occurs before classification. The reasoning behind this is that the last pooling layer is multi-dimensional, and it is not possible to use this

as is to perform classifications. Therefore, we need a one-dimensional vector, which we can produce by flattening the pooling output. In this way, we create a one-dimensional vector from the features extracted by the convolution and pooling layers, which are then used to perform the classification.

The classification accuracy of the CNN network can be seen in column *CNN* in Table 2. Here, it is clear that it outperforms the feed-forward network at every digit. In particular with the middle area digits where the feed-forward network struggled, the CNN network outperforms. The general increase in accuracy is between 1.9732% and 8.7302% as column *classification increase* shows in the Table 2.



```
# Checking parameters
[(name, param.shape) for name, param in net.named_parameters()]
✓ 0.0s
[('main.0.weight', torch.Size([16, 1, 3, 3])),
 ('main.0.bias', torch.Size([16])),
 ('main.3.weight', torch.Size([32, 16, 3, 3])),
 ('main.3.bias', torch.Size([32])),
 ('main.7.weight', torch.Size([10, 800])),
 ('main.7.bias', torch.Size([10]))]
```

Figure 2: Convolutional Neural Network model parameters

Parameters: The CNN model uses 12810 parameters. Ultimately, the intuition of this calculation is the same as with the Feed Forward network. Yet, due to the CNNs independence from the input size, we only care about the size of the *input channel*, *kernel size* and *output channel*, until the flattening has occurred. After flattening, the approach is the same as with the Feed Forward network. The result is inferred by the parameters in Figure 2. The parameters before flattening is calculated as seen in Equation 3.

$$params_before = (kernel_width \cdot kernel_height \cdot in_channel + bias) \cdot out_channel \quad (3)$$

Combining this we get the result for the parameters before flattening as in Equation 4

$$params_before = (3 \cdot 3 \cdot 1 + 1) \cdot 16 + (3 \cdot 3 \cdot 16 + 1) \cdot 32 = 4800 \quad (4)$$

Hereafter, we can utilise our previous approach for the Linear layer after flattening which leads to amount of parameters seen in Equation 5.

$$params_after = (800 \cdot 10) + 10 = 8010 \quad (5)$$

Putting it together, we get the total parameters seen in Equation 6

$$total_params = params_before + params_after = 4800 + 8010 = 12810 \quad (6)$$

Hereby the number of parameters in the CNN network is 88.289177774121% less compared to the feed forward network ($109386/12810 = 0.88289177774121 = 88.289177774121\%$).

800 input features:

```
Net(
  (main): Sequential(
    (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Flatten(start_dim=1, end_dim=-1)
    (7): Linear(in_features=800, out_features=10, bias=True)
    (8): LogSoftmax(dim=1)
  )
)
```

Figure 3: The properties of each layer in the Convolutional Neural Network

In Figure 3 we see the properties for each of the CNN layers. To understand the 800 input features, we can go through each layer of the network and calculate the output sizes. The output sizes of both the convolution and the pooling layers are calculated as seen for height in Equation 7 and width in Equation 8.

$$H_{out} = \lfloor \frac{input_h - kernel_h + padding_h + stride_h}{stride_h} \rfloor \quad (7)$$

$$W_{out} = \lfloor \frac{input_w - kernel_w + padding_w + stride_w}{stride_w} \rfloor \quad (8)$$

Based on the above, we can calculate the output of the various layers to understand why the input feature of the linear layer is 800, with the original input dimensions of 28X28 (the greyscale MNIST images). We will not take the ReLU() layers into account, as these are activation functions and do not affect the dimensions.

(0) Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1)):

$$output_size = \lfloor \frac{28 - 3 + 0 + 1}{1} \rfloor \times \lfloor \frac{28 - 3 + 0 + 1}{1} \rfloor = 26 \times 26$$

(2) MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False):

$$output_size = \lfloor \frac{26 - 2 + 0 + 2}{2} \rfloor \times \lfloor \frac{26 - 2 + 0 + 2}{2} \rfloor = 13 \times 13$$

(3) Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1)):

$$output_size = \lfloor \frac{13 - 3 + 0 + 1}{1} \rfloor \times \lfloor \frac{13 - 3 + 0 + 1}{1} \rfloor = 11 \times 11$$

(5) `MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)`:

$$output_size = \lfloor \frac{11 - 2 + 0 + 2}{2} \rfloor \times \lfloor \frac{11 - 2 + 0 + 2}{2} \rfloor = 5 \times 5$$

Now we are at the flattening layer preceding the Linear layer. Here, we flatten each of the examples in the minibatch, which in our case is 64. From the convolution block, we get an output in the form (batch_size, number of channel, height, width). Hereby for each example in the batch_size we should flatten the data. As shown earlier, the height and width of the last max pooling is 5×5 , and the output channel from the convolution in layer (3) is 32. Hereby, the flattened input with starting dimension 1 becomes $in_features = 5 \cdot 5 \cdot 32 = 800$.

Accuracy

Table 2: Performances between the feed forward network, and the CNN network.

Digit	Feed Forward	CNN	Difference	Classification increase (%)
0	0.973684	0.993421	0.019737	1,9732
1	0.962162	0.983784	0.021622	2,1622
2	0.930233	0.982558	0.052325	5,2325
3	0.903846	0.980769	0.076923	7,6923
4	0.932203	0.977401	0.045198	4,5198
5	0.896825	0.984127	0.087302	8,7302
6	0.929688	0.968750	0.039062	3,9062
7	0.939024	0.963415	0.024391	2,4391
8	0.951049	0.972028	0.020979	2,0979
9	0.958084	0.982036	0.023952	2,3952

3 U-Nets and MLOps

3.1 Describe the structure of the notebook

After installing missing packages and importing any necessary libraries, we set the global device variable to specify whether we want the notebook to run on CPU or GPU. We then define functions to download, process, evaluate and plot the dataset and models. These functions act as helper methods, covering tasks such as visualisation, model evaluation on data, saving and loading model and optimiser states, and plotting training and validation histories. When using Google Colab, we also mount Google Drive. We then load the dataset containing chest x-rays with lung segmentations and create DataLoader objects for the training, validation and test sets. We also plot a single example. The main training logic is defined within two functions, which we will explain in more detail as they contain the main logic for training a new model.

run-one-epoch: This function runs the training for one epoch, given the model and other parameters. It processes the samples in batches, calculates the training loss for each batch, performs backpropagation, and performs an optimisation step where parameters are updated. After processing all the batches, it evaluates the model on the evaluation data, logs the scores, and returns the average training loss along with evaluation metrics for the validation set.

training-loop: First we specify the metrics to evaluate the model, which in this case is the Macro F1 score. We then iterate over different epochs, starting at 0 or continuing training if the init-epoch parameter has been set. Within the loop, we call the run-one-epoch function, print the returned training and validation metrics, and append them to the train/validate history. At the end of the loop we save the model if a path has been provided. Once all epochs have been iterated, we return the training and validation history.

To start training, we initialise the model and optimiser and start training. Optionally, we can resume training from a previous epoch. The training process is initiated by calling the training loop function. Finally, we can plot the training loss, validation loss, and validation F1 score, and view the resulting segmentation of a sample image. In addition, we can evaluate the entire test dataset by printing the mean F1, the standard deviation of the F1 and the minimum F1.

3.2 Modification to log the training progress

To track the progress of the model in a simple way, we initialise a wandb project before running the training, as shown in [Figure 4](#). Now we can log any metric we want to wandb during training. We do this for each epoch in the training loop using the wandb.log function shown in [Figure 5](#).

This process produces the plots shown in [Figure 6](#), where we have an overview

```

wandb.init(project='adl-ass1-simple', entity='rstern')

# Run training
train_history, val_history = training_loop(
    model=model,
    loss=loss,
    optimizer=optimizer,
    train_loader=train_loader,
    val_loader=val_loader,
    init_epoch=init_epoch,
    n_epochs=10,
    metrics_dict=metrics,
    save_path="model.ckpt"
)

wandb.finish()

```

Figure 4: WandB Initialisation

```

if init_epoch == None:
    init_epoch = 0
try:
    for i in range(init_epoch, n_epochs):
        print(f"Epoch {i+1}/{n_epochs}")
        train_metrics, val_metrics = run_one_epoch(
            model=model,
            loss=loss,
            optimizer=optimizer,
            train_loader=train_loader,
            val_loader=val_loader,
            n_epochs=n_epochs,
            metrics_dict=metrics_with_loss,
        )
        print("  Mean epoch metrics:")
        print(f"    Training:  {pformat(train_metrics)}")
        print(f"    Validation: {pformat(val_metrics)}")
        train_history.append(train_metrics), val_history.append(val_metrics)

        # log to wandb
        wandb.log({'epoch': i+1, 'loss': val_metrics['loss'], 'loss-training': train_metrics, 'f1': val_metrics['f1']},

        if save_path:
            save_path_epoch = f"epoch_{i+1}_{save_path}"
            print(f"  Saving to: {save_path_epoch}")
            save_model(model, save_path_epoch, optimizer)
            old_path_epoch = f"epoch_{i}_{save_path}"
            if os.path.exists(old_path_epoch):
                os.remove(old_path_epoch)

```

Figure 5: Log to WandB



Figure 6: Metrics overview wandb

of the progress of training loss, validation loss and validation F1 score (the metrics we logged during training). We can see in this plot that while training loss and validation loss have decreased, the macro F1 score for the evaluation data has remained at a more or less constant low level, indicating that the model isn't performing as well as we'd like.

3.3 WandB Sweep

The concept of a sweep involves training the model several times with different hyper-parameters. By comparing the results of these models, we can identify the hyper-parameters that give the best results. First, we specify the different hyper-parameters and enumerate the possible values they could take shown in [Figure 7](#). Next, a configuration from the set of possible parameters is randomly selected and we continue to train our model as described above using our training loop. When initiating the training loop, it's important to adjust the parameter settings to match the selected sweep hyper-parameter to ensure the adaptability of our training methodology. It's worth noting that the initialisation process for the "wand" project differs slightly from the basic "wandb" project above. First, we use the "wandb.sweep" method to generate a sweep-id based on the sweep configurations ([Figure 7](#)). This id is then used when calling the "wandb.agent" method, where we start the sweep training and specify the number of different sweep runs to be attempted (in this case 5, see [Figure 8](#)). The initialisation of the model and optimiser, as well as the adjustment of the hyperparameters, takes place within a newly defined method called "train-with-sweep", which is invoked by the "wandb.agent" method. To log the metrics we use the same method "wandb.log" as shown in [Figure 5](#). The chosen hyperparameters are those parameters which most likely have an effect on our scores. We decided to exclude epochs to keep the computation reasonable.

3.4 Hyper-parameter Sweep

See [Figure 9](#) for an overview of the 5 different sweep configurations and the resulting loss. The best hyperparameters were found to be: batch size: 4, epochs: 10, learning rate: 0.01003, optimiser: adam. Choosing these, we obtained a loss of 0.01837. We must be careful not to interpret these hyperparameters as the general best. They indicate that this configuration seems to have advantages over the other chosen configurations. But in the end we only randomly tested 5

```

sweep_config = {
    'method': 'random',
    'metric': {'goal': 'minimize', 'name': 'loss'},
    'parameters': {
        'batch_size': {'values': [4,8,16,32]},
        'epochs': {'values': [10]},
        'learning_rate': {'distribution': 'uniform',
                          'max': 0.1,
                          'min': 0},
        'optimizer': {'values': ['adam', 'sgd']}
    }
}

sweep_id = wandb.sweep(sweep_config, project="adl-ass1-sweep")

```

Figure 7: Sweep Configuration

```

# run sweep
wandb.agent(sweep_id, function=train_with_sweep, count=5)

```

Figure 8: Run Sweep

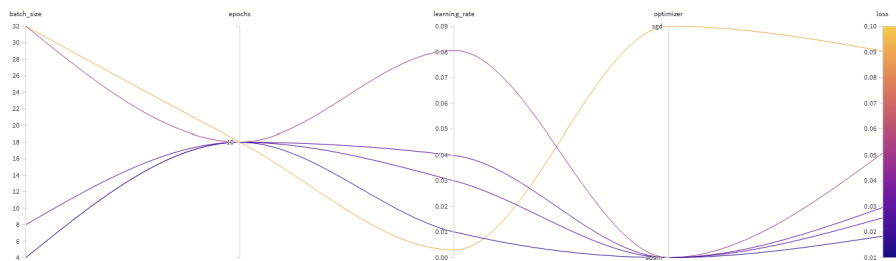


Figure 9: Hyper-parameter Sweep

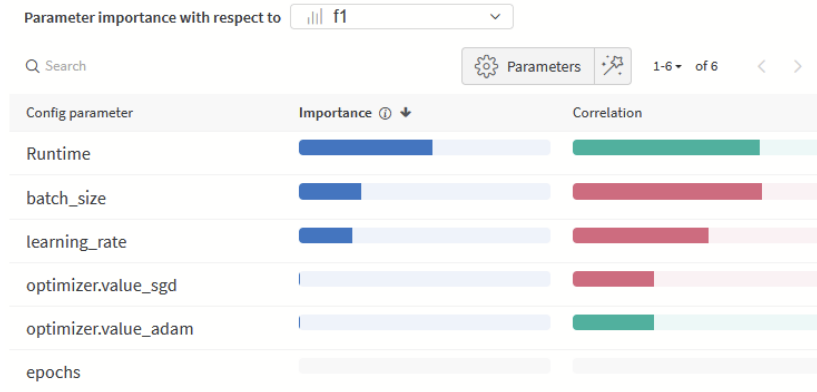


Figure 10: Correlation of parameters wrt. f1

configurations from our pre-defined hyperparameter space. There may be other configurations that give even better training results. The idea of a sweep is not to find the ultimate best, but to make a reasonable choice.

Additionally we can see the correlation of the hyper-parameters with the Macro F1 metric in [Figure 10](#). For example can see that the choice of optimizer is more heavy correlated to the score than the batch-size or the learning-rate.

References