Marked assignment 1: Understanding the performance of neural networks [20 points]

Making a neural network 'work' typically requires lots of trial and error with different architectures and training hyperparameters. However, there do exist certain (theoretical or numerical) results on how particular hyperparameter configurations affect performance. In this assignment we will explore some of these dependencies numerically. Specifically, your task will be to implement a fully-connected neural network and analyse how the performance of the model depends on the choice of hyperparameters (network width and depth, batch size, learning rate).

Hand-in format You will hand in a link to a Google Colab environment and the .ipynb itself via the Blackboard (see Assessments and Mark Schemes folder and then upload it into 'Coursework 1 Drop Box Spring 23'). Make sure to save your notebook such that you show your results (i.e. I'll need to be able to check the values you report in your results).

Getting ready Load the following libraries:

```
import numpy as np
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import torch.optim as optim
```

We will be working with the MNIST and CIFAR10 datasets. An example of how to download the CIFAR10 data is as follows:

```
train_set = torchvision.datasets.CIFAR10(
  root="./",
  download=True,
  train=True,
  transform=transforms.Compose([transforms.ToTensor()]),
)

test_set = torchvision.datasets.CIFAR10(
  root="./",
  download=True,
  train=False,
```

```
  transform=transforms.Compose([transforms.ToTensor()]),
)
```

You can pre-process the data in whichever way you wish: you can normalise it, or scale the pixel values (by dividing by 255). Just clarify what you did (if anything). For MNIST just replace CIFAR10 by MNIST.
Remember for MNIST the classes are the digits from 0-9. For CIFAR10 these are the classes of the above dataset:

```
classes = ('plane', 'car', 'bird', 'cat',
       'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Use your student ID (CID number) to set the seed:

```
# set seed for reproducibility
SEED = 'CID number'
np.random.seed(SEED)
torch.manual_seed(SEED)
```

Part I: Implementations [8 points]

In the first part we will be implementing the neural network and train and testing functions.

Task 1: Implement a fully-connected neural network [3 points] Implement a class (`class Net(nn.Module)`) with two functions:

- The function that initialises the layers that we will use. It has four parameters: 'dim': the dimension of the input, 'nclass': the number of output classes for our prediction problem, 'width': the width in each layer (we will consider all layers have the same width), 'depth': the depth of the neural network:

  ```
  def __init__(self, dim, nclass, width, depth):
          super().__init__()
          ... FILL IN THE REST ...
  ```

  You should define four types of layers: i) a layer that flattens the input, ii) a linear layer that takes the input and connects it to a hidden layer, iii) a linear layer that takes a hidden layer and connects it to the next hidden layer, iv) a ReLU activation layer. Use the `self.` to define the layers.

- The forward function that passes the input through the hidden layers and returns the output. The function should look as follows:

`def forward(self, input):`

This function should work for different numbers of hidden layers. You can do this by writing a for loop.

Task 2: Implement the data loading function [1 point] Implement a function that loads the data given a certain batchsize of the following form:

```
def loading_data(batch_size, train_set, test_set):
    ... FILL IN...
return trainloader, testloader
```

Use the `torch.utils.data.DataLoader` function.

Task 3: Implement a function that does one training epoch [1 point] This function should implement a single training epoch. Remember one epoch is a pass over all the training data. It should thus pass over all the items in the 'trainloader', compute the model output, compute the loss associated to the criterion, and take an update step for the optimization algorithm (the 'optimizer'). The function should have the following form:

`def train_epoch(trainloader, net, optimizer, criterion):`

and should return the loss (associated to the criterion we will pass it) over the train data (`loss = criterion(outputs, labels)`).

Task 4: Implement a function that does one test epoch [1 point] This function should take as input the test data and compute the loss (associated to the criterion) and the error (how many samples the model predicted incorrectly) over all the test datasets using a specific model configuration (the 'net'). The function should have the following form:

`def test_epoch(testloader, net, criterion):`

and should return two things: i) the *average* test loss over all the datapoints, ii) the error over all the datapoints where the error is computed as how many samples are predicted incorrectly (for this you will thus need to take the predicted class from the model and compare it to the true label; if they are the same, the model is correct; if different, the model is incorrect).

Task 5: Write a piece of code that sets the hyperparameters and that allows to run the train and test epochs [2 points] You will need to define the following hyperparameters:

- batch size
- dim (the dimension of the flattened input; it is 3072 for CIFAR10 and $28 * 28$ for MNIST)
- nclass (the number of classes; it is 10 for MNIST and CIFAR10)
- width
- depth
- the learning rate
- the number of training epochs

Then you will need to define i) the criterion (we will use the CrossEntropyLoss), ii) the optimizer (we can use Adam `optim.Adam` or SGD `optim.SGD`, but if the question doesn't specify which one to use you can try out others if you want). Then you will need to run all the functions: i) load the datasets using the function you created, ii) create a for loop that runs over the number of epochs and computes for each epoch the train and test outputs and prints *in each epoch* the results as follows:

```
print(f'Epoch: {epoch:03} | Train Loss: {train_loss:.04} |
    Test Loss: {test_loss:.04} | Test Error: {test_err:.04}')
```

Comment: I use 'error' to refer to 1-'accuracy'. Accuracy is thus the proportion of samples predicted correctly and error is the incorrect proportion.

Part II: Numerical exploration [10 points]

In this second part we will be working with the neural network. Our goal will be to understand how the performance (in terms of train and test error) depends on the choice of hyperparameters.

In each task, we will fix certain hyperparameters, but other hyperparameters are up to you to decide! *Report all the used hyperparameters in the text.* Include the tables as plain text in the notebook. Make sure that in the notebook you show that the train and test losses you present in the tables are indeed obtained from running your model (i.e. print out all your train and test losses). If you do not print

out the train and test losses or if we cannot see whether the train and test losses you reported in the table indeed were obtained from your code, you won't get any points. You will get partial points as long as your implementations are correct and correspond to the numbers you present in the tables. Full points can be obtained if you also managed to achieve some desired result (which is some pattern/result that should arise based on research results in recent years). In general: for all the numbers in the tables, keep all the parameters except the parameter in the table fixed for all your results.

Comment: How many epochs to use? In principle it's up to you to decide what you consider enough epochs (i.e. when the performance is good enough). Just report the used number.

Task 6: Analyse the performance for wide vs. deep neural networks [2 points] Fix the number of nodes per layer to 256. Choose the other hyperparameters yourself (and keep them fixed for all the results you present). Fill in the following table (include it in the notebook):

Depth	Train loss	Test loss
1		
5		
10		

Include in the notebook a discussion of your result. Some potential things you may observe: deep neural networks could lead to more overfitting on the data (smaller train error but worse test performance); also they may be harder to train and for example require a smaller learning rate (if you train with SGD).

Task 7: Compare SGD with small and large learning rates [2 points] Use in this setting the SGD optimiser (not Adam) with or without momentum. Fix the depth to 1 and width to 256. Fill in the following table (include it in the notebook):

Learning rate	Train loss	Test loss
0.001		
0.01		
0.1		

The learning rate controls how fast our model learns. Using a too small learning rate may lead to very slow learning; but using a too large learning rate may result in overshooting the minimum (i.e. not converging). What do you observe? Include a discussion of your observations in the notebook.

Task 8: Compare the performance for small and large batch sizes [3 points*] Fix the depth to 1 and fix the width to 64. Use the SGD optimizer (not Adam) with a learning rate of your choice and with or without momentum. Fill in the following table and present a plot of the train and test losses as a function of batch size (include both in the notebook) :

Batch size	Train loss	Test loss
8		
16		
32		
64		
128		
256		
512		

The intuition in deep learning is that noise has a regularising effect on the learned model. When we use a smaller batch size we increase the amount of noise that is used in the training process (because in each iteration we're using a different dataset over which we compute the loss). This should then result in a smaller generalisation error. See for example this paper [Keskar et al., 2016]. Do you observe any similar effect? In terms of computational efficiency, what are the trade-offs between small and large batches? What happens to the test loss as we increase the number of batches? What other observations do you have on the batch size?

* +1 point for being able to obtain some results and having a clear discussion of the results and +2 points if you manage to find a configuration that indeed matches the intuition in deep learning that a smaller batch size results in a smaller generalisation error.

Task 9: Analyse the train and test errors as a function of width [3 points*] Consider a neural network with *one* hidden layers. Choose the other hyperparameters and optimisation algorithm yourself. Fill in the below table and present a plot of the train and test losses as a function of width in your notebook.

Width	Train loss	Test loss
4		
8		
16		
32		
64		
128		
256		
512		
1024		

Discuss the results. What kind of width do you need to fit the data? Recent work [Neyshabur et al., 2018] has shown that the test error continues to decrease (or at least not increase) even if we make the number of parameters very large. If we were to use the standard generalisation bounds, we would conclude that at some point we overfit on the data and the test error would start to increase. However, with stochastic gradient descent there seems to be some implicit bias which does not allow this to happen.

* +1 point for being able to obtain some results and having a clear discussion of the results and +2 points if you manage to find a configuration that indeed shows that more parameters can result in a lower test loss.

Comments: The results from Figure 1 is something we'd expect to see.

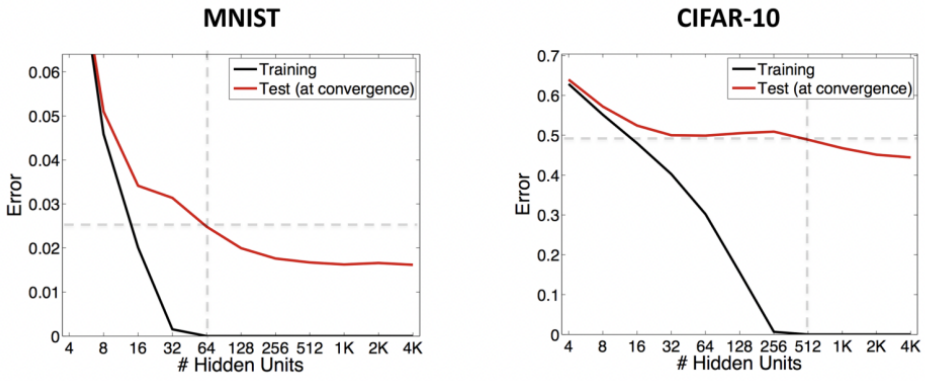

Figure: Training (empirical) and test (true) error for one-hidden-layer networks of increasing width, trained with SGD [20].

Figure 1: The results we'd expect to see; from Volkan Cehver's lectures `https://www.epfl.ch/labs/lions/wp-content/uploads/2022/01/lecture_08_2021.pdf`

Part III: Extension to CIFAR10 [2 points]

Task 10: Test the performance on CIFAR10 [2 points] Load the CIFAR10 dataset and train a neural network (you are free to choose the architecture and training algorithm) to the CIFAR10 dataset. Report in your notebook the train and test losses/accuracy and write a conclusion about which task you think is easier to learn for a fully-connected neural network - MNIST or CIFAR10.

Bonus task [1 point] The best *test* accuracy I get with the CIFAR10 dataset using *only* fully-connected layers is 0.55. Can you outperform this (using only fully-connected layers, but you can modify the depth, width, activation function and the training hyperparameters in whichever way you wish)? If you manage to outperform this, you will get one extra point.

Remark 0.1. *One can say we are now overfitting on the particular test data we choose, and probably we are. For this task we don't care about this. In general: a large topic of discussion is now around the fact that all of ML has overfitted on CIFAR10, MNIST and so on and now effort is being spent to collect new datasets.*

References

[Keskar et al., 2016] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

[Neyshabur et al., 2018] Neyshabur, B., Li, Z., Bhojanapalli, S., LeCun, Y., and Srebro, N. (2018). Towards understanding the role of over-parametrization in generalization of neural networks. *arXiv preprint arXiv:1805.12076*.