Queen Mary
University of London

Specific Learning Difficulties Cover Note

Student ID: 170189255

Advice for assessors and examiners

Guidelines for markers assessing coursework and examinations of students diagnosed with Specific Learning Difficulties (SpLDs) –
As far as the learning outcomes for the module allow, examiners are asked to mark exam scripts sympathetically, ignoring the types of errors that students with SpLDs make and to focus on content and the student's understanding of the subject. Specific learning difficulties such as Attention Deficit Disorders, dyslexia and or dyspraxia may affect student performance in the following ways:

- The candidate's spelling, grammar and punctuation may be less accurate than expected
- The candidate's organisation of ideas may be confused, affecting the overall structure of written work
- The candidate's proof reading may be weak with some errors undetected, particularly homophones and homonyms which can avoid spell checkers

Under examination conditions, these difficulties are likely to be exacerbated. Errors are likely to become more marked towards the end of scripts.

Useful approaches can include:

- Reading the passage quickly for content
- Including positive/constructive comments amongst the feedback so that students can work with specialist study skills tutors on developing new coping strategies
- Using clear English and when correcting; explain what is wrong and give examples
- Using non-red coloured pens for comments/corrections

Colleagues in schools are asked to ensure that students with specific learning difficulties access the support provided by the Disability and Dyslexia Service.

For more information regarding marking guidelines see DDS webpage http://www.dds.qmul.ac.uk/staffinfo/index.html and the Institutional Marking Practices for Dyslexic Students

Disability and Dyslexia Service
Student Services
Room 2.06 Francis Bancroft Building
www.dds.qmul.ac.uk
Tel: 020 7882 2756
Email: dds@qmul.ac.uk

Alteration or misuse of this document will result in disciplinary action

ECS629U: Artificial Intelligence Coursework

Task 1

I created my own data loader that extends the “torch.utils.data.Dataset” class. My Dataset formats the three dimensional dataset into a two dimensional dataset to make it easier for the trainer and model to handle. However, using the built in default PyTorch implementation would have worked just as well with some modification in the trainer and model.
The dataset is initialised with two tensors as its input (the features and labels of the functions respectively) and it’s “__getitem__” function returns a tuple containing the corresponding pairs of the features and labels. All the initialised datasets are stored in the default PyTorch data loader with a batch size defined in the variable “batch size” and with the “shuffle” flag (parameter) set to False. Changing the batch size affects the results of training and also the efficiency of training so its value can be modified to balance the two or prioritise one over the other.

I split the 2000 functions that were provided training into a portion that would be used for validating the model (200 of the 2000 functions) and a portion that will be used for training the model (1800 of the 2000 functions). Validating a model with a separate dataset, (I.E. data that is not the same as, or a subset of, the training data) enables you to identify instances of overfitting, as the validation data is likely completely different from the training data, so we can observe how the model behaves with data that it has not been trained with.

Task 2

I created two separate models for the encoder (“Encoder”) and the decoder (“Decoder”) components of the MLP. The “Model” model combines these models and trains them as one by initialising them in the same model and using them in the model’s forward functions. The output of the encoder is formatted by this model, such that the encoder’s output is appended to each feature of the decoder’s input (it appends the Rc to every Xt value, producing a 40 by rDim + 1 tensor). This allows the respective models to work in the same model, and thus allows them to be trained in the same model. All three models extend the “torch.nn.Module” class which allows them to function as PyTorch models.

The encoder is configured to take 2 features as its input as it handles Xc and Yc pairs (a subset of the input features for the function and their corresponding ground truth labels) for each function it’s fed as input. It is configured to feed the input into a linear layer which outputs a tensor of size hDim - which is a size we can define and modify. The last layer of the model outputs something of size rDim, which is also defined by the users. The mean of the final layer’s output is returned by the model’s forward function.

The decoder takes an input that is rDim + 1 long - which consists of Xt values that we want to predict, concatenated with the output of the encoder (which is rDim long). The encoder’s output is prepended onto each of the Xt values. The decoder also has a hDim size that users specify as the output of the first layer and input and output of subsequent layers. The model (it’s forward function) outputs a single value for each (Rc, Xt) value it is fed - which is a prediction (Yt).

The encoder and decoder use the Tanh activation function in between layers to add non-linarialities. The Tanh activation function seems to work better than popular alternative activation functions like ReLU, PReLU and sigmoid.

Task 3

I selected PyTorch's Adam optimizer to be used for both training and validating my model. I also used PyTorch's "nn.MSELoss" loss function to calculate the loss, which can be used for training, validation and for recording and graphing the improvement of the model overtime. The inbuilt optimiser and loss function are widely used and considered to be reliable for MLPs.

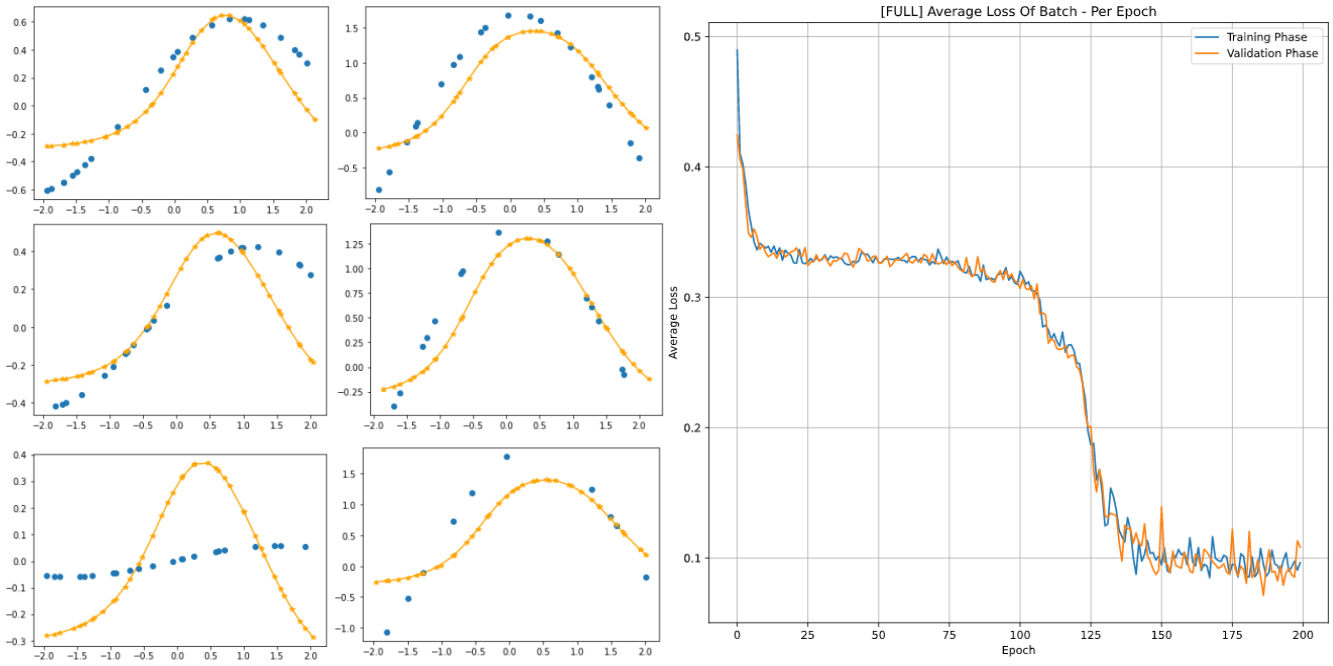
I set the optimizer's learning rate to 0.001 as learning rate values between 0.001 and 0.0015 seemed to yield the best results in terms of training a model that is ultimately accurate and and computationally inexpensive to train. Adding weight decay worsened the accuracy of the model, so it was left unset. Decreasing the batch size seems to improve the result (but at the cost of performance, hence I set it to 20) as did increasing the hDim to a number above 200 and the rDim to a number between 1 and 10.

Results of training the model with certain hyperparameters

Test 1 - Hyperparameters:

- *Batch size*: 100
- *Number of epochs*: 200
- *Learning rate*: 0.01
- *rDim*: 2
- *hDim*: 4

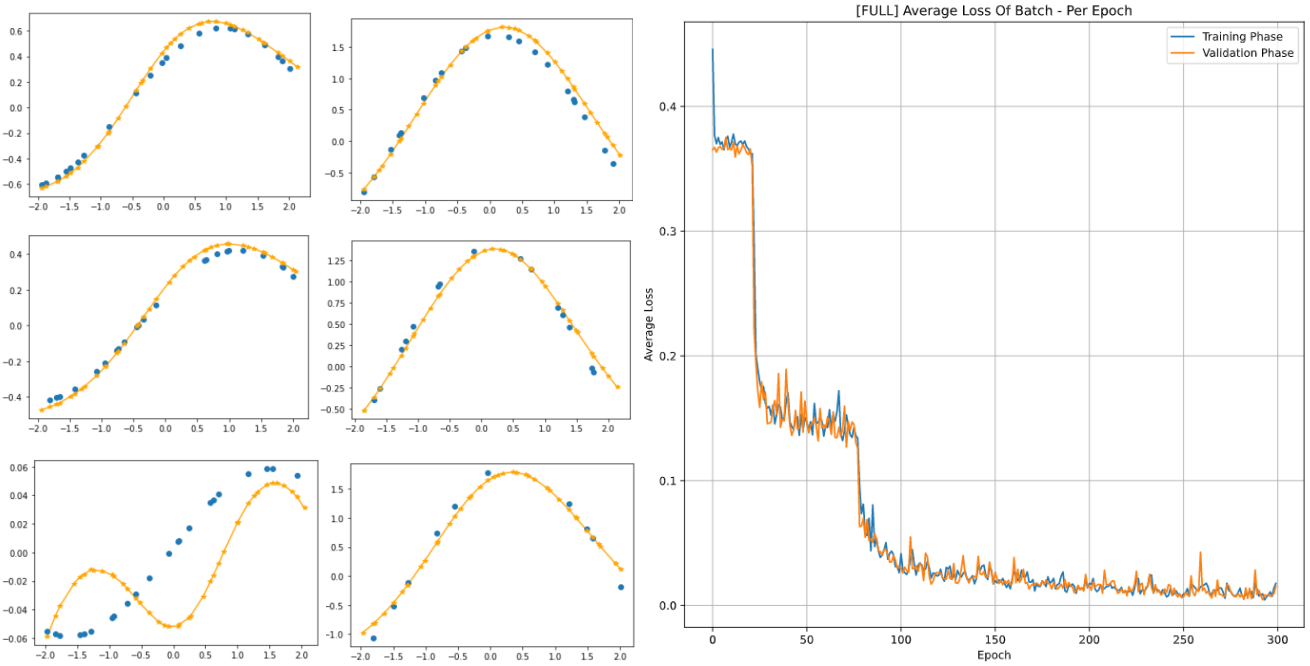
Results (test results and graph showing the average loss every epoch):



Test 2 - Hyperparameters:

- *Batch size*: 80
- *Number of epochs*: 300
- *Learning rate*: 0.001
- *rDim*: 6
- *hDim*: 200

Results (test results and graph showing the average loss every epoch):



Final Selection for Hyperparameters:

- *Batch size*: 20
- *Number of epochs*: 900
- *Learning rate*: 0.001
- *rDim*: 8
- *hDim*: 256

Results (test results and graph showing the average loss every epoch):