1.1 Data Preparation

To avoid the choice of hyper-parameters overfits the training data, it is a common practice to split the training dataset into the actual training data and validation data and perform hyper-parameter tuning based on results on validation data. Additionally, in deep learning, training data is often forwarded to models in **batches** for faster training time and noise reduction.

In our pipeline, we first load the entire MNIST data into the system, followed by a training/validation split on the training set. We simply use **the first 80% of the training set as our training data and use the rest training set as our validation data**. We also want to organize our data (training, validation, and test) in batches and use different combination of batches in different epochs for training data. Therefore, your tasks are as follows:

(a) follow the instruction in code to complete `load_mnist_trainval` in ./`utils.py` for training/validation split

(b) follow the instruction in code to complete `generate_batched_data` in ./`utils.py` to organize data in batches

You can test your data loading code by running:

```
$ python -m unittest tests.test_loading
```

2 Model Implementation

You will now implement two networks from scratch: a simple softmax regression and a two-layer multi-layer perceptron (MLP). Definitions of these classes can be found in ./`models`.

Weights of each model will be randomly initialized upon construction and stored in a weight dictionary. Meanwhile, a corresponding gradient dictionary is also created and initialized to zeros. Each model only has one public method called `forward`, which takes input of batched data and corresponding labels and returns the loss and accuracy of the batch. Meanwhile, it computes gradients of all weights of the model (even though the method is called forward!) based on the training batch.

2.1 Utility Function

There are a few useful methods defined in ./`models`/`_base_network.py` that can be shared by both models. Your first task is to implement them based on instructions in `_base_network.py`:

(a) **Activation Functions**. There are two activation functions needed for this assignment: `ReLU` and `Sigmoid`. Implement both functions as well as their derivatives in ./`models`/`_base_network.py` (i.e, `sigmoid`, `sigmoid_dev`, `ReLU`, and `ReLU_dev`). Test your methods with:

```
$ python -m unittest tests.test_activation
```

(b) **Loss Functions**. The loss function used in this assignment is Cross Entropy Loss. You will need to implement both Softmax function and the computation of Cross Entropy Loss in ./`models`/`_base_network.py`.

```
$ python -m unittest tests.test_loss
```

(c) **Accuracy**. We are also interested in knowing how our model is doing on a given batch of data. Therefore, you may want to implement the `compute_accuracy` method in ./`models`/`_base_network.py` to compute the accuracy of given batch.

2.2 Model Implementation

You will implement the training processes of a simple Softmax Regression and a two-layer MLP in this section. The Softmax Regression is composed by a fully-connected layer followed by a ReLU activation. The two-layer MLP is composed by two fully-connected layers with a Sigmoid Activation in between. Note that the Sofmax Regression model has no bias terms, while the two-layer MLP model does use biases. Also, don't forget the softmax function before computing your loss!

(a) Implement the forward method in `softmax_regression.py` as well as `two_layer_nn.py`. If the mode argument is train, compute gradients of weights and store the gradients in the gradient dictionary. Otherwise, simply return the loss and accuracy. Test:

```
$ python -m unittest tests.test_network
```

3 Optimizer

We will use an optimizer to update weights of models. An optimizer is initialized with a specific learning rate and a regularization coefficients. Before updating model weights, the optimizer applies L2 regularization on the

model:

$$J = L_{CE} + \frac{1}{2}\lambda \sum_{i=1}^{N} w_i^2 \tag{3}$$

where J is the overall loss and L_{CE} is the Cross-Entropy loss computed between predictions and labels.

You will also implement an vanilla SGD optimizer.The update rule is as follows:

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta J(\theta) \tag{4}$$

where θ is the model parameter, η stands for learning rate and the ∇ term corresponds to the gradient of the parameter.

In summary, your tasks are as follows:

(a) Follow instructions in the code and implement `apply_regularization` in `_base_optimizer.py` Remember, you may **NOT** want to apply regularization on **bias** terms!

(b) Implement the `update` method in `sgd.py` based on the discussion of update rule above.

Test your optimizer by running:

```
$ python -m unittest tests.test_training
```

4 Visualization

It is always a good practice to monitor the training process by monitoring the learning curves. Our training method in `main.py` stores averaged loss and accuracy of the model on both training and validation data at the end of each epoch. Your task is to plot the learning curves by leveraging these values. A sample plot of learning curves can be found in Figure 1.

(a) Implement `plot_curves` in ./`utils.py`. You'll get full marks on this question as long as your plot makes sense.

5 Experiments

Now, you have completed the entire training process. It's time to play with your model a little. You will use your implementation of the two-layer MLP for this section. There are different combinations of your hyper-parameters specified in the report template and your tasks are to tune those parameters

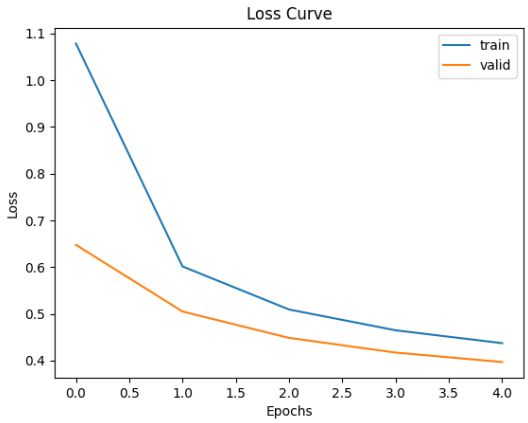

Figure 1: Example plot of learning curves

and report your observations by answering questions in the report template. We provide a default config file `config_exp.yaml` in ./`configs`. When tuning a specific hyper-parameter (e.g, the learning rate), please leave all other hyper-parameters as-is in the default config file.

(a) You will try out different values of learning rates and report your observations in the report file.

(b) You will try out different values of regularization coefficients and report your observations in the report file.

(c) You will try your best to tune the hyper-parameters for best accuracy.

(d) When tuning for best accuracy, tuning just epochs is not interesting. Tune at least 3 hyper-parameters (not including epochs). You may increase or decrease epochs it does not count as 1 of the 3.

(e) When tuning for best accuracy, the best model should have a marked improvement compared to the default hyper-parameters.

(f) When reporting observations be aware of applying good scientific methods.