VIROB Lab 2: Deep Learning

1 Intro

In this lab, we are going to train a simple Multi-Layer Perceptron (MLP) to classify images of handwritten digits in the MNIST dataset using the deep learning framework PyTorch.

1.1 MNIST Dataset

MNIST dataset contains images of a single digits (from 0 to 9) written by human as in Figure.1. These images have the size of 28×28 and have a single channel (i.e. gray scale).

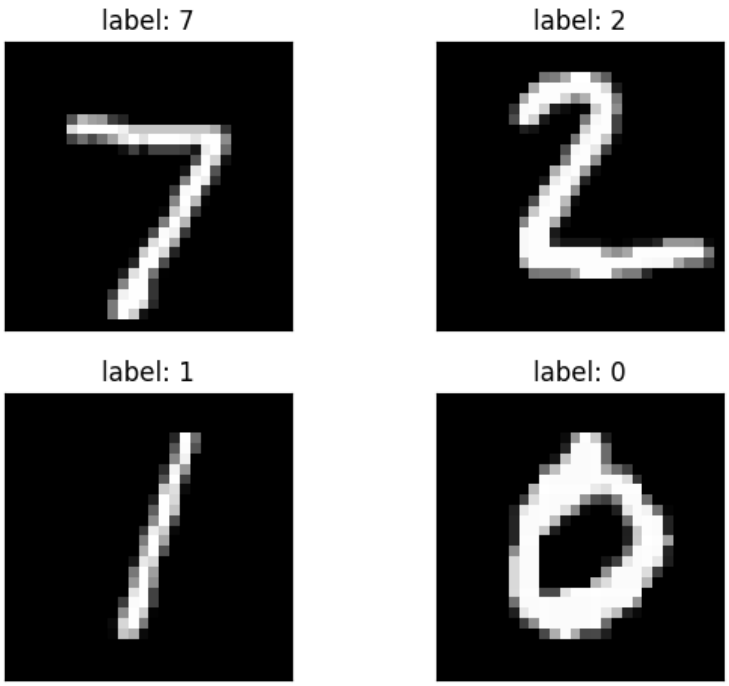

Figure 1: Sample images taken from MNIST dataset

This dataset contains 60000 data samples for training and 10000 data samples. Each data sample is a pair of **image** and **label**.

1.2 Model

1.2.1 Architecture

The MLP we train in this lab has the architecture shown in Figure.2. This MLP is made of two Linear (a.k.a fully connected or dense) layers. The first layer receives input of the size $(N, 28^2)$ and maps it to a tensor of size (N, N_HIDDEN). The second layer takes the output of the

first layer and maps it to another tensor of size $(N, 10)$. Here, N is number of data samples that are fed into the model and 10 is the number of classes in MNIST.

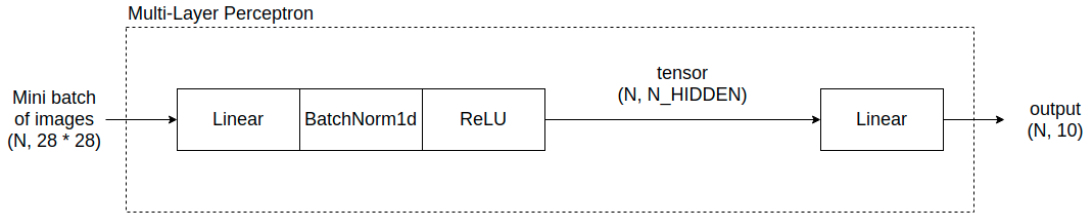

Figure 2: Architecture of an MNIST classifier

The output of the model (a tensor of size $(N, 10)$) is interpreted as following: element j-th of row i-th denotes the probability of image i-th belong to class j-th.

The first Linear layer is padded with a BatchNorm layer to accelerate the training process. The theory of this layer is outside of the scope of this lab. Interested readers can refer to this blog or the BatchNorm paper for more details. To help the model learn non-linear patterns, the output of the BatchNorm layer is activated by ReLU function defined in Eq.(1).

$$\texttt{ReLU}(x) = \texttt{max}(0, x) \tag{1}$$

Since the output of the last Linear layer is unbounded float value (in the range of $(-\infty, +\infty)$), it is not yet the class probability (which is float value in $[0, 1]$). This unbounded value is called **logit** and can be mapped into probability by Softmax function defined as following:

Let $\mathbf{x} = [x_0, x_1, ..., x_{n-1}]$ be a vector of unbounded float value (i.e. $x_i \in (-\infty, +\infty)$)

$$\texttt{Softmax}\left(x_i\right) = \frac{\exp\left(x_i\right)}{\sum_{j=0}^{n-1} \exp\left(x_j\right)} \tag{2}$$

The Softmax function is applied to each row of the MLP's output (size $(N, 10)$) to obtain another tensor of size $(N, 10)$. The row i-th of the resulted tensor contains element within the range of $[0, 1]$, thus representing the class probability of image i-th.

1.2.2 Loss Function

The MLP is trained by minimizing the cross entropy loss. Assume, $\mathbf{x} = [x_0, x_1, ..., x_9]$ is the model's predicted class **probability** for an image I which has the label l (l is the digit that I represents). The cross entropy loss for the data sample (I, l) is

$$\texttt{cross\_entopy}(\mathbf{x}, l) = -\log(x_l) \tag{3}$$

In practice, at each training step several data samples (a mini batch) are fed into the model and the `cross_entopy` loss is computed for all of these samples, then the average value is used as the final loss.

Intuitively, minimizing the cross entropy loss forces the model to push the predicted probability of the correct class higher. Since this probability is produced by Softmax, pushing the correct probability higher effectively lower probability of the wrong class.

2 Implementation

2.1 Training Loop

The training pipeline is illustrated in the Figure.3. At each training step, a mini batch of data - a set of (image, label) is taken from the dataset and fed to the model. The model performs

its prediction. Then, loss function uses the labels and model's prediction to compute a scalar number called the loss which quantifies how badly the model done on this mini batch. This loss is then propagated backward from the last layer to the first layer of the model to correct the model's parameters so that the loss function is minimized.

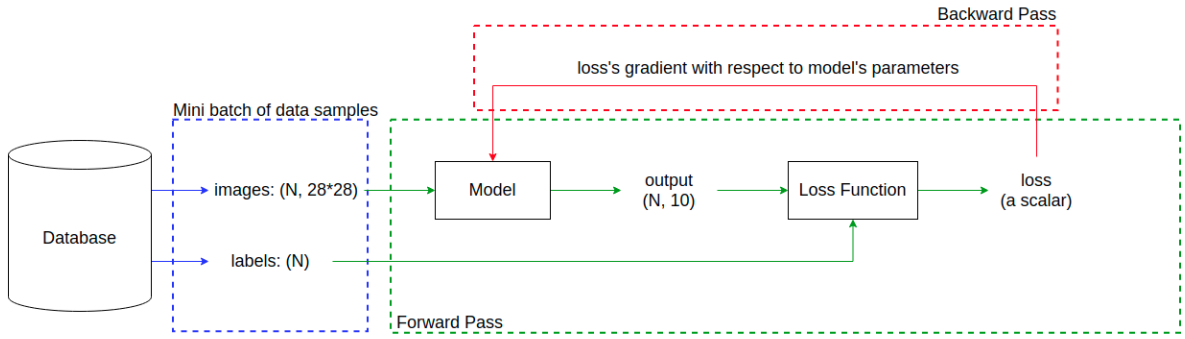

Figure 3: Training pipeline

In PyTorch framework, the backward pass is automatically defined once the model's forward pass is defined. With the backward pass has been already taken care of, our job are

1. Build the dataset from raw data
2. Define the model and loss function
3. Define the training loop
4. Get a coffee while model is being trained
5. Evaluate model's performance

The training loop and the evaluation are respectively implemented in `train.py` and `eval_classifier.py` No modification is required for them unless you want to try your own training loop or adding new metric for evaluation. To finish this lab, you need to do step 1 and 2 which is to build the dataset and define the model.

2.2 Expected Work

2.2.1 Dataset

The dataset is defined by the class `MNISTDataset` in `dataset.py` where a few blanks are made for you to fill in. The target of this script are

1. Parse raw data and store them into a database.
2. Provide indexing into database which is a function accept an index (an integer) as input and return a data sample - a pair of (image, label). This is carried out by method `__getitem__` which you need to fill in.
3. Provide function for batching individual data point - a pair of (image, label) into a mini batch of data (a tensor storing N images, a tensor storing N labels). This is carried out by method `collate_func` which you need to fill in.

2.2.2 Model

The model is defined by the class `MNISTClassifier` in `model.py` where you need to fill in some blanks.

The constructor of this class (i.e. its `__init__` method) defines the layers that make up the model and stores these layers in the attribute `self.net`. In addition, the loss function (cross entropy) is stored in the attribute `loss_fnc`.

The forward pass of the model is defined by the method `forward` which accepts a mini batch in the form of a Dictionary as input and return another Dictionary as output. The output of this method depends on the operation modes of the model (training or evaluating). If the model is in training mode, its loss function is invoked to compute loss. On the other hand, in evaluating mode, the `decode_prediction` method is invoked to compute class probability from logits and produce class prediction as the class receives the highest probability.

You are required to finish the `__init__` method to finish the model definition and `decode_prediction` to decode model's prediction

2.2.3 Training

Once you finish `dataset.py` and `model.py`, you can invoke `train.py` by opening a terminal in the directory containing `train.py` and type "python train.py 128" (128 represents the N_HIDDEN which can be set to any value that you wish). The training and then evaluation will take place.