Student name: Huyen Nguyen
Student ID: 11364650
EUID: hn0139
CSCE 5218 – Fall 2022

Project Report: Image Classification with Pytorch

1. Image Classification with the MNIST Dataset

1.1. Data preprocessing and exploration

The dataset is available on the Pytorch Dataset library, so we just need to load the dataset from Pytorch. The dataset originally included two pre-divided subsets: train subset (*n=60000*), and test subset (*n=10000*). The data contains raw images with pixel's values ranging from 0 to 255.

The raw image data were transformed into tensors and normalized to be in [-1, 1]. The transformed dataset was then loaded into Pytorch DataLoader which randomly divided data into batches. Note that the train set was shuffled well before dividing, while the test set was unnecessary. I chose a batch size of 32 by considering my available computational resource, but I also experimented with the model performance with a wide range of batch sizes. At this step, each data point has a size of [4, 1, 28, 28] *(torch.Size([4, 1, 28, 28]))*

The dataset has 10 class labels from 1 to 10. However, as the loss function built in Pytorch (e.g. *nn.CrossEntropyLoss()*) is able to handle one-hot encoding, we do not have to do the one-hot label encoding.

1.2. Model development

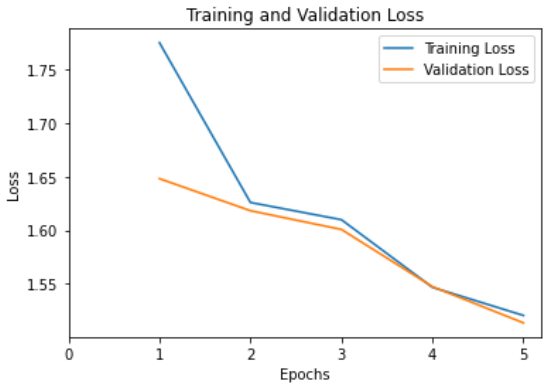
For model performance comparison between Pytorch and Keras, I tried to create a neural network model with similar architecture with the reference example. The neural network model was created with three fully-connected layers, excluding the input layers. First, the input data with the shape of (28x28) was flattened out to pass to the first hidden layer with 512 units and ReLU non-linear activation on top of it. The next hidden layer has the same number of units and ReLU activation function. The last layer (also called the output layer) has 10 units corresponding to the number of classes and applies a softmax activation function to turn the predicted output into probabilities. Specifically, the created network has the following properties:

```
Net(
(fc1): Linear(in_features=784, out_features=512, bias=True)
(fc2): Linear(in_features=512, out_features=512, bias=True)
(fc3): Linear(in_features=512, out_features=10, bias=True)
)
```

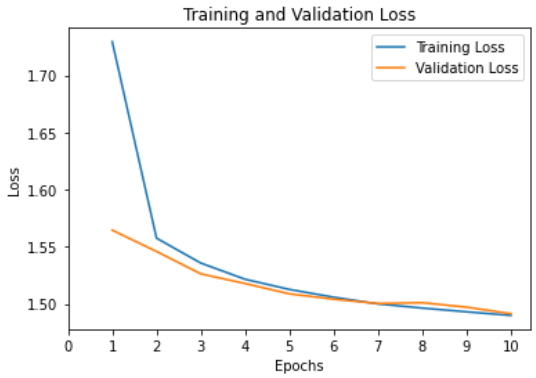
1.3. Model training and validation

All models were trained on CPU due to the resource limitation, but the codes were written in the way so that the model could be trained on GPU if available.

The model was trained on 5 epochs; the train set was used to train while the test set was used to validate as suggested by the reference example. In reality, it is suggested to partition further the train set into two subsets for training and validation, while the test set is only used for testing when the training has completed. We used Cross Entropy Loss, and Stochastic Gradient Decent with momentum = 0.9, and learning rate = 0.001 to train the model. Looking at the loss on training validation sets below, the model seems to be not converged yet. Very likely increasing the number of epochs could result in higher performance. On 5 epochs, the model got 0.951 accuracy on the test set (validation). This performance is compatible with the model built on Keras with a comparable architecture. The model with the best loss was saved to the checkpoint.



When we increased the number of epochs to 10 with the same batch size to compare the performance, the model performance improved, with **0.971 accuracy** on the validation set. The model performance does not indicate any overfitting issue.

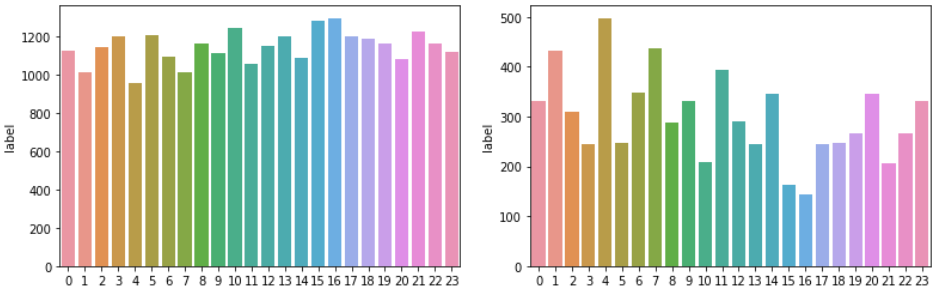


2. Image Classification with the American Sign Language Dataset

2.1. Data exploration and preprocessing

The dataset is unavailable on the Pytorch Dataset library, so we have to load it from the local disk. The data is in CSV format rather than raw images. The dataset originally contained two data subsets: a training set with shape (27455, 785), and a validation set (7172, 785) in which each row has 784-pixel values and a class label, representing an image. We separated X and y in each data subset, and then reshaped the X into the shape of *(1, 28, 28).* Several images are visualized below:

The class labels are from 0 to 23, corresponding to 24 alphabet letters. The data in classes are quite balanced on the training set but vary on the validation set. It was said to exclude the letter j and z.



Class distribution on training set **Class distribution on test set**

Again, we do not need to do one-hot label encoding as the example in Keras showed, since the loss function in Pytorch can handle that.

2.1.1. Data preprocessing

The data pixel values, ranging from 0 to 255, were normalized to be between -1 and 1. Further, in this task, I created a customized Dataset object *(class CustomDataset())* on Pytorch, which gets X and Y arrays as input, and output normalized X, and Y tensors. The datasets were then loaded into Pytorch DataLoader. The train data were randomly sampled and partitioned into batches (batch size = 32). The experiments on different batch sizes on this dataset indicated that smaller batches resulted in better performance in certain model.

2.2. Model development

2.2.1. Shallow Neural Network Model

2.2.1.1. Model Architecture

Following the second reference notebook, the neural network is created similarly in the first section. Specifically, the neural network model was created with three fully-connected layers, excluding the

input layers. First, the input data with the shape of (28x28) was flattened out to pass to the first hidden layer with 512 units and ReLU non-linear activation on top of it. The next hidden layer also has the same number of units and ReLU activation function. Finally, the last layer (also called output layer) has 24 units corresponding to the number of classes and applies a SoftMax activation function to turn the predicted output into probabilities. Specifically, the created network has the following layers:

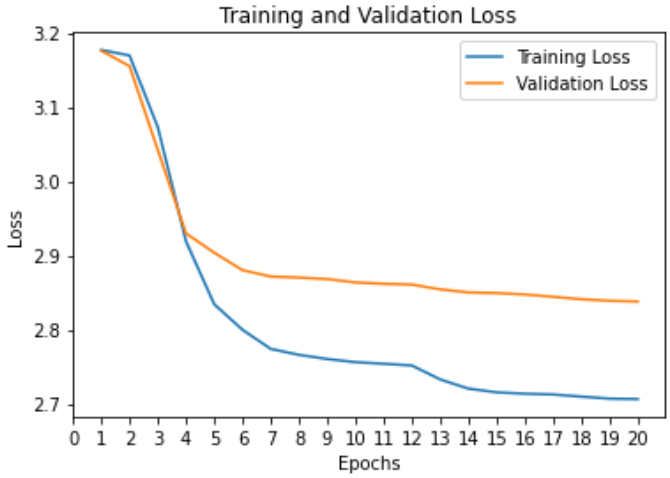
```
    Net(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=24, bias=True)
  (softmax): Softmax(dim=1)
  )
```

2.2.1.2. Model training and validation

The model was trained on 20 epochs with a batch size of 4; the train set was used to train while the test set was used to validate. I observed that this model performed better with smaller batch size. The batch size of 32 resulted in 0.25 accuracy, while the batch size of 4 had much better accuracy. This seems to be uncommon in reality.

Similarly, Cross Entropy Loss, and Stochastic Gradient Decent with momentum = 0.9, and learning rate = 0.001 were set to train the model.

The loss on training and validation sets over 20 epochs is shown below. The model seems to have the overfitting issue as it performed pretty well in training and much worse on the validation set. The best model with the lowest loss in validation was saved to the checkpoint, which got 0.4052 accuracy on the validation set.



This performance is much lower than the reported performance of the similar model architecture built on Keras (0.8480 accuracy). The model on Keras was also trained on 20 epochs but a batch size of 1. I would like to further investigate this issue later.

2.2.2. Convolutional Neural Network Model

2.2.2.1. Model Architecture

Similarly, the neural network is created similarly with the third reference notebook using Convolutional Neural Network (CNN). Specifically, the model consists of three Convolutional (Conv) Blocks, and a fully-connected block at the end. Each block consists of a Conv layer, a Batch normalization layer, and a Max pooling layer.

Specifically, in the first block, the input data with the shape of (1x28x28) was passed through the first Conv blocks to learn the local features. As the images were normalized into a single-color channel, the input channel of the first Conv layer is 1. A kernel size of 3 was used for filters, and a stride of 1 in all Conv layers in the network. That means a filter with a window size of (3, 3) shifts over the image one pixel at a time to obtain feature maps. After each Conv layer, we applied a BatchNorm layer to normalize the hidden inputs. Finally, the feature map was down-sampled by applying a Max pooling on them. This structure was repeated for two other Conv blocks, except for the second Conv blocks in which I did a Dropout of 2% nodes. The network with the number of units and layers was designed by following reference notebook 3.

After passing the input tensor over three above Conv blocks, it was flattened out into 1D to pass through two fully-connected layers. We also used a Dropout with a ratio of 0.3 between the two to reduce overfitting. In the last fully-connected layer, Softmax activation was applied to the output (24 classes) to turn them into probabilities. The whole model structure is shown below:

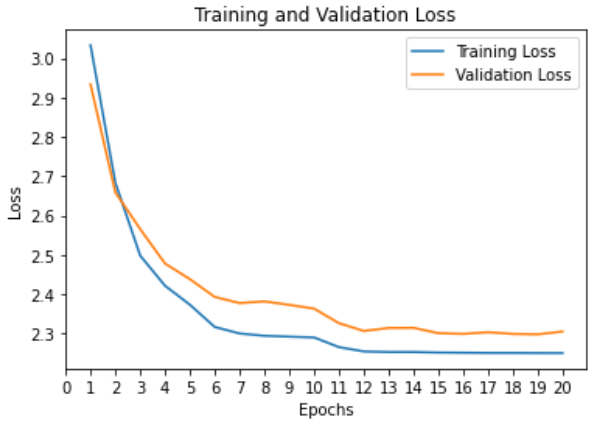
```
CNNNet(

  (conv1): Conv2d(1, 75, kernel_size=(3, 3), stride=(1, 1))

  (batchnorm1): BatchNorm2d(75, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)

  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)

  (conv2): Conv2d(75, 50, kernel_size=(3, 3), stride=(1, 1))

  (batchnorm2): BatchNorm2d(50, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)

  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)

  (conv3): Conv2d(50, 25, kernel_size=(3, 3), stride=(1, 1))

  (batchnorm3): BatchNorm2d(25, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)

  (pool3): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1,
ceil_mode=False)

  (dropout1): Dropout(p=0.2, inplace=False)

  (dropout2): Dropout(p=0.3, inplace=False)
```

```
    (fc1): Linear(in_features=100, out_features=512, bias=True)
    (fc2): Linear(in_features=512, out_features=24, bias=True)
    (softmax): Softmax(dim=1)
  )
```

2.2.2.2. *Model training and validation*

I used the same batch size (batch size = 32) and the number of epochs (epochs = 20) to train this CNN model. Further, Cross Entropy Loss, and Stochastic Gradient Decent with momentum = 0.9, and learning rate = 0.001 were set to train the model.

The loss on training and validation sets over 20 epochs is shown below. The best model with the lowest loss in validation (at epoch 19) was saved to the checkpoint whose accuracy is **0.9632.** This performance is similar to the CNN model built in Keras; this is understandable because similar model architecture and hyperparameters were used to train the CNN on Pytorch.



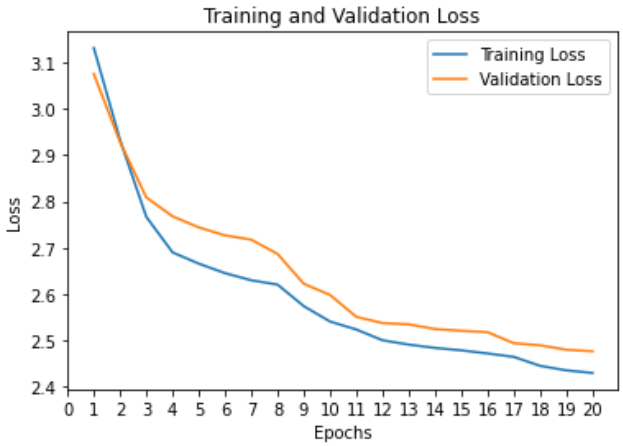
2.2.3. Convolutional Neural Network Model with Data Augmentation

2.2.3.1. *Data preprocessing with augmentation methods*

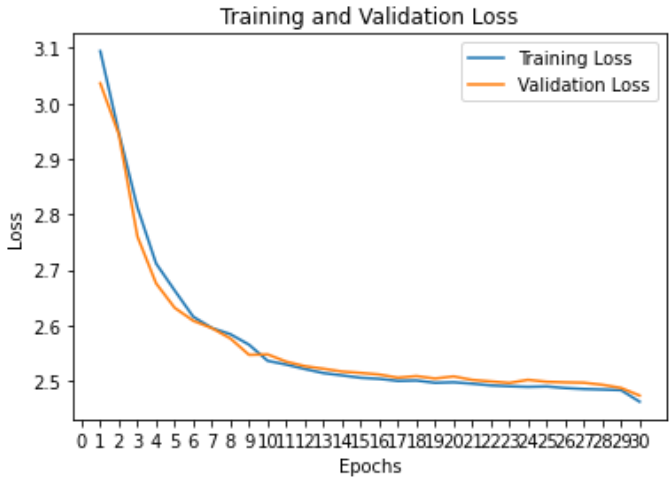
In this task, some data augmentation methods were used to improve the model's performance and its generalization on the unseen data. As suggested by the reference model with augmentation, several methods were applied as follows: randomly rotating the image in 10 degrees, randomly zooming the image (Note that we have to zoom the image and resize it in Pytorch), randomly shifting images horizontally and vertically, and randomly flipping images horizontally. Pytorch *transforms.Compose* was used to wrap all augmentation transformations. The same CNN model and training hyperparameters were used to train this model.

2.2.3.2. *Model training and validation*

The model was trained on 20 epochs with a batch size of 32. The best model with the lowest validation loss was saved to the checkpoint. The validation accuracy shows that the model performed much worse than the CNN model trained on the data without data augmentation, just 0.78 accuracy.



Looking at the loss of training and validation over epochs, it is likely that the model can achieve higher performance with more iterations. However, when increasing the number of epochs, the performance did not improve (accuracy = 0.78):



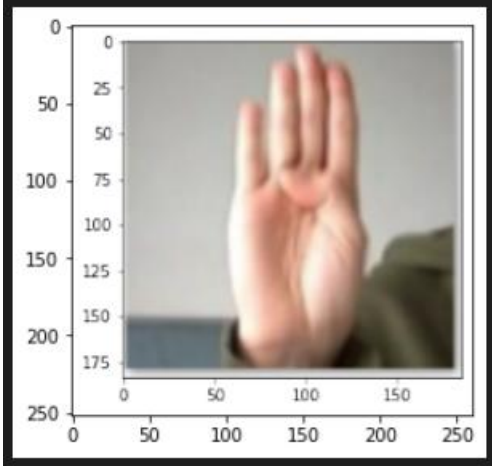
2.3. Model Deployment

The best CNN model (accuracy = 0.96) saved to the checkpoint was loaded to make predictions on the unseen data. Two raw images of sign language (letters a, and b) were preprocessed in the same way we did for the training data input, including rescaling and resizing into (1, 28, 28), converting to tensor, and finally normalizing (to be between [-1, 1]). Surprisingly, the model made perfect predictions on the two sample images, proving the model has a high performance and is reliable to use.

```
predict('../data/b.png')
```

✓ 0.9s

```
(1, 'b')
```



```
predict('../data/a.png')
```

✓ 0.6s

```
(0, 'a')
```