Q2_1

Neural Network Perception (NNP) with hidden layer=1, hidden size=40, batch size=60, epoch=5 is applied into the datasets Fashion-MNIST. In the condition of ReLU as transfer function, Cross-entropy loss function as performance index, Adam as optimization function, the model reach 86% accuracy and the training model takes 42s.

	Time	Accuracy	Hidden layer	Batch size
Q2_1	42.06s	86%	1	60

	Time	Accuracy	Transform function	Loss function
Q2_1	42.06s	86%	nn.ReLU()	nn.CrossEntropyLrectifieross()

The input size of NNP is 28x28 because the monochrome pattern pixels are 28x28. The code torchvision.datasets.MNIST() transforms the dataset into a torch with the shape (C,H,W) in the range (0,1). After setting up the data pipe, the code toch.utils.data.DataLoder() decides the batch size, the data volume in each iteration.

```
# ------------------------------------------------------------------------------
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
# ------------------------------------------------------------------------------
torch.manual_seed(200)

train_set = torchvision.datasets.FashionMNIST(root='./data_fashion', train=True, download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
```

Figure 1: Python code for data pipe

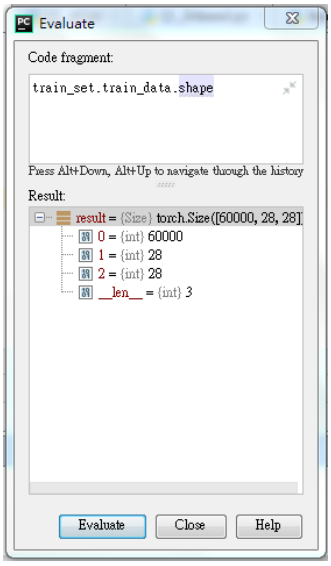


Figure 2: Pycharm debug to evaluate data shape: (6000,28,28)

The code for the computation of NNP is programmed using Pytorch, which deploys dynamic computational graphs. The dynamic computational graph defines the order of computations that are required to be performed. The Pytorch uses Object Oriented Programming feature to set up the framework for deep learning.

Q2_2

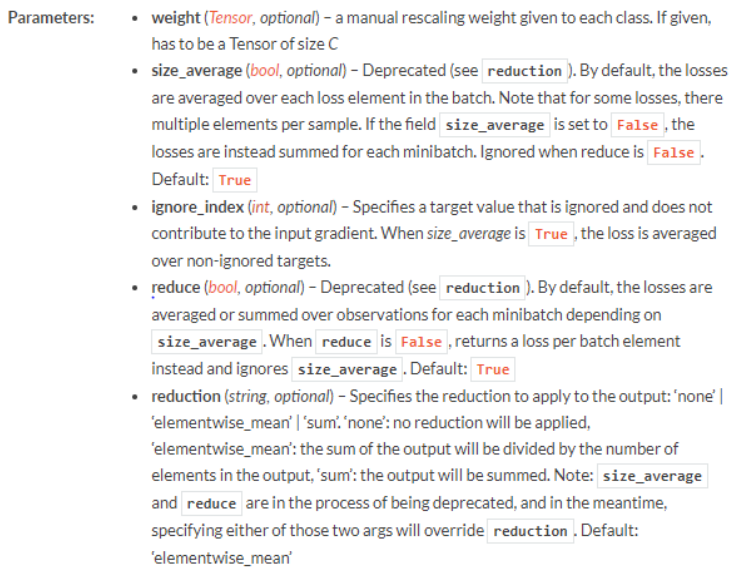
The programm ability to run on GPU is considered. Since CPU is the default operation for computational device, the object have to be transformed into cuda to run on GPU. GPU speeds up the computation of NNP because GPU adopts Compute Unified Device Architecture (CUDA) which is different from CPU.

```
net = Net(input_size, hidden_size, num_classes)
net.cuda()
```

Figure 3: Python code for transforming into cuda format

Q2_3

Program is set up to perform mini-batch gradient descent. Firstly, the data is transported by mini-batch. Secondly, the Loss functions such as torch.nn.CrossEntropyLoss(….. reduce=None,……) set reduce=True by default; as such, the losses are averaged or summed over observations for each minibatch depending on size_average. Figure 4 is the explanation from Pytorch official website.



Parameters:

- **weight** (*Tensor*, *optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size *C*
- **size_average** (*bool*, *optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **ignore_index** (*int*, *optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When *size_average* is `True`, the loss is averaged over non-ignored targets.
- **reduce** (*bool*, *optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string*, *optional*) – Specifies the reduction to apply to the output: 'none' | 'elementwise_mean' | 'sum'. 'none': no reduction will be applied, 'elementwise_mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: 'elementwise_mean'

Figure 4: Screenshot of Pytorch official document for illustrating paramours of loss function

Q2_4

By increasing the number of layers from 1 to 2 in the network, the computational time is increased from 42.06s to 44.74s with 6% increment. The accuracy remains roughly the same because the total number of weights and biases in the network are roughly the same.

The longer computing time is due to sequential calculation. Forward propagation comes before backward propagation. If more hidden layers are in NNP, more variables need to wait for the calculation from previous layers.

	Time	Accuracy	Hidden layer	hidden_size
Q2_1	42.06s	86%	1	40
Q2_4	44.74s	85%	2	20

Q2_5

Torch.optim.SGD() and torch.optim.Adam() are being compared. The difference between the performance for both optimization algorithms in terms of computing time and accuracy is up to 1.2%. Torch.optim.SGD() utilizes Averaged Stochastic Gradient Descent. On the other hand, torch.optim.Adam() utilizes first-order gradient-based optimization algorithm for stochastic objective functions, based on adaptive estimates of lower-order moments. Although torch.optim.Adam() has a lower memory requirements and has a more efficient calculation, the result shows that both algorithm has similar performance.

	Time	Accuracy	optimization algorithms	Loss function
Q2_1	42.06s	86%	torch.optim.Adam()	nn.CrossEntropyLrectifieross()
Q2_5	42.02s	85%	torch.optim.SGD()	nn.CrossEntropyLrectifieross()

Q2_6

Transform function nn.ReLU() has a shorter computing time than nn.Tanh(). The equation of nn.ReLU() is f(x)=max(0,x), and $\text{Tanh}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$. Comparing nn.ReLU, nn.Tanh needs to conduct exponential calculation, which is more complex. Therefore, NNP with the transfer function of nn.ReLU() uses less time to train the model.

	Time	Accuracy	Transform function	Loss function
Q2_1	42.06s	86%	nn.ReLU()	nn.CrossEntropyLrectifieross()
Q2_6	43.19s	86%	nn.Tanh()	nn.CrossEntropyLrectifieross()

Q2_7

Calculating the standard deviation of the gradients with respect to inputs can be used to find the potential key inputs. Each training iteration tends to minimize the loss function. When the loss function is partial differential to the input variables, at each iteration, the gradient of variable represents the change of each input variable to minimize the loss function.

In this section, the sequence of inputs with the highest 10 standard deviation of the gradients for each trail are [454,397,38,37,453 ,17 ,45,11,425 ,46]. Those numbers stand for the location on the picture resulting in drastic change as each trail increasing minimize the loss function. In other words, this location is the key potential feature when the NNP model is created.

The computation method used to calculate the standard deviation of the gradients is as the following: firstly, the average of gradient with respect to 784 inputs are calculated within mini-batch iteration. Secondly, the average of gradient within the whole 6000 data is calculated during an epoch iteration. Finally, the standard deviation of the average of gradient within different epoch iteration is calculated.

Q2_8

Using the dropout nodes can prevent over-fitting problem. It also speeds up the model training according to the test result. However, there is a potential risk of over simplifying the model when too many neurons are turned off during the training. Therefore, the parameter p in the nn.Dropout() needs to be optimized to create a better NNP model.

	Time	Accuracy	dropout nodes	Loss function
Q2_1	42.06s	86%	turn off	nn.CrossEntropyLrectifieross()
Q2_8	41.77s	85%	turn on	nn.CrossEntropyLrectifieross()

```
class Net4(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net4, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.2)
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)
        return out
# ----------------------------------------------------------------------------------------
# Choose the right argument for x
net4 = Net4(input_size, hidden_size, num_classes)
net4.cuda()
```

Figure 5: python code demonstration for dropout node

Q2_9

Computation by batch has the shortest training time, however, it has the worst accuracy. As for the computation by stochastic gradient descent (SGD), it has the longest training time, but there is no guarantee that this computation method has the highest accuracy. This is because computation by SGD is prone to over-fitting, especially when the datasets is large.

	Time	Accuracy	Hidden layer	Batch size

batch	34.54s	78%	1	6000
Mini-batch	40.96s	86%	1	60
SGD	431.90s	84%	1	1

Q2_10

The confusion matrix shows the correlation between the label and the prediction based on different classes. Figure 6 is the confusion matrix for Fashion-MNIST. The training model is based on Q2_1. The sum of all the number indicated by the diagonal arrows is the total number of correct predictions out of the dataset of 10000. Hence, the accuracy is calculated at 85.55% and the misclassification rate is 14.45%.

confusion matrix		prediction										
		0	1	2	3	4	5	6	7	8	9	All
label	0	759	0	18	66	4	0	135	2	16	0	1000
	1	2	948	5	40	4	0	0	0	1	0	1000
	2	12	0	803	14	132	0	35	0	4	0	1000
	3	18	2	15	919	19	0	20	0	7	0	1000
	4	0	0	103	50	808	0	34	0	5	0	1000
	5	0	0	0	1	0	865	0	77	8	49	1000
	6	84	0	139	66	122	0	571	0	18	0	1000
	7	0	0	0	0	0	9	0	955	0	36	1000
	8	0	0	3	8	4	3	4	6	972	0	1000
	9	0	0	0	0	0	4	1	40	0	955	1000
	All	875	950	1086	1164	1093	881	800	1080	1031	1040	10000

Figure 6: Confusion matrix for prediction

Q2_11

According to the NNP model, an arbitrary input correctly corresponds to the target associated with that input.

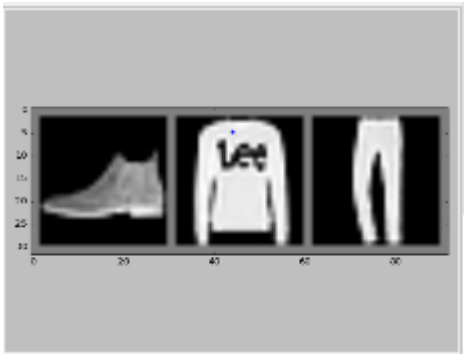


Figure 7: The arbitary input and its prediction