Advanced Deep Learning
Assingment 1

Ufuk Cem Birbiri

November 4, 2022

1 Questions and Answers

1.1 Exercise-1

This exercise is completed on a Colab Notebook with Pytorch.

1.2 Exercise-2

Q1. On line 49 of the code it is commented that we want to perform a GD based optimization. However on line 45 we invoked optim.SGD as the optimizer. Explain why in this case we are still performing a gradient descent on the whole dataset even if it seems that we are invoking a stochastic method.

Answer.1

Let's first talk about what SGD and GD are and, what their differences are. In both gradient methods, we try to minimize the error(risk) function by updating the parameters of the model in an iterative way. In GD, we go through all the samples in the train data to make an update on a parameter. On the other hand, we use only one training sample, or a subset of training samples to make an update on a parameter. If we use a subset of the training data, we call it as Mini-batch SGD.

In our case, we upload the dataset as a whole with the dimensions of [60000, 28, 28]. We feed the network with 60000 images all at once, by using Cross Entropy loss function. Since we don't separate the training data to batches and we pass the entire dataset at once, the SGD optimizer in Pytorch is used a regular GD. If we pass a subset of dataset at a time through the network, we would have used it as a SGD optimizer.

Q2. Discuss over-fitting issues by monitoring the train and test error curves.

Answer.2

Over-fitting is the state where the network learnt from the training data very well i.e the training error is so low, but it has a very bad test error. Under-fitting is the state of the model where the network learnt poorly from the training dataset, as well as the test error is bad. One should stop the training of the network where the test error is at the lowest point, therefore that model can have a good generalization on unseen data. The under-fitting and over-fitting concepts can be understood better in the Figure 1 on left.

The test and train error curves of the Exercise 1 can be seen In the Figure 1. The train and test loss in each epoch is saved them in a list. It is important to emphasize that the test set is used like a validation set, it was NEVER used as a training data, and there was no back-propagation for test set i.e. the network didn't learn anything from the test set. I just calculated the loss of the test set in each epoch. According to resulst, here is no significant decrease in the test loss after the 90th epoch. The train loss is at the minimum level. The number of training epochs as 100 is a good choose because the test loss is not increasing i.e. it is not in the over-fittin area yet.

Q3. Discuss what role does the choice of the network (i.e. number of layers and number of neurons per layer) have on the bias-variance trade-off. First describe your expectations based on theoretical analysis (arguing on the different capacity of the models) then test this expectations with a small experimental campaign. Is the expected behavior

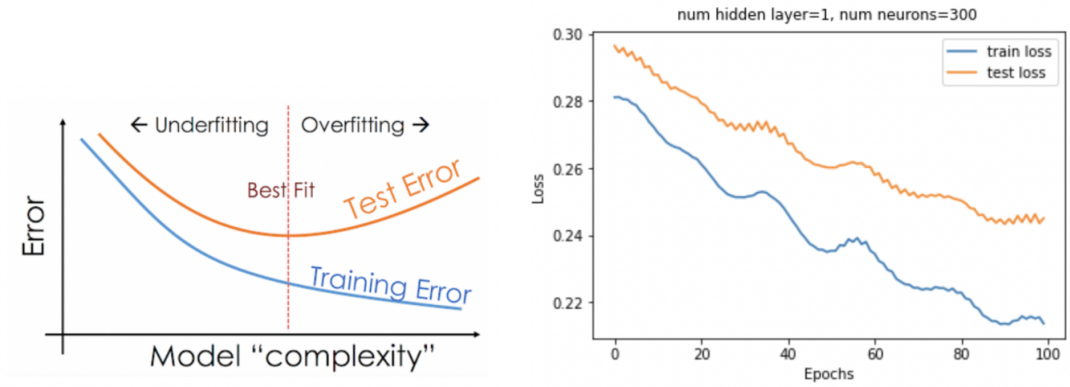

Figure 1: Left: The representation of over-fitting and under-fitting. The image is taken from [**?**]. Right: The train and test losses calculated in Exercise 1

confirmed by experimental results? Briefly discuss your findings.

Answer.3

The number of layers and number of neurons per layer are two significant parameters in the networks. I will discuss this question in two different ways:

1. The role of number of neurons (per layer):

In principle, we don't need very deep neural networks. Only one hidden layer with large number of neurons can theoretically approximate any reasonable function when there is sufficient training data. However, there are some problems in using very wide and shallow neural networks. The biggest problem is that very wide and shallow networks tend to learn from the training data very well, but it performs so bad in test data. The generalization of these networks are not good. In theory, if you train a wide and shallow network with every possible input, eventually this network will memorize the corresponding output. But that is impossible because in practice you will not have every possible training data in the dataset. Also, increasing the # of neurons instead of increasing the # of layers could not learn the various features at various levels similar to deep neural networks.
Let's investigate this idea. I built three different network with keeping the number of layers fixed(1 hidden layer only) and changing the number of neurons in that hidden layer. I fixed the number of layers in the network and we just play with the number of neurons in layers. The results are shown in the Table.1, and the train-test error plots are shown in Figure 2.

num hidden layers	**num neurons**	**test accuracy**	**train accuracy**
1	15	0.834	0.803
1	300	0.926	0.924
1	1000	0.927	0.945

Table 1: The details of different networks with accuracy results. All the parameters are fixed in the networks except the number of neurons in the hidden layer. The number of epoch is 100 and the learning rate is 0.1.

According to Table 1, when we increase the number of neurons in a hidden layer while keeping the other parameters fixed, there is a slightly increase in the model performance on test set. When the number of neurons is 15, the network could not learn same as the other networks. The test accuracy has the lowest value as 0.834. However, the networks have higher test accuracy results. The accuracy of network with 300 neurons(0.926) is similar to the accuracy of the network with 1000 neurons(0.927). The reason could be that the dataset is not big enough.How much we increase the number of neurons in a hidden layer, its test performance would not increase linearly. If we want a better result from the layer with 1000 neurons, we should have feed them with a larger dataset. In our experiment we used 60000 images for each layer so the results are very close(0.926 and 0.927). Also, in Figure 2, there is

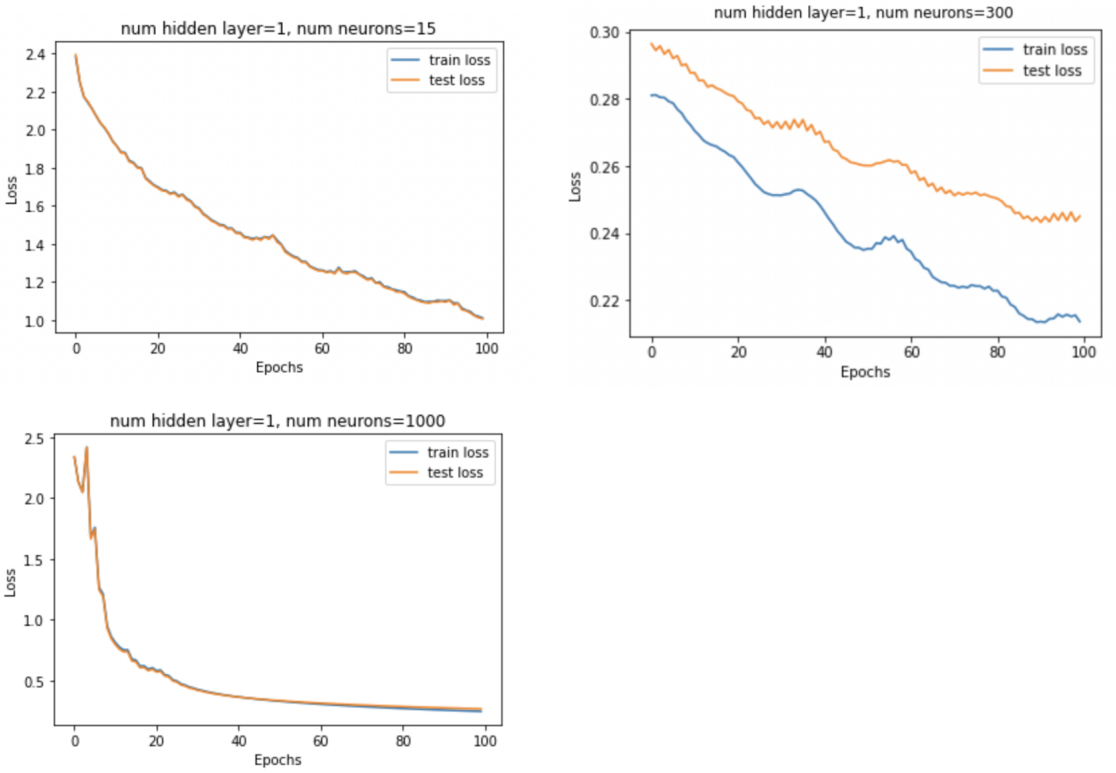

Figure 2: The test and error curves of networks with different nuumber of neurons.

no under-fitting or over-fitting in the train and test error curves.
The conclusion is that wide and shallow networks practically are not good idea. A large and well-collected training data is needed for the wide and shallow networks to learn the data. I think we should not trust to wide and shallow networks for good generalization because they don't learn different features in various layers.

2. The role of number of layers:
The main advantage of having multiple layers is that they could learn different features at various levels. In each layer, a different feature of the data is learned. For example in a CNN for human face classification, the first layer learns the edges, the second layer learns the shapes like nose or eyes, and the third layer can learn more higher-order features like faces. Having a multi-layer network is an advantage since it can learn all the intermediate and higher-order features from the raw data. This will reduce the test error and increase the generalization of the network. Also, as we increase the size of the model, we are giving more parameters that need to be learned, and therefore we are increasing the chance of over-fitting and memorizing the train data. When the number of layers in the network are getting larger, the vanishing gradient problem occurs.
Let's investigate this idea. I built 4 networks with different number of layers. In each network, the first hidden layer has 300 neurons and then 100 neurons in other layers. The last activation function is Sigmoid and the other activation functions are ReLU. An example of a network with 2 hidden layers can be seen in Figure 3.

I created 4 networks that has the number of hidden layers as 1, 2, 10, 20, respectively. The accuracy results and the information about the networks are given in Table 2.

The best test accuracy is in network with 2 hidden layers. There is a decrease in test accuracy when the number of layers is 5, that means no need for 5 layers. Max of 2 layers can be a good choice for this dataset. On the other hand the test accuracy is very low when the number of layers is 10 or 20. The reason is that the generalization is very bad in very deep networks because of the vanishing gradient problem. The gradients are getting so close to zero in further layers and network could not

```
# Now let us define the neural network we are using
net2 = torch.nn.Sequential(
    torch.nn.Linear(28*28, 300),
    torch.nn.ReLU(),
    torch.nn.Linear(300, 100),
    torch.nn.Sigmoid(),
    torch.nn.Linear(100, 10),
)
```

Figure 3: An example of a network with two hidden layers.

num hidden layers	num neurons	test accuracy	train accuracy
1	300	0.926	0.924
2	100	0.954	0.943
5	100	0.918	0.920 10
100	0.113	0.112	
20	100	0.097	0.100

Table 2: The details of different networks with accuracy results. The number of epoch is 100 and the learning rate is 0.1 for each experiment.

learn. The training and test error curves are shown in the Figure 4 for deeper networks. We can observe the vanishing gradient problem such that after some epochs the loss stop decreasing and stay same. That is because the network stop to learn.

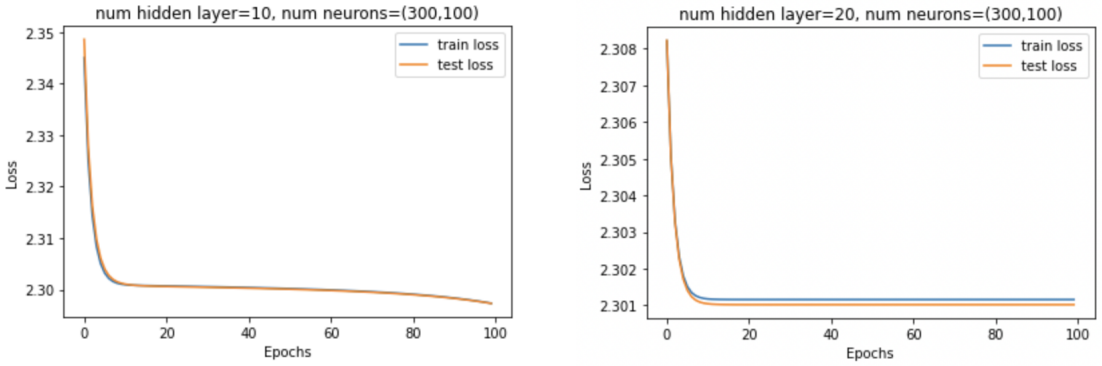

Figure 4: The training and test error curves of deep networks.

Q4. Discuss the benefits of using a cross entropy loss with respect to a quadratic loss.

Answer.4

In quadratic loss, let's assume there are k classes in the dataset and there is a probability vector [p1, p2, . . . ,pk] that has the probabilities of a sample classified by the k classes. The ground truth value of an instance is represented by a vector [a1, a2,. . . , ak] where the true class of the instance is 1(the ith) and others are 0. We can write the quadratic loss as:

$$\sum_i (p_i - a_i)^2$$

Square loss penalizes the model for making larger errors by squaring them. One disadvantage of this loss is that outliers are not handled properly. If the outlier error will be quite large, it is penalized by squaring it. Also if we plot a quadratic equation, we get a gradient descent with only one global minima without a local minima.

On the other hand, cross entropy loss is particularly useful if you have an unbalanced training data

for classification. It minimizes the distance between two probability distributions. More importantly the cross entropy has some nice properties, especially for sigmoid activation functions resulting in the gradient only depends on the neuron's output, the target and the neuron's input. This avoids learning slow-down and helps with the vanishing gradient problem from which deep neural networks suffer. Cross entropy loss for multi-class classification can be written as:

$$-\sum_i (y_i - log(\hat{y}_i))$$

where y_i is the predicted class and $\hat{y}_i$ is the true class. Since our problem is multi-class classification, and we use Sigmoid activation function in the final layer, cross entropy loss suits better than quadratic loss.

Q5. Q5. Why using a one-hot encoding? Wouldn't be simpler to use a single output? Hint: The answer has to do with the interplay between the loss and the sigmoidal activation functions.

Answer.5

First, let's analyse our network. There are 10 different classes in our dataset and the labels are [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. The problem becomes a multi-class classification. We transform labels to one hot encoding vectors before using them in the network. Also, our last activation function is Sigmoid, and the loss function is Torch.nn.CrossEntropyLoss().

Let's look at the explanation of the cross entropy loss in the Pytorch website[**?**]. It is indicated that this loss function is useful when you have a classification problem with C classes and it is good when the training data is unbalanced. Fortunately, our train data is not unbalanced since we have the class distribution of [5923 6742 5958 6131 5842 5421 5918 6265 5851 5949] when the class labels are [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] respectively. The input of Torch.nn.CrossEntropyLoss() should be a tensor of size C that do not need to be positive or sum to 1 .In our case it should be a tensor of size 10. The target of this criterion should be:

- Class indices in the range [0, C) (where C is the number of classes)
- Probabilities for each class.

Our target value(train_y) is a one-hot encoded vector where the correct class label is 1, and the others are zero which is a perfect target for Torch.nn.CrossEntropyLoss(). The dimension of our target labels is [60000, 10]. Our output is also a one-hot encoded vector with size [60000, 10].

OK, then how we create this output vector? The answer is the Sigmoid activation function in the last layer. The network that I am using is seen in Figure 5. The last Linear layer gives the output of a vector size [60000, 10], and before this linear layer the Sigmoid activation function assigns probabilities between 0 and 1. The sigmoid activation function's output is always between 0 and 1, so it is helpful for us to assign output probabilities of each class. Overall, sigmoid activation function + one-hot encoding is suitable for the cross entropy loss function in Pytorch for the multi-class classification task.

```
# Now let us define the neural network we are using
net = torch.nn.Sequential(
    torch.nn.Linear(28*28, 300),
    torch.nn.Sigmoid(),
    torch.nn.Linear(300, 10),
)


# Now we define the optimizer and the loss function
loss = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.1)
```

Figure 5: My model architecture

Q6. Test the sensibility of the gradient descent method with respect to the learning rate.

Answer.6
Choosing a appropriate learning rate is a crucial task in a neural network because a high learning rate may cause oscillations in gradient descent method, resulting in not be able to find the global(or local) minima of the gradient function. On the other hand, a low learning rate will make the learning process slow and could cause not to learn enough from the train data.
I tested 5 different learning rates while keeping the other parameters fixed. The accuracy results can be seen in Table 3

Learning rate	Accuracy
10	0.207
1	0.913
0.1	0.926
0.01	0.839
0.001	0.420

Table 3: The accuracy results using different learning rates.

According to results in Table 3, when we increase the learning rate(lr=10) the accuracy decrease since the gradient method makes big steps to find the global minima and it cannot find it. When we decrease the learning rate(lr=0.001), the accuracy also decrease since it is unable to reach to global minima. If we use lower learning rates, we should increase the number of epochs. Learning rate of 0.1 seems a suitable for this dataset and network of choice since it seems the best accuracy.

Q7. With the network architecture that is described in Q3. of Exercise 1 (line 32–33 of the code) do you experience any problem related to the vanishing of the gradient? Why?

Answer.7
Yes, I experienced vanishing gradient problem. In very deep neural networks, the value of product of the derivative is decreasing when you go deeper and at some point the partial derivative of the loss becomes very very close to zero. Therefore, the partial derivative becomes zero and the gradient vanishes.
Same thing happened in the Question 3, where when I built a deep network with 10 or 20 layers, the test accuracies had the lowest value(see Table 2). Also, the loss stop decreasing since we lost the gradients in deeper layers that can be seen in Figure 4.