Deep Learning 2020 **(Due: [2.12])**

Submission Assignment #[4]

Instructor: Jakub Tomczak *Name:* [Xingkai Wang], *Netid:* [xwg510]

1 Problem statement

Recurrent neural network (RNN) plays a significant role in dealing with sequence data. For example, there are many cases in which the information of the sequence determines the event itself. If we try to use this type of data to get useful output, we need a network that can access some prior knowledge about the data in order to fully understand the data.

In this research, we will test the performance of the three networks in processing sequence data. Firstly, we will use an simple sequence-to-sequence model which apply the linear layer to each token in the sequence. Secondly, a simple RNN network, called Elman network which is a three-layer network with the addition of a set of context units will be tested. Finally, the Long Short Term Memory (LSTM) network will be applied to finish our experiments. The final performance of three different networks will be compared and in this case we are interested in minimizing the cross entropy loss function:

$$Loss = -\sum(xi \cdot \log yi + (1 - xi) \cdot \log(1 - yi)) \tag{1.1}$$

2 Methodology

The initial approach of this research is building a Recurrent neural network (RNN) by using Pytorch. The structure of our RNN is that we have a embedding layer which transform the token to the vector, the embedding layer has 99430 inputs features because there are totally 99430 tokens and the outputs features are 300. In the second layer, we will test three different layers, which are MLP, ELman network, and LSTM. All three layers will have 300 inputs and 300 outputs. The outputs from second layer will input to a relu activation function and the global MaxPool along the time dimension will be applied which will eliminate the time dimension entirely. Then the outputs from golbal MaxPool layer will input into a fully connection layer and obtain the final outputs. Figure 1 shows the implementation of LSTM recurrent neural network.

```
class Net(nn.Module):
    def __init__(self):
        super(Net,self).__init__()
        self.emb = nn.Embedding(99430,300)
        self.lstm = nn.LSTM(300,300,batch_first = True)
        self.fc2 = nn.Linear(300,2)
    def forward(self,x):
        x = self.emb(x)
        x = torch.max(F.relu(self.lstm(x)[0]),1)[0]
        x = self.fc2(x)

        return x

net = Net()
```

Figure 1: Implementation of LSTM RNN

Before the data is fed into the neural network, we do some sequential preprocessing on the data. Firstly, all sentences are padded with value 0 to make sure they have the same length and all training data, validation data and test data are sliced out batches with batch size 200. The data will be input to the neural network in sequence order, for example, input the first token of all sentences in a batch at time t1, and input the second token of all sentences in a batch at time t2 and so on. The cross entropy loss function and Adam optimizer are chosen in the backpropagation process. Figure 2 presents the implementation of data padding.

```
def fixed_data(data):
    max_len = max([len(sent) for sent in data])
    for sent in data:
        temp = [w2i['.pad']] * (max_len - len(sent))
        sent.extend(temp)

    return data
```

Figure 2: Implementation of data padding

3 Experiments

In this research, the IMBd dataset will be used. We will test the accuracy on validation data after 1 training epoch. The batch size is set to 200. We will train on three different layers of MLP, ELMAN, and LSTM with four learning rates of 0.0001, 0.001, 0.003, and 0.01, and calculate the accuracy on the validation set. Finally, we will use the performance of the verification set to Determine the most appropriate learning rate to train the neural network, and test on the test set.

According to the machine learning experience, with enough tuning, the LSTM will perform best, followed by the Elman network, followed by the MLP. The expected results should follow this hypothesis, but the final hypothesis will based on the validation results and the final results on test data will confirm our hypothesis.

4 Results and discussion

Table 1 presents the validation accuracy on 3 layers with different learning rate after 1 epoch training. We test 0.0001, 0.001, 0.003, and 0.01 4 different learning rate respectively. Learning rate of 0.0001 does not work well on all three layers, since the learning rate is small so that the neural network can not converge to the optima after 1 epoch training. When the learning rate is larger to 0.001, the performance of all three layers improved, and became very similar. When the learning rate continues increase to 0.003, the performance of MLP layer does not change at all and the performance of Elman layer and LSTM layer improves not much only 2%. Finally, a larger learning rate of 0.01 is tested, when using 0.01, a very large loss occurs during the training process, seems this value is too large and does not fit the MLP layer and Elman layer. However, the learning speed of neural network with LSTM layer become faster and the performance improved to 90%.

	0.0001	0.001	0.003	0.01
MLP	75%	85%	85%	Nan
Elman	65%	84%	86%	Nan
LSTM	67%	84%	86%	90%

Table 1: Validation accuracy on 3 layers with different learning rate

According to the validation accuracy results above, the learning rate of 0.003 is chosen for both MLP layer and Elman layer, the learning rate of 0.01 is chosen for LSTM layer. Baesd on the validation accuracy and our hypothesis, the neural network with LSTM should have the best performance and then followed by Elman and MLP, and there is no much difference between the performance of Elman and MLP. Table 2 illustrate the final performanc on test data. As our expected, the neural network with LSTM layer has the highest accuracy, Elman and MLP followed and they have the same accuracy.

	MLP	Elman	LSTM
accuracy	86%	86%	87%

Table 2: test accuracy with 3 different layers

The final results prove the hypothesis which is that the LSTM should perform best and followed by Elman and MLP. However, in this experiment, due to the time limitation, we only set the training epoch to 1, which will definitely reduce the final performance of the neural network. In future research, we should increase the number of training epochs to make the neural network fully converge and test more different hyperparameters (learning rate, batch size, etc.) to see how 3 layers influence the performance of the neural network.

5 A code snippet

```
def create_batches(data, batch_size):
    batches = []
    i = 0
    while i < len(data):
        temp = data[i:i+200]
        batches.append(temp)
        i += 200

    return batches
```

Figure 3: Implementation of slicing out batches

```
class Net(nn.Module):
    def __init__(self):
        super(Net,self).__init__()
        self.emb = nn.Embedding(99430,300)
        self.fc1 = nn.Linear(300,300)
        self.fc2 = nn.Linear(300,2)
    def forward(self,x):
        x = self.emb(x)
        x = torch.max(F.relu(self.fc1(x)),1)[0]
        x = self.fc2(x)

        return x

net = Net()
```

Figure 4: Implementation of MLP network

```
class Elman(nn.Module):
    def __init__(self, insize = 300, outsize = 300, hsize = 300):
        super().__init__()
        self.lin1 = nn.Linear(600, 300)
        self.lin2 = nn.Linear(300,300)
    def forward(self, x, hidden = None):
        b, t, e = x.size()
        if hidden == None:
            hidden = torch.zeros(b, e, dtype = torch.float)

        outs = []
        for i in range(t):
            inp = torch.cat([x[:, i, :], hidden], dim = 1)
            out = F.relu(self.lin1(inp))
            out = self.lin2(out)
            outs.append(out[:, None, :])

        return torch.cat(outs, dim = 1), hidden
```

Figure 5: implementation of Elman network

```
class Net(nn.Module):
    def __init__(self):
        super(Net,self).__init__()
        self.emb = nn.Embedding(99430,300)
        self.rnn = nn.RNN(300,300,batch_first = True,nonlinearity = 'relu')
        self.fc2 = nn.Linear(300,2)
    def forward(self,x):
        x = self.emb(x)
        x = torch.max(self.rnn(x)[0],1)[0]
        x = self.fc2(x)

        return x

net = Net()
```

Figure 6: Implementation of torch rnn network

References

https://en.wikipedia.org/wiki/Recurrent_neural_network
Dishashree Gupta, 2017, "Fundamentals of Deep Learning – Introduction to Recurrent Neural Networks"