KAIST

AI502 Deep Learning

Homework 3 - Recurrent Neural Networks

Cappa Victor
20206080
School of computing
victor.cappa@kaist.ac.kr

HW3-Intro

In this homework, we implement and train several types of recurrent neural networks (RNNs), and use them to perform sentiment analysis on the IMDB movie dataset (binary classification problem) and, finally, we implement a model to predict the number of subway passengers on three Korean subway stations (regression problem).

HW3-Task1 - Sentiment analysis on IMDB movie dataset

The RNN input consists of padded input sequences with each element, corresponding to a single word, consisting in the given word embedding. As a result the inputs have a maximum (bounded) length, corresponding to the length of the longest sequence, and consist of one or more embedded words, with the remaining part being padding. As a result we are interested in having variable size inputs and we achieve this in this way:

```
embedded = self.embedding(text)
packed_seq = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths, enforce_sorted=False)
output, hidden = self.rnn(packed_seq)
return self.fc(hidden.squeeze(0))
```

The complete code can be found in the hw3_1(20206080).ipynb file.
With this solution we can achieve a test set accuracy improvement from 61% to 68%, just by applying this trick.
We analyze the performance of different RNN configurations, such as LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit), Stacked RNNs, and Bidirectional RNNs and validate the implemented neural networks architectures on the test set by means of the accuracy metric.
We report the experimental results of the different implementations, including the final train loss, validation loss, train accuracy and validation accuracy and the best test loss and test accuracy found by using the best model found during training and validated on the validation set:

Initial implementation	Train Loss: 0.6935 Train Acc: 50.70% Valid Loss: 0.6932 Valid Acc: 50.87% Test Loss: 0.6612 Test Acc: 61.03%
Fixed implementation	Train Loss: 0.3240 Train Acc: 86.55% Valid Loss: 0.6644 Valid Acc: 70.23% Test Loss: 0.6011 Test Acc: 68.82%
LSTM	Train Loss: 0.1659 Train Acc: 94.16% Valid Loss: 0.4480 Valid Acc: 84.30% Test Loss: 0.4349 Test Acc: 81.56%
GRU	Train Loss: 0.0235

	Train Acc: 99.37% Valid Loss: 0.6229 Valid Acc: 84.98% Test Loss: 0.3433 Test Acc: 85.90%
Stacked layers	Train Loss: 0.2503 Train Acc: 90.49% Valid Loss: 0.5735 Valid Acc: 75.39% Test Loss: 0.5558 Test Acc: 75.02%
Bidirectional	Train Loss: 0.1552 Train Acc: 94.25% Valid Loss: 0.7735 Valid Acc: 73.71% Test Loss: 0.6041 Test Acc: 70.54%
Dropout and stacked layers	Train Loss: 0.3833 Train Acc: 84.38% Valid Loss: 0.6533 Valid Acc: 70.62% Test Loss: 0.6486 Test Acc: 67.01%
Custom configuration	Train Loss: 0.0007 Train Acc: 100.00% Valid Loss: 0.8564 Valid Acc: 86.73% Test Loss: 0.3928 Test Acc: 84.64%

All the experimental results are gathered by keeping constant the former hyperparameter configuration.
As we can see the Fixed implementation allows to achieve a test accuracy of 68.82%, and allows us to drop the final train loss from 0.6935 in the Initial implementation to 0.3240, meaning the model is effectively fitting the training data (it is also shown in the plots on the hw3_1(20206080).ipynb file). The validation and test accuracy are similar, so the model generalizes well from the validation to the test set.
With LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit) we have a further improvement, allowing us to achieve a test accuracy of 81.56% and 85.90%, respectively. Moreover the final train loss is 0.1659 for LSTM and 0.0235 for GRU, meaning they are further minimizing the training loss compared to the former methods.
The Stacked layers implementation consists of an RNN with 3 stacked layers and achieves a test accuracy of 75.02% and final training loss of 0.2503. One of the reasons why this RNN version is not as performant as other methods such as LSTM and GRU is the vanishing gradient problem, due to the length of the resulting neural network structure.
Bidirectional achieves a final test performance of 70.54% and final training loss of 0.1552. With the final training loss being much lower compared to the Stacked one (0.2503).

Dropout with stacked layers consists in 3 stacked layers and a dropout of 0.5. It achieves a test accuracy of 67.01%, meaning the introduced dropout of 0.5 doesn't improve the overall performances but instead we incur in a drop in both test accuracy and training loss.
Finally, in the custom configuration we implemented a RNN architecture consisting in a bidirectional GRU (the best performing model so far) trained with the default hyperparameters, with the exception of a learning rate of 0.0005, a hidden size of 256 and 20 epochs. The resulting model finally overfits the training data, meaning we probably need to use some regularization techniques and, with the best model, we achieve a final test accuracy of 84.64%.

HW3-Task2 - Predict number of passengers in Korean subway

In the given task we try to learn a model able to predict the number of passengers in a Korean subway after one hour, specifically in Seoul Station, Gangnam Station, and Yeouido Station, in a time frame going from 05-06 am to 00-01 am of the next day.
Our model of choice is an LSTM that given a specific input encoding (containing several informations such as the number of passengers in the previous hour, the station one-hot encoding, day-of-week one-hot encoding, and timestamp one-hot encoding) is able to predict the number of passengers that will enter the subway in the next hour.
This is the implemented RNN model:

```
class LSTM(nn.Module):
    def __init__(self):
        super(LSTM, self).__init__()
        self.num_classes = NUM_CLASS
        self.num_layers = NUM_LAYERS
        self.input_size = INPUT_SIZE
        self.hidden_size = HIDDEN_SIZE
        self.sequence_length = SEQUENCE_LENGTH
        self.batch_size = BATCH_SIZE
        self.lstm=nn.LSTM(input_size=self.input_size, hidden_size=self.hidden_size,
num_layers=self.num_layers, batch_first=True)
        self.fc = nn.Linear(self.hidden_size, self.num_classes)

    def forward(self, x):
        h_0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).cuda(GPU_DEVICE)
        c_0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).cuda(GPU_DEVICE)
        ula, (h_out, c_out) = self.lstm(x, (h_0, c_0))
        fc_input = torch.cat([ula[i] for i in range(ula.size(0))])
        out = self.fc(fc_input)
        return out.view(x.size(0), self.sequence_length)
```

Moreover the function to calculate the full-batch RMSE of the test set have been implemented in the following way:

```
def measureRMSE(model, valid_loader):
    output_list = list()
    target_list = list()
    model.eval()
    for i, data in enumerate(valid_loader):
        input_x = data[0].type(torch.FloatTensor).cuda(GPU_DEVICE)
        target = data[1].type(torch.FloatTensor).cuda(GPU_DEVICE)
```

```
        output = model(input_x)
        output_list.append(output)
        target_list.append(target)
    full_batch_output = torch.cat(output_list)
    full_batch_target = torch.cat(target_list)
    full_batch_loss = RMSELoss(full_batch_output, full_batch_target)
    return full_batch_loss
```

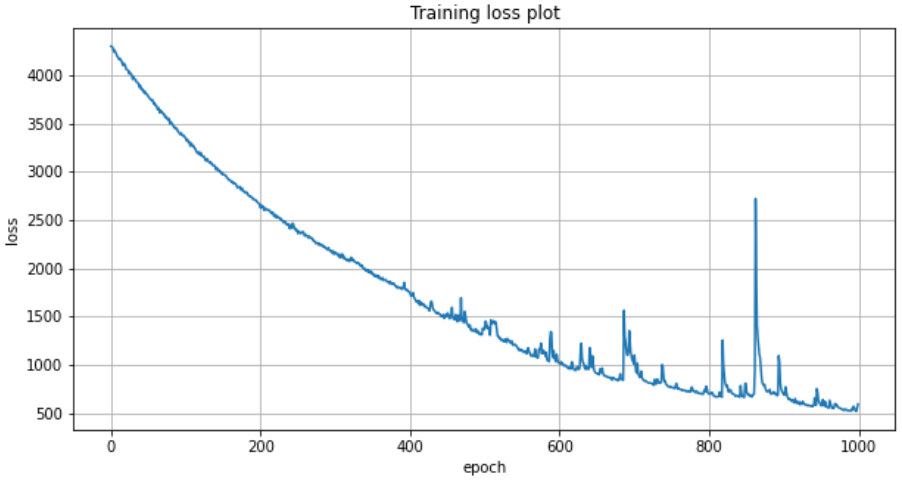
This function allows us to calculate the RMSE over the whole input dataset and returns a tensor with one element representing the resulting RMSE.
More informations about the implementation can be found on the hw3_(20206080).ipynb file.

1.Task2-Step1

In the first part we have a limited version of the input encoding only containing the previous number of passengers and the station one-hot encoding.
The resulting full-batch RMSE is:

RMSE on the test dataset: 653.3561401367188

As we can see it is pretty high, meaning that some hyperparameter optimization and some modifications to the LSTM model, included the input encoding are probably necessary in order to achieve better regression performances.
The resulting training loss have been plotted to better analyze the model's behaviour while training:



As we can see the training loss decreases along the epochs, starting from an initial training loss greater than 4000 to a final one close to 600.

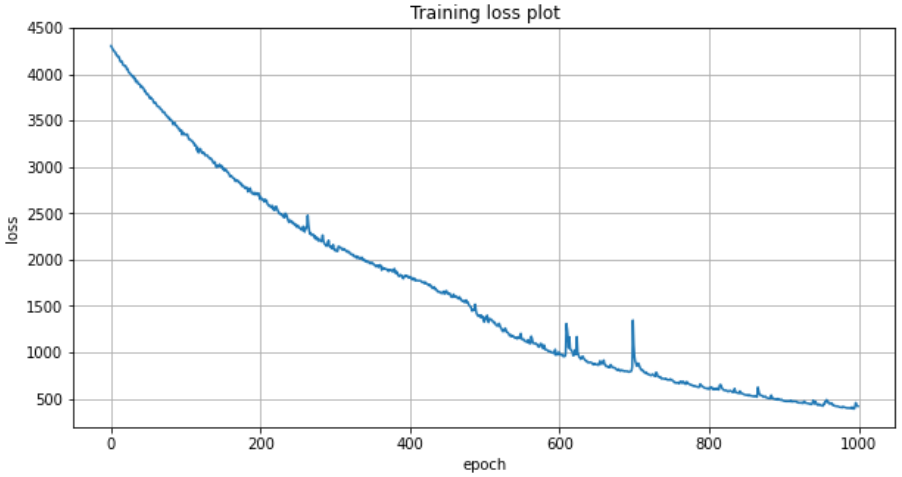
2.Task2-Step1

In the second part we use the complete input encoding we discussed in the previous section, including also the one-hot encoding of the timestamp and the day of the week.
The resulting full-batch RMS calculated on the test dataset is:

RMSE on the test dataset: 483.64715576171875

Therefore, including these additional informations in the input encoding allows us to achieve better test-set performances, thus enhancing our prediction capabilities.

This major improvement is due to the fact that the model is now able to discriminate whether it has to perform a prediction in a timestamp compared to another and which is the day of the week, and both of these are valuable information for our model.
The resulting training loss curve is shown in the following plot:



As in the previous scenario we start from an initial RMSE greater than 4000 and we achieve a final RMSE close to 400.