Task 1: MLP-based Encoder

1-1. The number of (hidden) layers

All conditions, other than the number of layers, kept the same as below.
- activation function: ReLU
- loss function: MSELoss()
- optimizer: Adam
- learning rate: 0.001
- n_epochs: 3

Number of Layers	Training Time (s)	Training Loss	MSE on Dev
1 (sample code)	137.07	0.0835626	0.09834197
2	269.96	0.08144311	0.088572696
3	406.20	0.09853185	0.11505561

As there is no noticeable improvement as the number of layers increased, I tried different optimizers on the model with 3 hidden layers.

1-2. Optimizer

All conditions, other than optimizer, kept the same as below.
- activation function: ReLU
- loss function: MSELoss()
- learning rate: 0.001
- n_epochs: 3
- number of layers: 3

Optimizer	Training Time (s)	Training Loss	MSE on Dev
Adam	406.20	0.09853185	0.11505561
RMSprop	492.86	0.091378376	0.09343084
SGD	186.90	0.085480385	0.09224715
Adagrad	181.27	0.06121099	0.067972414

In terms of training time, training loss, and MSE on dev data set, Adagrad optimizer shows the best performance. With Adagrad optimizer, I checked the performance of a more complex model with more hidden layers.

1-3. The number of (hidden) layers with Adagrad optimizer

All conditions, other than the number of layers, kept the same as below.
- activation function: ReLU
- loss function: MSELoss()
- n_epochs: 3
- learning rate: 0.001
- optimizer: Adagrad

Number of Layers	Training Time (s)	Training Loss	MSE on Dev
1 (sample code)	86.24	0.065490015	0.06491693
2	159.52	0.060033634	0.06184091
3	181.27	0.06121099	0.067972414
4	239.51	0.06775066	0.06881372
5	303.82	0.079901665	0.085808575

From the results, I could see that just adding additional hidden layers does not improve the model performance on the dev set. I would keep 3 layers in the following tasks.

1-4. The dimension of each hidden layer

So far, I set the dimension of hidden layer as (embd_dim, embd_dim) for all layers.
To see how the dimension of each hidden layer affects the performance, I tried a few varied cases below.

1) hidden_layer_1 = (embd_dim, embd_dim*2)
   hidden_layer_2 = (embd_dim*2, embd_dim*2)
   hidden_layer_3 = (embd_dim*2, embd_dim)

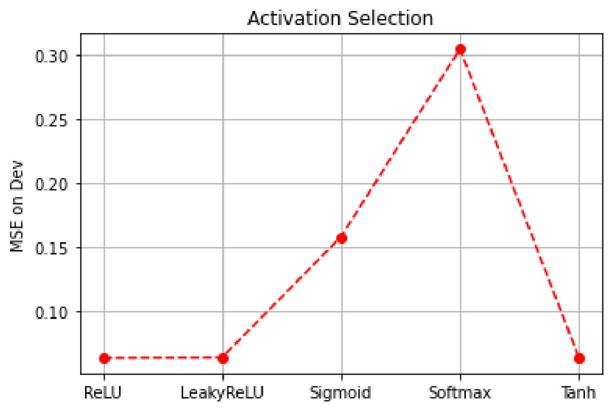
2) hidden_layer_1 = (embd_dim, embd_dim*2)
   hidden_layer_2 = (embd_dim*2, embd_dim)
   hidden_layer_3 = (embd_dim, embd_dim)

Case	Training Time (s)	Training Loss	MSE on Dev
previous result (embd_dim, embd_dim) for all hidden layers	181.27	0.06121099	0.067972414
1	384.82	0.05832457	0.06472107
2	263.98	0.061120078	0.06538708

Especially, by increasing the number of neurons in each hidden layer (case 1), I could see that the model performs better as the model gets complex it fits the training data better. However, in this assignment, for simplicity and in terms of computational expense, I would keep the dimension of all hidden layers as (embd_dim, embd_dim).

1-5. Activation function

I tried 5 different activation functions to compare the performance of each model. Activation functions that I tested were ReLU, LeakyReLU with a negative slope of 0.01, Sigmoid, Softmax, and Tanh.



Activation	MSE on Dev
ReLU	0.06392178
LeakyReLU	0.06425852
Sigmoid	0.15802404
Softmax	0.30420178
Tanh	0.06408245

ReLU, LeakyReLU, and Tanh activation functions showed similar and fairly good performance, on the other hand, Sigmoid and Softmax showed pretty poor performance. I will use ReLU as an activation function throughout this assignment.

1-6. Learning rate

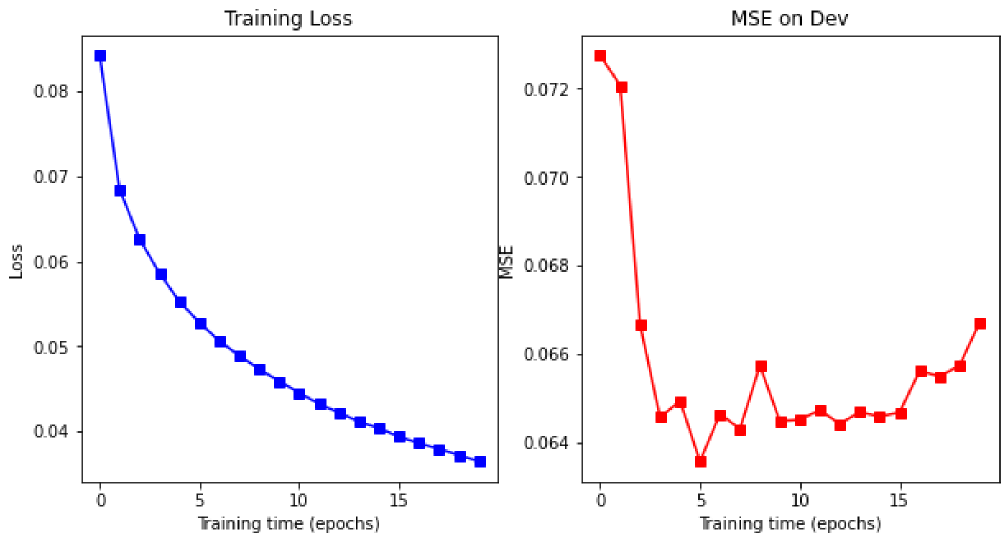
Also, tuned the learning rate parameter by comparing the performance of the model with different learning rates.

Learning Rate	Training Time (s)	Training Loss	MSE on Dev
0.0001	193.18	0.0760545	0.071361236
0.001	190.01	0.061967712	0.065630354
0.01	204.97	0.08404525	0.09379254

From the result, I fixed the learning rate as 0.001.

1-7. The number of epochs

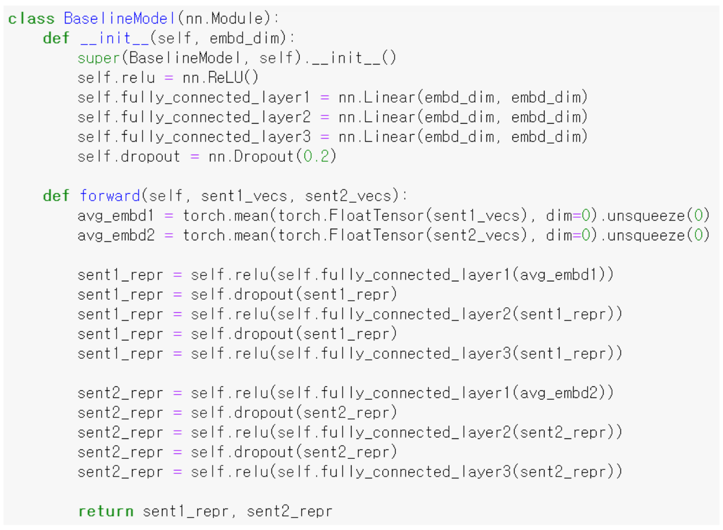
- activation function: ReLU
- loss function: MSELoss()
- learning rate: 0.001
- optimizer: Adagrad
- number of layers: 3



I observed training loss and MSE on the dev set over 20 epochs. As we can see in the graph, training loss is continuously decreased and near-zero (0.03646634) at the 20th epoch, whilst MSE on dev on the dev set is decreased up to the 5th epoch then started to increase. These results can be seen as a sign of overfitting. From these results, n_epoch = 6 seems an ideal option for the model, however, I applied regularisation to see if it helps the model performs better because it prevents the model from being overfitted to the training set.

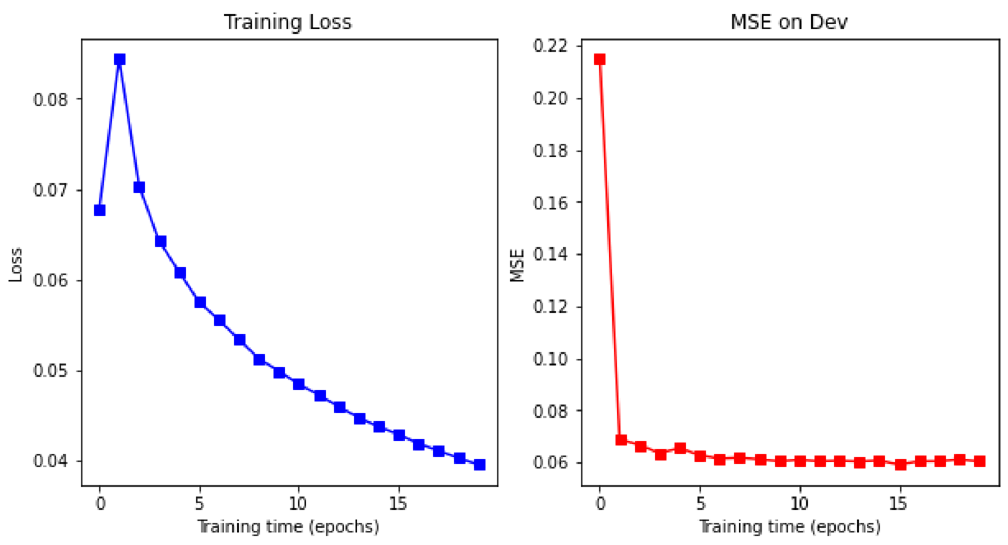
1-8. Regularisation (Dropout)

Used Model code



```
class BaselineModel(nn.Module):
    def __init__(self, embd_dim):
        super(BaselineModel, self).__init__()
        self.relu = nn.ReLU()
        self.fully_connected_layer1 = nn.Linear(embd_dim, embd_dim)
        self.fully_connected_layer2 = nn.Linear(embd_dim, embd_dim)
        self.fully_connected_layer3 = nn.Linear(embd_dim, embd_dim)
        self.dropout = nn.Dropout(0.2)

    def forward(self, sent1_vecs, sent2_vecs):
        avg_embd1 = torch.mean(torch.FloatTensor(sent1_vecs), dim=0).unsqueeze(0)
        avg_embd2 = torch.mean(torch.FloatTensor(sent2_vecs), dim=0).unsqueeze(0)

        sent1_repr = self.relu(self.fully_connected_layer1(avg_embd1))
        sent1_repr = self.dropout(sent1_repr)
        sent1_repr = self.relu(self.fully_connected_layer2(sent1_repr))
        sent1_repr = self.dropout(sent1_repr)
        sent1_repr = self.relu(self.fully_connected_layer3(sent1_repr))

        sent2_repr = self.relu(self.fully_connected_layer1(avg_embd2))
        sent2_repr = self.dropout(sent2_repr)
        sent2_repr = self.relu(self.fully_connected_layer2(sent2_repr))
        sent2_repr = self.dropout(sent2_repr)
        sent2_repr = self.relu(self.fully_connected_layer3(sent2_repr))

        return sent1_repr, sent2_repr
```

From the results, I could see that the overfitting issue was solved by using dropout regularisation. However, when I tested on the dev set once fully trained the model using dropout, I found out that it gives poor performance (0.15342864 of MSE) compared to the one that did not use the dropout regularisation. So, I decided not to use dropout and set the n_epochs as 6 from the result of 1-6.



1-9. Conclusion

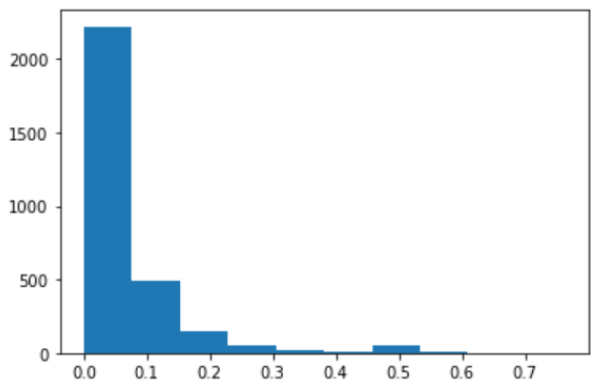
- Best model specification

activation function	ReLU
loss function	MSELoss()
optimizer	Adagrad
number of hidden layers	3
learning rate	0.001
n_epoch	6
dropout	No

With my best model, I achieved 0.061412487 of MSE on the dev set and the distribution of the squared errors is like below.

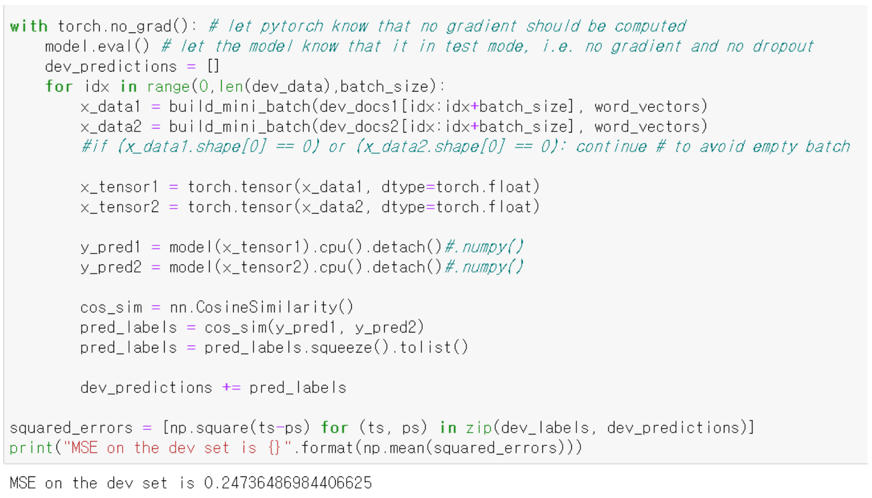
```
[205]: evaluate_trained_model(model, dev_data)
```

```
3000/? [00:03<00:00, 956.71it/s]

MSE of the method on the dev set: 0.061412487
```



Task 2: CNN-based Encoder

2-1. Before training the CNN model, I checked its performance on the dev set



```
with torch.no_grad(): # let pytorch know that no gradient should be computed
    model.eval() # let the model know that it in test mode, i.e. no gradient and no dropout
    dev_predictions = []
    for idx in range(0,len(dev_data),batch_size):
        x_data1 = build_mini_batch(dev_docs1[idx:idx+batch_size], word_vectors)
        x_data2 = build_mini_batch(dev_docs2[idx:idx+batch_size], word_vectors)
        #if (x_data1.shape[0] == 0) or (x_data2.shape[0] == 0): continue # to avoid empty batch

        x_tensor1 = torch.tensor(x_data1, dtype=torch.float)
        x_tensor2 = torch.tensor(x_data2, dtype=torch.float)

        y_pred1 = model(x_tensor1).cpu().detach()#.numpy()
        y_pred2 = model(x_tensor2).cpu().detach()#.numpy()

        cos_sim = nn.CosineSimilarity()
        pred_labels = cos_sim(y_pred1, y_pred2)
        pred_labels = pred_labels.squeeze().tolist()

        dev_predictions += pred_labels

squared_errors = [np.square(ts-ps) for (ts, ps) in zip(dev_labels, dev_predictions)]
print("MSE on the dev set is {}".format(np.mean(squared_errors)))
```

```
MSE on the dev set is 0.24736486984406625
```

Before training the CNN model, the MSE on the dev set is 0.24736486984406625. Detailed specification of the model is below.

- dropout_rate: 0.5
- filter_size: [2,3,4]
- filter_nums: [100]*len(filter_size)
- loss function: MSELoss()
- optimizer: Adam
- n_epochs: 10
- batch size: 50
- learning rate: 0.001

2-2. Learning rate

From the 2-1 model, I tried different learning rates to see how it affects the model performance. All conditions, other than the learning rate, kept the same.

Learning Rate	Training Loss	MSE on Dev
0.001	0.02898254	0.10902505755241698
0.01	0.02885877	0.0939330945127652
0.1	0.031280663	0.10028266030052005

From the result, I picked 0.01 as the value for the learning rate.

2-3. Mini-batch size

From the 2-2 model, I tried different mini-batch sizes to see how it affects the model performance. All conditions, other than the mini-batch size, kept the same.

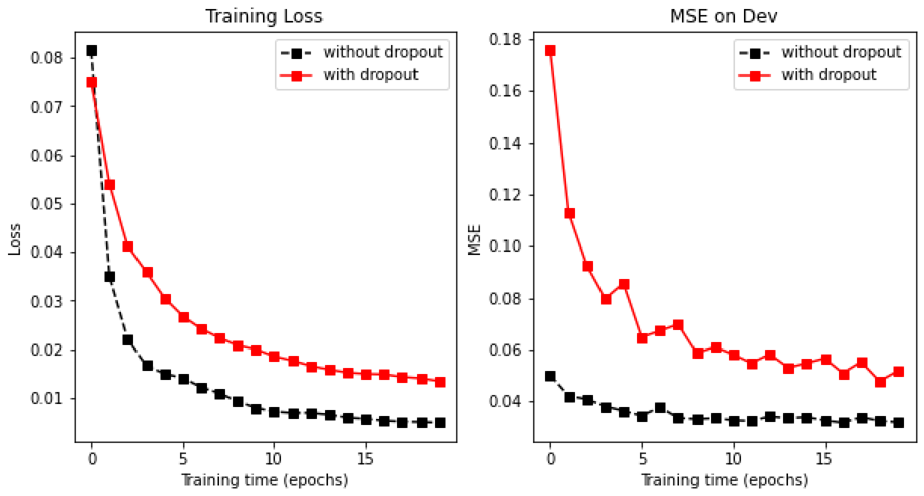
Mini-batch Size	Training Loss	MSE on Dev
20	0.034156214	0.12448833687814734
30	0.023302967	0.09083818506732848
50	0.018541062	0.08769613851085656
100	0.015206237	0.08576138282154218

From the result, I picked 50 as the value for the mini-batch size.

2-4. Dropout & number of epochs

From the 2-3 model, I tried different dropout rates to see how it affects the model performance. All conditions, other than the dropout rate, kept the same.

Dropout	Training Loss	MSE on Dev
0	0.008350984	0.03671873313092006
0.25	0.018832223	0.060639979800098956
0.5	0.033368997	0.11582394307821824
0.75	0.05615747	0.23904349497804356
1	0.27303126	0.27751311774078274



From the results, we can clearly see that the model shows poor performances when the dropout method was used. Also, by extending the number of epochs up to 20, no signs of overfitting are observed. Therefore, I would not apply the dropout method to my model and pick 17 as the value for the number of epochs because it achieved the lowest MSE on the dev set, 0.03168047401006368.

2-5. Filter sizes

I compared the model performance following the filter sizes. I fixed the number itself as 2, 3, 4, because I would not use more complex n-gram model than 4-gram. Instead, I made 3 different cases like below.

- [Case1] Fixed 1 size: [2], [3]. [4]
- [Case2] Contains 2 different sizes: [2,3], [2,4], [3,4]
- [Case3] Contains 3 different sizes: [2,3,4]

Filter sizes	Training Loss	MSE on Dev
[2]	0.009258554	0.03914324268903173
[3]	0.008376626	0.04011358781231733
[4]	0.008305512	0.03931700371052161
[2,3]	0.0062036193	0.03177025517449994
[2,4]	0.0052479743	0.03195301126581918
[3,4]	0.0047940915	0.033507742480541176
[2,3,4]	0.005086963	0.03105103419371978

There was no significant difference in MSE on the dev set between case 2 and case 3, however, case 3 showed a better performance. Also, I could see that if the filter size is fixed to one value, the model showed poor performance compared to the other two cases.

2-6. Conclusion

- Best model specification

loss function	MSELoss()
optimizer	Adam
learning rate	0.01
n_epoch	17
mini-batch size	50
dropout	No
filter size	[2,3,4]

With my best model, I achieved 0.03424460890371057 of MSE on the dev set.