1.

A.

My deep averaging network consisted of the following layers:

a. Embedding Layer
b. Averaging Layer
c. Dropout Layer (probability = 0.5%)
d. Linear Layer (embed_dim=50 -> 128)
e. ReLU Non-Linearity
f. Linear Layer (128 -> 64)
g. ReLU Non-Linearity
h. Linear Layer(64 -> 2)
i. Softmax Layer

I trained using AdamW as an optimizer, with a learning rate of 0.003, and weight decay of 1e-5, with a Cosine Annealed learning rate scheduler, annealed across one entire epoch, and a CrossEntropyLoss function. This along with an embedding size of 50, and not using the pretrained weights for the embedding layer, and using the default Pytorch layer initialization (kaiming initialization) I was able to get 77% accuracy on the dev set after 100 training epochs, which since I trained with batching (batch size 128) which I explain more below, I was able to train in ~20 seconds.

Results:

=====Train Accuracy=====

Accuracy: 6785 / 6920 = 0.980491;

Precision (fraction of predicted positives that are correct): 3528 / 3581 = 0.985200;

Recall (fraction of true positives predicted correctly): 3528 / 3610 = 0.977285;

F1 (harmonic mean of precision and recall): 0.981227;

=====Dev Accuracy=====
Accuracy: 678 / 872 = 0.777523;
Precision (fraction of predicted positives that are correct): 350 / 450 = 0.777778;
Recall (fraction of true positives predicted correctly): 350 / 444 = 0.788288;
F1 (harmonic mean of precision and recall): 0.782998;

Time for training and evaluation: 15.99 seconds
=====Results=====

```
{
  "dev_acc": 0.7775229357798165,
  "dev_f1": 0.7829977628635346,
  "execution_time": 15.986402750015259,
  "output": "Accuracy: 678 / 872 = 0.777523;\nPrecision (fraction of predicted positives that are correct): 350 / 450 = 0.777778;\nRecall (fraction of true positives predicted correctly): 350 / 444 = 0.788288;\nF1 (harmonic mean of precision and recall): 0.782998;\n"
}
```


Using an embedding size of 300, and changing the hidden layer sizes to roughly ~

a. Linear Layer (embed_dim=50 -> 512)
b. Linear Layer (512 -> 256)
c. Linear Layer (256 -> 2)

I was again still able to achieve 77% accuracy on the dev set without changing any other hyperparameters.
Results:

```
=====Train Accuracy=====
Accuracy: 6897 / 6920 = 0.996676;
Precision (fraction of predicted positives that are
correct): 3595 / 3603 = 0.997780;
Recall (fraction of true positives predicted
correctly): 3595 / 3610 = 0.995845;
F1 (harmonic mean of precision and recall): 0.996811;

=====Dev Accuracy=====
Accuracy: 673 / 872 = 0.771789;
Precision (fraction of predicted positives that are
correct): 347 / 449 = 0.772829;
Recall (fraction of true positives predicted
correctly): 347 / 444 = 0.781532;
F1 (harmonic mean of precision and recall): 0.777156;

Time for training and evaluation: 17.35 seconds
=====Results=====
{
  "dev_acc": 0.7717889908256881,
  "dev_f1": 0.7771556550951848,
  "execution_time": 17.347595691680908,
  "output": "Accuracy: 673 / 872 = 0.771789;\nPrecision
(fraction of predicted positives that are correct): 347
/ 449 = 0.772829;\nRecall (fraction of true positives
predicted correctly): 347 / 444 = 0.781532;\nF1
(harmonic mean of precision and recall): 0.777156;\n"
}
In order to achieve the same results in less epochs I
upped the learning rate to 0.058 and was able to
```

achieve similar results in just 15 epochs, using the embedding vectors of size 50. Results:

```
=====Train Accuracy=====
Accuracy: 6834 / 6920 = 0.987572;
Precision (fraction of predicted positives that are correct): 3562 / 3600 = 0.989444;
Recall (fraction of true positives predicted correctly): 3562 / 3610 = 0.986704;
F1 (harmonic mean of precision and recall): 0.988072;

=====Dev Accuracy=====
Accuracy: 675 / 872 = 0.774083;
Precision (fraction of predicted positives that are correct): 351 / 455 = 0.771429;
Recall (fraction of true positives predicted correctly): 351 / 444 = 0.790541;
F1 (harmonic mean of precision and recall): 0.780868;

Time for training and evaluation: 6.22 seconds
=====Results=====
{
  "dev_acc": 0.7740825688073395,
  "dev_f1": 0.7808676307007786,
  "execution_time": 6.22254490852356,
  "output": "Accuracy: 675 / 872 = 0.774083;\nPrecision (fraction of predicted positives that are correct): 351 / 455 = 0.771429;\nRecall (fraction of true positives predicted correctly): 351 / 444 = 0.790541;\nF1 (harmonic mean of precision and recall): 0.780868;\n"
}
```

Lastly I also experimented with other non-linearities (GELU, Tanh, PReLU), but both GELU and PReLU didn't show significant changes compared to ReLU, and Tanh was significantly worse than all of them, especially when using the 300-size embedding vectors. I also experimented with changing where the Dropout layer (and varying the probability from 0.25%-0.50%) was in the network but found that right after the embedding+averaging layer was the best location to better generalize the network to new data.

B.

In order to implement batching with ease I made a Dataset class and inherited from Pytorchs Dataset class in order to use their DataLoader. But to achieve this I needed to pad each sentence such that every sentence in a single batch was the same length. To do this, for every batch I pad each sentence (with 0's) so that they're the same length as the longest sentence in the batch, I do this on the fly when the batch is made every iteration, using a pad_batch() function, which is given to the DataLoader via the collat_fn parameter. Thus when I was done I now had a DataLoader object that I could iterate through and get a batches worth of training examples and their corresponding labels, and each sentence was padded on the fly per the longest sentence in each batch. This improved my training time and accuracy immensely.
Using the size 50 embedding layer without batching took ~130 seconds to train for 15 epochs achieving 50% accuracy on the dev set, with batching (batch size 128) it now takes ~6-7 seconds and achieves 77%. The

accuracy improvement comes from improving the approximation of stochastic gradient descent (technically AdamW in this case but ~same idea) to gradient descent, essentially the bigger the batch the closer SGD gets to being gradient descent (where in GD the batch size = training set size).

C.

With the GloVe initialization, the performance of my model either stays the same or slightly decreases. This is mostly from a combination of the small learning rate I am using along with the Annealed Cosine LR scheduler, which doesn’t give the rest of the network's weights after the embedding layer, enough room to change enough to improve the performance, with a higher learning rate and removing the scheduler, the Glove initializations improve the accuracy to 78%, but this takes a decent amount of tweaking the hyperparameters to achieve. Thus I found it easier to tweak the network by training the embedding layer rather than using the pretrained weights. I suspect using the pretrained weights in addition to allowing the embedding layer to learn (i.e. not turning off requires grad) would achieve the same if not better the results than training them from scratch

2.

Results from Viterbi algorithm:
Total time to tag the development set: 4 seconds
Accuracy: 37853 / 39995 = 0.946443

3.

Results from beam size 1:
Total time to tag the development set: 2 seconds
Accuracy: 37376 / 39995 = 0.934517

Results from beam size 3:
Total time to tag the development set: 3 seconds
Accuracy: 37770 / 39995 = 0.944368

Results from beam size 5:
Total time to tag the development set: 5 seconds
Accuracy: 37755 / 39995 = 0.943993

Result from beam size 50:
Total time to tag the development set: 24 seconds
Accuracy: 37441 / 39995 = 0.936142

In the same way that SGD is an approximation of GD, using a particular beam size affects the approximation of this algorithm to that of the Viterbi algorithm. Essentially with a big enough beam size you consider as many options as the Viterbi algorithm does, however, for most sentences, most of those tags you consider are almost never chosen, such as the period tag which is only used for periods, thus by limiting the size to a smaller beam, you can still achieve an accurate enough algorithm while taking the algorithm to being just O(n * T * k) where n is the sentence length, T is the number of tags, and k is the beam size. Essentially when k = T, you have the Viterbi algorithm. In summary decreasing beam size typically correlates with a lower accuracy (although not always and for most sentences

achieves similar accuracy to Viterbi) while decreasing the runtime. For this example a beam size of 1-5 runs faster than the Viterbi algorithm and achieves similar accuracy from ~93-94%