

Assignment 2

Understanding Attentions (20%)

1.2 Selection via Attention

(1.2.1)

Assume $q = [x_0 \ x_1 \ x_2]$ and we have large scalar $S = 1e5$.

We have $A = qK^\top = \begin{bmatrix} x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2} \\ x_0 k_{1,0} + x_1 k_{1,1} + x_2 k_{1,2} \\ x_0 k_{2,0} + x_1 k_{2,1} + x_2 k_{2,2} \\ x_0 k_{3,0} + x_1 k_{3,1} + x_2 k_{3,2} \end{bmatrix}^\top$,

$A' = \text{softmax}(A) = \begin{bmatrix} \frac{\exp(x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} \\ \frac{\exp(x_0 k_{1,0} + x_1 k_{1,1} + x_2 k_{1,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} \\ \frac{\exp(x_0 k_{2,0} + x_1 k_{2,1} + x_2 k_{2,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} \\ \frac{\exp(x_0 k_{3,0} + x_1 k_{3,1} + x_2 k_{3,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} \end{bmatrix}^\top$,

and finally $O = A'V =$

$$\begin{bmatrix} \frac{\exp(x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{0,0} + \frac{\exp(x_0 k_{1,0} + x_1 k_{1,1} + x_2 k_{1,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{1,0} \\ + \frac{\exp(x_0 k_{2,0} + x_1 k_{2,1} + x_2 k_{2,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{2,0} + \frac{\exp(x_0 k_{3,0} + x_1 k_{3,1} + x_2 k_{3,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{3,0} \\ \frac{\exp(x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{0,1} + \frac{\exp(x_0 k_{1,0} + x_1 k_{1,1} + x_2 k_{1,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{1,1} \\ + \frac{\exp(x_0 k_{2,0} + x_1 k_{2,1} + x_2 k_{2,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{2,1} + \frac{\exp(x_0 k_{3,0} + x_1 k_{3,1} + x_2 k_{3,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{3,1} \\ \frac{\exp(x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{0,2} + \frac{\exp(x_0 k_{1,0} + x_1 k_{1,1} + x_2 k_{1,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{1,2} \\ + \frac{\exp(x_0 k_{2,0} + x_1 k_{2,1} + x_2 k_{2,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{2,2} + \frac{\exp(x_0 k_{3,0} + x_1 k_{3,1} + x_2 k_{3,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{3,2} \end{bmatrix}^\top$$

We can see that to get $O = [v_{0,0} \ v_{0,1} \ v_{0,2}]$, we need the first term of A' ($\frac{\exp(x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})}$)

to be significantly larger than the other terms, so the results of $\frac{\exp(x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2})}{\sum_{j=0}^3 \exp(x_0 k_{j,0} + x_1 k_{j,1} + x_2 k_{j,2})} v_{0,i}$ dominate the sum in O .

Thus, $\exp(x_0 k_{0,0} + x_1 k_{0,1} + x_2 k_{0,2})$ needs to be significantly larger than the other terms in A , to achieve this we need a q that scales the first term of A to be larger than the other terms. This means that q needs to be a vector that align with k_0 and has a large magnitude, then the exponential function will do the amplification.

Therefore, the query vector should be $q = S \cdot k_0 = [S \cdot k_{0,0} \ S \cdot k_{0,1} \ S \cdot k_{0,2}] = [0.47S \ 0.65S \ 0.60S]$.

By setting a query vector to a specific scaled key vector, we are essentially focusing the model's attention to the corresponding value vectors.

(1.2.2)

Similar idea to 1.2.1, but instead of the first element of A' dominating the sum in O , we want the diagonal elements of A' to dominate the sum of each row of O . From 1.2.1, we know that we can achieve this by scaling each key vector by a large scalar to focus the model's attention to each of the value vectors.

Therefore, we have:

$$Q = S \cdot K = \begin{bmatrix} S \cdot k_0 \\ S \cdot k_1 \\ S \cdot k_2 \\ S \cdot k_3 \end{bmatrix} = \begin{bmatrix} S \cdot k_{0,0} & S \cdot k_{0,1} & S \cdot k_{0,2} \\ S \cdot k_{1,0} & S \cdot k_{1,1} & S \cdot k_{1,2} \\ S \cdot k_{2,0} & S \cdot k_{2,1} & S \cdot k_{2,2} \\ S \cdot k_{3,0} & S \cdot k_{3,1} & S \cdot k_{3,2} \end{bmatrix} = \begin{bmatrix} 0.47S & 0.65S & 0.60S \\ 0.64S & 0.50S & -0.59S \\ -0.03S & -0.48S & -0.88S \\ 0.43S & -0.83S & 0.35S \end{bmatrix}$$

(1.2.3)

It allows language models to highlight the important parts of a sequence, so the model can directly reference specific details from the input into the output, which might improve accuracy, and appropriateness in certain contexts. It might also give better continuity in longer outputs.

1.3 Averaging via Attention

(1.3.1)

Referring to the calculation in 1.2.1, we can see that if we want to average all the value vectors, we need each element of A' to be roughly equal. This means we need $x_0k_{0,0} + x_1k_{0,1} + x_2k_{0,2}$ to be roughly equal to $x_0k_{1,0} + x_1k_{1,1} + x_2k_{1,2}$ and so on.

We see a simple way to achieve this is to set each element in q to be the mean of each corresponding column in K and we can also scale it down to further to reduce the variance. So we can have $\mu_{k,0}k_{0,0} + \mu_{k,1}k_{0,1} + \mu_{k,2}k_{0,2}$ be roughly equal to $\mu_{k,0}k_{1,0} + \mu_{k,1}k_{1,1} + \mu_{k,2}k_{1,2}$ and so on.

Therefore, we have:

$$q = \frac{k_1 + k_2 + k_3 + k_4}{4} = \begin{bmatrix} \frac{0.47+0.64-0.03+0.43}{4} & \frac{0.65+0.50-0.48-0.83}{4} & \frac{0.60-0.59-0.88+0.35}{4} \end{bmatrix}.$$

(1.3.2)

Similar to 1.3.1, but instead of taking the mean of columns of all rows (key vectors) in K , we only take the mean of the first two rows (key vectors) in K . Depending on the result, we could scale up the query vector to slightly increase the attention on the first two value vectors.

Therefore, we have:

$$q = \frac{k_1 + k_2}{2} = \begin{bmatrix} \frac{0.47+0.64}{2} & \frac{0.65+0.50}{2} & \frac{0.60-0.59}{2} \end{bmatrix}.$$

(1.3.3)

It gives the model ability to condense different information from a sequence, which allows the model to better capture main ideas of a text. It might be good for summarization tasks.

2 Building Your Own Mini Transformer (40%)

2.2 Experiment with Your Implementation of Attention

2.2.1 Experiment 1: Dot-Product Attention v.s. Additive Attention

(2.2.1.1 Setup)

Attention mechanisms allow models to dynamically focus on different parts of the input data when producing an output. This experiment compares two attention mechanisms, dot-product attention and additive attention, and try to understand how different attention mechanisms affect the performance in the context of a mini-GPT model.

Both models has the same architecture except for how they compute the attention scores:

Dot-product attention computes the attention scores based on the dot product of query and key vectors and then scaled down with a scale factor. Assume we have query matrix Q , key matrix K , embedding dimensionality n_{embd} , and number of heads n_{head} , then the attention score matrix $A_{\text{dot-product}}$, is computed as follows:

$$A_{\text{dot-product}} = \frac{QK^{\top}}{\sqrt{n_{embd}/n_{head}}}$$

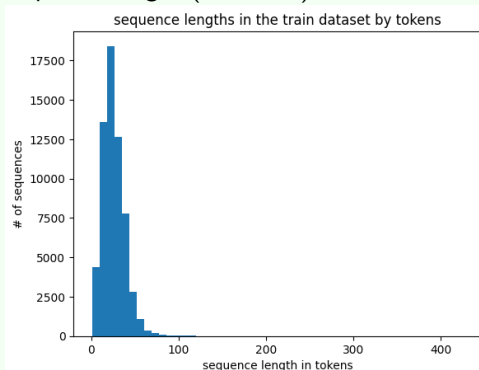
Additive attention computes attention scores by adding the query and key vectors followed by a nonlinear tanh transformation. Here, the attention score matrix A is computed as follows:

$$A_{\text{additive}} = W_2(\tanh(W_1([Q; K])))$$

where W_1 and W_2 are learned weight matrices.

Both models are trained on the same dataset which has the following properties:

- total training tokens: 1,561,375
- # of unique tokens: 80,663
- sequence length (in tokens) distribution :



(2.2.1.1 contd.)

Since a majority of tokens are below 100 tokens, we truncate to 100 tokens for performance reasons. Both models has 1.41 million parameters, and are trained for 1 epoch with a batch size of 32. The learning rate is set to $5e-4$.

My hypothesis is that dot-product attention will compute faster given its simplicity, while additive attention may give a slight improvement performance, since its nonlinearity gives it the ability to model more complex relationships.

(2.2.1.2 Results)

Table 1: Training Time and Loss Comparison

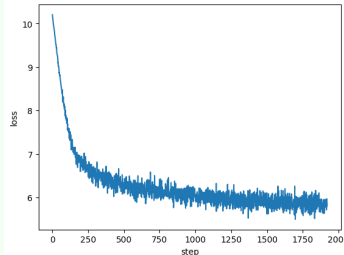
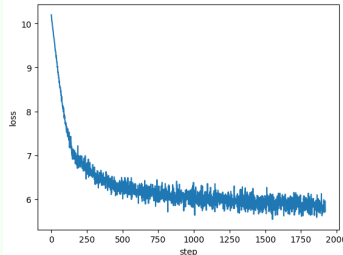
	Dot-Product	Additive Attention
Training Time	3:35	20:29
Final Training Loss	5.85	5.72
Traing Step vs. Loss		

Table 2: Testing Loss Comparison

#1: "After learning language models model natural language."

#2: "The quick brown fox jumps over the lazy dog."

#3: "Amidst the twilight, the ancient ruins whispered secrets long forgotten."

#4: "Quantum entanglement fascinates physicists and philosophers alike, bridging worlds with invisible threads."

	Dot-Product	Additive Attention
Sentence#1	9.459	9.114
Sentence#2	6.742	6.478
Sentence#3	8.199	7.895
Sentence#4	6.355	6.327

Table 3: Perplexity Comparison

	Dot-Product	Additive Attention
Sentence#1	12820.091	9083.743
Sentence#2	847.358	650.523
Sentence#3	3636.979	2684.190
Sentence#4	575.407	559.330
Train Perplexity	401.043	405.255
Dev Perplexity	380.805	385.998

(2.2.1.2 contd.)

Sample Outputs Comparison:

Dot-Product Attention:

1. <START> A age on evidence care nothing , United States are up the people , which is by the attacks for a barrel to big spending on this media when confirm the pledged , Price , low sea after the bring her Life , but she works in Steven Sox media , however . became a presence and then them attempt in December with an discovered of West organization days said , 2009 . <STOP>
2. <START> And over the House will keep Gordon tour by the department of all its being Big taxable many months were <UNK> , on its Radio feed in the new building . <STOP>
3. <START> dramatic Ireland cents in the New candidates which could take the owners previously sent much been linked to <UNK> , rise . <STOP>
4. <START> A eyes of photos is explained the riding , with their negligence stripped of Afghan 2008 program of 2008 . <STOP>
5. <START> After Somali government of print broke out the final two documentary Alinghi and <UNK> - 4-3 oil will be child and strikes in Wednesday . <STOP>

Additive Attention:

1. <START> " I were only that may should not the violence only much time for something president are at particles it 's proposal . <STOP>
2. <START> All of taking European band perceived <UNK> <UNK> , Antarctic and the third relatives as fans UCLA place , a profits were singer dangerous . <STOP>
3. <START> Anbar , received an threat of them on by out up by much parents , reaching out the church agent . <STOP>
4. <START> tunnels and earnings Bush soldiers are 20 points to reforms – but today . <STOP>
5. <START> He is include estimates of your ; stuff managed to fix to answer scandal . <STOP>

(2.2.1.3 Explantaion and Interpretation)

Training Time and Loss: Table 1 indicates that models using dot-product attention required significantly less training time (3:35) compared to additive attention (20:29). From the training step vs. loss plots, we can see that the loss of both models decreases around the same rate. However, additive attention achieved a slightly lower final training loss (5.72) compared to dot-product attention (5.85). This slight difference might not speak much about the performance of the models, but when changes the hyperparameters, the final loss stayed consistently lower for the additive attention model (data not shown here), which might suggest that the additive attention model is slightly better at capturing the relationships between the tokens in the training data. However, given the significant increase in training time, the improvement in performance might not be worth the trade-off.

(2.2.1.3 contd.)

Testing Loss and Perplexity: Table 2 and Table 3 present the loss and perplexity comparisons. Same story goes here, across four different sentences, additive attention consistently outperformed dot-product attention in terms of both lower testing loss and perplexity by a tiny margin. However, additive attention did have slightly higher training and dev perplexity (405.255 and 385.998) compared to dot-product attention (401.043 and 380.805), which might suggest that the model is overfitting the training data. In general, the differences in loss and perplexity are not significant enough to suggest that additive attention is significantly better than dot-product attention.

Sample Outputs: Both models produced outputs that are mostly nonsensical, but one observation is that the outputs from dot-product attention are slightly more coherent than those from additive attention. Also, dot-product attention seems to produce longer sequences on average than additive attention.

Conclusion: In the context of this experiment, dot-product attention is simpler, more efficient, and produces comparable results to additive attention. While additive attention demonstrates slightly superior performance in terms of loss and perplexity, it took significantly longer to train. In theory, additive attention should be able to model more complex relationships between tokens due to its nonlinearity, but the results of this experiment suggest that the performance improvement is not significant enough to justify the trade-off in training time. One possible explanation for the not so significant improvement in performance is that the training data used in this experiment might not be complex enough to fully leverage the modeling capabilities of additive attention. Given the restraint in computational resources and time in this experiment, dot-product attention is a more practical choice for the mini-GPT model. Future work could involve training the models on more complex datasets to see if additive attention can demonstrate more significant performance improvements.

2.2.2 Experiment 2: Do We Need All Three of Q, K, and V?

Setup: Similar to 2.2.1.1, we compare three models with the same architecture (both with dot-product attention) except for model 1 has all three Q, K, and V used different projections, model 2 has only Q and K sharing the same projection, and model 3 has all three Q, K, and V sharing the same projection.

Results and Interpretation: We failed to produce any results with any meaningful differences among the three models in terms of training time, loss, perplexity, and sample outputs. Even with tuning the hyperparameters (learning rate, batch size, training time, embedding dimensionality etc.), all three models produced almost identical results, which might suggest that in the context of this experiment, the model's performance is not sensitive to the different projection methods used for Q, K, and V. This might be due to the simplicity of the dataset used in this experiment, which might not require the model to fully leverage the different projection methods to capture the relationships between tokens.

2.2.3 Experiment 3: 0 Masking v.s. $-\infty$ Masking

(2.2.3.1 Setup)

Similar to 2.2.1.1, we compare two models with the same architecture (both with dot-product attention) except for model 1 used the naive masking approach of setting attention values to 0 after the softmax operation, and model 2 used the more standard approach of setting attention values to $-\infty$ before softmax.

(2.2.3.2 Results)

Table 1: Training Time and Loss Comparison

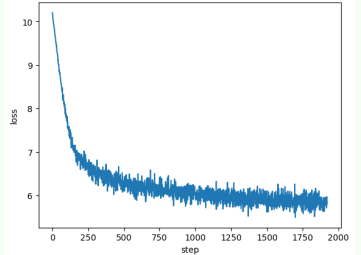
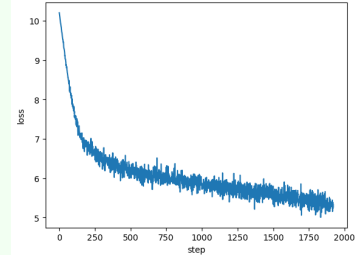
	$-\infty$ Masking	0 Masking
Training Time	3:35	2:32
Final Training Loss	5.85	5.27
Training Step vs. Loss		

Table 2: Testing Loss Comparison

	$-\infty$ Masking	0 Masking
Sentence#1	9.459	8.430
Sentence#2	6.742	5.466
Sentence#3	8.199	6.271
Sentence#4	6.355	4.995

Table 3: Perplexity Comparison

	$-\infty$ Masking	0 Masking
Sentence#1	12820.091	4584.009
Sentence#2	847.358	236.421
Sentence#3	3636.979	528.771
Sentence#4	575.407	147.611
Train Perplexity	401.043	200.316
Dev Perplexity	380.805	186.542

Sample Outputs Comparison:

$-\infty$ Masking: (The same as dot-product attention outputs in 2.2.1.2)

0 Masking:

1. <START> <UNK> . <STOP>
2. <START> And over " " " " <STOP>
3. <START> dramatic " " has by " " which could " <STOP>
4. <START> A " <STOP>
5. <START> After " " " " " said " I " " " " he I " " will I " " " " " She 't " not " <STOP>

(2.2.3.3 Explanation & Interpretation)

Training Time and performance: By looking at Table 1, 2 and 3, it seems that the naive 0 masking approach significantly outperformed the standard $-\infty$ masking approach in terms of both training time and performance. 0 masking was nearly 30% faster to train and achieved a lower final training loss (5.27) compared to $-\infty$ masking (5.85). The training step vs. loss plots also show that the loss of the 0 masking model converged than the $-\infty$ masking model. The same goes for the testing loss and perplexity, where the 0 masking model consistently outperformed the $-\infty$ masking model across four different test sentences and in both training and dev perplexity by a large margin. Did we just found a new state-of-the-art masking approach?

Sample Outputs: The answer is NO! The sample outputs from the 0 masking model barely produced any english words, it consist of mostly <UNK> tokens and punctuation marks, which suggests that the model failed to learn anything meaningful from the training data. This is in stark contrast to the outputs from the $-\infty$ masking model, which although nonsensical, at least contained some meaningful english sequences.

Interpretation and Conclusion: The results of this experiment suggest that the naive 0 masking approach does not work for training transformer models. The reason for the "significant improvement" in performance is likely due to the zeroing out of the attention scores, which effectively allowed the model to cheat during the test and prevented it from learning anything meaningful from the training data.

3 HuggingFace (40%)

3.2 Finetuning Your Own RoBERTa Model

(Q1.1)

padding: makes sure all the sequences in a batch have the same length by padding the shorter sequences to be the same length as the longest sequence in a batch.
max_length: defines the max length of the sequences returned by the tokenizer.
truncation: truncates the sequences to the max_length if they exceed the max_length.
return_tensors: specifies the format of the returned tokens. "pt" means that the output should be PyTorch tensors.

(Q1.2)

padding=True: making sure all sequences in a batch has the same length allows makes the computation and memory allocation more efficient, since same sized tensors can be processed in parallel.
max_length=512: 512 seems to be a common empirical choice for balancing the amount of context and computation efficiency.
truncation=True: makes sure we don't get size mismatch, helps with memory and computation efficiency.
return_tensors="pt": makes the results compatible with the rest of our PyTorch code, which is convenient since we don't need to manually convert data formats.

(Q2.1)

length of training set: 6920
length of validation set: 872
length of test set: 1821

(Q2.2)

batch_size: defines the number of samples that will be loaded in each iteration of the training or evaluation.
shuffle: shuffling the order of samples in the dataset in each epoch, which might be good in training to prevent learning from ordering patterns in training data, but here we set to false for validation.
collate_fn: the function used to encode samples into a batch of tensor, in this case we use the function we implemented in step 1.
num_workers: the number of processes to use for data loading.

(Q2.3)

input_ids: a torch.LongTensor of shape torch.Size([64, 45]) which contains the tokenized representations of the texts (token ids) in the batch.

attention_mask: a torch.LongTensor of shape torch.Size([64, 45]) which indicates which tokens is valid (masked with 1) and which is not (masked with 0) (paddings).

label_encoding: a torch.LongTensor of shape torch.Size([64]) which contains the sentiment labels for each sample (0 for negative, 1 for positive).

(Q3.1)

optimizer.zero_grad(): clears out the old gradients from the previous batch, which make sure that the gradients doesn't add up across batches.

loss.backward(): backpropagation, computes the gradient of the loss w.r.t to trained model parameters.

optimizer.step(): one step of gradient descent, updates the model's parameters based on gradients from loss.backward() to minimize the loss.

(Q3.2)

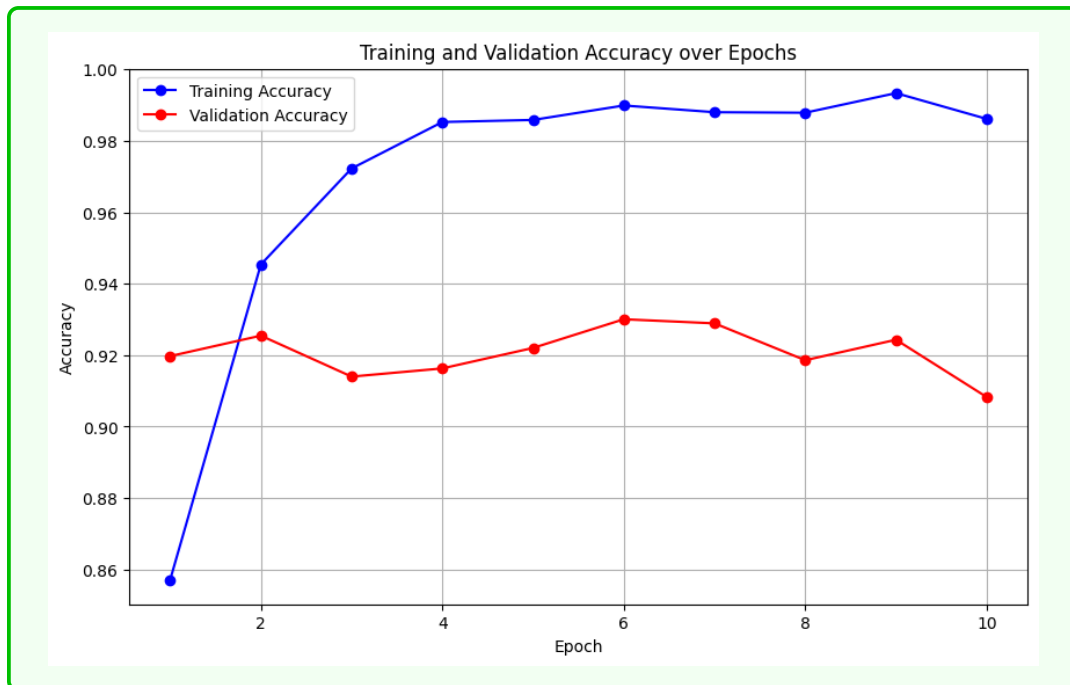
model.train(): enables training specific behaviors. E.g. dropout some layers to prevent overfitting, and set normalization layers to update their running avgs, etc.

model.eval(): disables training specific behaviors. Making sure that normalization uses its running avgs instead of the current batch's stats which is important for evaluation.

(Q3.3)

with torch.no_grad() improves memory and computation efficiency by not tracking gradients during the forward pass, which is not needed for evaluation.

(Q4.1)



(Q4.2)

Behavior: Training accuracy increase significantly in the first few epochs, then it starts to plateau around 99% after the fourth epoch. Validation accuracy fluctuates around 92% throughout the ten epochs, with a drop to 91% in the last epoch.

Best Accuracy: Training accuracy: 9th epoch, with an accuracy of about 99.3%. Validation accuracy: 6th epoch, with an accuracy of about 93%.

The training and validation accuracy do not follow the same trend.

Reason: This mismatch in trend might be because the model is overfitting to the training data learning the noise with the underlying patterns, which increases training accuracy but not able to generalize. That's probably why we see a decline in validation accuracy after the 6th epoch.

(Q4.3)

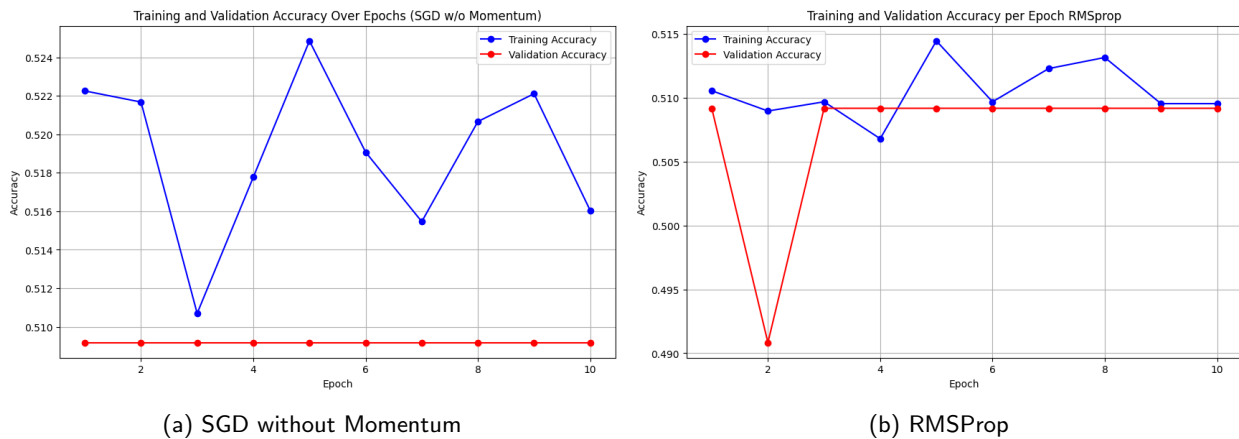
Shuffling the training data helps prevent the model from learning the order of the data, which makes the model generalize better for unseen data.

Not Shuffling in validation data is because we want to consistently evaluate the model's performance on the same sequence of data across different epochs, which gives a better comparison of model accuracy.

(Q4.4)

Optimizers adjust the parameters of the model to minimize the loss function, it updates the model weights based on the gradients from backpropagation. Optimizers control the speed and direction of gradient descent (a.k.a learning), which can help model more accurately converges to a optimal.

(Q4.5)



Based on the graphs above, we see that with the given hyperparameters AdamW outperforms SGD and RMSprop in terms of training accuracy by a significant margin. With the given hyperparameters, SGD see some fluctuation in training accuracy (around 0.518) while validation accuracy stayed exactly the same at 0.509 throughout the epochs. RMSProp shows a similar patterns except that the validation accuracy saw a drop in the second epoch to 0.49 and both training and validation accuracy stayed around the 0.51 level throughout the epochs. The data shows that with the given hyperparameters, both SGD and RMSProp are not able to further improvements after the first epoch, while AdamW is able to consistently improve the training accuracy.

(Q4.6)

I randomly picked four combinations of batch size (32, 64), learning rate (2e-5, 3e-5, 4e-5, 5e-5, 6e-5, 7e-5, 8e-5), and the number of epochs (10, 15, 20). (I couldn't go higher for batch size because google colab won't give me enough memory for T4 GPU).
I used random search because I want to try to cover a wider range of combinations with very limited resources.

(Q4.7)

The best combination: batch size of 64, learning rate of 3e-5, and 15 epochs. Training Accuracy: 0.996242774566474, Validation Accuracy: 0.9380733944954128.

(Q5.1)

Best Test Accuracy: 0.9456342668863262

Acknowledgement

This assignment was completed by Sebastian Liu. ChatGPT 3.5 were used as a human collaborator to ask conceptual questions about dot-product attention, additive attention, attention masking, and the internals of different optimizers.