IFT6135 Assignment 2 Practical

Simon Chamorro

April 10, 2022

1 Problem 1: LSTM

Question 3: Figure 1 shows the results for the 6 experiments that we ran to compare hyperparameters and optimizers for the LSTM.

Question 4a: *Among the 6 configurations, which hyperparameters + optimizer would you use if you were most concerned with wall-clock time?*

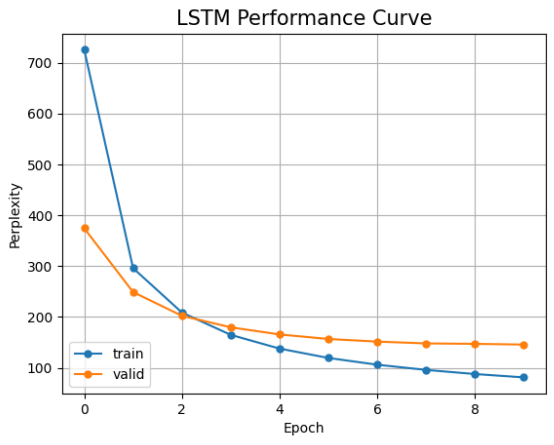
Experiment number 4 is the fastest to train. However, its performance is significantly lower to experiments 1, 2, 5 and 6. I would choose configuration number 2, since the training time difference is negligible compared to experiment 4, and performance is a lot superior. Experiment 2 uses a one-layer LSTM and adamw for optimization.

Question 4b: *With generalization performance?*

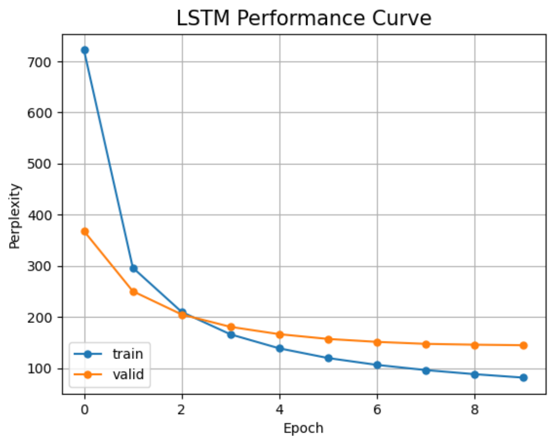
I would choose configuration number 2. In fact, the model using adamw with a one-layer lstm is the model that performs best on the test set after training for 10 epochs as we can see in table 1.

Exp	Model	Training Time (s)	Performance (perplexity)
1	l=1, optimizer=adam	38.79	148.62
2	l=1, optimizer=adamw	38.91	**148.12**
3	l=1, optimizer=momentum	38.24	1600.20
4	l=1, optimizer=sgd	38.20	2088.86
5	l=2, optimizer=adamw	45.47	152.85
6	l=4, optimizer=adamw	55.73	162.85

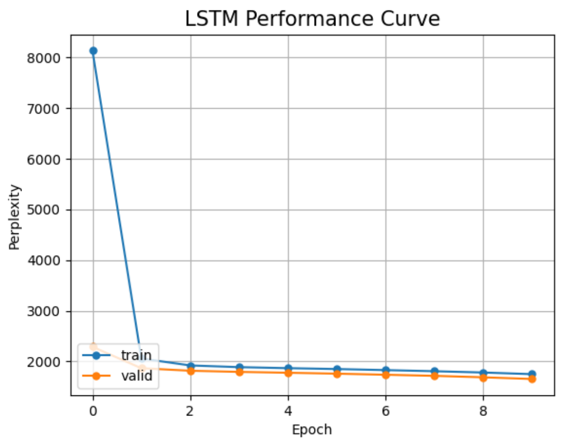
Table 1: **LSTM Results and Training Times per Epoch.**



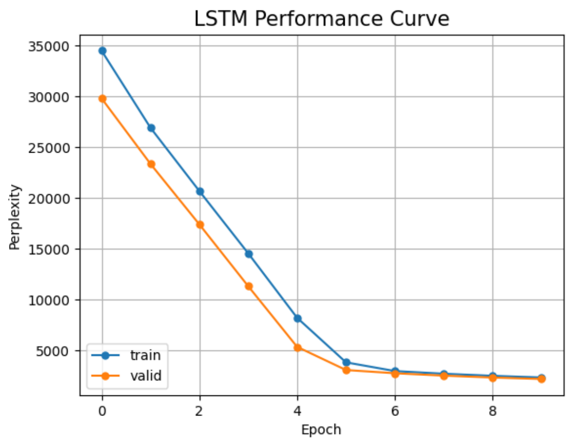
(a) l=1, optimizer=adam



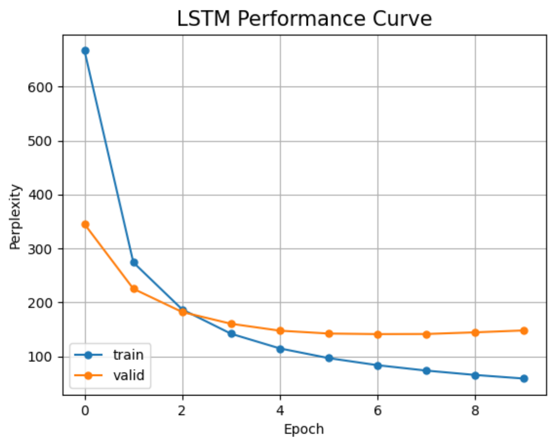
(b) l=1, optimizer=adamw



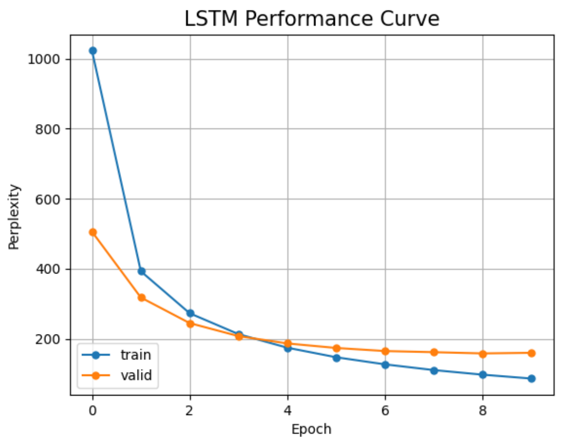
(c) l=1, optimizer=momentum



(d) l=1, optimizer=sgd



(e) l=2, optimizer=adamw



(f) l=4, optimizer=adamw

Figure 1: **LSTM Learning Curves.**

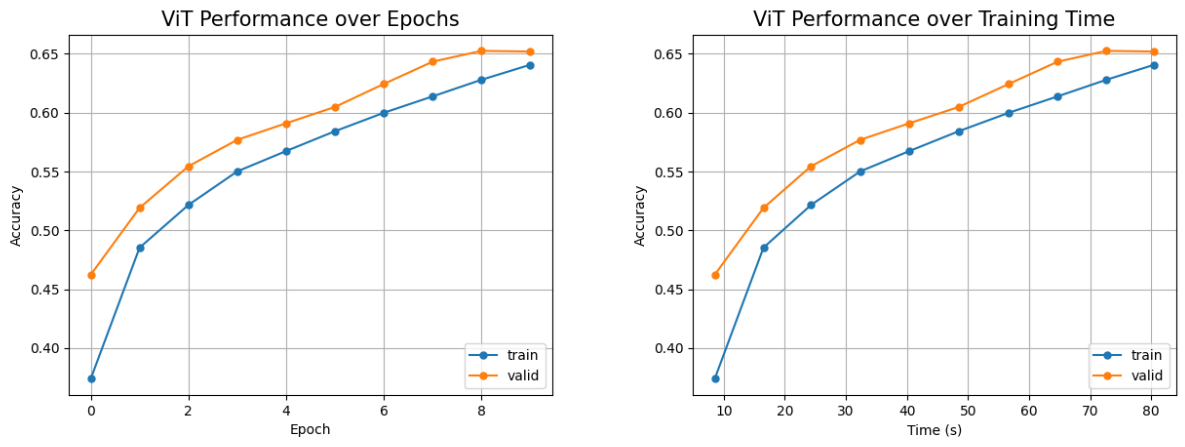
2 Problem 2: ViTs

Attention Mechanism, Question 3: *Using the functions you have implemented, complete the function forward() in the class MultiHeadedAttention. You may implement the different affine projections however you want (do not forget the biases), and you can add modules to the init () function. How many learnable parameters does your module have, as a function of num heads and head size?*

Multi-headed attention requires the equivalent of four linear fully connected layers, with matrices of parameters W_Q, W_V, W_K, W_Y and biases b_Q, b_V, b_K, b_Y. Each W matrix has $num\_heads^2 \times head\_size^2$ parameters and each bias tensor b has $num\_heads \times head\_size$ parameters. So in total, there are $4 \times (num\_heads^2 \times head\_size^2 + num\_heads \times head\_size)$ learnable parameters

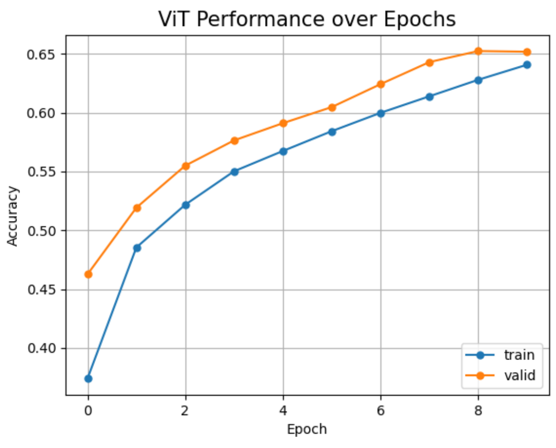
3 Problem 3: Training ViT Models

Questions 1-2: Figures 2 to 8 show the learning curves for all the ViT experiments. Accuracy is plotted over epochs as well as over training time. Table 2 presents the training, validation, and test results for each model as well as the mean training time per epoch.

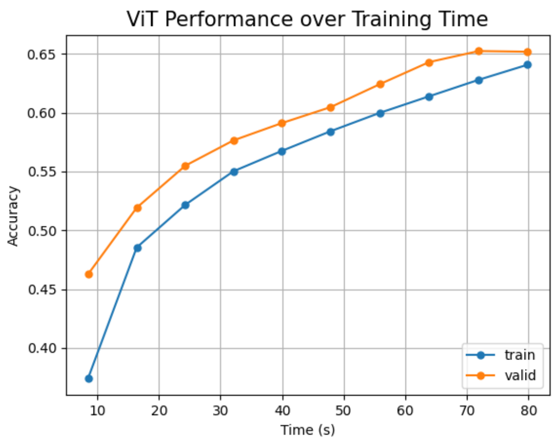


(a) Accuracy over Epochs (b) Accuracy over Time

Figure 2: **ViT Learning Curve Experiment 1.** l=2, optimizer=adam, block=prenorm

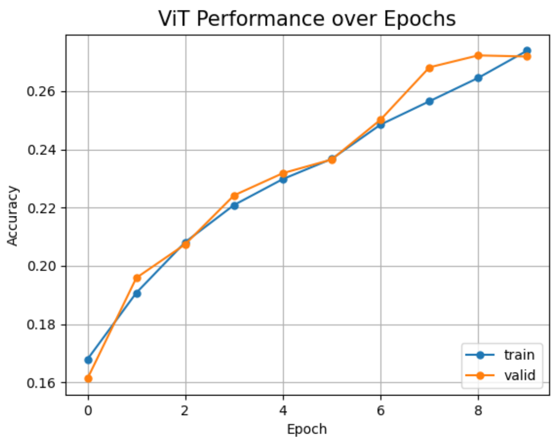


(a) Accuracy over Epochs

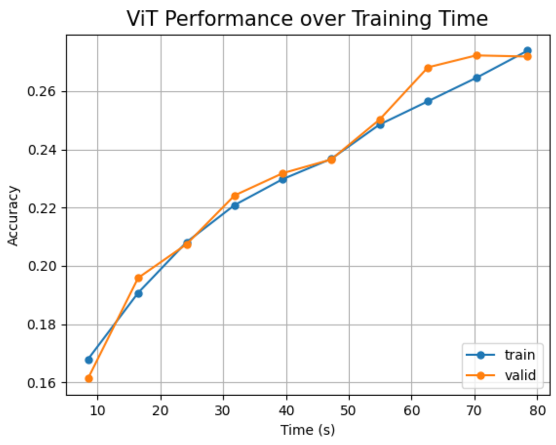


(b) Accuracy over Time

Figure 3: **ViT Learning Curve Experiment 2.** l=2, optimizer=adamw, block=prenorm

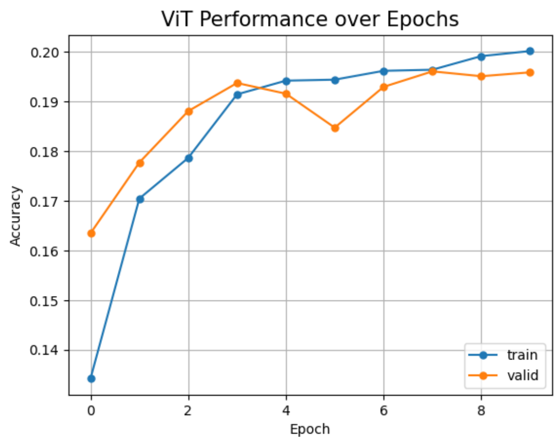


(a) Accuracy over Epochs

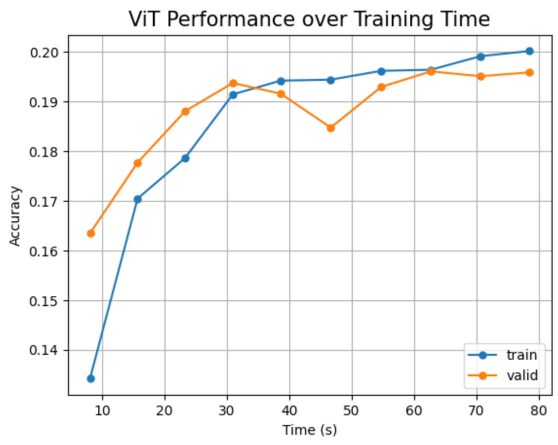


(b) Accuracy over Time

Figure 4: **ViT Learning Curve Experiment 3.** l=2, optimizer=momentum, block=prenorm

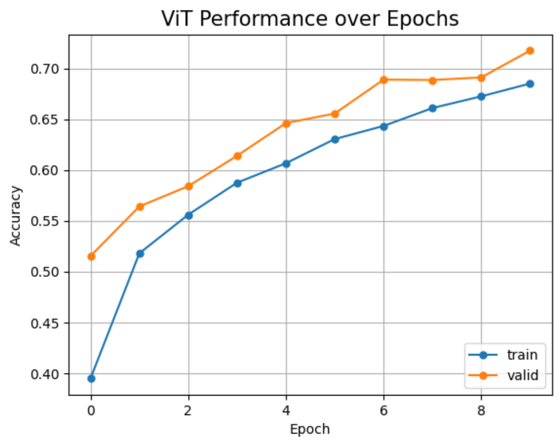


(a) Accuracy over Epochs

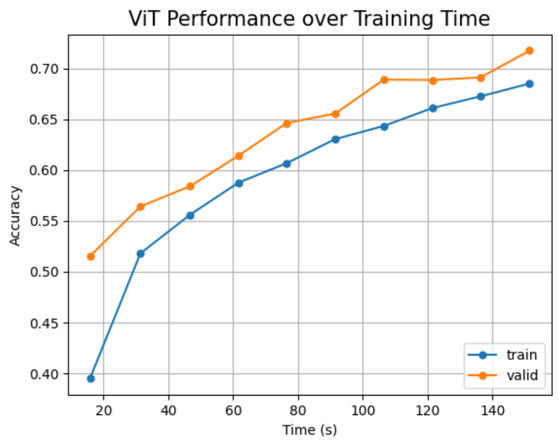


(b) Accuracy over Time

Figure 5: **ViT Learning Curve Experiment 4.** l=2, optimizer=sgd, block=prenorm

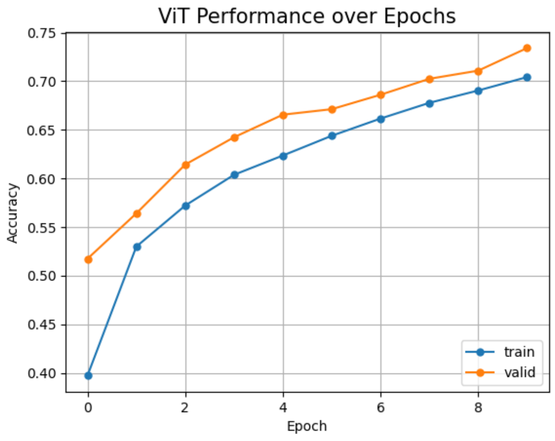


(a) Accuracy over Epochs

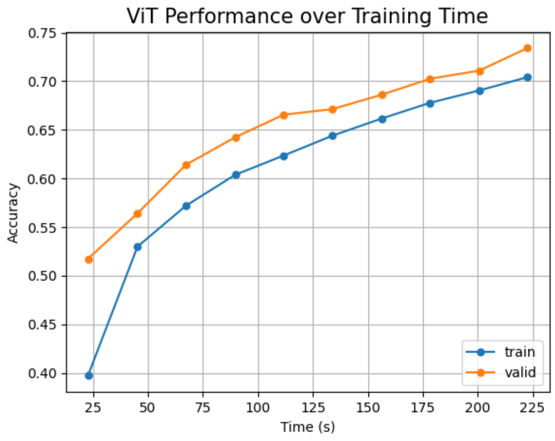


(b) Accuracy over Time

Figure 6: **ViT Learning Curve Experiment 5.** l=4, optimizer=adamw, block=prenorm

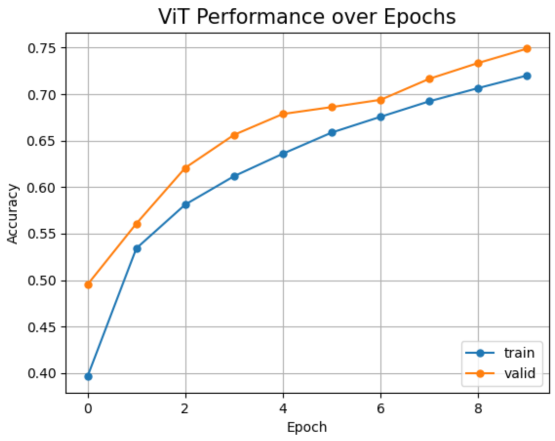


(a) Accuracy over Epochs

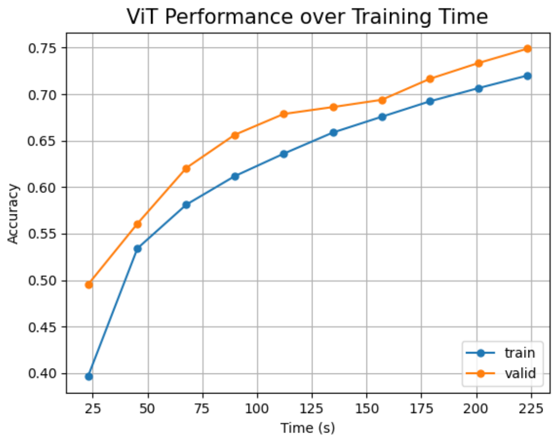


(b) Accuracy over Time

Figure 7: **ViT Learning Curve Experiment 6.** l=6, optimizer=adamw, block=prenorm



(a) Accuracy over Epochs



(b) Accuracy over Time

Figure 8: **ViT Learning Curve Experiment 7.** l=6, optimizer=adamw, block=postnorm

	Model				**Accuracy (%)**		
Exp	**Layers**	**Optimizer**	**Block**	**Train Time (s)**	**Train**	**Valid**	**Test**
1	2	adam	prenorm	8.05	64.08	65.25	61.63
2	2	adamw	prenorm	7.99	64.07	65.23	61.61
3	2	momentum	prenorm	7.85	27.39	27.23	28.88
4	2	sgd	prenorm	7.85	20.02	19.61	19.65
5	4	adamw	prenorm	15.14	68.52	71.76	65.91
6	6	adamw	prenorm	22.27	70.43	73.42	66.89
7	6	adamw	postnorm	22.36	**72.01**	**74.90**	**67.97**

Table 2: **ViT Results.** This table presents the train, validation, and test results for each model, as well as the mean training time per epoch. For the train and validation results, the max result over all epochs is presented.

Question 3: *Among the first 6 configurations, which hyperparameters + optimizer would you use if you were most concerned with wall-clock time? With generalization performance?*

If I was most concerned about clock time, I would choose configuration number 1. In fact, configurations 1 to 4 all have a very similar training time, and configuration 1 has the best performance out of the four. However, if I was most concerned with generalization performance, I would choose configuration 7, as it achieves the best results. This configuration uses 6 layers, which explains why its training time is higher.

Question 4: *Between the experiment configurations 1-4 at the top of run exp vit.py, only the optimizer changed. What difference did you notice about the four optimizers used? What was the impact of weight decay, momentum, and Adam?*

We can see that adam and adamw performed significantly better compared to momentum and sgd over the 10 epochs of training. First of all, we notice that the sgd experiment reaches a 19.61% accuracy on the validation set at the end of training. There is also a sign of overfitting towards the end of training. Then, the momentum experiment performs better, reaching a validation accuracy of 27.23%. We can seet that the learning curve does not seem to plateau, so we can assume that the network woudl continue to improve if it was trained for more epochs. Finally, the adam and adamw both show a much better performance, reaching 65.25% and 65.23% respectively for validation accuracy. In conclusion, by comparing these four experiments, we see that momentum has s positive effect on training and helps avoid overfitting and that adam and adamw show faster learning and achieve better results. We cannot conclude on the performance of adam vs adamw only from these experiments as the results are very similar. Maybe running multiple seeds would help to see a clearer distinction.

Question 5: *Compare experiments 6 and 7. Which model did you think performed better (PreNorm or PostNorm)? Why?*

In our experiments, the PostNorm block slightly outperformes the PreNorm block. However, the difference is quite small, so more experiments would be necessary to make a conclusion. For example, running multiple seeds and averaging the results could help. In fact, it studies have shown that usually PreNorm blocks should perform better in transformers. Wang et al. [2019] explain why PreNorm is preferable. In the PreNorm block, the skip connection allows the gradient to bypass the layer normalization unit, which helps prevent gradient vanishing or exploding. This is more important as networks get deeper, which is probably why it does not seem to have a big impact in our experiments since the networks are small.

Question 6: *In configurations 1- 7, you trained a transformer with various hyper-parameter settings. Given the recent high profile transformer based models, are the results as you expected? Speculate as to why or why not. How do they compare with CNN architectures given below (see assignment).*

Table 3 shows the number of parameters that each model has, as well as their performance. As we can see, they don't nearly match the performance of the given CNNs (GoogleNet, ResNet, ResNetPreAct. DenseNet) and they have an order of magnitude more parameters. The highest performance of our experiments is a 67.97% test accuracy whereas all the CNNs have an accuracy over 89%. If we compare the ratio of performance versus number of parameters, it is normal to obtain much better performance from CNN architectures. In fact, CNNs have a lower number of parameters by design, since the kernels are convolved through all the image, applying the same weights to multiple inputs. On the other hand, Transformers have much more parameters. For example, state-of-the art ViT implementations can have up to hundreds of thousands of parameters.

Exp	# of Params	Valid Accuracy	Test Accuracy
1	1,086,730	65.25	61.63
2	1,086,730	65.23	61.61
3	1,086,730	27.23	28.88
4	1,086,730	19.61	19.65
5	2,140,938	71.76	65.91
6	3,195,146	73.42	66.89
7	3,195,146	**74.90**	**67.97**

Table 3: **ViT Number of Parameters.** This table presents the validation and test results for each model, as well as their number of parameters.

Question 7: *For each of the experiment configurations above, measure the average steady-state GPU memory usage (nvidia-smi is your friend!). Comment about the GPU memory footprints of each model, discussing reasons behind increased or decreased memory consumption where applicable.*

Table 4 shows the GPU memory consumption during training for each model. This measure was taken by querying nvidia-smi once per epoch and averaging the results. It is important to note, however, that this reports the total consumption of the card, not only the training process. As we can see, memory usage is quite stable for the first 4 experiments, and it scales up with the number of model parameters, since these need to be stored in the GPU. We can see that memory usage scales linearly with the number of parameters, a two-layer increase yields a 400 to 500 MiB memory footprint increase in practice.

Exp	**Layers**	**# of Params**	**GPU Memory Usage (MiB)**
1	2	1,086,730	2,341.9
2	2	1,086,730	2,332.9
3	2	1,086,730	2,389.5
4	2	1,086,730	2,361.0
5	4	2,140,938	2,813.4
6	6	3,195,146	3,202.2
7	6	3,195,146	3,254.6

Table 4: **ViT Memory Usage.** This table presents the GPU memory usage for each model, as well as their number of parameters and number of layers. Note that this is the memory consumption of the graphics card averaged during training, it does not mean that it is all used by the model, or by the process training the model.

Question 8: *Comment on the overfitting behavior of the various models you trained, under different hyperparameter settings. Did a particular class of models overfit more easily than the others? Can you make an informed guess of the various steps a practitioner can take to prevent overfitting in this case?*

Out of the experiments that we ran, the only one that seems to overfit is experiment number 4, which uses SGD, this experiment does not use any regularisation technique during the optimisation. I think there are a few steps that practitioners can take to avoid overfitting. First, regularisation is important. For example, weight decay seems to help during our experiments. Also, it is important to make sure that we have enough data for the capacity of the model so it won't overfit, and early-stopping should be used to stop training when the model has the best generalization performance.

References

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. *arXiv preprint arXiv:1906.01787*, 2019.