Written Assignment

1 RNNs and Attention [20pt]

For any successful deep learning system, choosing the right network architecture is as important as choosing a good learning algorithm. In this question, we will explore how various architectural choices can have a significant impact on learning. We will analyze the learning performance from the perspective of vanishing /exploding gradients as they are backpropagated from the final layer to the first.

1.1 Warmup: A Single Neuron RNN

Consider an n layered fully connected network that has scalar inputs and outputs. For now, assume that all the hidden layers have a single unit, and that the weight matrices are set to 1 (because each hidden layer has a single unit, the weight matrices have a dimensionality of $\mathbb{R}^{1\times 1}$).

1.1.1 Effect of Activation - ReLU

Lets say we're using the ReLU activation. Let x be the input to the network and let $f : \mathbb{R}^1 \to \mathbb{R}^1$be the function the network is computing. Do the gradients necessarily have to vanish or explode as they are backpropagated? Answer this by showing that $0 \leq \left|\frac{\partial f(x)}{\partial x}\right| \leq 1$.

From the chain rule, we know the following,

$$\frac{\partial f}{\partial x} = \frac{\partial h_n}{\partial h_{n-1}} \cdot \frac{\partial h_{n-1}}{\partial h_{n-2}} \cdots \frac{\partial h_1}{\partial x}$$

Given a ReLU activation and weights set to 1,

$$\frac{\partial h_1}{\partial x} = \begin{cases} 0 & x < 0 \\ w_1 = 1 & x \geq 0 \end{cases}$$

This is true for each hidden state h_i. Thus we have bounded $0 \leq \left|\frac{\partial f(x)}{\partial x}\right| \leq 1$. This means that the gradients will not explode since they are bounded to 1. They may still vanish as they are lower bounded at 0.

1.1.2 Effect of Activation - Different weights [5pt]

Solve the problem in 1.1.1 by assuming now the weights are not 1. You can assume that the i-th hidden layer has weight w_i. Do the gradients necessarily have to vanish or explode as they are backpropagated? Answer this by deriving a similar bound as in Sec 1.1.1 for the magnitude of the gradient.

In this new case, we can make no assumptions about the weights w_i. However, the above use of the chain rule still applies, and we can still compute the hidden layer gradients. For some layer i,

$$\frac{\partial h_i}{\partial h_{i-1}} = \begin{cases} 0 & h_{i-1} < 0 \\ w_i & h_{i-1} \geq 0 \end{cases}$$

The lower bound of 0 still holds for $\left|\frac{\partial f(x)}{\partial x}\right|$ as it only takes one unit to be negative for the entire gradient to become 0.
To aid in solving the upper bound, lets assume none of the gradients are zero and plug into the chain rule formulation.

$$\frac{\partial f}{\partial x} = w_n \cdot w_{n-1} \cdot \cdots \cdot w_1$$

This reveals that the gradient bound is dependent on the bound of $w_i \forall i \in [1, n]$. If all the w_i's are less than 1, then the gradients may vanish, but not explode. This is because the product of many values less than 1 will approach 0. On the other hand, if all the w_i's are greater than 1, then we can say that the gradients may explode but cannot vanish. This is because the products of values greater than 1 always increases.
I said may in the previous cases, because for a shallow network, the product might not be a significant value.
In cases where there is a mix of the range of w_i, we cannot say anything with certainty about the bound.
Thus the bound for $\left|\frac{\partial f(x)}{\partial x}\right|$ is,

$$0 \leq \left|\frac{\partial f(x)}{\partial x}\right| \leq \prod_{i=1}^{n} |w_i|.$$

As required.

1.2 Matrices and RNN

We will now analyze the recurrent weight matrices under Singular Value Decomposition. SVD is one of the most important results in all of linear algebra. It says that any real matrix $M \in \mathbb{R}^m$ can be written as $M = U\Sigma V^T$ where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are square orthogonal matrices, and $\Sigma \in \mathbb{R}^{m\times n}$ is a rectangular diagonal matrix with nonnegative entries on the diagonal (i.e. $\Sigma_{ii} \geq 0$ for $i \in \{1, \ldots, \min(m, n)\}$ and 0 otherwise). Geometrically, this means any linear transformation can be decomposed into a rotation/flip, followed by scaling along orthogonal directions, followed by another rotation/flip.

1.2.1 Gradient through RNN [5pt]

Let say we have a very simple RNN-like architecture that computes $x_{t+1} = \sigma(Wx_t)$. You can view this architecture as a deep fully connected network that uses the same weight matrix at each layer. Suppose the largest singular value of the weight matrix is $\sigma_{\max}(W) = \frac{1}{4}$. Show that the largest singular value of the input-output Jacobian has the following bound:

$$0 \leq \sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \left(\frac{1}{16}\right)^{n-1}$$

Hint: if C = AB, then $\sigma_{\max}(C) \leq \sigma_{\max}(A)\sigma_{\max}(B)$. Also, the input-output Jacobian is the multiplication of layerwise Jacobians.
We can start by understanding the gradient $\frac{\partial x_n}{\partial x_1}$.

$$\frac{\partial x_n}{\partial x_1} = \frac{\partial}{\partial x_1}\left(\sigma(Wx_{n-1})\right) = \frac{\partial x_n}{\partial x_{n-1}} \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdots \frac{\partial x_2}{\partial x_1}.$$

And each term is given by

$$\begin{aligned}\frac{\partial x_i}{\partial x_{i-1}} =& \frac{\partial}{\partial x_{i-1}}\sigma(Wx_{i-1}) = \operatorname{diag}(\sigma'(Wx_{i-1}))W = \operatorname{diag}(\sigma(Wx_{i-1})(1-\sigma(Wx_{i-1})))W \\ =& D_i W\end{aligned}$$

Where the Jacobian is diagonal due to the element-wise sigmoid. This gives,

$$\frac{\partial x_n}{\partial x_1} = D_{n-1}WD_{n-2}W\cdots D_1W$$

By the hint, we know that

$$\sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \prod_{i=1}^{n-1}\sigma_{\max}(D_i)\cdot\sigma_{\max}(W)^{n-1},$$

so we can use this.
Using the known property of SVD: if $W = U\Sigma V^T$, then $W^i = U\Sigma^i V^T$, we can solve for $\sigma_{\max}(W^{n-1})$. And noting that $\sigma_{\max}(W) = \frac{1}{4}$

$$\sigma_{\max}(W^{n-1}) = \sigma_{\max}(U\Sigma^{n-1}V^T) = \left(\frac{1}{4}\right)^{n-1}$$

For $\sigma_{\max}(D)$, we need to note that the derivative of the sigmoid function peaks at $\frac{1}{4}$ and is bounded above by this value. Since each D_i is an element-wise sigmoid derivative of Wx_{i-1}, this bound is valid for D_i. For diagonal matrices, the singular values are equal to the absolute values of the diagonal entries. So the maximum value in the diagonal matrix is the maximum singular value. Thus we have that,

$$\prod_{i=1}^{n-1}\sigma_{\max}(D_i) = \left(\frac{1}{4}\right)^{n-1}.$$

Combining the identity from the hint, we get,

$$\sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \left(\frac{1}{4}\right)^{n-1}\left(\frac{1}{4}\right)^{n-1} = \left(\frac{1}{16}\right)^{n-1}$$

And now we discuss the lower bound. The singular values of a matrix are always greater than or equal to zero. Thus no further explanation is required to show that the lower bound for $\sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right)$ is 0.
Thus we have shown that,

$$0 \leq \sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \left(\frac{1}{16}\right)^{n-1}$$

as required.

1.3 Relative Attention

We implement relative attention, as in traditional Softmax attention, however we now include a new term that allows us to reweight the attention scores based on the distance between inputs.

Now assume that we are working with a single-headed local self-attention mechanism on a one-dimensional sequence $x \in \mathbb{R}^n : x_i \geq 0 \forall i$ and W_Q, W_K, $W_V \in \mathbb{R}$.

$$\alpha_{i,j}(Q, K, V, p) = \left(\frac{Q_i K_j}{\sqrt{d_k}} + p_{i-j} \right)$$

where (p_k) is a sequence of scalars and Q, $K \in \mathbb{R}^n$. Note that here we allow the index k to take on negative values.

$$\text{ReltativeAttention}(Q, K, V, p) = \text{softmax}(\alpha(Q, K, p))V$$

And our attention layer is simply:

$$\text{RelAttnLayer}(x; W_Q, W_K, W_V, p) = \text{ReltativeAttention}(W_Q x, W_K x, W_V x, p)$$

1.3.1 Implement a 1D Convolution [5pts]

Implement the following 1D-convolution with kernel $w = [2, 0, 1]^T$, as using the RelAttnLayer. Specifically, you need to implement:

$$\text{Conv1D}(x; w)_i = \sum_{j=-1}^{1} x_{i+j} w_{j+2}$$

You may ignore the behavior of your implementation in the edge cases when $i > n - 1$ and $i < 1$. Please specify the p, W_Q, W_K, and W_V you used and argue why they closely approximate this function in the relevant range.

Lets begin by computing the convolution for the given weights.

$$\text{Conv1D}(x)_i = 2x_{i-1} + 0 \cdot x_i + 1 \cdot x_{i+1} = 2x_{i-1} + 1 \cdot x_{i+1}$$

This means each value in the output will be $y_i = 2x_{i-1} + x_{i+1}$. The computation will look as below.

$$\text{softmax}(\alpha(Q, K, p)) \cdot W_V x$$

So we want the output of the softmax to look like $[2, 0, 1]^T$. We can start by setting W_Q and W_K to zero and specifying p. With $W_V = 1$, we need

$$\sigma(p_{-1})x_{i-1} + \sigma(p_0)x_i + \sigma(p_1)x_{i+1}$$

such that

$$\sigma(p_{-1}) = 2 \quad \sigma(p_0) = 0 \quad \sigma(p_1) = 1.$$

For $\sigma(p_0)$, we pick $p_0 = -\infty$. As for the other two, we cannot find values that satisfy, so we can approximate by proportional values. One such case is $p_{-1} = \ln 2$ and $p_1 = -\ln 2$. This leads to the following.

$$\sigma(p_{-1} = \ln 2) = \frac{2}{3} \quad \sigma(p_1 = -\ln 2) = \frac{1}{3}$$

And this meets the proportion requirement. To summarize what we have found so far, the parameters are listed below.

$$p_{-1} = \ln 2 \quad p_0 = -\infty \quad p_1 = -\ln 2$$

$$W_Q = W_K = 0 \quad W_V = 1$$

This computes an approximation $\frac{2}{3}x_{i-1}+\frac{1}{3}\cdot x_{i+1}$. However, we can eliminate this factor by changing W_V from 1 to 3. This will lead to scaling the entire result by 3, removing the denominators and achieving our desired result.
Thus the final choice of parameters is as listed below.

$$p_{-1} = \ln 2 \quad p_0 = -\infty \quad p_1 = -\ln 2$$

$$W_Q = W_K = 0 \quad W_V = 3$$

1.3.2 Implement Max Pooling [5pts]

Now we will use RelAttnLayer to approximate 1d max-pooling with a stride of 1 and a window size of $2k+1$ around the current input. Recall that max pooling with a stride 1 and width of $2k+1$ is simply:

$$\text{MaxPool}(x)i = \max_{-k\leq m\leq k} x_{m+i}$$

Again you may ignore the edge cases in your solutions $i > n-k$ and $i < k+1$. Please specify the p, W_Q, W_K, and W_V you used and argue why they approximately implement this function in the relevant range.

The choice of parameters is listed below.

$$W_Q = 1, \quad W_K = 1, \quad W_V = 1.$$

$$p = \begin{cases} p_m = 0 & \text{for } m \in \{-k, -k+1, \dots, k\} \\ p_m = -\infty & \text{for } |m| > k. \end{cases}$$

Under these parameters, the attention score is as follows, under the assumption that $d_k = 1$ due to scalar weights. This factor is constant so actually does not affect the computation.

$$\alpha_{i,j} = \frac{Q_i K_j}{\sqrt{d_k}} + p_{i-j} = x_i\, x_j + 0 = x_i\, x_j,$$

For a fixed i, note that x_i is constant over the softmax computation. Thus, the scores $\alpha_{i,j}$ are monotonic in x_j, and the softmax will assign the largest weight to the index j corresponding to the largest x_j in the window $\{x_{i-k}, \dots, x_{i+k}\}$. In the limit of a large multiplier the softmax becomes nearly one-hot at the maximum value. Hence, the weighted sum

$$\text{RelAttnLayer}(x)_i = \sum_{j=i-k}^{i+k} \text{softmax}(\alpha_{i,j})\, x_j$$

will approximate

$$\max_{-k\leq m\leq k} x_{i+m},$$

which is exactly the definition of max pooling. Our choice of parameters is

$$W_Q = 1, \quad W_K = 1, \quad W_V = 1,$$

$$p_m = 0 \quad \text{for } m \in \{-k, \dots, k\}, \quad p_m = -\infty \quad \text{for } |m| > k.$$

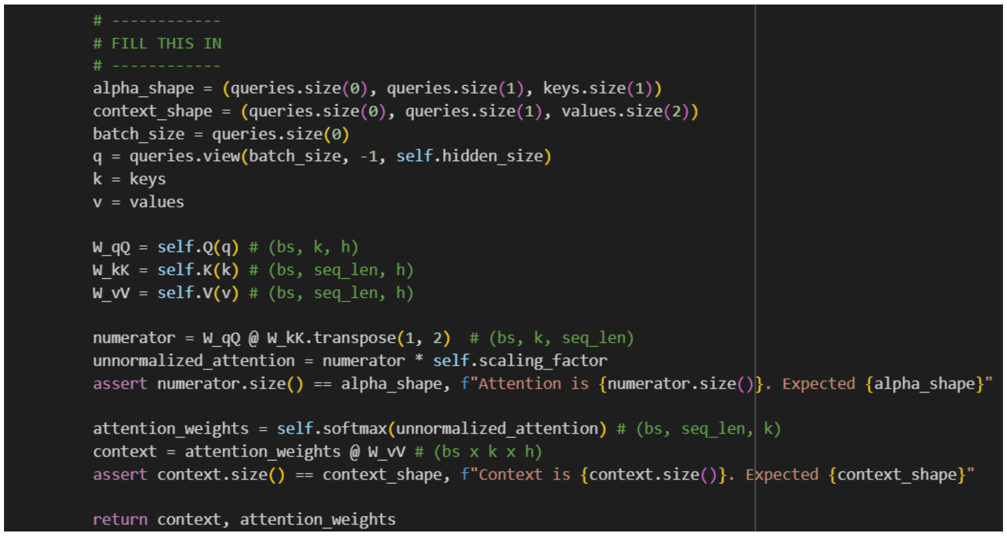
This ensures that the relative attention layers approximates a 1D max pooling with stride 1 and window size $2k+1$.

2 NMT

2.1 Scaled Dot Product Attention

All code available in the “nmr.ipynb”.

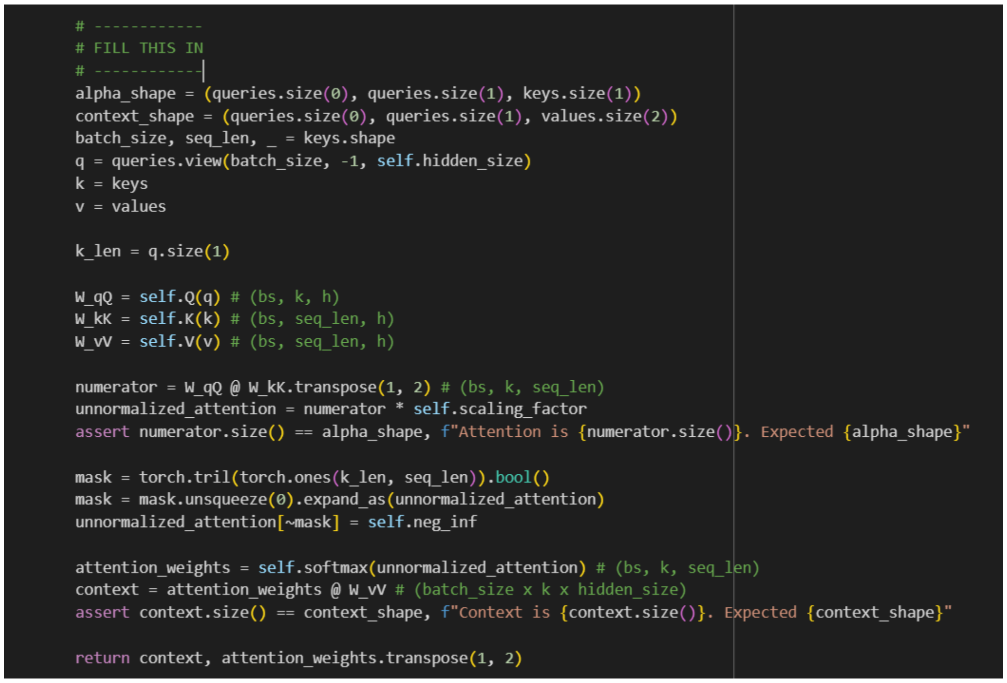
2.1.1 Scaled Dot Product Attention Code



```python
# ------------
# FILL THIS IN
# ------------
alpha_shape = (queries.size(0), queries.size(1), keys.size(1))
context_shape = (queries.size(0), queries.size(1), values.size(2))
batch_size = queries.size(0)
q = queries.view(batch_size, -1, self.hidden_size)
k = keys
v = values

W_qQ = self.Q(q) # (bs, k, h)
W_kK = self.K(k) # (bs, seq_len, h)
W_vV = self.V(v) # (bs, seq_len, h)

numerator = W_qQ @ W_kK.transpose(1, 2)  # (bs, k, seq_len)
unnormalized_attention = numerator * self.scaling_factor
assert numerator.size() == alpha_shape, f"Attention is {numerator.size()}. Expected {alpha_shape}"

attention_weights = self.softmax(unnormalized_attention) # (bs, seq_len, k)
context = attention_weights @ W_vV # (bs x k x h)
assert context.size() == context_shape, f"Context is {context.size()}. Expected {context_shape}"

return context, attention_weights
```

Figure 1: Scaled Dot Product Attention Code

2.1.2 Causal Scaled Dot Product Attention Code



```python
# ------------
# FILL THIS IN
# ------------
alpha_shape = (queries.size(0), queries.size(1), keys.size(1))
context_shape = (queries.size(0), queries.size(1), values.size(2))
batch_size, seq_len, _ = keys.shape
q = queries.view(batch_size, -1, self.hidden_size)
k = keys
v = values

k_len = q.size(1)

W_qQ = self.Q(q) # (bs, k, h)
W_kK = self.K(k) # (bs, seq_len, h)
W_vV = self.V(v) # (bs, seq_len, h)

numerator = W_qQ @ W_kK.transpose(1, 2) # (bs, k, seq_len)
unnormalized_attention = numerator * self.scaling_factor
assert numerator.size() == alpha_shape, f"Attention is {numerator.size()}. Expected {alpha_shape}"

mask = torch.tril(torch.ones(k_len, seq_len)).bool()
mask = mask.unsqueeze(0).expand_as(unnormalized_attention)
unnormalized_attention[~mask] = self.neg_inf

attention_weights = self.softmax(unnormalized_attention) # (bs, k, seq_len)
context = attention_weights @ W_vV # (batch_size x k x hidden_size)
assert context.size() == context_shape, f"Context is {context.size()}. Expected {context_shape}"

return context, attention_weights.transpose(1, 2)
```

Figure 2: Causal Scaled Dot Product Attention Code

2.1.3 Positional Embedding Discussion

Transformers have no understanding of order as it looks at the whole sequence at the same time, so positional encoding is needed to give it an idea of order and sequence structure. Sinusoidal positional encodings are continious and allow the model to generalize better to longer sequences and allows more pattern learning beased on token distance, as opposed to the raw absolute positions of OHE.

2.1.4 Loss Curves and Discussion

	Dataset Size	
Hidden Size	Small	Large
32	0.799	1.156
64	0.478	0.569

Table 1: Table of the lowest validation loss for each run.

Increasing hidden size from 32 to 64 significantly improves model performance, as shown by lower validation loss on both the small (from 0.799 to 0.478) and large datasets (from 1.156 to 0.569). This demonstrates that larger models are better able to learn intricate patters, even on smaller datasets. Increasing the dataset size, however, does not always lead to better generalization: for hidden size 32, the larger dataset resulted in worse performance, probably because of underfitting or insufficient model capacity. On the other hand, the model with hidden size 64 benefits from more data, achieving better generalization. The loss curves from `save_loss_comparison_by_hidden` and `save_loss_comparison_by_dataset` show that larger models converge faster and to lower loss values. These results are consistent with expectations: larger models need sufficient data to generalize well, but can also overfit or underperform if their capacity exceeds what the data supports.

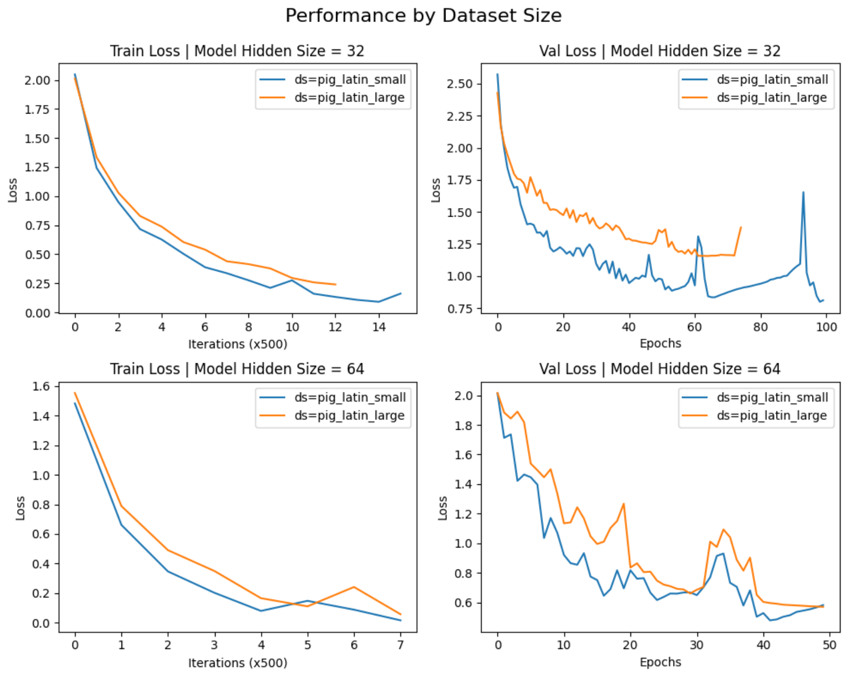


Figure 3: Loss curves produced by running `save_loss_comparison_by_dataset`.

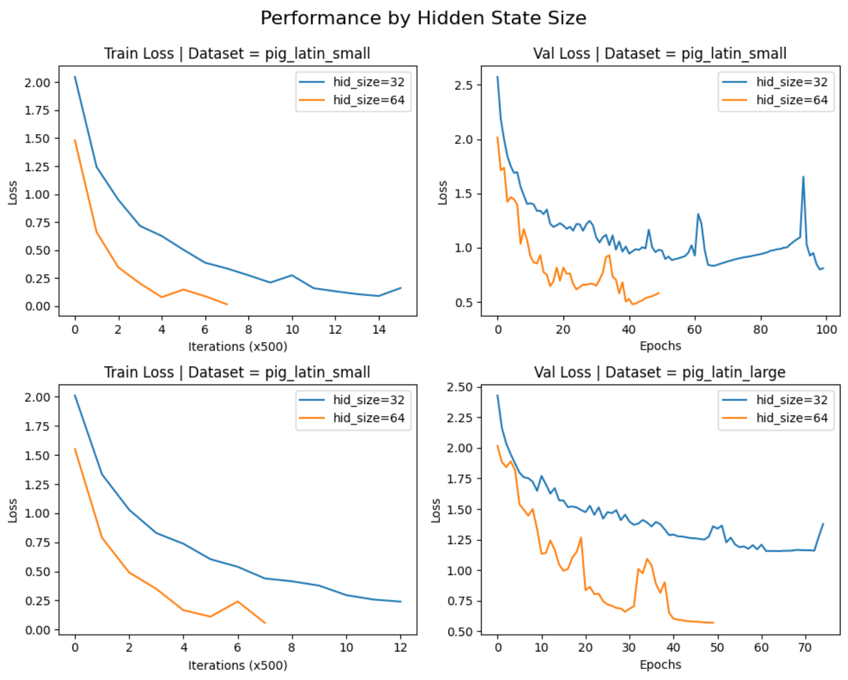
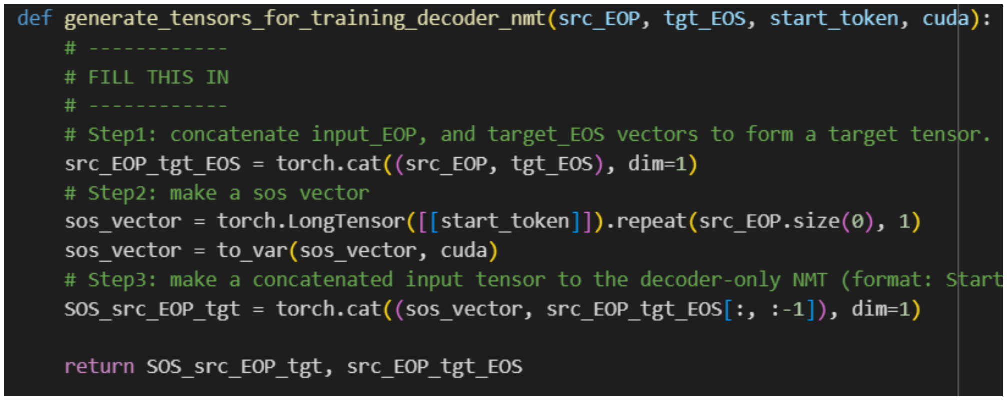


Figure 4: Loss curves produced by running `save_loss_comparison_by_hidden`.

2.2 Decoder Only NMT

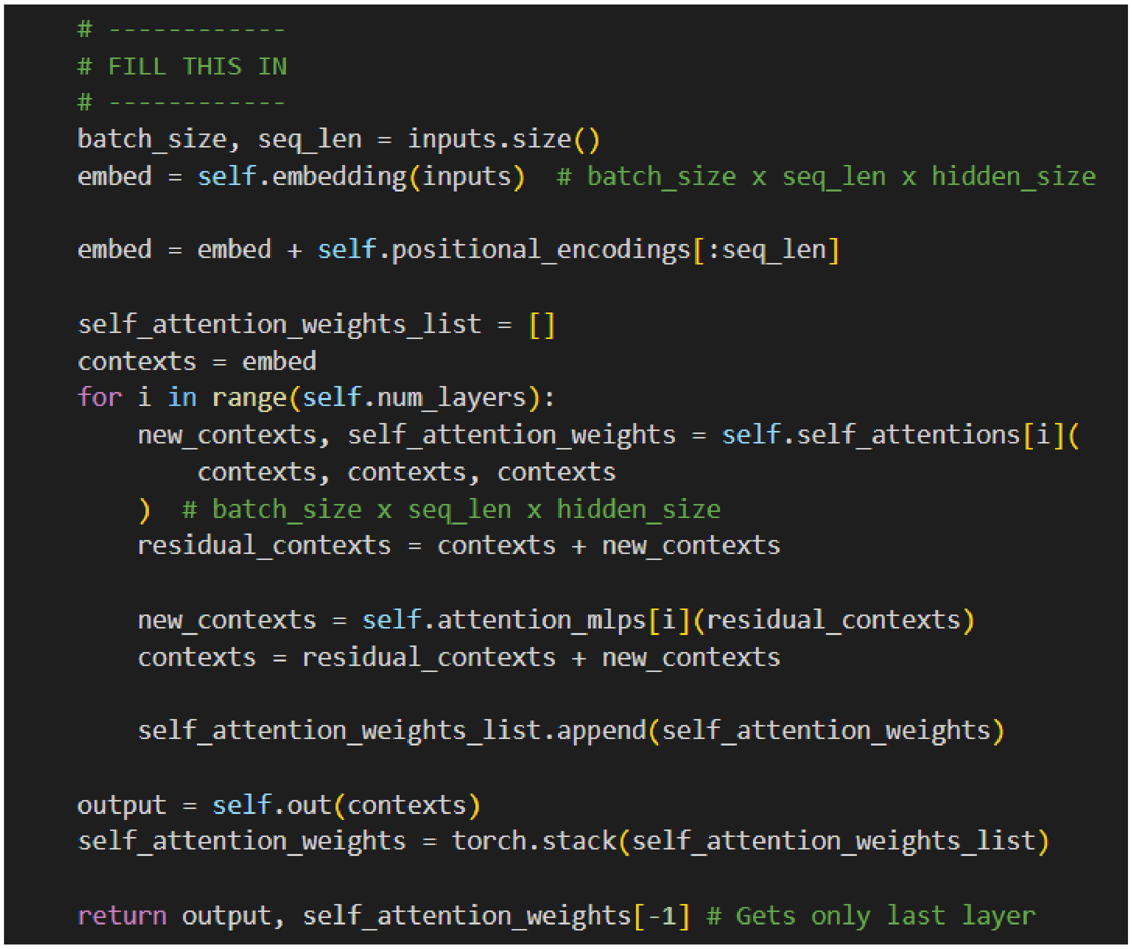
2.2.1 Token Processing Code



```
def generate_tensors_for_training_decoder_nmt(src_EOP, tgt_EOS, start_token, cuda):
    # ------------
    # FILL THIS IN
    # ------------
    # Step1: concatenate input_EOP, and target_EOS vectors to form a target tensor.
    src_EOP_tgt_EOS = torch.cat((src_EOP, tgt_EOS), dim=1)
    # Step2: make a sos vector
    sos_vector = torch.LongTensor([[start_token]]).repeat(src_EOP.size(0), 1)
    sos_vector = to_var(sos_vector, cuda)
    # Step3: make a concatenated input tensor to the decoder-only NMT (format: Start
    SOS_src_EOP_tgt = torch.cat((sos_vector, src_EOP_tgt_EOS[:, :-1]), dim=1)

    return SOS_src_EOP_tgt, src_EOP_tgt_EOS
```

Figure 5: Token Processing Function Code

2.2.2 Transformer Decoder Forward Code



```
    # ------------
    # FILL THIS IN
    # ------------
    batch_size, seq_len = inputs.size()
    embed = self.embedding(inputs)  # batch_size x seq_len x hidden_size

    embed = embed + self.positional_encodings[:seq_len]

    self_attention_weights_list = []
    contexts = embed
    for i in range(self.num_layers):
        new_contexts, self_attention_weights = self.self_attentions[i](
            contexts, contexts, contexts
        )  # batch_size x seq_len x hidden_size
        residual_contexts = contexts + new_contexts

        new_contexts = self.attention_mlps[i](residual_contexts)
        contexts = residual_contexts + new_contexts

        self_attention_weights_list.append(self_attention_weights)

    output = self.out(contexts)
    self_attention_weights = torch.stack(self_attention_weights_list)

    return output, self_attention_weights[-1] # Gets only last layer
```

Figure 6: Transformer Decoder Forward Code

2.2.3 Decoder Only Discussion

Now, run the training and testing code block to see the generated translation using a decoder-only model. Comment on the pros and cons of the decoder-only approach. How is the quality of your generated results compared to the ones using the encoder-decoder model?

The decoder only approach is superior with a loss of 0.2, which is far less than all the encoder-decoder models. And this is with the small dataset, which is remarkable and amplifies the gap in loss even further. Neither the 64, nor the 32 hidden state models got the translation correct when trained on the small dataset, but the decoder only model did.
Some pros for decoder only include, it is better in performance, it requires less data to train, and it is simpler (as it does not have an encoder).
Some cons might be having implicit $\langle EOP \rangle$ as this may lead to less interpertability. And due to not having an encoder, we compute the encoding each time step which may lead to time losses for long inputs. The enc-dec has an edge here as it only computes the encoding once.

2.3 Scaling Law and IsoFLOP Profiles

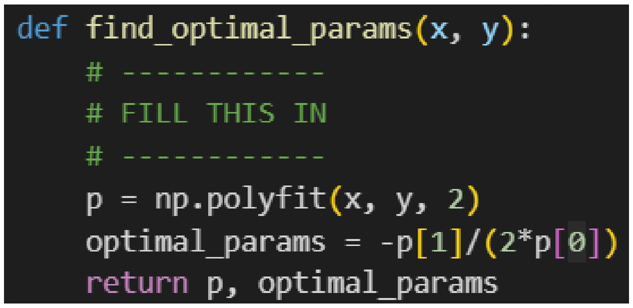
2.3.1 Model Observations

Hidden Size	Approx. FLOPs	Validation Loss
16	256	1.655
32	1024	1.054
48	2304	0.757
64	4096	0.487
96	9216	0.314
128	16384	0.425

Table 2: Model Size and Validation Loss.

The table and plots show that increasing hidden size, and thus FLOPs does indeed lead to decreased validation loss, but only to a certain extent. Past 96, going to 128, the validation loss actually increases again suggesting that 96 is closer to the optimal parameter. This highlights the importance of balancing model capacity with generalization to avoid overfitting.

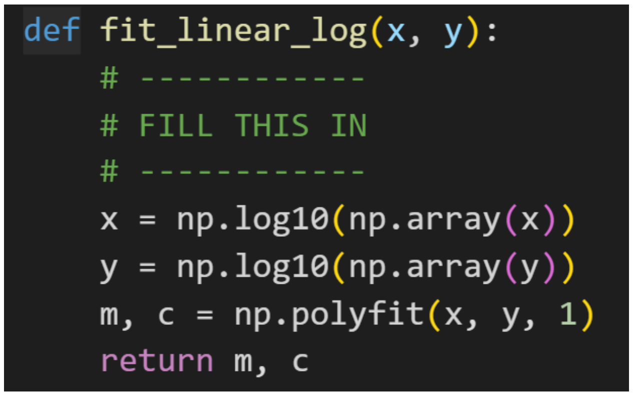
2.3.2 Quadratic Fit Code



```python
def find_optimal_params(x, y):
    # ------------
    # FILL THIS IN
    # ------------
    p = np.polyfit(x, y, 2)
    optimal_params = -p[1]/(2*p[0])
    return p, optimal_params
```

Figure 7: Code to compute the quadratic fit parameters.

2.3.3 Optimal Model Linear Fit



```python
def fit_linear_log(x, y):
    # ------------
    # FILL THIS IN
    # ------------
    x = np.log10(np.array(x))
    y = np.log10(np.array(y))
    m, c = np.polyfit(x, y, 1)
    return m, c
```

Figure 8: Linear Fit Code

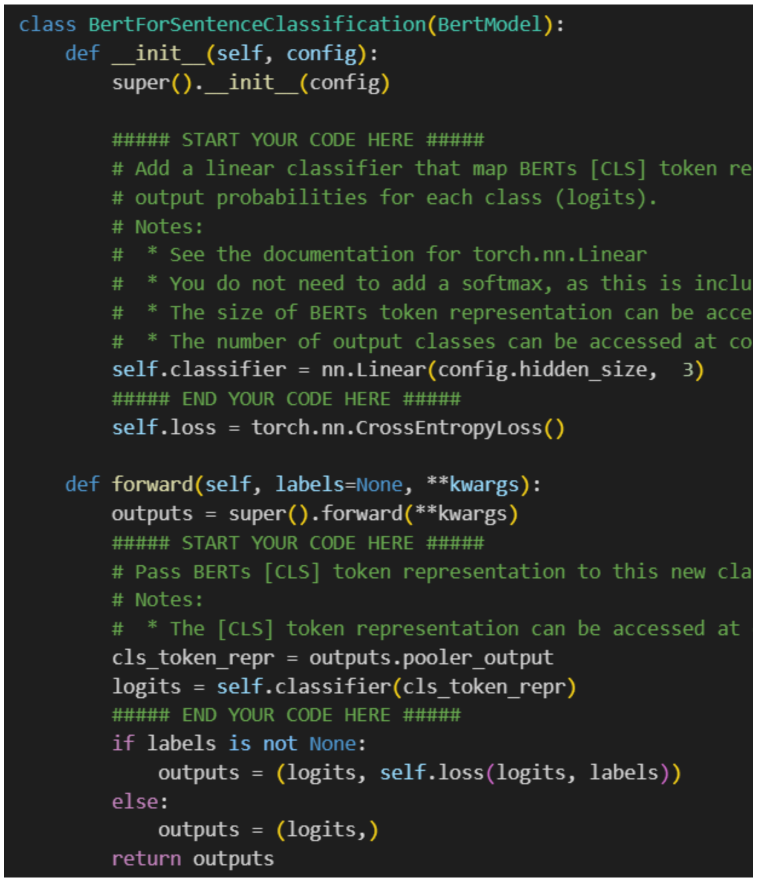
Based on the plot, the optimal number of parameters for 1e15 FLOPs is 1e10.

2.3.4 Optimality Discussion

No, the setup in Section 2.2.3 is not compute optimal. The model has too many parameters (260,000) relative to the available training tokens and compute budget. To align with the compute-optimal scaling shown in the plots, the hidden size should be reduced (e.g., to 64 or 48).

3 Fine-tuning Pretrained Language Models (LMs)

3.1 Bert Classifier Head Code



```
class BertForSentenceClassification(BertModel):
    def __init__(self, config):
        super().__init__(config)

        ##### START YOUR CODE HERE #####
        # Add a linear classifier that map BERTs [CLS] token re
        # output probabilities for each class (logits).
        # Notes:
        #  * See the documentation for torch.nn.Linear
        #  * You do not need to add a softmax, as this is inclu
        #  * The size of BERTs token representation can be acce
        #  * The number of output classes can be accessed at co
        self.classifier = nn.Linear(config.hidden_size,  3)
        ##### END YOUR CODE HERE #####
        self.loss = torch.nn.CrossEntropyLoss()

    def forward(self, labels=None, **kwargs):
        outputs = super().forward(**kwargs)
        ##### START YOUR CODE HERE #####
        # Pass BERTs [CLS] token representation to this new cla
        # Notes:
        #  * The [CLS] token representation can be accessed at
        cls_token_repr = outputs.pooler_output
        logits = self.classifier(cls_token_repr)
        ##### END YOUR CODE HERE #####
        if labels is not None:
            outputs = (logits, self.loss(logits, labels))
        else:
            outputs = (logits,)
        return outputs
```

Figure 9: Bert Classifier Head Code.

3.2 Freezing Pretrained Weights

Open the notebook and run the cells under Question 3 to fine-tune only the classifiers weights, leaving BERTs weights frozen. After training, answer the following questions (no more than four sentences total)

- Compared to fine-tuning (see Question 2), what is the effect on train time when BERTs weights are frozen? Why? (1-2 sentences)
- Compared to fine-tuning (see Question 2), what is the effect on performance (i.e. validation accuracy) when BERTs weights are frozen? Why? (1-2 sentences)

The train time is far less when only training the classifier compared to fine-tuning as we see a decrease from 1 second per epoch to 0, or an increase from 15 batches per second to 60. This is a result of having far less trainable parameters and thus gradients to store and compute and weights to update as the classifier head is far smaller than the rest of the model.

The performance takes a big hit, and drops from 92% to 74% as the model is only learning the new data in a tiny fraction of layers (the classifier). Freezing the model parameters does not allow these parameters to learn anything, and since this is the bulk of the model, it severely restricts the models learning.

3.3 Effect of Pretraining Data

Compared to fine-tuning BERT with the pretrained weights from MathBERT (see Question 2), what is the effect on performance (i.e. validation accuracy) when we fine-tune BERT with the pretrained weights from BERTweet? Why might thsis be the case? (2-3 sentences)

When fine-tuned from the BERTweet weights, it achieves a 72% validation accuracy, which is no where near MathBERT, even when it is frozen. This is expected as BERTweet is not trained on the domain of math, rather twitter, so it needs to learn more and is catching up. MathBERT was trained on math initially, so it has a head start in understanding math content and can focus on learning the classification task, while BERTweet must also learn the mathematical basis for the classifications.

4 Connecting Text and Images with CLIP

4.2 Prompting CLIP

I used the prompt “two goldfish” immediately and it was successful, so I tried “two nemo” which also worked so I then tried “two nemos”, which does not work, interestingly.