CSC413A3

siweitang

March 2023

1 Robustness and Regularization

a Adversarial Examples

i Prediction under Attack

$$\begin{aligned}
f(x;w) = w^T x &\Rightarrow \nabla_x f(x;w) = w \\
&\Rightarrow x' = x - \epsilon w \\
\Rightarrow f(x';w) = w^T x' &\Rightarrow w^T(x - \epsilon w) = w^T x - \epsilon w^T w
\end{aligned}$$

b Gradient Descent and Weight Decay

i Closed Form Ridge Regression Solution

$$\begin{aligned}
\nabla_w(\frac{1}{2n}\|Xw - t\|^2 + \lambda\|w\|^2) &= \frac{1}{2n} \cdot X^T \cdot 2(Xw - t) + \lambda \cdot 2w = 0 \\
\Rightarrow w(\frac{1}{n} \cdot X^T X + 2\lambda I) &= \frac{1}{n} X^T t \\
&\Rightarrow w* = (X^T X + 2\lambda n \cdot I)^{-1} \cdot X^T t
\end{aligned}$$

2 Trading off Resources in Neural Net Training

a Effect of batch size

i 2.1.1 Batch size vs. learning rate

a) As batch size increases, gradient noise will decrease, which allows for bigger optimal learning rate. If B is large, the increase of optimal learning rate is more obvious.

b Training steps vs. batch size

a) C has the most efficient batch size. For batch sizes smaller than C, increase batch size would reduce the training steps significantly with fewer change in computation. If the batch size becomes larger than C, the decrease of training steps is small.

b) Point A: Regime: noise dominated. Potential way to accelerate training: seek parallel compute.
Point B: Regime: curvature dominated. Potential way to accelerate training: use high order optimizers.

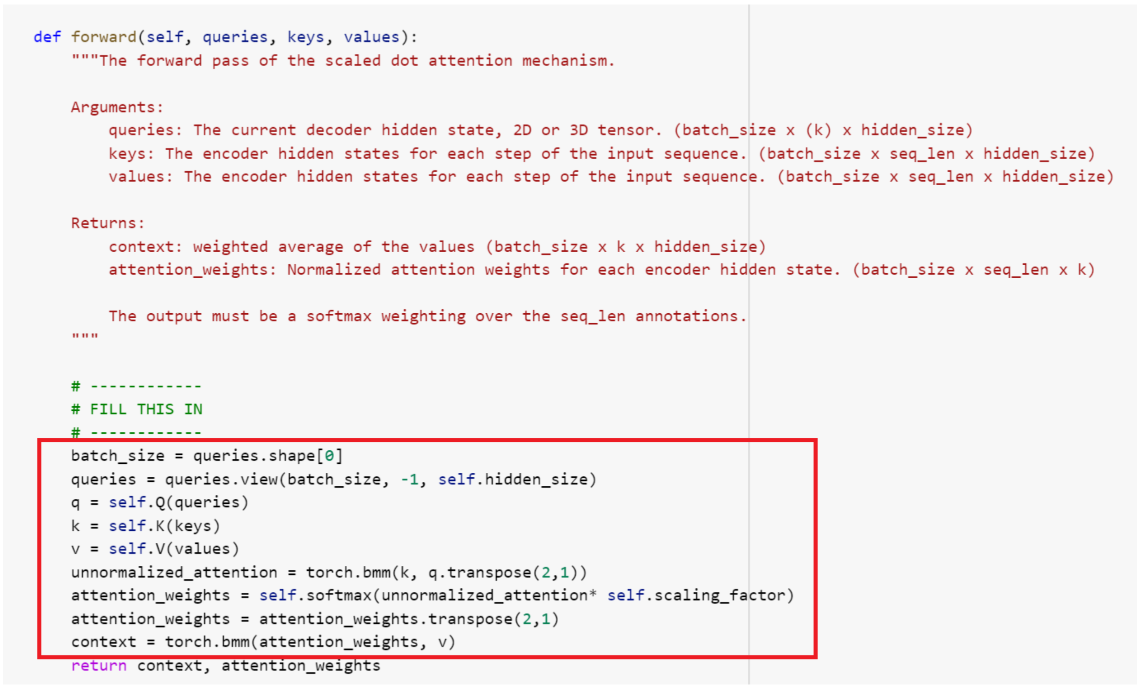
c Model size, dataset size and compute

a) C is the best option. From figure 3, keep training won't change test loss much. From figure 4, increasing batch size also won't help much.

3 Neural machine translation (NMT)

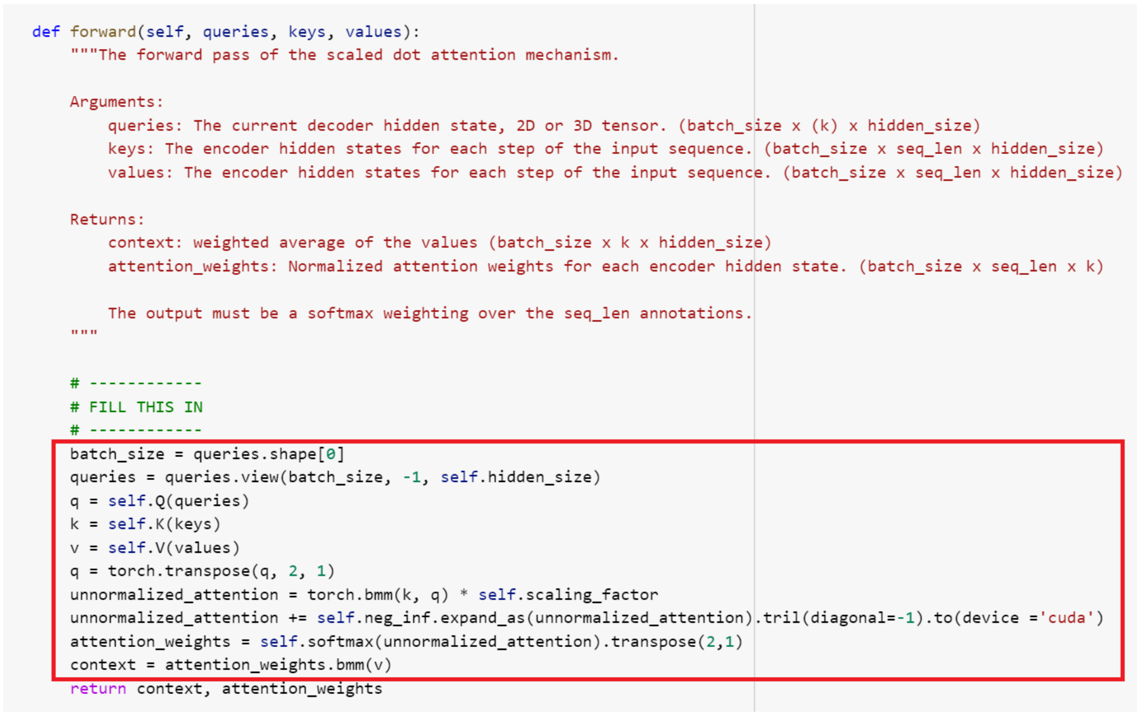
a Scaled Dot Product Attention

ScaledDotProduct:



```
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    Arguments:
        queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidden_size)
        keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
        values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

    Returns:
        context: weighted average of the values (batch_size x k x hidden_size)
        attention_weights: Normalized attention weights for each encoder hidden state. (batch_size x seq_len x k)

        The output must be a softmax weighting over the seq_len annotations.
    """

    # ------------
    # FILL THIS IN
    # ------------
    batch_size = queries.shape[0]
    queries = queries.view(batch_size, -1, self.hidden_size)
    q = self.Q(queries)
    k = self.K(keys)
    v = self.V(values)
    unnormalized_attention = torch.bmm(k, q.transpose(2,1))
    attention_weights = self.softmax(unnormalized_attention* self.scaling_factor)
    attention_weights = attention_weights.transpose(2,1)
    context = torch.bmm(attention_weights, v)
    return context, attention_weights
```

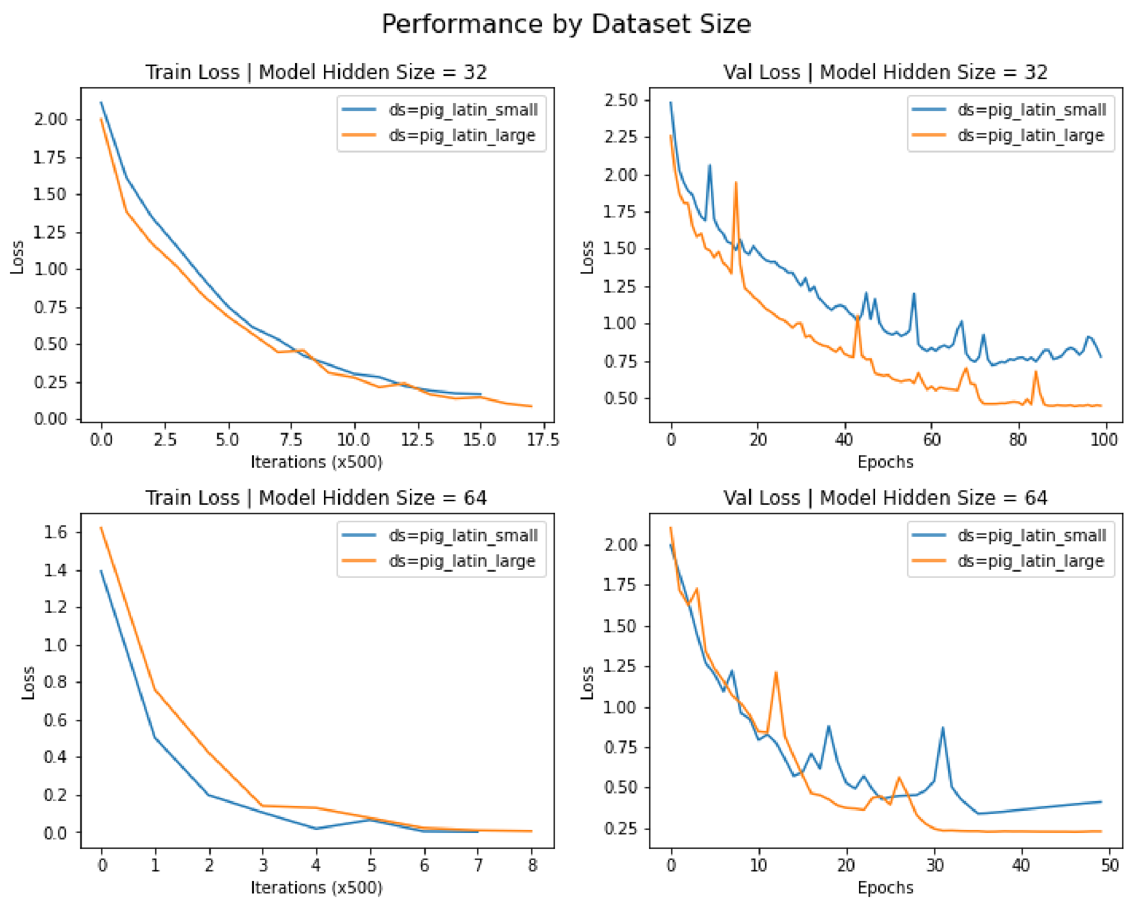
CausalScaledDotProduct:

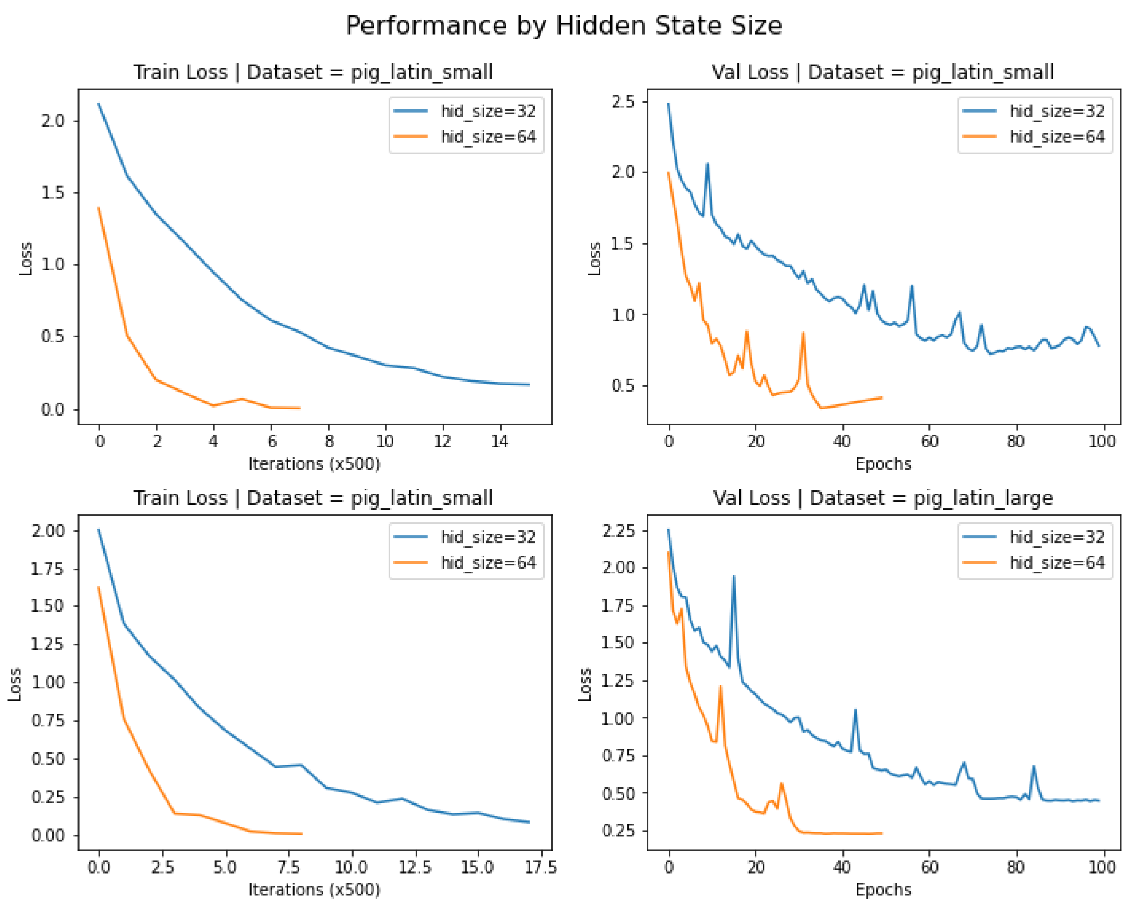


```
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    Arguments:
        queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidden_size)
        keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
        values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

    Returns:
        context: weighted average of the values (batch_size x k x hidden_size)
        attention_weights: Normalized attention weights for each encoder hidden state. (batch_size x seq_len x k)

        The output must be a softmax weighting over the seq_len annotations.
    """

    # ------------
    # FILL THIS IN
    # ------------
    batch_size = queries.shape[0]
    queries = queries.view(batch_size, -1, self.hidden_size)
    q = self.Q(queries)
    k = self.K(keys)
    v = self.V(values)
    q = torch.transpose(q, 2, 1)
    unnormalized_attention = torch.bmm(k, q) * self.scaling_factor
    unnormalized_attention += self.neg_inf.expand_as(unnormalized_attention).tril(diagonal=-1).to(device ='cuda')
    attention_weights = self.softmax(unnormalized_attention).transpose(2,1)
    context = attention_weights.bmm(v)
    return context, attention_weights
```

i Answer to question 3

We need to have positional encoding since the order of words in language affects the meaning. The advantage of position encoding is:
The range of sine and cosine function is within [-1, 1], adding a vector or integer means the position of word gives large position values for later words in sentence, lower the effect of word embedding.
Also, unlike other encoding, sine and cosine function does not require the hidden dimension larger than the sequence length.



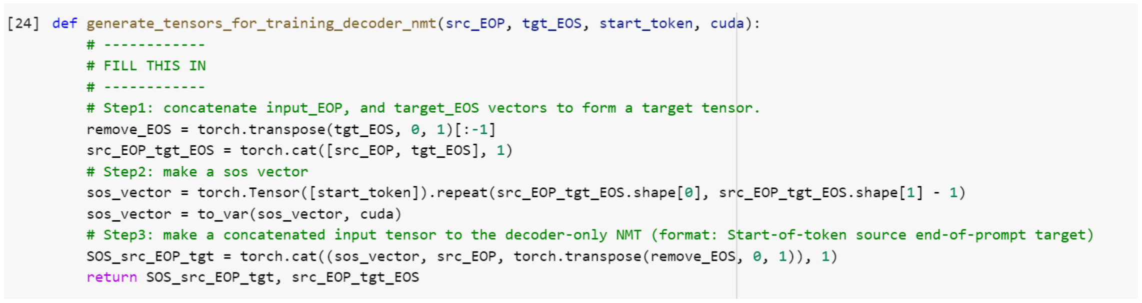


The lowest validation loss is 0.22571481138561467.

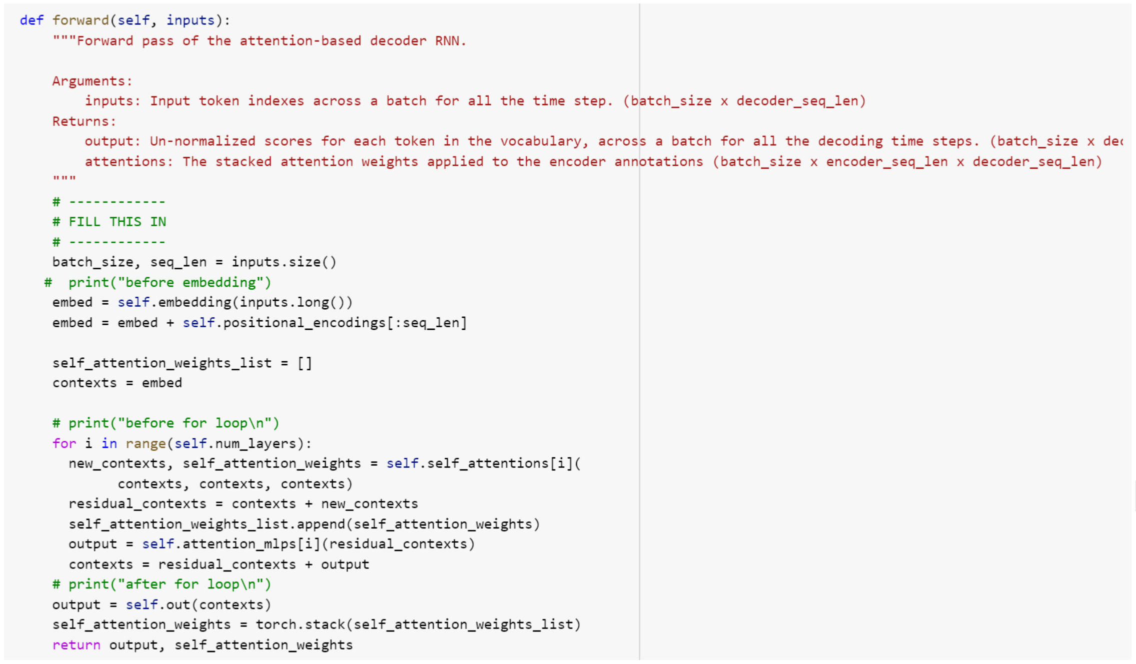
ii Answer to question 4

Increase the hidden size makes the lowest validation loss much smaller, and makes the loss function converges quicker.
With smaller dataset, loss function would converge after a certain number of iterations, with large dataset, the loss function would keep decreasing and not converge. Also, larger dataset would makes the validation loss lower.

b Decoder Only NMT



```
[24] 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.
         remove_EOS = torch.transpose(tgt_EOS, 0, 1)[:-1]
         src_EOP_tgt_EOS = torch.cat([src_EOP, tgt_EOS], 1)
         # Step2: make a sos vector
         sos_vector = torch.Tensor([start_token]).repeat(src_EOP_tgt_EOS.shape[0], src_EOP_tgt_EOS.shape[1] - 1)
         sos_vector = to_var(sos_vector, cuda)
         # Step3: make a concatenated input tensor to the decoder-only NMT (format: Start-of-token source end-of-prompt target)
         SOS_src_EOP_tgt = torch.cat((sos_vector, src_EOP, torch.transpose(remove_EOS, 0, 1)), 1)
         return SOS_src_EOP_tgt, src_EOP_tgt_EOS
```



```
def forward(self, inputs):
    """Forward pass of the attention-based decoder RNN.

    Arguments:
        inputs: Input token indexes across a batch for all the time step. (batch_size x decoder_seq_len)
    Returns:
        output: Un-normalized scores for each token in the vocabulary, across a batch for all the decoding time steps. (batch_size x dec
        attentions: The stacked attention weights applied to the encoder annotations (batch_size x encoder_seq_len x decoder_seq_len)
    """
    # ------------
    # FILL THIS IN
    # ------------
    batch_size, seq_len = inputs.size()
  #  print("before embedding")
    embed = self.embedding(inputs.long())
    embed = embed + self.positional_encodings[:seq_len]

    self_attention_weights_list = []
    contexts = embed

    # print("before for loop\n")
    for i in range(self.num_layers):
      new_contexts, self_attention_weights = self.self_attentions[i](
            contexts, contexts, contexts)
      residual_contexts = contexts + new_contexts
      self_attention_weights_list.append(self_attention_weights)
      output = self.attention_mlps[i](residual_contexts)
      contexts = residual_contexts + output
    # print("after for loop\n")
    output = self.out(contexts)
    self_attention_weights = torch.stack(self_attention_weights_list)
    return output, self_attention_weights
```

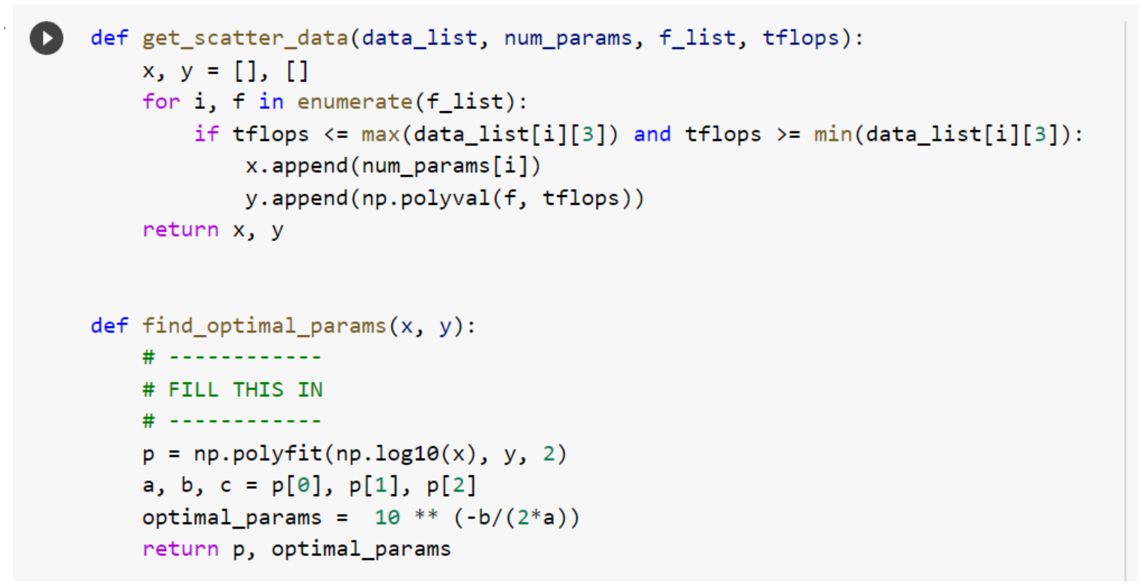
Pros: Decoder-only NMT is simpler and smaller then encoder and decoder model, so it can be trained more quickly and generate translation faster. It also reduces the computation cost, and decoder-only model is more flexible, has wider usage.
Cons: Without encoder component, the model may not be able to capture the full meaning of the input sentence, leading to vague translations. Also, the validation loss is much larger.

c Scaling Law and IsoFLOP Profiles

i Answer to question 1

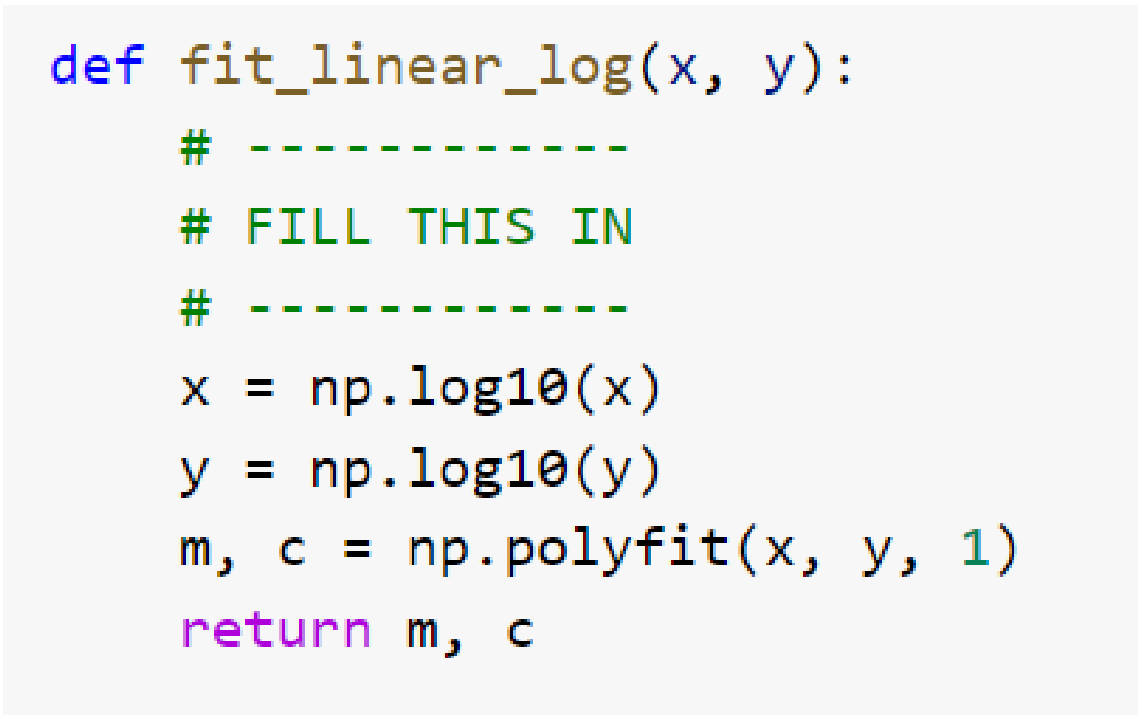
No, a larger model does not always have a smaller validation loss. We can see that from plot. A larger model has more parameters. However, this doesn't guarantee that it will perform better than a smaller model. The performance of a model also depends on other factors such as the choice of hyperparameters.

ii Answer to question 2



```
def get_scatter_data(data_list, num_params, f_list, tflops):
    x, y = [], []
    for i, f in enumerate(f_list):
        if tflops <= max(data_list[i][3]) and tflops >= min(data_list[i][3]):
            x.append(num_params[i])
            y.append(np.polyval(f, tflops))
    return x, y


def find_optimal_params(x, y):
    # ------------
    # FILL THIS IN
    # ------------
    p = np.polyfit(np.log10(x), y, 2)
    a, b, c = p[0], p[1], p[2]
    optimal_params =  10 ** (-b/(2*a))
    return p, optimal_params
```

d Answer to question 3

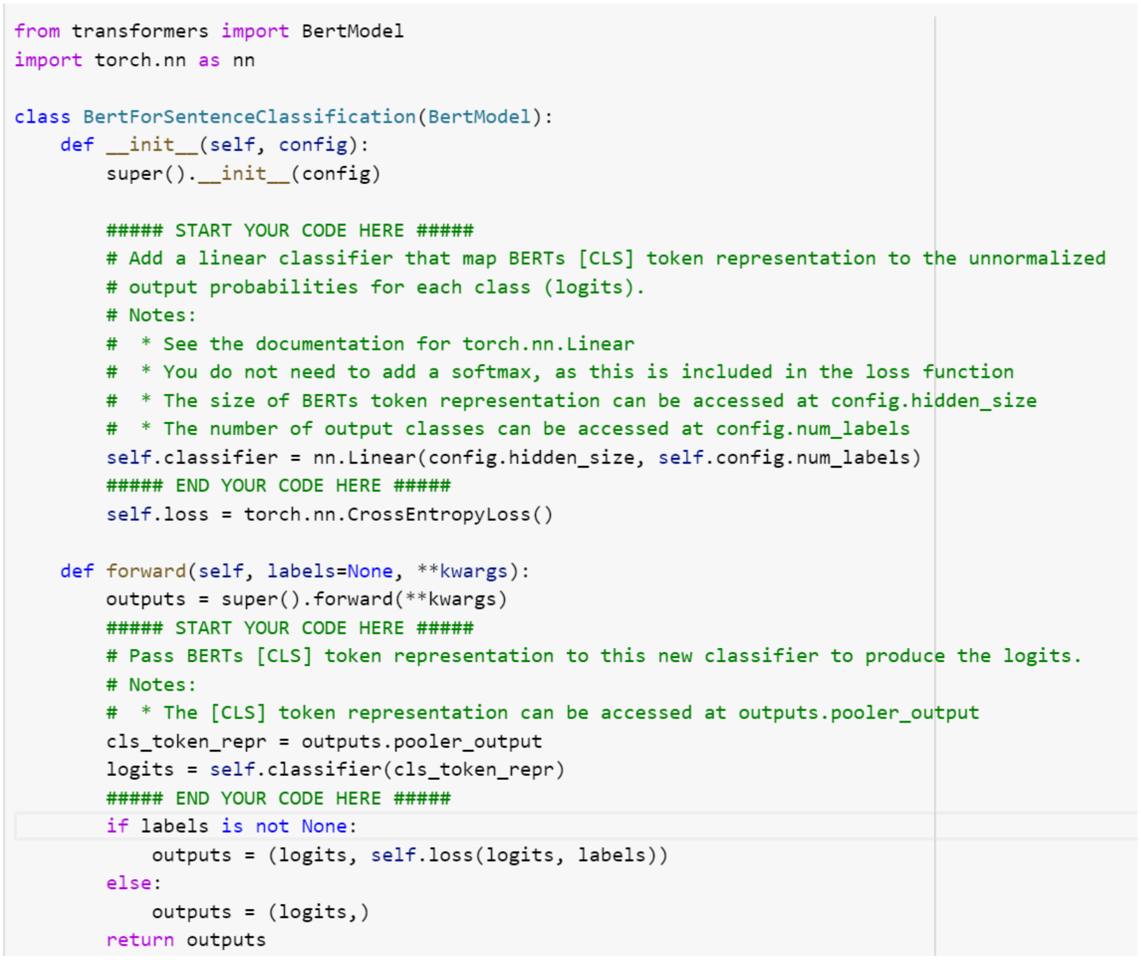


```
def fit_linear_log(x, y):
    # ------------
    # FILL THIS IN
    # ------------
    x = np.log10(x)
    y = np.log10(y)
    m, c = np.polyfit(x, y, 1)
    return m, c
```

e Answer to question 4

The training setup in section 3.2.3 does not compute optimal. We can increase the dataset size to compute optimal.

4 Fine-tuning Pretrained Language Models (LMs)



```
from transformers import BertModel
import torch.nn as nn

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

        ##### START YOUR CODE HERE #####
        # Add a linear classifier that map BERTs [CLS] token representation to the unnormalized
        # 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 included in the loss function
        #  * The size of BERTs token representation can be accessed at config.hidden_size
        #  * The number of output classes can be accessed at config.num_labels
        self.classifier = nn.Linear(config.hidden_size, self.config.num_labels)
        ##### 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 classifier to produce the logits.
        # Notes:
        #  * The [CLS] token representation can be accessed at outputs.pooler_output
        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
```

a Answer to question 3

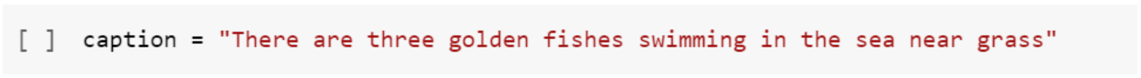
when BERTs weights are frozen, training time is much shorter compared to question 2, since as the model parameters are not updated during the training process. This means that the forward and backward passes during training can be computed much faster since there are no gradients to compute or update.
Also, the accuracy is lower and does not change, since the model will not be able to learn from the data during training. This means that the performance of the model will likely be worse than if the weights were allowed to be updated during training.

b Answer to question 4

training time is a bit longer and the accuracy is still lower, it may causes by the pretraining data is not representative of the task data.

5 Connecting Text and Images with CLIP

It is easy since I just describe the picture content.



```
[ ] caption = "There are three golden fishes swimming in the sea near grass"
```