1.1.1

(a) As the batch size B increases, the noise in the minibatch gradient decreases. This is because averaging over a larger batch reduces the effect of the noise. Therefore, the gradient estimate becomes more accurate and less stochastic with a larger batch size. Then we should use a larger learning rate because the update direction will be more stable and correct. Hence, the convergence process will become faster.

1.1.2

(a) Point C has the most efficient batch size. We can see both batch size and training steps increase exponentially. Hence, the best trade-off point is that there is no extremely large value of batch size and training steps. Point C is in the middle position where there is no extremely high values.

(b)

- Point A: Regime: noise dominated. Potential way to accelerate training: use higher order optimizer
- Point B: Curvature dominated. Potential way to accelerate training: see parallel compute

1.2

(a) C

From Figure 3 and Figure 2, we can see that larger models tend to have lower test loss when dataset size/compute/number of steps is fixed. Model size determines the lower bound of test loss in every plot. The reason is that a larger model can capture more complex patterns and generalize better, resulting in a lower test loss.

2.1.1

```
    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 k x seq_len)

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

        # ------------
        # FILL THIS IN (with the help from ChatGPT)
        # ------------
        q = self.Q(queries)  # (batch_size x k x hidden_size)
        k = self.K(keys)     # (batch_size x seq_len x hidden_size)
        v = self.V(values)   # (batch_size x seq_len x hidden_size)

        # Compute scaled dot product attention
        unnormalized_attention = torch.bmm(q, k.transpose(1, 2)) * self.scaling_factor  # (batch_size x k x seq_len)

        # Apply softmax to get attention weights
        attention_weights = self.softmax(unnormalized_attention)  # (batch_size x k x seq_len)

        # Compute the context vector as the weighted sum of the values
        context = torch.bmm(attention_weights, v) # (batch_size x k x hidden_size)
        return context, attention_weights
```

2.1.2

```python
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 (with the help from ChatGPT)
    # ------------
    q = self.Q(queries)  # (batch_size x k x hidden_size)
    k = self.K(keys)     # (batch_size x seq_len x hidden_size)
    v = self.V(values)   # (batch_size x seq_len x hidden_size)

    # Scaled dot product
    unnormalized_attention = torch.bmm(q, k.transpose(1, 2)) / self.scaling_factor  # (batch_size x k x seq_len)

    # causal mask s
    seq_len = unnormalized_attention.size(-1)
    mask = torch.tril(torch.ones(seq_len, seq_len)) == 0  #
    mask = mask.to(unnormalized_attention.device)
    unnormalized_attention = unnormalized_attention.masked_fill (mask, self.neg_inf)

    attention_weights = self.softmax(unnormalized_attention)  # (batch_size x k x seq_len)

    # Compute the context vector
    context = torch.bmm(attention_weights, v)  # (batch_size x k x hidden_size)

    return context, attention_weights
```

2.1.3

The attention itself is orderless, but the characters in one word has the nature of order. By adding a unique positional encoding to each character embedding, the output of model now will be different for the same set of characters but different order, which can help the model to learn the sequential information within the data.

This sin/cos position embedding can allow the model to generalize to sequence lengths it hasn't seen before, because it encodes positions by capturing relative distances between tokens, rather than fixed indices. Hence, this relative embedding makes the extrapolating possible, because any length of input now can use this way.

2.1.4

Best validation loss

	Small dataset	**Large dataset**
Small model	1.080198118319878	0.8789122584462166
Large model	0.7387400166346476	0.6220258412882685

Effect of increasing dataset size:

- Iterations: The training loss tends to decrease more slowly for the larger dataset

compared to the smaller dataset, because more training data will simply bring more loss. Also, the training process will also slow down because the compute resource is fixed.

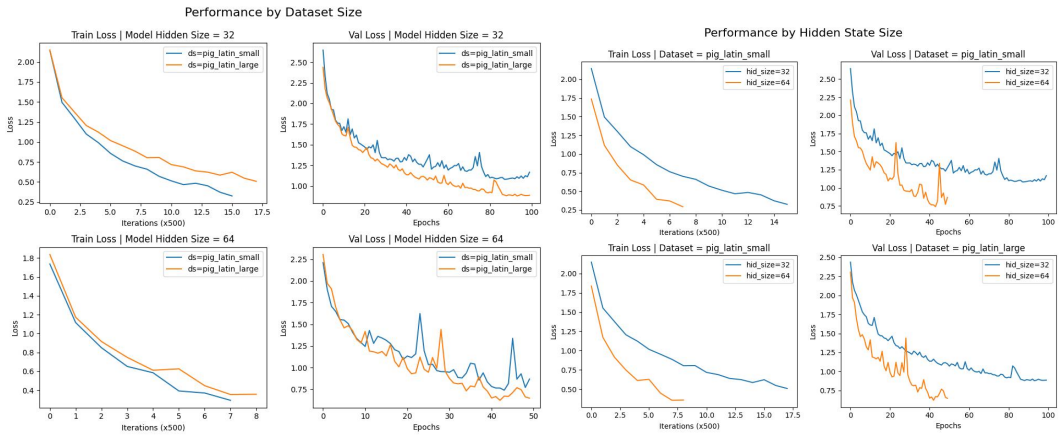
- Generalization: Model trained by larger dataset yields a lower validation loss. This indicates that a larger dataset helps the model to generalize better. More training examples can help the model to reduces the effect of the noise (from Question 1), which can reduce overfitting.

Effect of increasing model size

- Iterations: For both small and large dataset, the training loss and validation loss are lower with larger model compared to smaller model. This indicates the model now can learn the training data more effectively and make more rapid progress per iteration.
- Generalization: The validation loss also decreases with a larger hidden size. This indicates generalization also becomes better. One possible reason is that larger model can capture more types of patterns in the data by having more parameters.

These results align with my expectation:

- Larger hidden size helps the training loss to converge faster in terms of iterations, because it improves model's ability in learn more information in every iteration by having more trainable parameters.
- Larger dataset size helps the model to generalize better, because more wider variety of examples is fed into the model. Then the model can have a better performance on unseen data. In other words, the model will have lower validation loss.



2.2.1

```python
def generate_tensors_for_training_decoder_nmt(src_EOP, tgt_EOS, start_token, cuda):
    # ------------
    # FILL THIS IN (Help by ChatGPT)
    # ------------
    # 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)

    # Step 2:
    sos_vector = torch.LongTensor([start_token])
    sos_vector = sos_vector.squeeze()
    sos_vector = sos_vector.expand(src_EOP.size(0), 1)
    sos_vector = to_var(sos_vector, cuda)

    # Step 3: Concatenate SOS vector with src_EOP and shifted tgt_EOS for the decoder input
    SOS_src_EOP_tgt = torch.cat((sos_vector, src_EOP, tgt_EOS[:, :-1]), dim=1)

    return SOS_src_EOP_tgt, src_EOP_tgt_EOS
```

2.2.2

```python
    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. (b
            attentions: The stacked attention weights applied to the encoder annotations (batch_size x encoder_seq_len x deco
        """
        # ------------
        # FILL THIS IN  (Help by ChatGPT)
        # ------------
        batch_size, seq_len = inputs.size()

        encoded = self.embedding(inputs)  # batch_size x seq_len x hidden_size

        # Add positinal embeddings from self.create_positional_encodings.
        encoded = encoded + self.positional_encodings[:seq_len]

        annotations = encoded
        self_attention_weights =[]

        for i in range(self.num_layers):
            new_annotations, layer_attention_weights = self.self_attentions[i](
                annotations, annotations, annotations
            )  # batch_size x seq_len x hidden_size
            residual_annotations = annotations + new_annotations
            new_annotations = self.attention_mlps[i](residual_annotations)

            annotations = residual_annotations + new_annotations
            self_attention_weights.append(layer_attention_weights)

        output = self.out(annotations)
        self_attention_weights = torch.stack(self_attention_weights, dim=0)
        return output, self_attention_weights
```

2.2.3

Output by decoder-only:

```
source:         the air conditioning is working
translated:     ithathay away ongday iway oray
```

Output by encoder-decoder:

```
source:         the air conditioning is working
translated:     ethay airway onditiningway isway orkingway
```

We can see the quality of the output from decoder-only model is actually worse.

Pros:

- Decoder-only model has a simpler structure, which makes training process faster.
- Also, since there are less parameters, the computation cost is also less.

Cons:

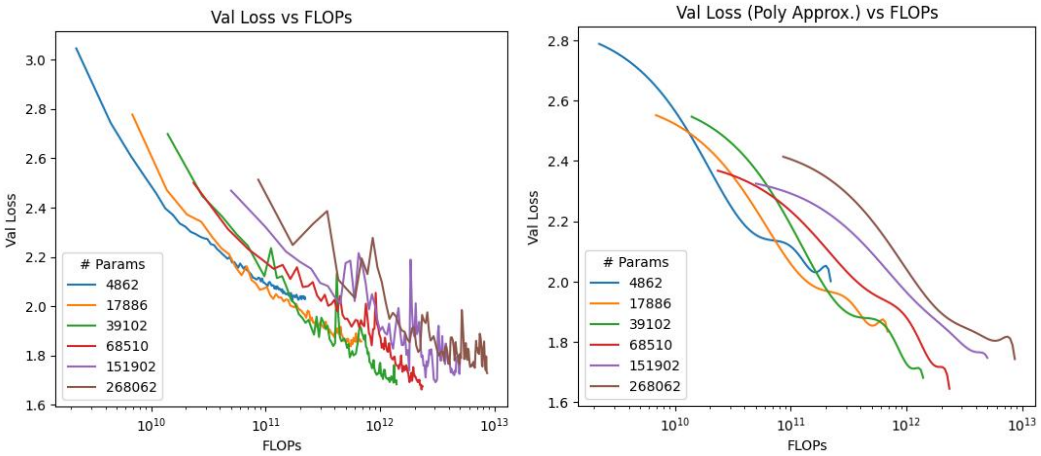
- Decoder-only model generates the output token based on the previously generated tokens. The errors made early in the sequence may propagate and affect the future

output

2.3.1

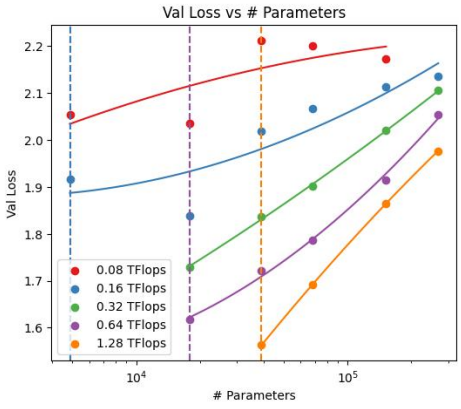
In general, for any model, the validation loss decreases as FLOPS increases. Also, the initial value of FLOP for larger model is higher because more computation is required for training larger model from the beginning.

However, Larger models do not always have a smaller validation loss when FLOPS is fixed. For low FLOP budgets, smaller models may actually outperform larger ones, showing that optimal model size is dependent on the compute resources available.



2.3.2

```python
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
    # ------------
    # Fit a quadratic polynomial (degree 2) to the data in log space
    logx=np.log10(x)
    logy=np.log10(y)
    p = np.polyfit(logx, y, 2)


    optimal_params = -p[1] / (2 * p[0])

    optimal_params = 10**optimal_params
    if optimal_params < min(x):
        optimal_params = min(x)
    elif optimal_params > max(x):
        optimal_params = min(x)



    return p, optimal_params
```



2.3.3

```python
def fit_linear_log (x, y):
    # ------------
    # FILL THIS IN
    # ------------
    log_x = np.log10(x)
    log_y = np.log10(y)


    m, c = np.polyfit(log_x, log_y, 1)


    return m, c
```

From the output:

$$\text{Log(optimal \#parameters)}=0.7894 * \text{Log}_{10}(\text{FLOPS}) -4.9885$$

Under this budget,

$$\text{FLOPs}=1e15$$

$$\text{Log}_{10}(\text{FLOPs}) = 15$$

Then

$$\text{Log(optimal \#parameters)} = 0.7894*15 -4.9885 = 6.8$$

Then

$$\text{optimal \#parameters} = 10^{6.8}$$

Then the optimal number of parameters with this budget is $10^{6.8}$

2.3.4

From ‘Training and evaluation code’ in part 2, there is a line:

```python
print(
    "Epoch: {:3d} | Flops (T): {}, Tokens (M) {}, Train Loss {:.3f}, Val Loss {:.3f} | Gen: {:20s}".format(
        epoch, total_flops/1e12, total_tokens/1e6,
        mean_train_loss, mean_val_loss, gen_string)
)
```

Output of 2.2.3 shows:

‘Flops (T): 8.6272094844, Tokens (M): 5.0001’

So, for the model in 2.2.3

$$\text{FLOPS} = 8.6272^{12} = 169998121950 = 1.7 * 10^{11} \text{ (we assume this is our budget)}$$

$$\text{\#Parameters} = 268062 = 2.68 * 10^{5} \text{ (from the plot in Step 1)}$$

$$\text{\#Tokens}= 5.0001^{6}=15626=1.5* 10^{4}$$

$$\text{Now, } \log_{10}(\text{FLOPS}) = 11$$

Then, by the equations:

$$\text{Log(optimal \#parameters)}=0.7894 * \text{Log}_{10}(\text{FLOPS}) -4.9885 = 3.7$$

$$\text{Log(optimal \#tokens)} = 0.3009 * \text{Log}_{10}(\text{FLOPS}) + 3.0486 = 6.3$$

$$\text{optimal \#parameters} = 1* 10^{3.7}$$

$$\text{optimal \#tokens} = 1* 10^{6.3}$$

The optimal number of parameters is $1* 10^{3.7}$,and the optimal number of tokens is $1* 10^{6.3}$, with this budget (FLOPS = $1.7 * 10^{11}$). The model in 2.2.3 should have less parameters and more tokens.

3.1

```python
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, 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_representation = outputs.pooler_output
        logits = self.classifier (cls_token_representation)
        ##### END YOUR CODE HERE #####
        if labels is not None:
            outputs = (logits, self.loss(logits, labels))
        else:
            outputs = (logits,)
        return outputs
```

3.3

- When the pretrained weights are frozen, the training time (1min 1sec) is a lot shorter. Only the classifier's weights are trained during backpropagation, which significantly reduces the number of computations.
- When the pretrained weights are frozen, the validation accuracy is lower, because the contextualized representation in the last layer can not be adapted to capture the special patterns and difference of the verbal arithmetic dataset.

3.4

when the pretrained weights are from BERTweet, the validation accuracy is lower.
BERTweet was pre-trained on the dataset consisted of tweets, which is very irrelevant from the arithmetic, so this model is not able to capture the important nuances of mathematical calculation in the generated contextualized representation.

4.2

caption = ‘clownfish’
The search process is easy, because it is very clear that the class of the object in the image is clownfish.