CSC413 A3 Writeup

Richard Yan

March 2023

1 Robustness and Regularization

1.1 Adversarial Examples

1.1.1 Bounding FGSM

1.1.2 Prediction under Attack

$$\begin{aligned} f(\mathbf{x}^{'}; \mathbf{w}) &= \mathbf{w}^\top \mathbf{x}^{'} \\ &= \mathbf{w}^\top (\mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})) \\ &= \mathbf{w}^\top \mathbf{x} - \epsilon \mathbf{w}^\top \nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w}) \\ &= \mathbf{w}^\top \mathbf{x} - \epsilon \mathbf{w}^\top \mathbf{w} \\ &= \mathbf{w}^\top \mathbf{x} - \epsilon \|\mathbf{w}\|^2 = f(\mathbf{x}; \mathbf{w}) - \epsilon \|\mathbf{w}\|^2 \end{aligned}$$

The answer from ChatGPT in the handout put in the y target label, but that's omitted in our handout.

1.2 Gradient Descent and Weight Decay

1.2.1 Toy Example

1.2.2 Closed Form Ridge Regression Solution

$$\begin{aligned} \nabla_{\mathbf{w}} \frac{1}{2n} \|X\mathbf{w} - \mathbf{t}\|_2^2 + \lambda \|\mathbf{w}\|_2^2 &= \frac{1}{n} X^\top (X\mathbf{w} - \mathbf{t}) + 2\lambda \mathbf{w} \\ \mathbf{w}^*_{ridge} &= (X^\top X - 2n\lambda \mathbf{I})^{-1} X^\top \mathbf{t} \end{aligned}$$

The answer from ChatGPT in the handout somehow miss a scalar 2 when deriving the gradient of the regularization term.

1.2.3 Adversarial Attack under Weight Decay

1.2.4 The Adversary Strikes Back

2 Trading off Resources in Neural Net Training

2.1 Effect of batch size

2.1.1 Batch size vs. learning rate

(a) As batch size increase, the minibatch gradient $g_B(w)$ will be closer to the true gradient, because variance will decrease as batch size increase, while mean is the true gradient. Hence as batch size increase, learning rate can tend to increase as well, because the minibatch gradient is closer to the true gradient, hence larger learning rate can be used without being affected by the noise at a large scale.

2.1.2 Training steps vs. batch size

(a) Point C is most efficient, because when batch size is smaller than C (e.g. point A), increasing batch size reduce training steps proportionally (usually because our compute can run bigger batch), while when batch size is bigger than C (e.g. point B), increasing batch size didn't have a significant reduce in training steps (usually because that's the maximum batch size our compute can hold).

(b) A is curvature dominated, and to accelerate training we could use higher order optimizers. B is noise dominated, and to accelerate training we could seek parallel compute.

2.2 Model size, dataset size and compute

(a) Increase the model size is the best option, because same batch size and more steps will be training on a model that has plateaued loss for more steps (since it already has somewhat adequate performance), which probably won't have a lot of positive impact on test performance (maybe even little negative impact for overfitting), and same steps but larger batch size will be used to achieve faster convergence (less training steps needed), which also is training on a plateaued loss, and not going to impact test performance as well.

3 Neural machine translation (NMT)

3.1 Transformers for NMT (Attention Is All You Need)

1.

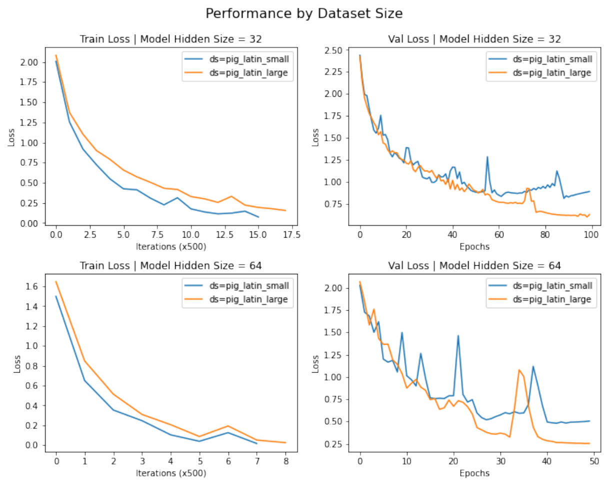
```python
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=2)
        self.scaling_factor = torch.rsqrt(
            torch.tensor(self.hidden_size, dtype=torch.float)
        )

    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]
        q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        k = self.K(keys)
        v = self.V(values)
        unnormalized_attention = q @ k.transpose(1, 2) * self.scaling_factor
        attention_weights = self.softmax(unnormalized_attention).transpose(1, 2)
        context = attention_weights.transpose(1, 2) @ v
        return context, attention_weights
```

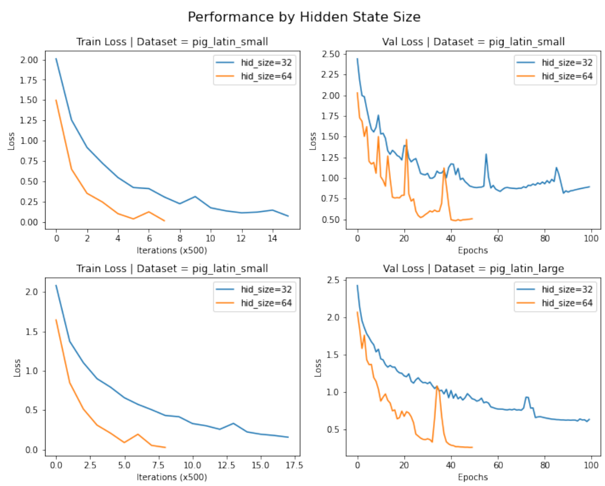
2.

```python
class CausalScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(CausalScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size
        self.neg_inf = torch.tensor(-1e7)

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=2)
        self.scaling_factor = torch.rsqrt(
            torch.tensor(self.hidden_size, dtype=torch.float)
        )

    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]
        q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        k = self.K(keys)
        v = self.V(values)
        unnormalized_attention = q @ k.transpose(1, 2) * self.scaling_factor
        mask = torch.tril(unnormalized_attention)
        unnormalized_attention[mask == 0] += self.neg_inf
        attention_weights = self.softmax(unnormalized_attention).transpose(1, 2)
        context = attention_weights.transpose(1, 2) @ v
        return context, attention_weights
```

3. Using positional encoding can let our model know the information of the position of a token in the sequence, and that is useful for example, a word in the start of sequence has different POS than the same word being in the middle. The advantage of using this positional encoding method rather than one-hot encoding is that it generalize better when encountering longer sequence input than what has been encountered in the training set, and also that using one-hot encoding will learn better the earlier positions because they are encountered more (hence biased).

4.





Lowest validation loss for:
h=32, data=small: 0.8132491590789496
h=32, data=large: 0.6012894445485336
h=64, data=small: 0.48120205478223327
h=64, data=large: 0.2535756330960678

3.2 Decoder Only NMT

1.

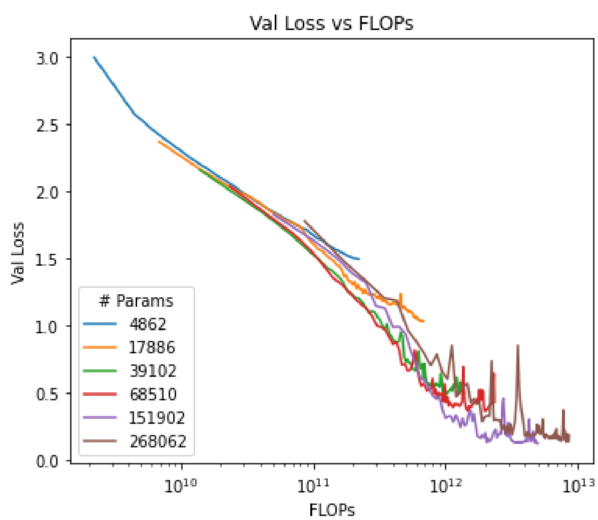
```
def generate_tensors_for_training_decoder_nmt(src_EOP, tgt_EOS, start_token, cuda):
    # ------------
    # FILL THIS IN
    # ------------
    # Step1: concatenate input_EOP, and target_EOS vectors to form a target tensor.
    src_EOP_tgt_EOS = torch.cat((src_EOP, tgt_EOS), dim=1)
    # Step2: make a sos vector
    sos_vector = torch.Tensor([start_token]).expand(src_EOP.shape[0], 1).long()
    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, tgt_EOS[:, :-1]), dim=1)
    return SOS_src_EOP_tgt, src_EOP_tgt_EOS
```

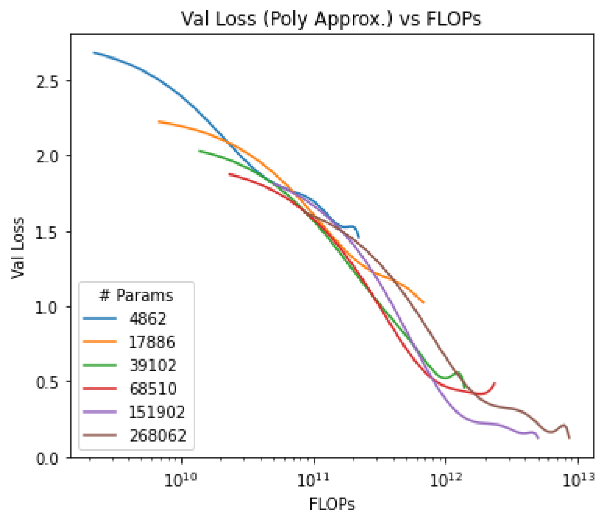
2.
```python
def forward(self, inputs):
    """Forward pass of the attention-based decoder RNN.

    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. (batch_size x decoder_seq_len x vocab_size)
        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()
    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
```

3. The pros are that the training time is pretty much faster than the encoder-decoder version, but the cons are that the accuracy aren't as good as the encoder-decoder version. But considering the fact that we only use the small vocabulary in this training, comparing with the encoder-decoder version that use small vocabulary as well, the result is actually a little bit better (may because of twice hidden size). It's not as good as those translation based on the large vocab set. The lowest validation loss indeed place between the models that use small and large vocab size (h=64).

3.3 Scaling Law and IsoFLOP Profiles

1.





In general, when number of FLOPs aren't too big, increasing # FLOPs indeed get lower validation loss. But when # FLOPs reach a certain amount, there isn't much affect on validation loss anymore. So larger model isn't necessarily always better, but when model is relatively small, increasing its size will give better result.

2.

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

3.

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

The optimal # params will be 2e7 for 1e15 FLOPs, and that's based on the best fitted line on the plot.

4. We used 8T FLOPs, so according to the plots, we should have 1e6 params and 1e7 tokens. So we should both decrease the amount of tokens and amount of parameters.

4 Fine-tuning Pretrained Language Models (LMs)

1.

```python
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 = torch.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_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
```

2.

3. The training time significantly reduced when the BERTs weights are frozen, and this is simply because we don't need to backprop all the way into BERT to finish an epoch, rather we only need to backprop until the new (and the last) classifier linear layer to finish an epoch.
However, the validation accuracy is significantly low when BERTs weights are frozen, and this is because BERT model isn't built for our task in the previous hand, so it's probably not going to have the CLS token output for what we desire (maybe contained a lot of other information that we don't need), so fine-tuning on BERTs weights significantly increase the performance.

4. A little lower performance than fine-tuning on MathBERT, but much higher than freezing BERTs weights. This may be because of the pre-training data from MathBERT is too specific on math data (which is highly suitable for our task), where tweets data aren't really having a lot of relation with the task we are performing, hence lower performance.

5.

5 Connecting Text and Images with CLIP

1.

2. Actually got it in the first try, seems pretty easy, but maybe with a larger data set it will be much harder. The prompt is "crownfish in front of coral".