Joseph Blom and Grayson Snyder
MSAI337: Deep Learning for Natural Language Processing
Professor David Demeter
3 June 2025

Homework 3

1) f) Accuracies: Pre: 0.26 Test: 0.64 Valid: 0.66. Dropout rate 0.1, 768 in features in linear classifier outputting to one logit per choice, 185 trainable parameters → ~110M fine-tuned parameters, total training time: 3m17.5s, inference time: 2.5s

g) This approach used the bert-base-uncased model with a dropout layer and linear classification head taking 768 dimension and outputting a single logit. We used the adam optimizer and out loss function was cross-entropy over softmax-normalized logits.

In terms of limitations of the approach, one is the processing of each choice independently. It could improve by doing a single sequence with [SEP] distinguishing options. The fixed maximum length could also cause long sentences to be prematurely truncated, so adjusting sequence length would be necessary.

2) a) Accuracies: Pre: 0.22 Test: 0.44 Valid: 0.46. Dropout rate 0.1, 768 in features in linear classifier outputting to one logit per choice, 68 trainable parameters → ~1.19M fine-tuned parameters, total training time: 1m12.5s, inference time: 2.5s

b) This approach used a model architecture that started with the pretrained bert-base-uncased, freezing the original BERT parameters and inserting the adapter modules after each transformer layer. The adapters consist of down-projection with ReLU, up-projection, layer normalization, and residual connections.

c) Adapter module:

```
class Adapter(nn.Module):
    def __init__(self, hidden_size, bottleneck_size=64):
        super().__init__()
        self.down_project = nn.Linear(hidden_size, bottleneck_size)
        self.activation = nn.ReLU()
        self.up_project = nn.Linear(bottleneck_size, hidden_size)
        self.layer_norm = nn.LayerNorm(hidden_size)

    def forward(self, x):
        residual = x
        x = self.down_project(x)
        x = self.activation(x)
        x = self.up_project(x)
        return self.layer_norm(x + residual)
```

Adapter Injection:

```python
class BertWithAdapters(nn.Module):
    def __init__(self, adapter_bottleneck=64):
        super().__init__()
        config = BertConfig.from_pretrained("bert-base-uncased")
        self.bert = BertModel.from_pretrained("bert-base-uncased", config=config)

        # Freeze all BERT parameters
        for param in self.bert.parameters():
            param.requires_grad = False

        self.adapters = nn.ModuleList(
            [Adapter(config.hidden_size, adapter_bottleneck) for _ in range(config.num_hidden_layers)]
        )

        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(config.hidden_size, 1)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask, output_hidden_states=True)
        hidden_states = list(outputs.hidden_states)  # List of layer outputs

        # Inject adapters into each layer's output
        for i in range(len(self.adapters)):
            hidden_states[i + 1] = self.adapters[i](hidden_states[i + 1])

        # Use the [CLS] token from the last adapted hidden state
        pooled_output = hidden_states[-1][:, 0, :]
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        return logits.squeeze(-1)
```

d) The adapter fine-tuning had much less trained parameters - ~1.19M vs ~110M. The accuracy was worse by ~20%, but training time was about half as long, though the inference time was not noticeably different. This method of fine-tuning is lower performance but more efficient with its smaller parameter usage.

3) a) Accuracies: Pre: 0.26 Test: 0.50 Valid: 0.51. Dropout rate 0.1, LoRA Dropout 0.05, 768 in features in linear classifier outputting to one logit per choice, 46 trainable parameters → ~0.2M fine-tuned parameters, total training time: 2m18s, inference time: 2.5s

b) This approach used a model architecture that started with the pretrained bert-base-uncased, freezing the original BERT parameters and inserting the LoRA modules only to the query and value projections in self-attention layers. LoRA was configured with default rank and alpha, a dropout of 0.05, and applied only to query and value projections.

c) LoRA Implementation

```python
class LoRALinear(nn.Module):
    def __init__(self, original_linear, r=4, alpha=16, dropout=0.0):
        super().__init__()
        self.original = original_linear  # Keep original Linear layer
        self.r = r
        self.alpha = alpha
        self.scaling = alpha / r
        self.dropout = nn.Dropout(dropout) if dropout > 0.0 else nn.Identity()

        in_features = original_linear.in_features
        out_features = original_linear.out_features

        # Freeze original layer
        for param in self.original.parameters():
            param.requires_grad = False

        self.lora_A = nn.Parameter(torch.randn((r, in_features)) * 0.01)
        self.lora_B = nn.Parameter(torch.randn((out_features, r)) * 0.01)

    def forward(self, x):
        return self.original(x) + (self.dropout(x) @ self.lora_A.T @ self.lora_B.T * self.scaling)
```

LoRA Application:

```python
def apply_lora_to_self_attention(attention, r=4, alpha=16, dropout=0.0):
    for proj_name in ['query', 'value']:
        original_proj = getattr(attention, proj_name)
        lora_wrapped = LoRALinear(original_proj, r=r, alpha=alpha, dropout=dropout)
        setattr(attention, proj_name, lora_wrapped)


class BertDataset(Dataset):…


class BertClassifier(nn.Module):
    def __init__(self, r=4, alpha=16, dropout=0.1, lora_dropout=0.05):
        super().__init__()
        self.bert = BertModel.from_pretrained("bert-base-uncased")
        self.dropout = nn.Dropout(dropout)
        self.classifier = nn.Linear(768, 1)

        # Freeze all pretrained BERT parameters
        for param in self.bert.parameters():
            param.requires_grad = False

        # Inject LoRA into attention layers
        for layer in self.bert.encoder.layer:
            apply_lora_to_self_attention(layer.attention.self, r=r, alpha=alpha, dropout=lora_dropout)
```

d) The LoRA fine-tuning had much less trained parameters - ~0.2M vs ~110M. The accuracy was worse by ~14%, but training time and inference time was not noticeably different. Despite much smaller parameter space, the performance is not awful in comparison, especially when considering the amount of resources that were used to train each model.

4) a) Accuracies: Pre: 0.26 Test: 0.33 Valid: 0.36. Dropout rate 0.1, 3 trainable parameters → 4.6K fine-tuned parameters, total training time: 2m13s, inference time: 2.5s

b) This approach used a model architecture that started with the pretrained bert-base-uncased, freezing the original BERT parameters and adding learnable prefix embeddings with a linear classifier. Prefix tokens were concatenated at the beginning of input sequences and the attention mask was extended to include them. The CLS token position also was adjusted so it would skip the prefix.

c) Prefix Embedding Implementation:

```python
class PrefixTunedBERT(nn.Module):
    def __init__(self, prefix_length=5):
        super().__init__()
        self.prefix_length = prefix_length
        self.bert = BertModel.from_pretrained("bert-base-uncased")
        self.config = self.bert.config
        for param in self.bert.parameters():
            param.requires_grad = False  # freeze BERT

        self.prefix_embeddings = nn.Embedding(prefix_length, self.config.hidden_size)
        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(self.config.hidden_size, 1)

    def forward(self, input_ids, attention_mask):
        batch_size = input_ids.size(0)

        # Create prefix token ids: (batch_size, prefix_length)
        prefix_tokens = torch.arange(self.prefix_length, device=input_ids.device).unsqueeze(0).expand(batch_size, -1)

        # Get prefix embeddings: (batch_size, prefix_length, hidden_size)
        prefix_embed = self.prefix_embeddings(prefix_tokens)

        # Get input embeddings: (batch_size, seq_len, hidden_size)
        inputs_embeds = self.bert.embeddings(input_ids)

        # Concatenate: (batch_size, prefix_length + seq_len, hidden_size)
        inputs_embeds = torch.cat([prefix_embed, inputs_embeds], dim=1)

        # Update attention mask: (batch_size, prefix_length + seq_len)
        prefix_attention_mask = torch.ones(batch_size, self.prefix_length, device=attention_mask.device)
        attention_mask = torch.cat([prefix_attention_mask, attention_mask], dim=1)

        # Pass through BERT using inputs_embeds directly
        outputs = self.bert(inputs_embeds=inputs_embeds, attention_mask=attention_mask)
        cls_output = outputs.last_hidden_state[:, self.prefix_length, :]  # skip prefix, use real CLS

        logits = self.classifier(self.dropout(cls_output))
        return logits.squeeze(-1)
```

Prefix Embedding Application:

```python
def train_model(model, dataloader, optimizer, criterion, epoch):
    model.train()
    total_loss = 0
    start_time = time.time()
    for batch in dataloader:
        input_ids = batch['input_ids'].view(-1, batch['input_ids'].size(-1)).to(device)
        attention_mask = batch['attention_mask'].view(-1, batch['attention_mask'].size(-1)).to(device)
        labels = batch['label'].to(device)

        optimizer.zero_grad()
        logits = model(input_ids, attention_mask)

        logits = logits.view(-1, 4)

        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    epoch_time = time.time() - start_time
    print(f"Epoch {epoch} - Loss: {total_loss/len(dataloader):.4f} - Time: {timedelta(seconds=epoch_time)}")
    return epoch_time
```

d) The prefix tuning was vastly more parameter efficient than the full fine-tuning, using less than one hundredth of a percent of the parameters. Unfortunately, the accuracy was almost half that of the full fine-tuning. There was not a major difference in training or inference time. This method of fine-tuning performed the worst out of all the options attempted.

Model:	Pre-training Accuracy:	Testing Accuracy:	Validation Accuracy:
Baseline	0.26	0.64	0.66
Adapter	0.22	0.44	0.46
LoRA	0.26	0.50	0.51
Prefix	0.26	0.33	0.36