ECE661 Homework 3: RNN and Transformers

Yiming Mao

February 27, 2025

1 True/False Question

1.1: T. Long Short-Term Memory (LSTM) networks use three gates—forget gate, input gate, and output gate—to regulate the flow of information. The forget gate specifically determines how much of the past cell state should be retained or discarded. However, while the input gate and cell update mechanism can also influence the state dynamics, they can can achieve effects in the exact same way.

1.2: T. The time complexity of single-head attention is $O(n^2d)$, and in multi-head attention, with h heads, the complexity remains $O(n^2d)$ due to parallel computation.

1.3: T. BPE is an unsupervised subword tokenization algorithm that starts with individual characters and gradually merges the most frequent adjacent character pairs into subwords. Traditional word-level tokenization has a large vocabulary, leading to more memory usage and out-of-vocabulary (OOV) issues when encountering unseen words.

1.4: F. In transformers, positional embeddings are **added** (not multiplied) to token embeddings to encode positional information.

1.5: F. In an encoder-decoder transformer, the encoder outputs act as the **Key** and **Value**, while the decoder-generated representation is the **Query** in the second multi-head attention block.

1.6: F. Tokens are the basic units of input and output in a language model. In natural language processing tasks, tokens typically represent words, subwords, or characters.

1.7: F. In transformer models, each token undergoes a linear projection to compute the Query (Q), Key (K), and Value (V) matrices using weight matrices of size d×d. The complexity for computing Q, K, V is $O((N+M)d^2)$, not $O((N+M)d)$.

1.8: T. FlashAttention improves the computational efficiency of multi-head attention by optimizing memory locality and reducing redundant operations.

1.9: F. Humans do not directly modify the LLM weights; instead, they provide preference data used to train a reward model. The actual training of the LLM is handled by reinforcement learning algorithms (e.g., PPO), not by direct human involvement. Human feedback is indirectly used to guide the model rather than manually adjusting weights.

1.10: F. KV-Cache is crucial for efficient LLM training and inference, reducing redundant computations by storing past key-value pairs.

2 Lab (1): Recurrent Neural Network for Sentiment Analysis

2(a):

Code:

```
def load_imdb(base_csv:str = './IMDBDataset.csv'):
    """
    Load the IMDB dataset
    :param base_csv: the path of the dataset file.
    :return: train, validation and test set.
    """
    # Add your code here.
    df = pd.read_csv(base_csv)
```

```
    X = df['review']
    y = df['sentiment']
    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
        random_state=12)
    x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train, test_size
        =0.125, random_state = 12)


    print(f'shape of train data is {x_train.shape}')
    print(f'shape of test data is {x_test.shape}')
    print(f'shape of valid data is {x_valid.shape}')
    return x_train, x_valid, x_test, y_train, y_valid, y_test

x_train, x_valid, x_test, y_train, y_valid, y_test = load_imdb()
```

Output:

```
shape of train data is (35000,)
shape of test data is (10000,)
shape of valid data is (5000,)
```

2(b):

Code:

```
def build_vocab(x_train:list, min_freq: int=5, hparams=None) -> dict:
    """
    build a vocabulary based on the training corpus.
    :param x_train:  List. The training corpus. Each sample in the list is a string
        of text.
    :param min_freq: Int. The frequency threshold for selecting words.
    :return: dictionary {word:index}
    """
    # Add your code here. Your code should assign corpus with a list of words.

    all_words = []
    for sample in x_train:
        sample = sample.lower()
        sample = sample.translate(str.maketrans('', '', string.punctuation))
        words = sample.split()
        all_words.extend(words)

    corpus = Counter(all_words)
    corpus = {word: freq for word, freq in corpus.items() if word.lower() not in
        stopwords.words('english')}
    corpus_ = [word for word, freq in corpus.items() if freq >= min_freq]
    # creating a dict
    vocab = {w:i+2 for i, w in enumerate(corpus_)}
    vocab[hparams.PAD_TOKEN] = hparams.PAD_INDEX
    vocab[hparams.UNK_TOKEN] = hparams.UNK_INDEX
    return vocab
```

2(c):

Code:

```
def tokenize(vocab: dict, example: str)-> list:
    """
    Tokenize the give example string into a list of token indices.
    :param vocab: dict, the vocabulary.
```

```
    :param example: a string of text.
    :return: a list of token indices.
    """
    # Your code here.
    example = example.lower()
    example = example.translate(str.maketrans('', '', string.punctuation))
    tokens = example.split()
    unk_index = vocab.get("<UNK>", 1)
    token_indices = [vocab.get(word, unk_index) for word in tokens]

    return token_indices
```

2(d):

Code:

```
    def __getitem__(self, idx: int):
        review = self.x[idx]
        label = self.y[idx]

        token_ids = tokenize(self.vocab, review)

        if len(token_ids) > self.max_length:
            token_ids = token_ids[:self.max_length]

        binary_label = 1 if label == "positive" else 0

        return {
            'ids': token_ids,
            'length': len(token_ids),
            'label': binary_label
        }
```

2(e):

Code:

```
    def __init__(
        self,
        vocab_size: int,
        embedding_dim: int,
        hidden_dim: int,
        output_dim: int,
        n_layers: int,
        dropout_rate: float,
        pad_index: int,
        bidirectional: bool = False,
        **kwargs):

        super().__init__()

        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim,
            padding_idx=pad_index
        )

        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=n_layers,
```

```
            batch_first=True,
            bidirectional=bidirectional,
            dropout=dropout_rate if n_layers > 1 else 0
        )

        lstm_output_dim = hidden_dim * 2 if bidirectional else hidden_dim
        self.fc = nn.Linear(lstm_output_dim, output_dim)

        self.dropout = nn.Dropout(dropout_rate)

        if "weight_init_fn" not in kwargs:
            self.apply(init_weights)
        else:
            self.apply(kwargs["weight_init_fn"])

    def forward(self, ids: torch.Tensor, length: torch.Tensor):
        embedded = self.embedding(ids)

        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, length.cpu(),
            batch_first=True, enforce_sorted=False)
        packed_output, (hidden, _) = self.lstm(packed_embedded)

        lstm_output, _ = nn.utils.rnn.pad_packed_sequence(packed_output, batch_first=
            True)

        if self.lstm.bidirectional:
            hidden = torch.cat((hidden[-2], hidden[-1]), dim=1)
        else:
            hidden = hidden[-1]

        dropped = self.dropout(hidden)

        prediction = self.fc(dropped)

        return prediction
```

2(f):

Dynamics of training loss, validation loss and model prediction of the first test set:

```
epoch: 1
train_loss: 0.689, train_acc: 0.541
valid_loss: 0.676, valid_acc: 0.563
epoch: 2
train_loss: 0.558, train_acc: 0.716
valid_loss: 0.389, valid_acc: 0.835
epoch: 3
train_loss: 0.314, train_acc: 0.871
valid_loss: 0.363, valid_acc: 0.850
epoch: 4
train_loss: 0.220, train_acc: 0.917
valid_loss: 0.349, valid_acc: 0.873
epoch: 5
train_loss: 0.172, train_acc: 0.940
valid_loss: 0.401, valid_acc: 0.862

test_loss: 0.320, test_acc: 0.873

Actual Sentiment: positive
Predicted Sentiment: positive, Probability: 0.9957
```

Output:

Training loss consistently decreases as training progresse. The validation loss follows the training loss pattern until Epoch 4, but slightly increases at Epoch 5. This suggests a slight overfitting trend, where the model learns training data well but generalizes slightly worse on validation data. The test accuracy (87.3) is close to validation accuracy (86.2), indicating good generalization. The model predicted the correct sentiment with high confidence (99.57). This suggests that the model has learned effective sentiment features. Therefore, it meets my expection.

3 Lab (2) Large Langugae Model for Text Generation

3(a):

Code:

```
# your code here: load the model and tokenizer
model = AutoModelForCausalLM.from_pretrained("gpt2")
tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = model.config.eos_token_id

def generate_text(model, tokenizer, prompt, max_length):


    # your code here: tokenize the prompt
    inputs = tokenizer(prompt, return_tensors="pt", padding=True, truncation=True)
    input_ids = inputs.input_ids
    attention_mask = inputs.attention_mask

    # your code here: generate token using the model
    gen_tokens =  model.generate(input_ids, attention_mask=attention_mask, max_length
        =max_length)

    # your code here: decode the generated tokens
    gen_text = tokenizer.decode(gen_tokens[0], skip_special_tokens=True)
    print(gen_text)

generate_text(model, tokenizer, "GPT-2 is a langugae model based on transformer
    developed by OpenAI", 100)
```

Output:

GPT-2 is a langugae model based on transformer developed by OpenAI. It is a simple, fast, and scalable model of the human brain. It is based on the concept of the "brain as a machine".

The model is based on the concept of the "brain as a machine". The model is based on the concept of the "brain as a machine". The model is based on the concept of the "brain as a machine". The model is based on the

3(b):

Code:

```
tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token
```

```
# your code here: load the dataset
dataset = load_dataset("wikitext", "wikitext-2-raw-v1")

# get 10% of dataset
dataset_train = dataset["train"].select(range(len(dataset["train"]) // 10))
dataset_valid = dataset["validation"].select(range(len(dataset["validation"]) // 10))

# your code here: implement function that tokenize the dataset and set labels to be
    the same as input_ids
def tokenize_function(examples):
    tokenized = tokenizer(examples['text'], padding='max_length', truncation=True,
        return_tensors="pt")
    tokenized["labels"] = tokenized["input_ids"].clone()
    return tokenized

# your code here: tokenize the dataset (you may need to remove columns that are not
    needed)
tokenized_datasets_train = dataset_train.map(tokenize_function, batched=True,
    remove_columns=["text"])
tokenized_datasets_valid = dataset_valid.map(tokenize_function, batched=True,
    remove_columns=["text"])


tokenized_datasets_train.set_format("torch")
tokenized_datasets_valid.set_format("torch")

# your code here: create datacollator for training and validation dataset
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)

train_dataloader = DataLoader(tokenized_datasets_train, shuffle=True, batch_size=4,
    collate_fn=data_collator)
valid_dataloader = DataLoader(tokenized_datasets_valid, batch_size=4, collate_fn=
    data_collator)

# Test the DataLoader
for batch in train_dataloader:
    print(batch['input_ids'].shape)
    print(batch['attention_mask'].shape)
    print(batch['labels'].shape)
    break

print("DataLoader is working correctly!")
```

3(c):

Code:

```
def evaluate_perplexity(model, dataloader):
    model.eval()
    model.to(device)
    total_loss = 0
    total_length = 0
    loss_fn = nn.CrossEntropyLoss(reduction='sum')

    with torch.no_grad():
        for batch in dataloader:
            # your code here: get the input_ids, attention_mask, and labels from the
                batch
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
```

```
            labels = batch['labels'].to(device)

            # your code here: forward pass
            outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
            logits = outputs.logits

            # Shift so that tokens < n predict n
            shift_logits = logits[..., :-1, :].contiguous()
            shift_labels = labels[..., 1:].contiguous()

            # your code here: calculate the loss
            loss = loss_fn(shift_logits.view(-1, shift_logits.size(-1)), shift_labels
                .view(-1))

            total_loss += loss.item()
            total_length += attention_mask.sum().item()

    # Calculate perplexity
    perplexity = torch.exp(torch.tensor(total_loss / total_length))

    return perplexity.item()


perplexity = evaluate_perplexity(model, valid_dataloader)
print(f"Initial perplexity: {perplexity}")
```

Output:

```
Initial perplexity: 42.9958610534668
```

3(d):

Code:

```
# your code here: report validation and training loss every epoch
    evaluation_strategy='epoch',
    logging_strategy='epoch',
    save_strategy='epoch',
    logging_dir='./logs',
)

# your code here: create a Trainer object
trainer = Trainer(
    model=model,                    # Your GPT-2 model
    args=training_args,             # Training arguments
    train_dataset=tokenized_datasets_train, # Training dataset
    eval_dataset=tokenized_datasets_valid,  # Validation dataset
    tokenizer=tokenizer,            # GPT-2 tokenizer
)
```

fine-tuned perplexity: 27.341947555541992

Generate some text using the fine-tuned model:

GPT–2 **is** a langugae model based on transformers developed by OpenAI , **and is** based on the same basic principles as the CATAS model . The CATAS model **is** based on the CATAS model of the interaction between the two nuclei of the CNS . The CATAS model **is** based on the interaction between the CNS **and** the CNS + the interaction between the CNS **and** the CNS + the interaction between the CNS **and** the CNS + the interaction between the CNS **and** the CNS + the

3(e):

Code:

```
from peft import LoraConfig

peft_config = LoraConfig(
    lora_alpha=16,
    lora_dropout=0.1,
    r=64,
    bias="none",
    task_type="CAUSAL_LM",
)

# your code here: load GPT2 model and add the lora adapter
model = AutoModelForCausalLM.from_pretrained("gpt2")

model_lora = get_peft_model(model, peft_config)
model_lora.print_trainable_parameters()


training_args = TrainingArguments(
    output_dir="./gpt2-lora-wikitext-2",
    overwrite_output_dir=True,
    num_train_epochs=3,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    eval_steps=400,
    save_steps=800,
    warmup_steps=500,
    prediction_loss_only=True,
)

# your code here: set trainer and train the model
trainer = Trainer(
    model=model_lora,
    args=training_args,
    train_dataset=tokenized_datasets_train,
    eval_dataset=tokenized_datasets_valid,
    tokenizer=tokenizer,
)

trainer.train()

ppl = evaluate_perplexity(model_lora, valid_dataloader)
print(f"Perplexity after lora finetuning: {ppl}")
```

3(f):
Finetuning time of fully finetuning: 14m 30.4s
Finetuning time of LoRA finetuning: 8m 51.9s

Reason: LoRA finetuning updates fewer parameters instead of the full model, avoids expensive backpropagation for frozen layers and uses efficient low-rank matrix updates.

Perplexity of pre-trained GPT-2: 42.9958610534668
Perplexity of finetuned GPT-2: 27.341890335083008
Perplexity of LoRA finetuned GPT-2: 42.94930648803711

Generated text of pre-trained GPT-2: GPT-2 is a langugae model based on transformer developed by OpenAI. It is a simple, fast, and scalable model of the human brain. It is based on the concept of the "brain as a machine".

The model is based on the concept of the "brain as a machine". The model is based on the concept of the "brain

as a machine". The model is based on the concept of the "brain as a machine". The model is based on the

Generated text of finetuned GPT-2:GPT-2 is a langugae model based on transformers developed by OpenAI , and is based on the same basic principles as the CATAS model . The CATAS model is based on the CATAS model of the interaction between the two nuclei of the CNS . The CATAS model is based on the interaction between the CNS and the CNS + the interaction between the CNS and the CNS + the interaction between the CNS and the CNS + the interaction between the CNS and the CNS + the

Generated text of LoRA finetuned GPT-2: GPT-2 is a langugae model based on transformers developed by OpenAI. It is a model that is based on the concept of a "transformation" of the human brain.

The model is based on the concept of a "transformation" of the human brain.

The model is based on the concept of a "transformation" of the human brain.

Reason: 1. Fully fine-tuned GPT-2 significantly lowers perplexity (from 42.99 $\rightarrow$ 27.34). This means the model learns better probability distributions from the fine-tuning dataset. 2. LoRA fine-tuned GPT-2 has almost the same perplexity as the pre-trained model (42.95 42.99). LoRA does not fully update all model parameters. It only updates small adapter layers, leading to less significant improvements in perplexity. 3. Fully fine-tuned GPT-2 produces more diverse text but with overuse of "CNS" (potential dataset bias).