ECE 661: Homework #3
RNN and Transformers

Hai Li

ECE Department, Duke University — February, 2025

Objectives

This assignment aims to provide hands-on experience with Recurrent Neural Networks (RNNs) and Large Language Models (LLMs) through two main labs:

Lab 1: Implement an LSTM model for sentiment analysis using the IMDB dataset. Students will build a data loader, create a vocabulary, implement tokenization, and construct an LSTM model for binary sentiment classification.

Lab 2: Explore Large Language Models using GPT-2. Students will generate text with a pre-trained model, prepare a dataset, evaluate model performance using perplexity, fine-tune the model, and compare full fine-tuning with LoRA (Low-Rank Adaptation) fine-tuning.

Through these labs, students will gain practical experience in natural language processing tasks, model implementation, and working with state-of-the-art language models.

Warning: You are asked to complete the assignment independently.

This lab has 100 points plus 10 bonus points, yet your final score cannot exceed 100 points. The submission deadline will be **11:55pm, Wednesday, March 5**. We provide a template named `LabRNN.ipynb` and `LabLLM.ipynb` to start with, and you are asked to develop your own code based on this template. You will need to submit two independent files including:

1. A self-contained PDF report, which provides answers to all the conceptual questions and clearly demonstrates all your lab results and observations and required code snippet. Remember, **do NOT generate PDF from your jupyter notebook to serve as the report**, which can increase the TA's burden of grading.
2. `code.zip`, a zipped code file which contains 2 jupyter notebooks `LabRNN.ipynb`, `LabLLM.ipynb`, respectively for the two labs.

Note that 20 percent of the grade will be deducted if the submissions doesn't follow the above guidance.

Note that TAs hold the right to adjust grading based on the returned homeworks. We make sure that the grading rule is consistent among all students. Also, the results given for the Labs (for example the reported accuracies) are obtained from the specific runtime when TAs were working on the answers. We do not expect you to get exactly the same numbers; yet, it is necessary that your results show the same trends/patterns/observations in order to receive full credits.

1 True/False Questions (30 pts)

For each question, please provide a short explanation to support your judgment or fix the wrong statement.

Problem 1.1 (3pts) Long short-term memory (LSTM) is able to configure how much the history information to keep. Particularly, the forget gate is designed to control how much of the previous cell state to retain. we cannot adjust the rest parameters within one LSTM cell to achieve identical effect as changing the forget gate.

True. In an LSTM, the forget gate is responsible for deciding how much of the past information should be kept or discarded. While the input and output gates regulate how new information is added and how the cell state influences the output, the forget gate specifically controls memory retention. Since each gate has a distinct function, adjusting other parameters cannot fully replace the role of the forget gate in determining how much past information remains.

Problem 1.2 (3 pts) In a simple self-attention, the time complexity is $O(n^2d)$ for query, key, and value (q, k, v) with sequence length n and dimensionality d. For multi-head attention, where the number of heads is h and the projections W^Q, W^K, $W^V \in \mathbb{R}^{d\times(d/h)}$, the time complexity remains $O(n^2d)$.

True. In multi-head attention, each head works with a smaller part of the data, with size d/h. This means the cost for each head is $O(n^2(d/h))$. Since there are h heads, the total cost becomes h x O(n² (d/h)) , which simplifies to $O(n^2d)$, because the h cancels out. So, the overall time complexity is still $O(n^2d)$, the same as for single-head attention.

Problem 1.3 (3pts) Byte Pair Encoding (BPE) is a "Bottom-up" tokenization technique and is used in GPT and BERT. Compared to traditional word-level tokenization, it reduces the vocabulary size significantly and provides better handling of unknown words.

True. BPE is a bottom-up tokenization technique used in models like GPT and BERT. Unlike traditional word-level tokenization, which splits text into fixed vocabulary words, BPE merges the most frequent pairs of characters or subword units. This reduces the overall vocabulary size and creates more flexible subword tokens. So, BPE improves the model's ability to handle unknown or out-of-vocabulary words by breaking them down into smaller units.

Problem 1.4 (3pts) In transformers, positional embedding is multiplied on the embedding of each token to represent the position relationship between tokens.

False. Transformers add positional embeddings to token embeddings instead of multiplying them. This addition helps the model understand word order while still processing tokens in parallel. Without it, transformers wouldn't capture the sequence structure.

Problem 1.5 (3pts) In an encoder-decoder transformer model, the information captured by the encoder is used as the "Query" and "Value" in the second multi-head attention block in the decoder. (Hint: Refer to Lecture 10 slides).

False. In an encoder-decoder transformer, the encoder's output is used as the Key and Value, not the Query, in the second multi-head attention block of the decoder. The Query comes from the decoder's previous layer. The decoder uses the encoder's output (Key and Value) to attend to relevant information from the input sequence. Therefore, the encoder does not provide the Query in this attention block.

Problem 1.6 (3pts) In LLMs, a token stands for a word in the input prompt or output prompt.

False. In LLMs, a token is not necessarily a whole word. Instead, a token can also be a subword, character, or punctuation mark, depending on how the tokenization is done.

Problem 1.7 (3pts) Assume an LLMs input token length is N and generated token length is M and the hidden dimension is d. In decoding phase the computation complexity for computing Q,K,V in a layer of a single forward pass is $O((N + M)d)$.

False. Computing Q, K, and V for each token requires multiplying its embedding by a weight matrix of size d × d, which takes $O(d^2)$ time per token. Since there are (N + M) tokens in total, the overall time complexity is $O((N + M) d^2)$, not O((N + M) d).

Problem 1.8 (3pts) Flash attention is proposed to improve the computation locality of multihead attention computation in LLMs.

True. Flash Attention improves efficiency by optimizing memory usage and making attention calculations faster. It reduces unnecessary memory use and speeds up computations, especially for long sequences, making it better for large models.

Problem 1.9 (3pts) In RLHF, human directly involve in the training process of LLMs, providing feedback to the LLM generated contents and train it accordingly.

True. In RLHF, humans are directly involved by providing feedback on the model's generated responses. This feedback is then used to train a reward model, which helps fine-tune the LLM to produce better outputs.

Problem 1.10 (3pts) KV-Cache is not helpful in training large language model.

True. KV-Cache is primarily used during inference, not training. It stores previously computed key-value pairs to avoid redundant computations, making generation faster and more efficient. During training, however, all tokens are processed simultaneously, so KV-Cache is not needed.

2 Lab (1): Recurrent Neural Network for Sentiment Analysis (35pts)

Google colab is a useful tool for modeling training. In order to run ipynb document, you need to upload your jupyter notebook document to google drive. Then double click the file in the google drive, which will lead you to google colab. Moreover, it'd be more efficient to use GPU to train your LSTM and Transformer models in this assignment. To use GPU, please check Runtime > Change run-time type, and select GPU as the hardware accelerator. You may click on side bar document icon to upload your dataset and python file there.

In this lab, you will learn to implement an LSTM model for sentiment analysis. Sentiment analysis [1, 2] is a classification task to identify the sentiment of language. It usually classifies data into two to three labels: positive, negative or/and neutral. IMDB [3] is one of the most widely used dataset for binary sentiment classification. It uses only positive and negative labels. IMDB contains 50,000movie review data collected from popular movie rating service IMDB. You may find more details at https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews.

To start this assignment, please open the provided Jupyter Notebook LabRNN.ipynb. We have provided implementation for the training recipe, including optimizers, learning rate schedulers, and weight initialization. You may NOT change any of the hyperparameter settings (i.e., the object HyperParams) in this lab.

Grading Instructions. For this part, we mainly grade on the logic and conciseness of the code. If a mistake is found in this part that lead to erroneous conclusions for questions in the later part, we will consider this and provide partial/full credit for the later part, to avoid applying the same penalty twice. **Please attach your added code in the report under corresponding question.** Note that TAs and Professor have the final discretion to adjust grade according to the given submission.

(a) (5 pts) Implement your own data loader function. First, read the data from the dataset file on the local disk. Then split the dataset into three sets: train, validation, and test by $7:1:2$ ratio. Finally return x_train, x_valid, x_test, y_train, y_valid and y_test, where x represents reviews and y represent labels.

I completed the load_imdb function by loading the dataset and initially splitting it into training (30%) and temporary (70%) sets. Then, the temporary set is further divided into validation (20%) and test (10%) sets, aiming for a 7:1:2 split (train:valid:test). Additionally, I transformed the "positive" labels into 1 and "negative" into 0 using np.where(). The code used is as follows:

```
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"]
    #y = np.where(df['sentiment'] == 'positive', 1, 0)
                                (variable) y_temp: Any
    x_train, x_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
    x_valid, x_test, y_valid, y_test = train_test_split(x_temp, y_temp, test_size=(0.66666666666), random_state=42)

    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
```

```
[21] data_path = '/content/drive/MyDrive/2023 - Duke/2025-01/ECE661_ComputerEngineeringMachineLearning&DeepNeuralNets/Assigments/HW3/IMDBDataset.csv'

    X_train, X_valid, X_test, y_train, y_valid, y_test = load_imdb(data_path)
```

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

(b) (5 pts) Implement the build_vocab function to build a vocabulary based on the training corpus. You should first compute the frequency of all the words in the training corpus. Remove the words that are in the STOP_WORDS. Then filter the words by their frequency (≥ min_freq) and finally generate a corpus variable that contains a list of words.

I'm using the next code:

```python
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.

    corpus = Counter()

    for review in x_train:
        words = review.split()
        words = [word.lower() for word in words]
        words = [word for word in words if word not in hparams.STOP_WORDS]

        corpus.update(words)

    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
```

```python
hparams = HyperParams()


vocab = build_vocab(X_train, min_freq=5, hparams=hparams)
vocab
```

```
{'As': 2,
 'much': 3,
 'as': 4,
 'I': 5,
 'love': 6,
 'trains,': 7,
 "couldn't": 8,
 'stomach': 9,
 'this': 10,
 'movie.': 11,
 'The': 12,
 'premise': 13,
 'that': 14,
 'one': 15,
 'could': 16,
 'steal': 17,
 'a': 18,
 'locomotive': 19,
 'and': 20,
 'from': 21,
 'to': 22,
 'Chicago': 23,
 'without': 24,
```

(c) (5 pts) Implement the tokenization function. For each word, find its index in the vocabulary. Return a list of integers that represents the indices of words in the example.

```
[23] 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.

        words = example.split()
        words = [word.lower() for word in words]
        words = [word for word in words if word not in hparams.STOP_WORDS]

        token_indices = [vocab.get(word, vocab[hparams.UNK_TOKEN]) for word in words]


        return token_indices
```

```
'''
vocab
{'much': 2,
 'love': 3,
 'trains,': 4,
 'stomach': 5,
 'movie.': 6,
 'premise': 7,
 'one': 8,
 ....
'''

example_text = "I love the premise of the movie."

token_indices = tokenize(vocab, example_text)
token_indices
```

```
[3, 7, 6]
```

(d) (5 pts) Implement the __getitem__function in the IMDB class. Given an index i, you should return the i-th review and label. The review is originally a string. Please tokenize it into a sequence of token indices. Use the max_length parameter to truncate the sequence so that it contains at most max_length tokens. Convert the label string ('positive' / 'negative') to a binary index, such as 'positive' is 1 and 'negative' is 0. Return a dictionary containing three keys: 'ids', 'length', 'label' which represent the list of token ids, the length of the sequence, the binary label.

```
class IMDB(Dataset):
    def __init__(self, x, y, vocab, max_length=256) -> None:
        """
        :param x: list of reviews
        :param y: list of labels
        :param vocab: vocabulary dictionary {word:index}.
        :param max_length: the maximum sequence length.
        """
        self.x = x
        self.y = y
        self.vocab = vocab
        self.max_length = max_length

    def __getitem__(self, idx: int):
        """
        Return the tokenized review and label by the given index.
        :param idx: index of the sample.
        :return: a dictionary containing three keys: 'ids', 'length', 'label' which represent the list of token ids, the length of the sequence, the binary label.
        """
        # Add your code here.
        review = self.x[idx]
        label = self.y[idx]

        token_indices = tokenize(self.vocab, review)
        token_indices = token_indices[:self.max_length]
        label = 1 if label == 'positive' else 0

        return {'ids': token_indices, 'length': len(token_indices), 'label': label}


    def __len__(self) -> int:
        return len(self.x)


def collate(batch, pad_index):
    batch_ids = [torch.LongTensor(i['ids']) for i in batch]
    batch_ids = nn.utils.rnn.pad_sequence(batch_ids, padding_value=pad_index, batch_first=True)
    batch_length = torch.Tensor([i['length'] for i in batch])
    batch_label = torch.LongTensor([i['label'] for i in batch])
    batch = {'ids': batch_ids, 'length': batch_length, 'label': batch_label}
    return batch

collate_fn = collate
```

(e) (5 pts) In _init_, a LSTM model contains an embedding layer, an lstm cell, a linear layer, and a dropout layer. You can call functions from Pytorch's nn library. For example, nn.Embedding, nn.LSTM, nn.Linear...

(5 pts) In forward, decide where to apply dropout. The sequences in the batch have different lengths. Write/call a function to pad the sequences into the same length. Apply a fully-connected (fc) layer to the output of the LSTM layer. Return the output features which is of size [batch size, output dim].

```
def init_weights(m):
    if isinstance(m, nn.Embedding):
        nn.init.xavier_normal_(m.weight)
    elif isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.LSTM) or isinstance(m, nn.GRU):
        for name, param in m.named_parameters():
            if 'bias' in name:
                nn.init.zeros_(param)
            elif 'weight' in name:
                nn.init.orthogonal_(param)

class LSTM(nn.Module):
    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,
        **kwargs):
        """
        Create a LSTM model for classification.
        :param vocab_size: size of the vocabulary
        :param embedding_dim: dimension of embeddings
        :param hidden_dim: dimension of hidden features
        :param output_dim: dimension of the output layer which equals to the number of labels.
        :param n_layers: number of layers.
        :param dropout_rate: dropout rate.
        :param pad_index: index of the padding token.we
        """
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments. (you can use nn.LSTM, nn.Embedding, nn.Linear, nn.Dropout)
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_index)

        self.lstm = nn.LSTM(
                  embedding_dim,
                  hidden_dim,
                  num_layers=n_layers,
                  bidirectional=bidirectional,
                  dropout=dropout_rate if n_layers > 1 else 0,
                  batch_first=True
              )


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

        self.pad_index = pad_index

        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):
        """
        Feed the given token ids to the model.
        :param ids: [batch size, seq len] batch of token ids.
        :param length: [batch size] batch of length of the token ids.
        :return: prediction of size [batch size, output dim].
        """
        # Add your code here.

        embedded = self.dropout(self.embedding(ids))
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, length.cpu().int(), batch_first=True, enforce_sorted=False)
        packed_output, (hidden, cell) = self.lstm(packed_embedded)

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

        hidden = self.dropout(hidden)
        prediction = self.fc(hidden)

        return prediction
```

(f) (5 pts) Train the LSTM model for 5 epochs with default configuration. Do you observe a steady and consistent decrease of the loss value as training progresses? Report your observation on the learning dynamics of training loss, and validation loss on the IMDB dataset. Besides, show the model prediction of the first test set. Do they meet your expectations and why?

During the training of the LSTM model, we observe a **steady decrease** in the **training loss**, which indicates that the model is successfully learning. The training loss drops from 0.675 in epoch 1 to 0.054 in epoch 5, and **the training accuracy increases from 55.6% to 98.4%.** This suggests that the model is fitting the training data well.

However, the validation loss shows some fluctuations. After an initial decrease from 0.627 in epoch 1 to 0.277 in epoch 2, it increases again to 0.465 by epoch 5. This could be a sign of overfitting, as the model's performance on the validation set is not improving consistently alongside the training set. Validation accuracy also fluctuates.

It is worth noting that 5 epochs might not be enough to fully optimize the model, especially when dealing with a complex task like sentiment analysis. Further tuning, such as increasing the number of epochs, adjusting the learning rate, or adding regularization techniques, could potentially improve the validation loss and accuracy.

Regarding the test set prediction, the model predicts "Negative" sentiment with a confidence of 0.5196. While the model correctly identifies the negative sentiment, the confidence is relatively low. This could be due to the model's uncertainty, as sentiment analysis, especially on nuanced or less clear texts, can be challenging. Given the relatively few epochs, it's reasonable to expect that further optimization could lead to higher confidence in predictions.

3 Lab (2) Large Langugae Model for Text Generation (45 pts)

In this lab, we also recommend you using Google colab to finish the model training. You will learn to use hugging face transformers to run and fine-tune large language models. Please follow the instruction in jupyter notebook to finish the task and attach your implemented code in the report under the corresponding question.

(a) (10 pts) Load the pre-trained GPT-2 and generate some text. It is recommended to use .generate() for token generation and .decode to decode token into text.

```
# your code here: load the model and tokenizer
model = AutoModelForCausalLM.from_pretrained("gpt2").to(device)
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").to(device)
    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)
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
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 (

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 bas
```

(b) (10 pts) Prepare the dataset. We are using wiki-text as test and training dataset, but it is a relative large dataset, we are only using 10% of data for practice.

```
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)
    tokenized["labels"] = tokenized["input_ids"].copy()
    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!")
```

```
README.md: 100% 10.5k/10.5k [00:00<00:00, 593kB/s]
test-00000-of-00001.parquet: 100% 733k/733k [00:00<00:00, 5.38MB/s]
train-00000-of-00001.parquet: 100% 6.36M/6.36M [00:00<00:00, 60.4MB/s]
validation-00000-of-00001.parquet: 100% 657k/657k [00:00<00:00, 44.4MB/s]
Generating test split: 100% 4358/4358 [00:00<00:00, 5596.79 examples/s]
Generating train split: 100% 36718/36718 [00:00<00:00, 358123.73 examples/s]
Generating validation split: 100% 3760/3760 [00:00<00:00, 107714.47 examples/s]
Map: 100% 3671/3671 [00:05<00:00, 735.96 examples/s]
Map: 100% 376/376 [00:00<00:00, 745.96 examples/s]
torch.Size([4, 1024])
torch.Size([4, 1024])
torch.Size([4, 1024])
DataLoader is working correctly!
```

(c) (10 pts) Implement the inference of GPT-2, and evaluate its performance. We use perplexity as the evaluation metric, which is a common metric for LLMs.

```python
def evaluate_perplexity(model, dataloader):
    model.eval()
    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(model.device)
            attention_mask = batch["attention_mask"].to(model.device)
            labels = batch["labels"].to(model.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}")
```

```
`loss_type=None` was set in the config but it is unrecognised.Using the default loss: `ForCausalLMLoss`.
Initial perplexity: 42.9958610534668
```

The initial perplexity of the pre-trained GPT-2 model on the WikiText dataset is **42.9958**. This value reflects the performance of the model without any further adjustments, as it is the model trained by OpenAI.

(d) (5 pts) Finetune the GPT-2 model on wiki-text dataset. Show how the train and validation loss change in different epoch. Hugging face provide convenient function to train a transformers, there is no need to implement your own training function. After finetuning, load the finetuned model and evaluate the perplexity and try to generate some text

Below is the code used for fine-tuning the model, along with the training and validation loss results during the three training epochs:

Lab 2(d) Fine-tune GPT2 on wiki-text

Generated code may be subject to a license | YasminSalehi/emotion-aware-chatbot

```
# Set up training arguments
training_args = TrainingArguments(
    output_dir="./gpt2-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=False,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    logging_dir="./logs",
    logging_strategy="epoch",
)

# your code here: create a Trainer object
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets_train,
    eval_dataset=tokenized_datasets_valid,
    data_collator=data_collator
)

trainer.train()
trainer.save_model()
```

```
/usr/local/lib/python3.11/dist-packages/transformers/training_args.py:1575: FutureWarning: `evaluation_strategy` is c
  warnings.warn(
wandb: WARNING The `run_name` is currently set to the same value as `TrainingArguments.output_dir`. If this was not i
wandb: Using wandb-core as the SDK backend.  Please refer to https://wandb.me/wandb-core for more information.
wandb: WARNING If you're specifying your api key in code, ensure this code is not shared publicly.
wandb: WARNING Consider setting the WANDB_API_KEY environment variable, or running `wandb login` from the command lir
wandb: No netrc file found, creating one.
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
wandb: Currently logged in as: barbara-patricia-flores-rios (barbara-patricia-flores-rios-duke-university) to https:/
Tracking run with wandb version 0.19.7
Run data is saved locally in /content/wandb/run-20250305_162600-d11ftx9h
Syncing run ./gpt2-wikitext-2 to Weights & Biases (docs)
View project at https://wandb.ai/barbara-patricia-flores-rios-duke-university/huggingface
View run at https://wandb.ai/barbara-patricia-flores-rios-duke-university/huggingface/runs/d11ftx9h
[1377/1377 12:00, Epoch 3/3]
```

Epoch	Training Loss	Validation Loss
1	3.669900	3.339453
2	3.147600	3.354737
3	2.881200	3.388916

As seen in the loss values, the training loss decreased steadily over the epochs, indicating that the model was learning and fitting the training data well. However, the validation loss began to increase after the first epoch, suggesting that the model's performance on unseen data was deteriorating. This is a common sign of overfitting.

Test fine-tuned model

```
# your code here: load the fine-tuned model
model_finetuned = AutoModelForCausalLM.from_pretrained("./gpt2-wikitext-2").to(device)
perplexity = evaluate_perplexity(model_finetuned, valid_dataloader)
print(f"fine-tuned perplexity: {perplexity}")
```

```
fine-tuned perplexity: 27.162353515625
```

Generate some text using the fine-tuned model

```
[8] # load the fine-tuned model

    tokenizer = AutoTokenizer.from_pretrained("gpt2")

    # generate text
    generate_text(model_finetuned, tokenizer, "GPT-2 is a langugae model based on transformers developed by OpenAI", 100)
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
GPT-2 is a langugae model based on transformers developed by OpenAI , and is a novel approach to the formation of a new class of microarray based on the
```

After fine-tuning the GPT-2 model on the WikiText dataset, we obtained a **perplexity of 27.162**, which indicates a significant improvement in the model's performance. This reduction in perplexity shows that the fine-tuned model has learned to generate more coherent and accurate text compared to the pre-trained model (**42.9958**.). This result demonstrates how fine-tuning has allowed the model to generate more relevant and specific text, showing an improvement in the model's ability to understand and generate coherent sequences based on the given context.

(e) (5 pts) Use LoRA to finetune the GPT-2 model. Hugging face peft has implemented LoRA, directly call the function to realize LoRA finetuning.

I'm using the next code

Lab 2(e) Parameter efficient fine-tuning (LoRA)

finetune the base gpt model through LoRA

```
from peft import LoraConfig, get_peft_model

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_lora = AutoModelForCausalLM.from_pretrained("gpt2").to(device)
model_lora = get_peft_model(model_lora, peft_config)

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=False,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    logging_dir="./logs",
    logging_strategy="epoch",
)

# 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,
    data_collator=data_collator
)


trainer.train()
trainer.save_model()

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

```
/usr/local/lib/python3.11/dist-packages/peft/tuners/lora/layer.py:1264: UserWarning: fan_in_fan_out is set to False but the target module is `Conv1D`. Setting fan_in_fan_out to
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/transformers/training_args.py:1575: FutureWarning: `evaluation_strategy` is deprecated and will be removed in version 4.46 of  Transfo
  warnings.warn(
```

[1377/1377 09:27, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	4.171500	3.743232
2	3.835000	3.551432
3	3.741100	3.520424

The training loss showed a gradual decrease over the epochs, indicating that the model was learning effectively from the training data. Specifically, the training loss decreased from **4.17** in epoch 1 to **3.74** in epoch 3.

The validation loss also improved but remained relatively stable. It decreased from **3.74** in epoch 1 to **3.52** in epoch 3. This suggests that the model was generalizing well to the validation data.

Evaluate lora fine-tuned model on wiki-text

compare the text generated by the fully fine-tuned model and LoRA fine-tuned model and the pre-trained model. Do you see any difference in the quality of the generated text? Try to explain why. (Hint: trust your result and report as it is.)

```
[10] generate_text(model_lora, tokenizer, "GPT-2 is a langugae model based on transformers developed by OpenAI", 100)
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
GPT-2 is a langugae model based on transformers developed by OpenAI , which is based on the concept of a "transformation" of the human l
```

Compare the perplexity of the fully fine-tuned model and LoRA fine-tuned model. Do you see any difference in the perplexity? Try to explain why.

```
ppl = evaluate_perplexity(model_lora, valid_dataloader)

print(f"Perplexity after lora finetuning: {ppl}")
```

```
Perplexity after lora finetuning: 30.29522132873535
```

(f) (5 pts) Compare the finetuning time of fully finetuning and LoRA finetuning, explain the reason. Besides, compare the perplexity and generated text of Pre-trained GPT-2 and finetuned, LoRA finetuned GPT-2, explain the reason.

In terms of training time, fine-tuning with LoRA took about **9 minutes** for the entire process, compared to **12 minutes** for the full fine-tuning process in part (d). This reduction in time is a key advantage of LoRA, as it achieves comparable results with fewer parameters, making the training process more efficient.
After fine-tuning with LoRA, I evaluated the perplexity, which was **30.295**, slightly higher than the perplexity of the fully fine-tuned model (**27.162**). This indicates that while LoRA fine-tuning may not achieve the exact same level of performance as full fine-tuning, it still delivers competitive results with fewer trainable parameters.
In terms of text generation, the output from the LoRA fine-tuned model was qualitatively similar to the fully fine-tuned model. The text generated by both models was coherent, but the LoRA fine-tuned model was more efficient in terms of computational resources. This highlights LoRA's ability to fine-tune large models effectively while maintaining good performance.