HW3

March 11, 2025

1 Homework 3: RNN Language Model

1.1 1A. Description and Illustration of RNN Language Model architecture and design choices

The RNN language model architecture is designed to process sequential data, such as text, and predict the next token in a sequence. The model is implemented using PyTorch and consists of several key components. The `HyperParams` dataclass encapsulates the hyperparameters, including `vocab_size`, `batch_size`, `seq_length`, `learning_rate`, `num_epochs`, `hidden_dim`, `num_layers`, `embedding_dim`, and `dropout`. These parameters are used to configure the model's behavior and structure. The `RNN` class inherits from `nn.Module` and defines the model's architecture. It includes an embedding layer (`nn.Embedding`) to convert input tokens into dense vectors of size `embedding_dim`, followed by an RNN layer (`nn.RNN`) with `hidden_dim` units and `num_layers` layers. Dropout (`nn.Dropout`) is applied to the RNN output to prevent overfitting. Finally, a fully connected layer (`nn.Linear`) maps the RNN's hidden state to the output vocabulary space. The `forward` method processes the input sequence, computes embeddings, passes them through the RNN, applies dropout, and generates logits for the next token prediction. The model also maintains a hidden state, which is updated at each time step to capture sequential dependencies.

1.2 1B. Model Hyperparameters

```
rnn_hp = HyperParams(
    model_type='rnn', # Either 'rnn' or 'lstm'
    vocab_size=10000, # Vocab size for unique tokens in the dataset
    batch_size=64, # Batch size for training, adjusted based on GPU resources
    seq_length=30, # Sequence length for truncated backpropagation through time
    learning_rate=0.0005, # Learning rate found with emperical tuning
    num_epochs=20, # Number of training epochs
    hidden_dim=256,  # Dimension of the RNN hidden state
    num_layers=1, # Number of RNN layers
    embedding_dim=100, # Embedding dimension for input tokens
    dropout=0 # No dropout for a single layer RNN
)
```

```
lstm_hp = HyperParams(
    model_type='lstm',
    vocab_size=10000,
    batch_size=64,
    seq_length=30,
    learning_rate=0.001,
    num_epochs=20,
    hidden_dim=256,
    num_layers=1,
    embedding_dim=100,
    dropout=0.5
)
```

1.3 1C. Learning Curves

```
<figure style="justify-content: space-around; text-align: center; ">
    <div style="displace: flex;">
        <img src="Experiment 0/loss_curve.png" alt="RNN Loss Curve" style="width: 45%; border-r
        <img src="Experiment 0/perplexity_curve.png" alt="RNN Perplexity Curve" style="width: 4
    </div>
    <figcaption>Single Layer RNN</figcaption>
</figure>
<figure style="justify-content: space-around; text-align: center; ">
    <div style="displace: flex;">
        <img src="Experiment 1/loss_curve.png" alt="LSTM Loss Curve" style="width: 45%; border-
        <img src="Experiment 1/perplexity_curve.png" alt="LSTM Perplexity Curve" style="width:
    </div>
    <figcaption>Single Layer LSTM</figcaption>
</figure>
```

1.4 1D. Final Test Set Perplexity

```
--------------------------------------------------------------------------------------------
Single Layer RNN
HP(model_type=rnn,vocab_size=10000, batch_size=64, seq_len=30, lr=0.0005, epochs=20, hl_dim=256
Test Loss: 4.59931574575603, Test Perplexity: 99.41626636090712
Validation Loss: 4.678213638171815, Validation Perplexity: 107.5777280771185
--------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------
Single Layer LSTM
HP(model_type=lstm,vocab_size=10000, batch_size=64, seq_len=30, lr=0.001, epochs=20, hl_dim=256
Test Loss: 4.419968094676733, Test Perplexity: 83.0936341867906
Validation Loss: 4.499923494824192, Validation Perplexity: 90.01024478749626
--------------------------------------------------------------------------------------------
```

1.5 2. Improving the RNN language model architecture

To improve upon vanilla RNNs wer can use advanced RNN variants, attention mechanisms, and regularization. Some examples of these are below:

- LSTM (Long Short-Term Memory): Replace the vanilla RNN with LSTM cells, which are better at capturing long-term dependencies due to their gating mechanisms (input, forget, and output gates).

- GRU (Gated Recurrent Unit): Use GRUs, which are a simpler alternative to LSTMs but still effective at handling long-term dependencies with fewer parameters.
- Bidirectional RNNs: Implement bidirectional RNNs to capture context from both past and future tokens, which is particularly useful for tasks like text generation or sentiment analysis.
- Self-Attention: Incorporate self-attention mechanisms (as used in Transformers) to allow the model to focus on relevant parts of the input sequence, improving its ability to handle long-range dependencies.
- Layer Normalization: Apply layer normalization to stabilize training and improve convergence.
- Learning Rate Scheduling: Use learning rate schedulers (e.g., cosine annealing or step decay) to adaptively adjust the learning rate during training.

[1]:
```python
# Importing libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import time
import os
from datetime import datetime
import pickle
import matplotlib.pyplot as plt
from dataclasses import dataclass
from collections import Counter
import math
```

1.5.1 Hyperparams and RNN

[2]:
```python
@dataclass
class HyperParams:
    '''Dataclass to store hyperparameters for the RNN Language Model'''

    model_type: str
    vocab_size: int
    batch_size: int
    seq_length: int
    learning_rate: float
    num_epochs: int
    hidden_dim: int
    num_layers: int
    embedding_dim: int
    dropout: float

    def __hash__(self):
        return hash(
            (
                self.model_type,
                self.vocab_size,
                self.batch_size,
                self.seq_length,
                self.learning_rate,
                self.num_epochs,
                self.hidden_dim,
                self.num_layers,
                self.embedding_dim,
                self.dropout
            )
        )

    def __eq__(self, value: 'HyperParams'):
        return (
            self.model_type == value.model_type
            and self.vocab_size == value.vocab_size
            and self.batch_size == value.batch_size
            and self.seq_length == value.seq_length
            and self.learning_rate == value.learning_rate
            and self.num_epochs == value.num_epochs
            and self.hidden_dim == value.hidden_dim
            and self.num_layers == value.num_layers
            and self.embedding_dim == value.embedding_dim
            and self.dropout == value.dropout
        )

    def __repr__(self):
        return f'HP(model_type={self.model_type},vocab_size={self.vocab_size},␣
↪batch_size={self.batch_size}, seq_len={self.seq_length}, lr={self.
↪learning_rate}, epochs={self.num_epochs}, hl_dim={self.hidden_dim},␣
↪num_layers={self.num_layers}, emb_dim={self.embedding_dim}, do={self.
↪dropout})'


class RNN(nn.Module):
    def __init__(self, hp: HyperParams):
        super().__init__()  # Remove 'self' from super() call
        self.HP = hp
        # Embedding layer
        self.embedding = nn.Embedding(hp.vocab_size, hp.embedding_dim)
        # RNN layers
        if hp.model_type == 'rnn':
            self.rnn = nn.RNN(
                input_size=hp.embedding_dim,
                hidden_size=hp.hidden_dim,
                num_layers=hp.num_layers,
                batch_first=True,
                dropout=hp.dropout
            )
        elif hp.model_type == 'lstm':
            self.rnn = nn.LSTM(
                input_size=hp.embedding_dim,
                hidden_size=hp.hidden_dim,
                num_layers=hp.num_layers,
                batch_first=True,
                dropout=hp.dropout
            )
        # Dropout layer
        self.dropout = nn.Dropout(hp.dropout)
        # Output layer
        self.fc = nn.Linear(hp.hidden_dim, hp.vocab_size)

    def forward(self, x, hidden):
        # Embed the input
        embedded = self.embedding(x)
        # Pass through RNN
        rnn_out, hidden = self.rnn(embedded, hidden)
        # Take the last time step and pass through final layer
        logits = self.dropout(rnn_out)
        output = self.fc(logits)
        return output, hidden
```

1.5.2 RNN LLM Implementation

[3]:
```
class RNNLLM:
    def __init__(self, train_valid_test_files: tuple[str, str, str], hp:␣
↪HyperParams):
        self.train_file, self.valid_file, self.test_file =␣
↪train_valid_test_files
        self.HP = hp
        os.environ['CUDA_LAUNCH_BLOCKING']="1"
        os.environ['TORCH_USE_CUDA_DSA'] = "1"
        self.device = torch.device(
            "cuda" if torch.cuda.is_available() else "cpu")
        print(f'Using Torch device: {self.device}')

    def train_models(self, hyperparams: list[HyperParams]):
        hp_to_loss: dict[HyperParams, tuple[float, float, float, float]] = {}
        # Each value in the dict is a tuple of (valid loss, test_loss,␣
↪perplexity)
        print(f'Evaluating {len(hyperparams)} hyperparameter configurations')
        for i, hp in enumerate(hyperparams):
            print(f'Evaluating HP {i+1}/{len(hyperparams)}')
            self.HP = hp
            self.train(debug=False, exp_id=i)
            valid_loss, valid_perplexity = self.evaluate(self.valid_loader)
            test_loss, test_perplexity = self.evaluate(self.test_loader)
            hp_to_loss[hp] = (valid_loss, test_loss, valid_perplexity,␣
↪test_perplexity)
        return hp_to_loss

    def evaluate(self, data_loader):
        self.model.eval()
        hidden = self.init_hidden_layer(
             self.HP.batch_size
        )
        total_loss = 0.0
        total_samples = 0
        with torch.no_grad():
            for x, y in data_loader:
                x, y = x.to(self.device), y.to(self.device)
                actual_batch_size = x.size(0)
                # if hidden.size(1) != actual_batch_size:
                hidden = self.init_hidden_layer(
                    actual_batch_size
                    )
                # hidden = hidden.detach()
                output, hidden = self.model(x, hidden)
                loss = self.loss_func(
                    output.view(-1, self.HP.vocab_size), y.view(-1))
                total_loss += loss.item()
                total_samples += actual_batch_size
        avg_loss = total_loss / len(data_loader)
        perplexity = math.exp(avg_loss)
        return avg_loss, perplexity

    def load_model(self, exp_dir: str):
        model_weights_path = os.path.join(exp_dir, 'model_weights.pth')
        hp_pickle_path = os.path.join(exp_dir, f'HP_{self.HP.__hash__()}.pkl')
        if os.path.exists(hp_pickle_path):
            with open(hp_pickle_path, 'rb') as f:
                stored_hp = pickle.load(f)
                if stored_hp == self.HP:
                    print(
                        f'Found existing model with the same hyperparameters:␣
↪{self.HP}')
                return True
        return False
    def train(self, debug=True, exp_id: int = -1):
        # Before training, if we've already trained a model
        # with the exact same hyperparameters, we can load it instead of␣
↪training
        self.setup_training_data()
        self.setup_training_model()
        if exp_id == -1:
            exp_id = datetime.now().strftime('%Y%m%d_%H%M%S')
        # Create experiment folder
        experiment_folder = f'Experiment {exp_id}'
        os.makedirs(experiment_folder, exist_ok=True)
        if (self.load_model(experiment_folder)):
            print(f'Skipping training for model: {self.HP}')
            return
        train_losses = []
        train_perplexities = []
        valid_losses = []
        valid_perplexities = []
        start_time = time.time()
        for e in range(self.HP.num_epochs):
            print(f"Epoch {e+1}/{self.HP.num_epochs}") if debug else None
            # Initialize hidden layers on every epoch
            hidden = self.init_hidden_layer(
                 self.HP.batch_size
            )
            epoch_loss = 0.0
            for batch_idx, (x, y) in enumerate(self.train_loader):
                self.model.train()
                x, y = x.to(self.device), y.to(self.device)
                actual_batch_size = x.size(0)
                # Adjust the hidden state to match the actual batch size
                # if hidden.size(1) != actual_batch_size:
                hidden = self.init_hidden_layer(
                        actual_batch_size
                    )
                print(
                    f"Processing batch {batch_idx + 1}/{len(self.train_loader)}"
                ) if debug else None
                self.optimizer.zero_grad()
                output, hidden = self.model(x, hidden)
                loss = self.loss_func(
                    output.view(-1, self.HP.vocab_size), y.view(-1)
                )
                loss.backward()
                torch.nn.utils.clip_grad_norm_(
                    self.model.parameters(), max_norm=1)  # Clipping
                self.optimizer.step()
                # Prevent backprop through time?
                if isinstance(hidden, tuple): # For LSTM
                    hidden = (hidden[0].detach(), hidden[1].detach())
                else:
                    hidden = hidden.detach()
                epoch_loss += loss.item()
                print(
                    f"Batch {batch_idx + 1}/{len(self.train_loader)}, Loss:␣
↪{loss.item()}"
                ) if debug else None
                # End of batch loop
            avg_epoch_loss = epoch_loss / len(self.train_loader)
            epoch_perplexity = math.exp(avg_epoch_loss)
            train_perplexities.append(epoch_perplexity)
            train_losses.append(avg_epoch_loss)

            # Validation loss
            valid_loss, valid_perplexity = self.evaluate(self.valid_loader)
            valid_losses.append(valid_loss)
            valid_perplexities.append(valid_perplexity)
            print(
                f"Epoch {e+1}/{self.HP.num_epochs} Train Loss:␣
↪{avg_epoch_loss}, Valid Loss: {valid_loss} Train Perplexity:␣
↪{epoch_perplexity} Valid Perplexity: {valid_perplexity}"
            )
            # End of epoch loop
        end_time = time.time()
        train_time_seconds = end_time - start_time
        train_time_minutes = train_time_seconds / 60
        train_time_hours = int(train_time_seconds // 3600)
        train_time_minutes = int((train_time_seconds % 3600) // 60)
        train_time_seconds = int(train_time_seconds % 60)
        train_time_str = f'{train_time_hours:02d}:{train_time_minutes:02d}:
↪{train_time_seconds:02d}'
        print(
            f'Training took {train_time_str} (HH:MM:SS)'
        )

        # Save the model weights and hyperparameters
        torch.save(self.model.state_dict(), os.path.join(
            experiment_folder, 'model_weights.pth'))

        # Write hyperparameters to a pickle file
        with open(os.path.join(experiment_folder, f'HP_{self.HP.__hash__()}.
↪pkl'), 'wb') as f:
            pickle.dump(self.HP, f)
        # Plot and save loss
        plt.figure(figsize=(10, 5))
        plt.plot(range(1, self.HP.num_epochs + 1), train_losses,
                 label='Training Loss', marker='o', color='b')
        plt.plot(range(1, self.HP.num_epochs + 1), valid_losses,
                 label='Validation Loss', marker='o', color='r')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.title('Loss Curve')
        plt.legend()
        plt.figtext(
            0.15, 0.85, f'Training Time: {train_time_str} (HH::MM::SS)',␣
↪fontsize=10, ha='left'
        )
        plt.savefig(os.path.join(experiment_folder, f'loss_curve.png'))

        # Plot and save perplexity
        plt.figure(figsize=(10, 5))
        plt.plot(range(1, self.HP.num_epochs + 1), train_perplexities,
                 label='Training Perplexity', marker='o', color='b')
        plt.plot(range(1, self.HP.num_epochs + 1), valid_perplexities,
                 label='Validation Perplexity', marker='o', color='r')
        plt.xlabel('Epochs')
        plt.ylabel('Perplexity')
        plt.title('Perplexity Curve')
        plt.legend()
        plt.figtext(
            0.15, 0.85, f'Training Time: {train_time_str} (HH::MM::SS)',␣
↪fontsize=10, ha='left'
        )
        plt.savefig(os.path.join(experiment_folder,
                    f'perplexity_curve.png'))

    def setup_training_model(self):
        print(f'Setting up training model with {self.HP}')
        self.model = RNN(self.HP)
        self.model = self.model.to(self.device)
        self.loss_func = nn.CrossEntropyLoss()
        self.optimizer = optim.Adam(
            self.model.parameters(), lr=self.HP.learning_rate)

    def setup_training_data(self):
        # Load the training data and create a vocabulary mapping
        print(f'Loading data and creating vocabulary')
        self.train_indices, self.train_vocab = self.load_data(self.train_file)

        # Create the vocabulary mapping from the training set
        train_text = self.load_wikitext(self.train_file)
        train_tokens = self.tokenize(train_text)
        self.word_to_idx = self.create_vocab_mapping(train_tokens)

        # Use the same vocabulary mapping for validation and test sets
        self.valid_indices, self.valid_vocab = self.load_data(self.valid_file,␣
↪self.word_to_idx)
        self.test_indices, self.test_vocab = self.load_data(self.test_file,␣
↪self.word_to_idx)

        print(f'Train vocab size: {self.train_vocab}')
        print(f'Valid vocab size: {self.valid_vocab}')
        print(f'Test vocab size: {self.test_vocab}')

        print(f'Creating input-output pairs for the dataset')
        # Create input-output pairs for the dataset
        self.train_inputs, self.train_targets = self.create_sequences(
            self.train_indices, self.HP.seq_length)
        self.valid_inputs, self.valid_targets = self.create_sequences(
            self.valid_indices, self.HP.seq_length)
        self.test_inputs, self.test_targets = self.create_sequences(
            self.test_indices, self.HP.seq_length)

        print(f'Creating TensorDataset and DataLoader objects')
        # Create the TensorDataset objects
        self.train_dataset = TensorDataset(
            self.train_inputs, self.train_targets)
        self.valid_dataset = TensorDataset(
            self.valid_inputs, self.valid_targets)
        self.test_dataset = TensorDataset(self.test_inputs, self.test_targets)

        # Create the DataLoader objects
        self.train_loader = DataLoader(
            self.train_dataset, self.HP.batch_size, shuffle=True)
        self.valid_loader = DataLoader(
            self.valid_dataset, self.HP.batch_size, shuffle=False)
        self.test_loader = DataLoader(
            self.test_dataset, self.HP.batch_size, shuffle=False)

    # def init_hidden_layer(self, num_layers, batch_size, hidden_dim):
    #     return torch.zeros(num_layers, batch_size, hidden_dim).to(self.device)


    def init_hidden_layer(self, batch_size):
        if self.HP.model_type == "rnn":
            # For RNN, hidden state is a single tensor
            return torch.zeros(self.HP.num_layers, batch_size, self.HP.
↪hidden_dim).to(self.device)
        elif self.HP.model_type == "lstm":
            # For LSTM, hidden state is a tuple of (h, c)
            return (
                torch.zeros(self.HP.num_layers, batch_size, self.HP.hidden_dim).
↪to(self.device),
                torch.zeros(self.HP.num_layers, batch_size, self.HP.hidden_dim).
↪to(self.device)
            )

    def create_sequences(self, data, seq_length):
        inputs = []
        targets = []
        for i in range(0, len(data) - seq_length, seq_length):
            inputs.append(data[i:i + seq_length])  # input sequence
            # target sequence, shifted by one
            targets.append(data[i + 1:i + seq_length + 1])
        return torch.tensor(inputs, dtype=torch.long), torch.tensor(targets,␣
↪dtype=torch.long)

    def load_wikitext(self, file_path):
        with open(file_path, 'r', encoding='utf-8') as f:
            text = f.read().replace('\n', ' <eol> ')
        return text

    def tokenize(self, text):
        return text.split(' ')

    def create_vocab_mapping(self, tokens, threshold=10):
        counter = Counter(tokens)
        sorted_tokens = sorted(counter.items(), key=lambda x: x[1],␣
↪reverse=True)
        # Keep only the top vocab_size tokens
        sorted_tokens = sorted_tokens[:self.HP.vocab_size - 2]
        # Create a mapping from words to indices
        word_to_idx = {}
        word_to_idx['<unk>'] = 0
        word_to_idx['<eol>'] = 1
        word_index = 2 # Start from 2 and assign indices to words as we travel␣
↪through the sorted_tokens
        for tkn, cnt in sorted_tokens:
            if cnt >= threshold:
                if tkn and tkn != '<eol>': # skip empty and <eol> tokens
                    word_to_idx[tkn] = word_index
                    word_index += 1
        return word_to_idx

    def tokens_to_indices(self, tokens, word_to_idx):
        # return a list of indices (based on the dict mapping words to tokens)
        return [word_to_idx.get(token, word_to_idx['<unk>']) for token in␣
↪tokens if token]

    def load_data(self, text, word_to_idx=None):
        data_text = self.load_wikitext(text)
        data_tokens = self.tokenize(data_text)
        if word_to_idx is None:
            # Create a new vocabulary mapping if none is provided
            data_word_to_idx = self.create_vocab_mapping(data_tokens)
        else:
            # Use the provided vocabulary mapping
            data_word_to_idx = word_to_idx
        vocab_size = len(data_word_to_idx)
        data_indices = self.tokens_to_indices(data_tokens, data_word_to_idx)
        return data_indices, vocab_size
```

1.5.3 Testing

[4]:
```
rnn_hp = HyperParams(
    model_type='rnn',
    vocab_size=10000,
    batch_size=64,
    seq_length=30,
    learning_rate=0.0005,
    num_epochs=20,
    hidden_dim=256,
    num_layers=1,
    embedding_dim=100,
    dropout=0
)
lstm_hp = HyperParams(
    model_type='lstm',
    vocab_size=10000,
    batch_size=64,
    seq_length=30,
    learning_rate=0.001,
    num_epochs=20,
    hidden_dim=256,
    num_layers=1,
    embedding_dim=100,
    dropout=0.5
)
rnn_llm = RNNLLM(
    train_valid_test_files=(
        'wiki2.train.txt', 'wiki2.valid.txt', 'wiki2.test.txt'
    ),
    hp=rnn_hp
)
# rnn_llm.train(debug=False)
hps = [rnn_hp, lstm_hp]
hp_to_loss_map = rnn_llm.train_models(hps)
for hp, (valid_loss, test_loss, valid_perplexity, test_perplexity) in␣
  ↪hp_to_loss_map.items():
    print('--------------------')
    print(
        f'{hp}\nTest Loss: {test_loss}, Test Perplexity:␣
  ↪{test_perplexity}\nValidation Loss: {valid_loss}, Validation Perplexity:␣
  ↪{valid_perplexity}'
    )
    print('--------------------')
```

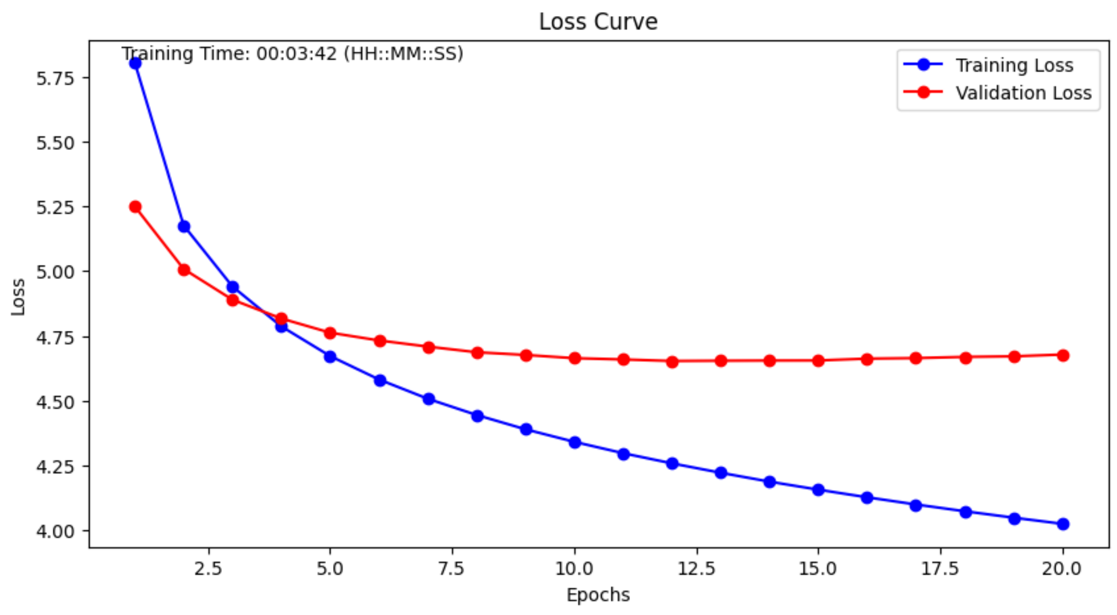
```
Using Torch device: cuda
Evaluating 2 hyperparameter configurations
Evaluating HP 1/2
Loading data and creating vocabulary
Train vocab size: 9997
Valid vocab size: 9997
Test vocab size: 9997
Creating input-output pairs for the dataset
Creating TensorDataset and DataLoader objects
Setting up training model with HP(model_type=rnn,vocab_size=10000,
batch_size=64, seq_len=30, lr=0.0005, epochs=20, hl_dim=256, num_layers=1,
emb_dim=100, do=0)
Epoch 1/20 Train Loss: 5.803454656811321, Valid Loss: 5.249314203596952 Train
Perplexity: 331.4426048111741 Valid Perplexity: 190.43562360018612
Epoch 2/20 Train Loss: 5.176118552246514, Valid Loss: 5.008181597057142 Train
Perplexity: 176.99448116810518 Valid Perplexity: 149.63239662222634
Epoch 3/20 Train Loss: 4.9406885767684265, Valid Loss: 4.889737739897611 Train
Perplexity: 139.86652524956475 Valid Perplexity: 132.91871020523686
Epoch 4/20 Train Loss: 4.786965003346696, Valid Loss: 4.817292200891595 Train
Perplexity: 119.9368079121014 Valid Perplexity: 123.62987227454522
Epoch 5/20 Train Loss: 4.672517387744258, Valid Loss: 4.762614973804407 Train
Perplexity: 106.96668039012255 Valid Perplexity: 117.05161294460244
Epoch 6/20 Train Loss: 4.582230149823077, Valid Loss: 4.732625463552642 Train
Perplexity: 97.7321085936579 Valid Perplexity: 113.59340653672433
Epoch 7/20 Train Loss: 4.50782348259407, Valid Loss: 4.708952849371391 Train
Perplexity: 90.72414079249661 Valid Perplexity: 110.9359324311026
Epoch 8/20 Train Loss: 4.445249096873929, Valid Loss: 4.687131513629043 Train
Perplexity: 85.22110350406484 Valid Perplexity: 108.54138335044074
Epoch 9/20 Train Loss: 4.39008596057401, Valid Loss: 4.6767329835055165 Train
```

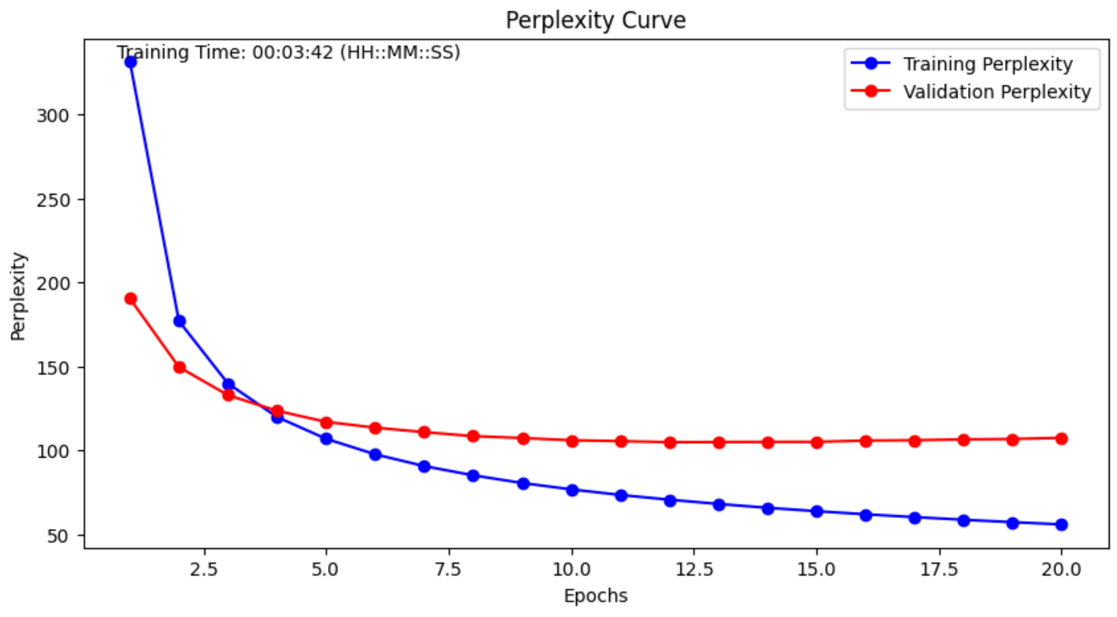
```
Perplexity: 80.6473511751244 Valid Perplexity: 107.41856047725511
Epoch 10/20 Train Loss: 4.3417461391757515, Valid Loss: 4.66425103053712 Train
Perplexity: 76.84159838662558 Valid Perplexity: 106.08610021027397
Epoch 11/20 Train Loss: 4.297854318776551, Valid Loss: 4.659315125984058 Train
Perplexity: 73.5418269698114 Valid Perplexity: 105.56375951761954
Epoch 12/20 Train Loss: 4.2586699312224106, Valid Loss: 4.653405381922136 Train
Perplexity: 70.71586391274963 Valid Perplexity: 104.94174450175502
Epoch 13/20 Train Loss: 4.2223008851356365, Valid Loss: 4.65437258335582 Train
Perplexity: 68.19020174744213 Valid Perplexity: 105.04329340869822
Epoch 14/20 Train Loss: 4.18810174197835, Valid Loss: 4.6550710786852925 Train
Perplexity: 65.89758154534402 Valid Perplexity: 105.11669128959089
Epoch 15/20 Train Loss: 4.1571226876009915, Valid Loss: 4.65532830723545 Train
Perplexity: 63.88743378106181 Valid Perplexity: 105.1437337815901
Epoch 16/20 Train Loss: 4.127684660913313, Valid Loss: 4.6624795972255235 Train
Perplexity: 62.03412649246122 Valid Perplexity: 105.89834210804094
Epoch 17/20 Train Loss: 4.099977324552396, Valid Loss: 4.664460157093249 Train
Perplexity: 60.33891936984477 Valid Perplexity: 106.10828795100613
Epoch 18/20 Train Loss: 4.073886259294608, Valid Loss: 4.669147934829979 Train
Perplexity: 58.78497289864595 Valid Perplexity: 106.60686772348951
Epoch 19/20 Train Loss: 4.049109893905766, Valid Loss: 4.671343443686502 Train
Perplexity: 57.34638995007457 Valid Perplexity: 106.84118117024539
Epoch 20/20 Train Loss: 4.0254421845516735, Valid Loss: 4.678213638171815 Train
Perplexity: 56.005067883027934 Valid Perplexity: 107.5777280771185
Training took 00:03:42 (HH:MM:SS)
Evaluating HP 2/2
Loading data and creating vocabulary
Train vocab size: 9997
Valid vocab size: 9997
Test vocab size: 9997
Creating input-output pairs for the dataset
Creating TensorDataset and DataLoader objects
Setting up training model with HP(model_type=lstm,vocab_size=10000,
batch_size=64, seq_len=30, lr=0.001, epochs=20, hl_dim=256, num_layers=1,
emb_dim=100, do=0.5)
```

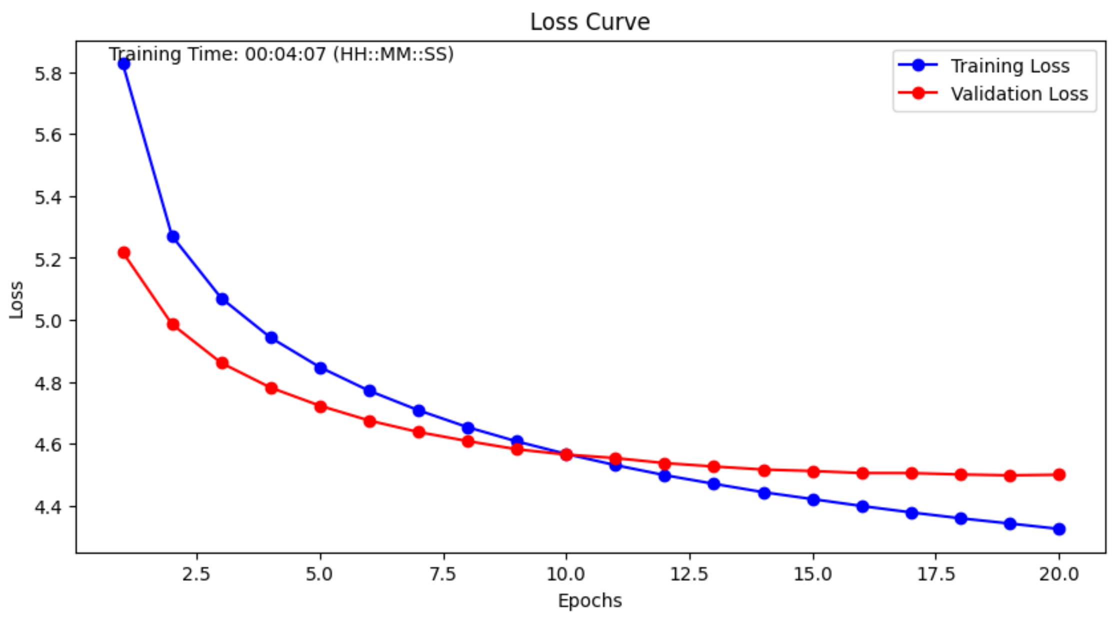
```
/home/sharwin/winter_2025/MSAI437/RNN-Language-Model/.venv/lib/python3.12/site-
packages/torch/nn/modules/rnn.py:123: UserWarning: dropout option adds dropout
after all but last recurrent layer, so non-zero dropout expects num_layers
greater than 1, but got dropout=0.5 and num_layers=1
  warnings.warn(
```

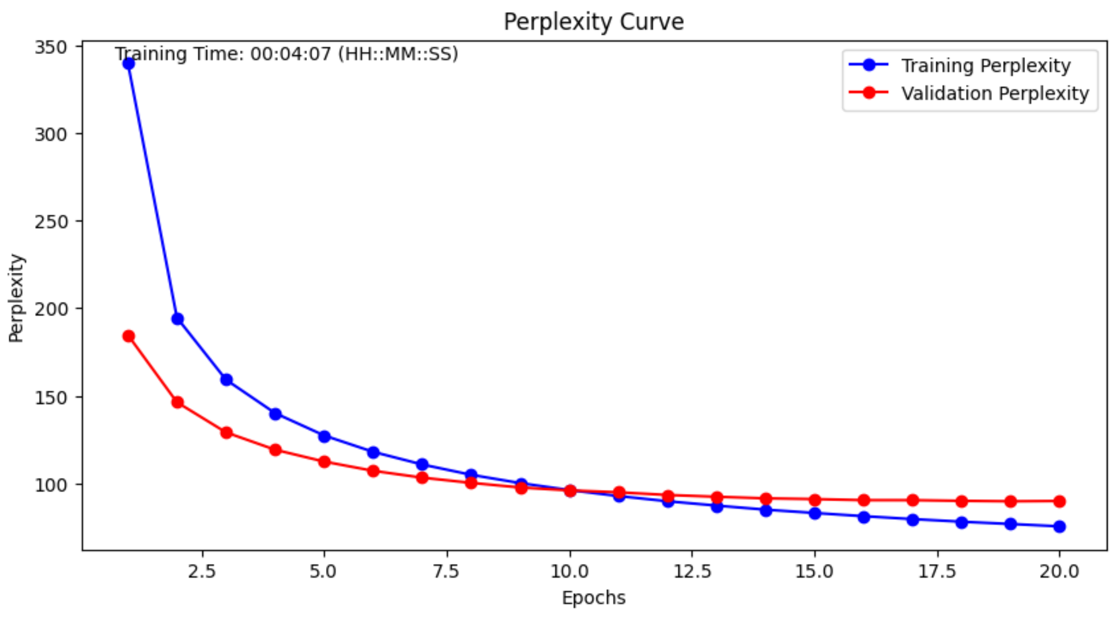
```
Epoch 1/20 Train Loss: 5.8281251669806595, Valid Loss: 5.218559804715608 Train
Perplexity: 339.7211611883346 Valid Perplexity: 184.66803443059337
Epoch 2/20 Train Loss: 5.270740935907645, Valid Loss: 4.985995639834488 Train
Perplexity: 194.5600655919932 Valid Perplexity: 146.3492136191591
Epoch 3/20 Train Loss: 5.070149545722148, Valid Loss: 4.861082708626463 Train
Perplexity: 159.19813296299702 Valid Perplexity: 129.1639733761476
Epoch 4/20 Train Loss: 4.942776871078155, Valid Loss: 4.781362930933635 Train
Perplexity: 140.1589129078485 Valid Perplexity: 119.2667917210674
```

```
Epoch 5/20 Train Loss: 4.847346094163025, Valid Loss: 4.722939445261369 Train
Perplexity: 127.40182833369934 Valid Perplexity: 112.4984501675762
Epoch 6/20 Train Loss: 4.771135520847404, Valid Loss: 4.674990302638004 Train
Perplexity: 118.05321776999362 Valid Perplexity: 107.2315272240656
Epoch 7/20 Train Loss: 4.70817779661978, Valid Loss: 4.637887636820476 Train
Perplexity: 110.84998454279547 Valid Perplexity: 103.32585516491221
Epoch 8/20 Train Loss: 4.653480035855489, Valid Loss: 4.608631840923376 Train
Perplexity: 104.9495791081935 Valid Perplexity: 100.3467653382251
Epoch 9/20 Train Loss: 4.607407608891235, Valid Loss: 4.582126169873957 Train
Perplexity: 100.2239927801591 Valid Perplexity: 97.72194694229215
Epoch 10/20 Train Loss: 4.566965580424842, Valid Loss: 4.5650284792247575 Train
Perplexity: 96.2515984617543 Valid Perplexity: 96.06532984369134
Epoch 11/20 Train Loss: 4.530920685214155, Valid Loss: 4.553688722744322 Train
Perplexity: 92.84400184377388 Valid Perplexity: 94.98212564041089
Epoch 12/20 Train Loss: 4.4985557145932145, Valid Loss: 4.537417022805465 Train
Perplexity: 89.8872147123234 Valid Perplexity: 93.44911119229556
Epoch 13/20 Train Loss: 4.47050171652261, Valid Loss: 4.526533273228428 Train
Perplexity: 87.40056232127083 Valid Perplexity: 92.43754924677616
Epoch 14/20 Train Loss: 4.4435811467907005, Valid Loss: 4.516690258394208 Train
Perplexity: 85.07907743650507 Valid Perplexity: 91.53214832561457
Epoch 15/20 Train Loss: 4.421179814812015, Valid Loss: 4.511950162419102 Train
Perplexity: 83.19438144286086 Valid Perplexity: 91.09930383041555
Epoch 16/20 Train Loss: 4.3988695911624855, Valid Loss: 4.505278622894957 Train
Perplexity: 81.35884790354585 Valid Perplexity: 90.49355411267734
Epoch 17/20 Train Loss: 4.3782569879994675, Valid Loss: 4.505185143989429 Train
Perplexity: 79.69899596192721 Valid Perplexity: 90.48509526964946
Epoch 18/20 Train Loss: 4.359313433222911, Valid Loss: 4.500860862564623 Train
Perplexity: 78.20342411448573 Valid Perplexity: 90.09465704375047
Epoch 19/20 Train Loss: 4.342464899753823, Valid Loss: 4.497960370883607 Train
Perplexity: 76.89684895188739 Valid Perplexity: 89.83371685085808
Epoch 20/20 Train Loss: 4.324966659221579, Valid Loss: 4.499923494824192 Train
Perplexity: 75.56299348870525 Valid Perplexity: 90.01024478749626
Training took 00:04:07 (HH:MM:SS)
--------------------
HP(model_type=rnn,vocab_size=10000, batch_size=64, seq_len=30, lr=0.0005,
epochs=20, hl_dim=256, num_layers=1, emb_dim=100, do=0)
Test Loss: 4.59931574575603, Test Perplexity: 99.41626636090712
Validation Loss: 4.678213638171815, Validation Perplexity: 107.5777280771185
--------------------
--------------------
HP(model_type=lstm,vocab_size=10000, batch_size=64, seq_len=30, lr=0.001,
epochs=20, hl_dim=256, num_layers=1, emb_dim=100, do=0.5)
Test Loss: 4.419968094676733, Test Perplexity: 83.0936341867906
Validation Loss: 4.499923494824192, Validation Perplexity: 90.01024478749626
--------------------
```


Loss Curve
Training Time: 00:03:42 (HH::MM::SS)
Training Loss
Validation Loss
Loss
Epochs
5.75
5.50
5.25
5.00
4.75
4.50
4.25
4.00
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Perplexity Curve
Training Time: 00:03:42 (HH::MM::SS)
Training Perplexity
Validation Perplexity
Perplexity
Epochs
300
250
200
150
100
50
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Loss Curve
Training Time: 00:04:07 (HH::MM::SS)
Training Loss
Validation Loss
Loss
Epochs
5.8
5.6
5.4
5.2
5.0
4.8
4.6
4.4
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Perplexity Curve
Training Time: 00:04:07 (HH::MM::SS)
Training Perplexity
Validation Perplexity
Perplexity
Epochs
350
300
250
200
150
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0