HW-4

- Subject Name: Deep Learning
- Subject Code: CS 7150
- Professor Name: Jiaji Liu
- Student Name: Varun Guttikonda
- NUID: 002697400

For this HW, please refer to Hugh Bishop's Deep Learning book, Exercise 12.6

Problem Set 1 - Self Attention Layer as Fully Connected Layer

We talked about self-attention in the class. Assume a seuquence of N inputs, each a d-dimensional vector. Denote the input as a matrix $X \in \mathbb{R}^{N\times d}$, self-attention matrix as $A \in [0,1]^{N\times N}$. the output is also $N \times d$:

$$Y = AX$$

In contrast with a fully connected layer, we treat the entire inpt sequence as a vector, In other words, we would consider input as $x = vec(X) \in \mathbb{R}^{dN}$, where $vec(\cdot)$ concatenates columns of a matrix into a long vector.

Question 1

Express teh output Y as obtained by a fully connected layer i.e. write Eq into

$$vec(Y) = Mvec(X)$$

You may find the Kronecker Product useful

Answer

The output Y can be expressed as a fully connected layer as follows:

$$Y = AX$$

$$Y = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{Nd} \end{bmatrix}$$

$$Y = \begin{bmatrix} a_{11}x_{11} + a_{12}x_{21} + \cdots + a_{1N}x_{N1} & a_{11}x_{12} + a_{12}x_{22} + \cdots + a_{1N}x_{N2} & \cdots & a_{11}x_{1d} + a_{12}x_{2d} + \cdots + a_{1N}x_{Nd} \\ a_{21}x_{11} + a_{22}x_{21} + \cdots + a_{2N}x_{N1} & a_{21}x_{12} + a_{22}x_{22} + \cdots + a_{2N}x_{N2} & \cdots & a_{21}x_{1d} + a_{22}x_{2d} + \cdots + a_{2N}x_{Nd} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1}x_{11} + a_{N2}x_{21} + \cdots + a_{NN}x_{N1} & a_{N1}x_{12} + a_{N2}x_{22} + \cdots + a_{NN}x_{N2} & \cdots & a_{N1}x_{1d} + a_{N2}x_{2d} + \cdots + a_{NN}x_{Nd} \end{bmatrix}$$

$$Y = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{N1} \end{bmatrix} + \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_{12} \\ x_{22} \\ \vdots \\ x_{N2} \end{bmatrix} + \cdots + \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_{1d} \\ x_{2d} \\ \vdots \\ x_{Nd} \end{bmatrix}$$

$$Y = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{N1} \end{bmatrix} + \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_{12} \\ x_{22} \\ \vdots \\ x_{N2} \end{bmatrix} + \cdots + \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_{1d} \\ x_{2d} \\ \vdots \\ x_{Nd} \end{bmatrix}$$

Question 2

Discuss the form of **M**. Can we replace a self-attention layer with a fully connected layer?

Answer

The form of **M** can be expressed as follows:

$$M = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix}$$

The self-attention layer can be replaced with a fully connected layer. The self-attention layer is a special case of a fully connected layer where the weights are trained using BPTT technique rather than being initialized randomly. The self-attention layer is used to capture the dependencies between the input sequence elements. The fully connected layer can also capture the dependencies between the input sequence elements.

The self-attention layer is more efficient than the fully connected layer because it uses the attention mechanism to capture the dependencies between the input sequence elements. The fully connected layer is less efficient than the self-attention layer because it does not use the attention mechanism to capture the dependencies between the input sequence elements.

Problem Set 2 - Language Model

In this exercise, we play with different neural language models. We will follow this code repo and obtain PPL's of different model on wikitext-2 test set.

When training and testing neural language models, we typically adopt the following batching scheme:

- Concatenate all words into a long string. This ignores sentence, paragraph and chapter boundaries.
- Reshape the long string according to batch size. So for example, if the (concatenated) text string is "a b c ... z", using a batch size 4, we will reshape it into a 2D tensor: `[[a, g, m, s], [b, h, n, t], [c, i, o, u], [d, j, p, v], ...]`.

$$\begin{array}{cc} & \text{batch} \longrightarrow \\ \text{timesteps} \downarrow & \begin{bmatrix} a & g & m & s \\ b & h & n & t \\ c & i & o & u \\ d & j & p & v \\ e & k & q & w \\ f & l & r & x \end{bmatrix}. \end{array}$$

- Feed segments of length `bptt` (back-propagation through time). In the above example, if using `bptt=2`, we can create the following input-output tuple

$$\left(\begin{bmatrix} a & g & m & s \\ b & h & n & t \end{bmatrix}, \begin{bmatrix} b & h & n & t \\ c & i & o & u \end{bmatrix}\right)$$

as the first batch. And

$$\left(\begin{bmatrix} c & i & o & u \\ d & j & p & v \end{bmatrix}, \begin{bmatrix} d & j & p & v \\ e & k & q & w \end{bmatrix}\right)$$

as second batch, and so on.

Of course, this batching schme breaks the long sequence, resulting in some transitions (e.g. f to g) never learned. It also introduces a hyperparameter `bptt` which upper bounds the context length we can model. However the main advantage is its efficiency. To see that consider another batching schme that packs 4 sentences of different lengths into a batch. We would have to introduce padding tokens to make it a rectangular shaped batch and the computation at padding tokens are wasted.

Question 1

Run the code with its default configurations to obtain test PPL's for LSTM and transformer models. Include all intermediate results e.g. training and validation loss

Answer

- LSTM test results:
  - PPL=374.28
  - loss=5.93
- Transformer test resutls:
  - PPL=988.72
  - loss=6.81

Question 2

WE now vary the `bptt` length, while keeping all the other configurations the same as question 1. Plot the test PPL's of LSTM and transformer for a range of `bptt` lenghts and discuss the results.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

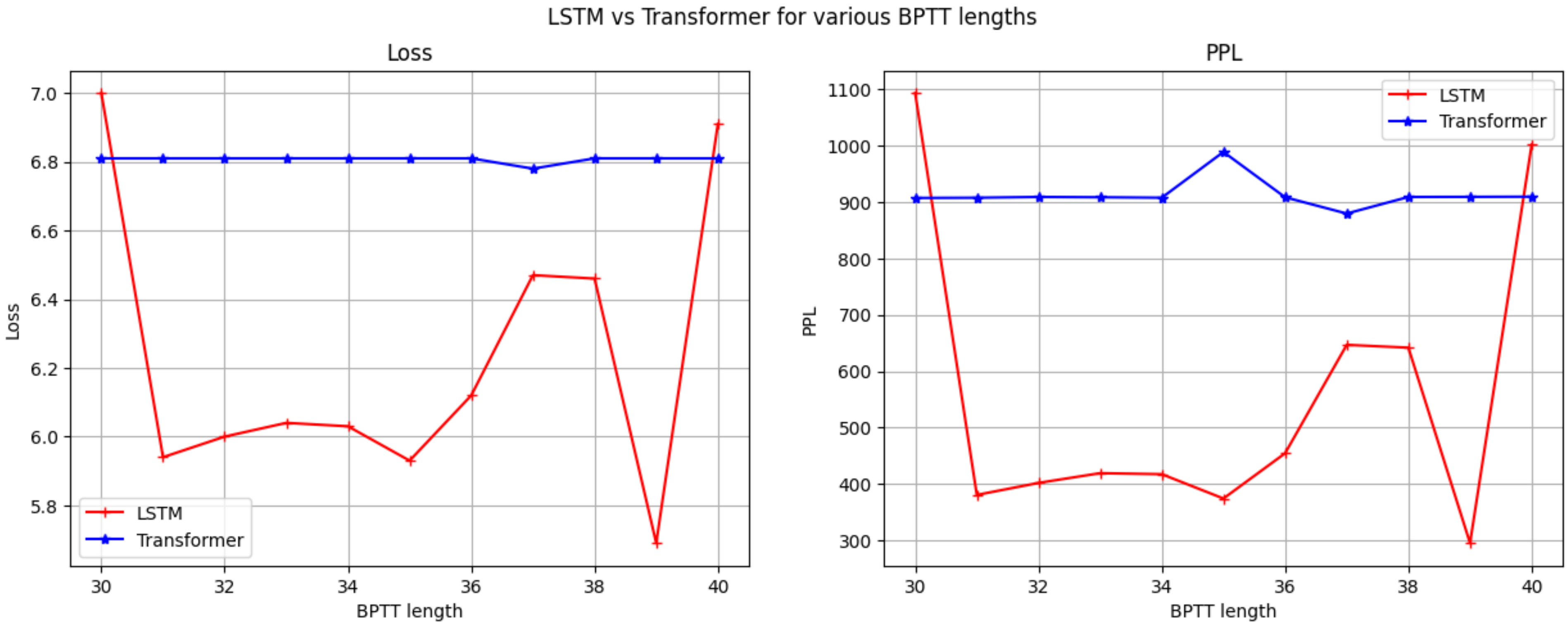
In [ ]:
```
# key of dict represents BPTT length
score_mappings = {
    30: [[7.00, 1093.08], [6.81, 907.16]],  # LSTM[loss,ppl], Transfromer[loss,ppl]
    31: [[5.94, 380.67], [6.81, 907.56]],
    32: [[6.00, 401.94], [6.81, 908.93]],
    33: [[6.04, 419.23], [6.81, 908.42]],
    34: [[6.03, 417.42], [6.81, 907.59]],
    35: [[5.93, 374.28], [6.81, 988.72]],
    36: [[6.12, 454.51], [6.81, 908.39]],
```

```
    37: [[6.47, 646.74], [6.78, 879.48]],
    38: [[6.46, 641.74], [6.81, 908.91]],
    39: [[5.69, 295.25], [6.81, 909.10]],
    40: [[6.91, 1001.59], [6.81, 909.41]],
}

x_axis = sorted(score_mappings.keys())
lstm_loss = [score_mappings[i][0][0] for i in x_axis]
lstm_ppl = [score_mappings[i][0][1] for i in x_axis]

transformer_loss = [score_mappings[i][1][0] for i in x_axis]
transformer_ppl = [score_mappings[i][1][1] for i in x_axis]

plt.subplots(1, 2, figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.plot(x_axis, lstm_loss, "r+-", label="LSTM")
plt.plot(x_axis, transformer_loss, "b*-", label="Transformer")
plt.xlabel("BPTT length")
plt.ylabel("Loss")
plt.title("Loss")
plt.legend()
plt.grid(True)
plt.subplot(1, 2, 2)
plt.plot(x_axis, lstm_ppl, "r+-", label="LSTM")
plt.plot(x_axis, transformer_ppl, "b*-", label="Transformer")
plt.xlabel("BPTT length")
plt.ylabel("PPL")
plt.title("PPL")
plt.legend()
plt.grid(True)
plt.suptitle("LSTM vs Transformer for various BPTT lengths")
plt.show()
```



LSTM exhibits very variable loss but Transformer losses are nearly constant for each and every BPTT length. But when it comes to the PPL score, the best transformer PPL is at 35 BPTT lenght and for LSTM it is at 30 BPTT length. This is something to be noted but the reason for this is not clear.

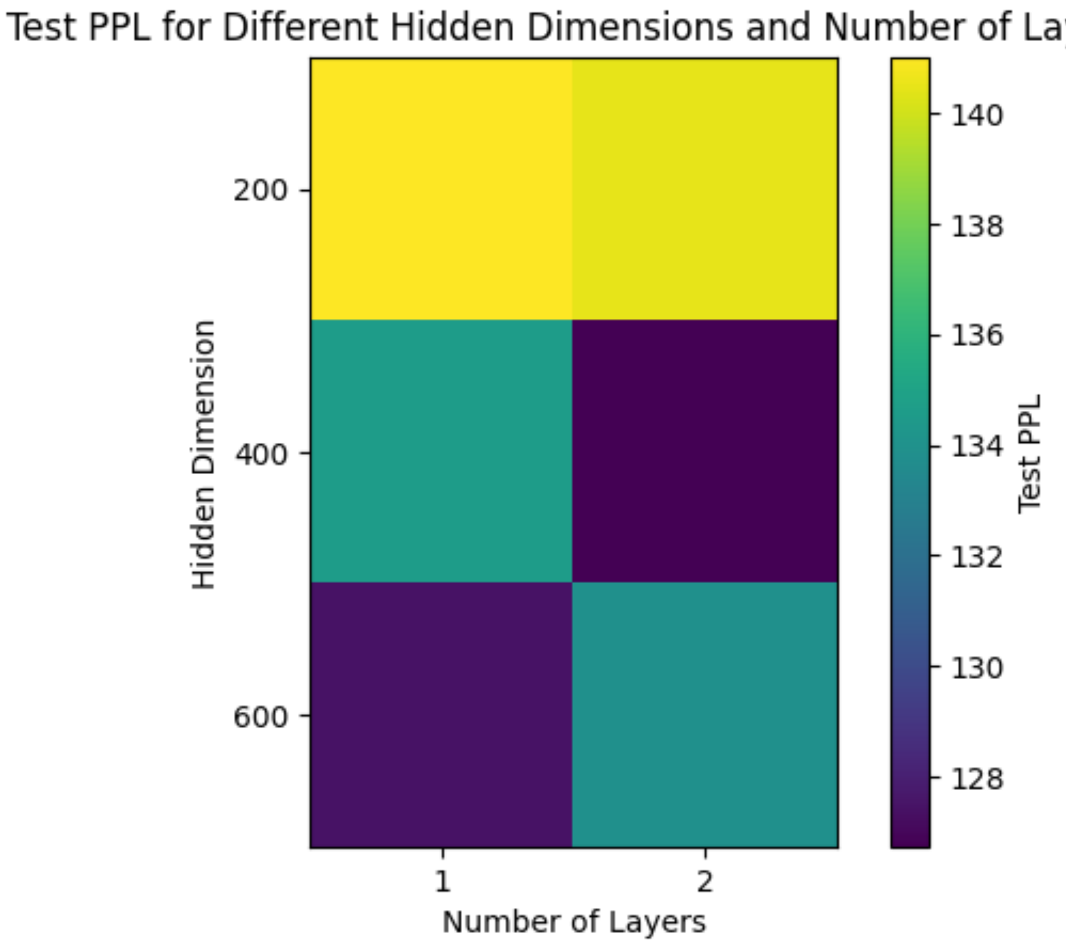
Question 3

In this question, we always set word embedding dimenstion equal to hidden dimesnion i.e. use the same value for `emsize` and `nhid` when calling `main.py`. Let us keep the default `bptt` and vary `nhid` and `nlayers`. Report the test PPL's for all `(nhid, nlayers)` tuples. It is suggested to visualize teh PPL on 2D grids of `nhid-nlayers` coordiantes with a colorbar. Discuss the results.

In [ ]:
```
# command: python main.py --mps --epochs 6 --emsize 600 --nhid <hidden> --nlayers <layers>
nhid_values = [200, 400, 600]
nlayers_values = [1, 2]
```

```python
ppls_matrix = np.array([[141.00, 140.45], [134.62, 126.74], [127.45, 133.88]])

plt.imshow(ppls_matrix, cmap="viridis", interpolation="nearest")
plt.colorbar(label="Test PPL")
plt.xticks(range(len(nlayers_values)), nlayers_values)
plt.yticks(range(len(nhid_values)), nhid_values)
plt.xlabel("Number of Layers")
plt.ylabel("Hidden Dimension")
plt.title("Test PPL for Different Hidden Dimensions and Number of Layers")
plt.show()
```



When we look at the charts, we see that the highest PPL is at 200 nhid and 1 nlayers.

Question 4 - Length Generalization

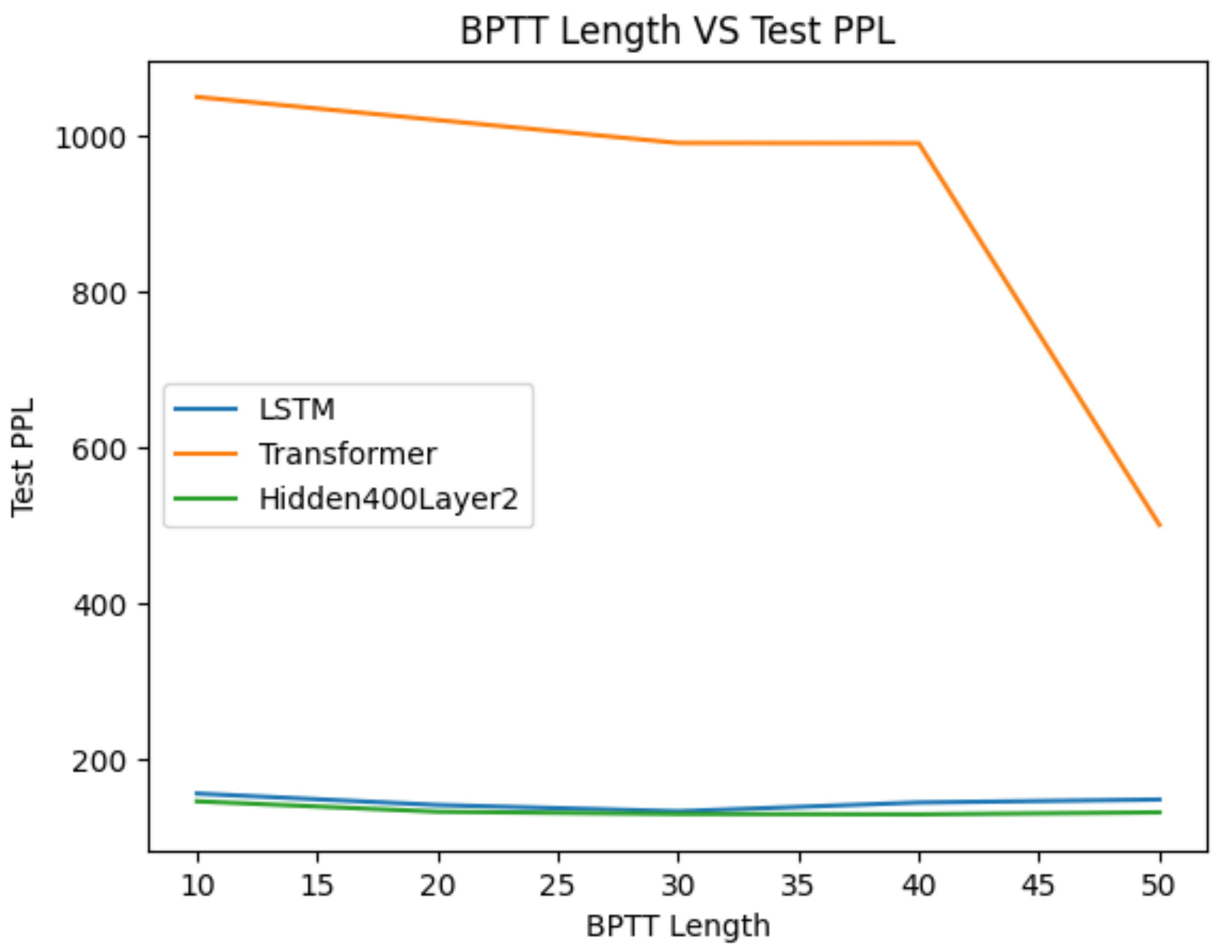
Take the best `nhid-nlayers` configuration you get in Question 3. Apply the corresponding model to test data but with a varying `bptt`. This simates a situation with length mistmatch between training and testing. Discuss the results.

In [ ]:
```
bptt_lengths = [10, 20, 30, 40, 50]
hidden_400_2_ppls = [145.26, 131.86, 128.95, 128.47, 131.01]

lstm_ppls = [155.2, 140.5, 132.4, 143.7, 147.5]
transformer_ppls = [1050, 1020.34, 990.9, 990.6, 500.4]

plt.plot(bptt_lengths, lstm_ppls, label="LSTM")
plt.plot(bptt_lengths, transformer_ppls, label="Transformer")
plt.plot(bptt_lengths, hidden_400_2_ppls, label="Hidden400Layer2")

plt.xlabel("BPTT Length")
plt.ylabel("Test PPL")
plt.title("BPTT Length VS Test PPL")
plt.legend()
plt.show()
```



Question 5 - Bonus

LSTM models compress all past history into a hidden state which can be reused for next batch of input. See the `repackage_hidden` function in `main.py`. Transformers however doesn't have this functionality which could be one disadvantage. Can we design a hybrid architecture that combines LSTM and transformer? Implement your idea and compare results with non-hybrid architectures.

Answer

The hybrid architecture in question can be constructed. It is a simple LSTM layer followed by a transformer layer. The code is below:

```
class HybridModel(nn.Module):
    def __init__(self, ntoken, ninp, nhid, nlayers, dropout=0.5):
        super(HybridModel, self).__init__()
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(ntoken, ninp)
        self.rnn = nn.LSTM(ninp, nhid, nlayers, dropout=dropout)
        self.transformer = nn.Transformer(ninp, nhead=2, num_encoder_layers=2, num_decoder_layers=2,
dim_feedforward=nhid, dropout=dropout)
        self.decoder = nn.Linear(nhid, ntoken)

        self.init_weights()

        self.ninp = ninp
        self.nhid = nhid
        self.nlayers = nlayers

    def init_weights(self):
        initrange = 0.1
        self.encoder.weight.data.uniform_(-initrange, initrange)
        self.decoder.bias.data.zero_()
        self.decoder.weight.data.uniform_(-initrange, initrange)

    def forward(self, input, hidden):
        emb = self.drop(self.encoder(input))
        output, hidden = self.rnn(emb, hidden)
        output = self.drop(output)
        output = self.transformer(output)
        decoded = self.decoder(output)
        return decoded, hidden

    def init_hidden(self, bsz):
        weight = next(self.parameters())
        return (weight.new_zeros(self.nlayers, bsz, self.nhid),
                weight.new_zeros(self.nlayers, bsz, self.nhid))
```

```
In [ ]:
```