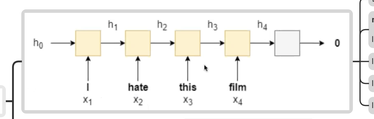
Part 2 word-level recurrent language model

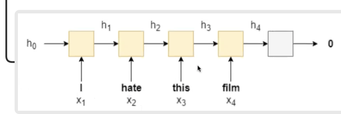
Introduction to RNN (Recurrent Neural Network)

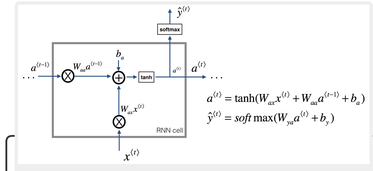
$h_t = \mathrm{RNN}(x_t, h_{t-1})$



- [1,3,3,4,9]
- next token prediction task
  - I hate this film
  - I --> m --> hate
  - I hate --> m --> this
  - I hate this --> m --> film

learn embedding based on
- current information (a word) x_t
- previous information (words before the current word) h_(t-1)





$$a^{\langle t\rangle} = \tanh(W_{aa}x^{\langle t\rangle} + W_{aa}a^{\langle t-1\rangle} + b_a)$$
$$\hat{y}^{\langle t\rangle} = soft\max(W_{ya}a^{\langle t\rangle} + b_y)$$

W here is the learnable parameters to do linear transformation --> we have three W here, one for a, one for x, and one for y

tanh is the non-linear activation function --> without this our model would only be a linear model

softmax --> change the ouput to be the prediction

$h_t = f(Wx_t + Uh_{t-1} + b)$

- wa<t-1> + b (previous)
- wx<t> + b (current)
- wa<t-1> + wx<t> + b
- mathematical proof
  - non-linear stuff into linear model
    - approximate every function in the world
- non-linear activation
  - tanh
  - relu
  - leakyrelu
- step 1: previous/current linear transformed
- step 2: sum up
- step 3: non linear activation
- if conducting **a classification task**
  - step 1: linear transform the output of step 3
  - step 2: softmax the results to get the probability distribution
- linear transformation
  - y = **ax** + b

Datasets & DataLoaders

Loading dataset

```
from torchtext.datasets import PennTreebank
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
```

```
#!pip install torchdata
!pip install portalocker
```

- particularly useful when you need to ensure that only one process or thread can access a specific file at a time
- a package uses to lock file

```
train_iter, valid_iter, test_iter = PennTreebank(
    root='data',
    split=('train', 'valid', 'test'))

num = 0
for _ in train_iter:
    num +=1

print("Train size: {num}")
```

load the Peentreebank dataset --> and split it into train/valid/test set

training set size --> 42608

Dataset preprocess

```
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torch.utils.data import Dataset
```

1. load the english tokenizer
2. create the vocabulary using build_vocab_from_iterator()

```
tokenizer = get_tokenizer('basic_english')
vocab = build_vocab_from_iterator(map(tokenizer, train_iter), specials=['<unk>'])
vocab.set_default_index(vocab['<unk>'])
```

- build_vocab_from_iterator()
  - **train_iter** is an iterator over the training dataset, where **each item** is **a piece of text** (like a sentence or a document).
  - **map(tokenizer, train_iter)** applies the tokenizer to **each item** in train_iter, **splitting the text into tokens.**
    - 'I have a car' — ['I', 'have', 'a', 'car']
    - 'I have a cat' — ['I', 'have', 'a', 'cat']
  - **specials=['<unk>']** adds a special token <unk> to the vocabulary, which **stands for "unknown."**
- vocab.set_default_index(vocab['<unk>'])
  - **vocab['<unk>'] retrieves the index** assigned to **the <unk> token** in the vocabulary.
  - **set_default_index(vocab['<unk>'])** sets this index **as the default** for the **vocabulary**.
    - why?
      - if you try to **get the index of a token that is not in the vocabulary**, it **will return the index of <unk> instead of raising an error**.
      - --> crucial for **handling out-of-vocabulary tokens** during model inference.

```
print("Vocabulary size: {len(vocab)}")
print("The first 10 words in the vocabulary: {vocab.get_itos()[:10]}")
```

```
Vocabulary size: 9922
The first 10 words in the vocabulary: ['<unk>', 'the', 'to', 'of', '<n>', 'a', 'in', 'and', '.', ',']
```

vocabulary size and top 10 words in the vocabulary

```
# Map vocabulary tokens back to their indices.
vocab(['here', 'is', 'an', 'example'])

[1293, 23, 30, 617]


# But out-of-vocabulary tokens will be converted to <unk> (the default index)
vocab(['here', 'is', 'an', 'unknown', 'token', 'asdfghj'])

[1293, 23, 30, 1815, 20200, 0]
```

- get corresponding index of words in the vocabulary
- 'asdfghj' --> haven't shown in the vocabulary --> its index is 0, which is <unk>

```
def data_process(raw_text_iter dataset.IterableDataset) -> torch.Tensor:
    """Converts raw text into a flat Tensor."""
    data = [torch.tensor(vocab.tokenizer(item), dtype=torch.long) for item in raw_text_iter]
    return torch.cat(tuple(filter(lambda t: t.numel() > 0, data)))
```

takes an iterable dataset as input --> contain the raw text data

1. it iterates over each item in the dataset
2. for each item, first tokenize the text into tokens using tokenizer
3. convert these tokens into their corresponding indices in the vocabulary. --> and convert to long tensor
4. filter out any empty tensors
5. **concatenate** all these tensors **into a single flat tensor**

```
train_iter, valid_iter, test_iter = WikiText2(
    root='data',
    split=('train', 'valid', 'test'))
train_stream = data_process(train_iter)
valid_stream = data_process(valid_iter)
test_stream = data_process(test_iter)
```

process the iterable dataset

- a cat sit --> train_iter
  - a --> 5
  - cat --> 10
  - sit --> 15
  - [5, 10, 15]

```
train_stream

tensor([  9, 3849, 3869,  881,    9, 20000,   83, 3849,   88,    0],
       device='cuda:0')
```

get a sequence of word indices for the text

```
max_seq_length=10
from typing import Tuple, List
def create_data_tuples(source: torch.Tensor):
    return [[source[i:i + max_seq_length], source[i+1:i+ max_seq_length + 1]]
            for i in range(len(source) - max_seq_length)]
```

- max_seq_length --> the maximum length of the input and target sequences to be generated.
- takes a tensor as input
- iterate through the input tensor to generate **pairs of sequences,** each one has two elements.

- sentence () --> model
  - input sequence (**serves as the input to a model**) from current index i to 'i + max_seq_length'
  - target sequence (**expected output of the model**) from index i + 1 to 'i + max_seq_length + 1'

Example

Given a ``source`` tensor `'[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]'` and `'max_seq_length'` set to `'10'`, the function would create a single pair (since it stops before the last `'max_seq_length'` tokens), where:

* The input sequence would be `'[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]'`
* The target sequence would be `'[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]'`

- teacher forcing ratio --> 0.5
- 1 --> m --> 5
- **1, 5 --> m -->**
- **1, 2 --> m -->**

the model **learns to predict the next token** in a sequence **given the previous tokens**, aiding in tasks like **text generation or auto-completion**.

Create a custom dataset

```
from torch.utils.data import Dataset
class LMDataset(Dataset):
    def __init__(self, source):
        self.source = source
        self.data = create_data_tuples(source)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]
```

define a LMDataset

each dataset has a lot of generated sequence pairs

```
train_dataset = LMDataset(train_stream)
valid_dataset = LMDataset(valid_stream)
test_dataset = LMDataset(test_stream)
```

```
train_dataset[0]

[tensor([   9,  3849,  3869,   881,     9, 20000,    83,  3849,    88,     0],
       device='cuda:0'), tensor([ 3849,  3869,   881,     9, 20000,    83,  3849,    88,     0, 3869])]
```

preparing your data for training with dataloaders

```
from torch.utils.data import DataLoader

batch_size = 64
eval_batch_size = 32

train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_dataloader = DataLoader(valid_dataset, batch_size=eval_batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=eval_batch_size, shuffle=True)
```

for training set, each batch has 64 pairs of sequences

for valid/test set, each batch has 32 pairs of sequences

shuffle = True, shuffling the data order in these datasets

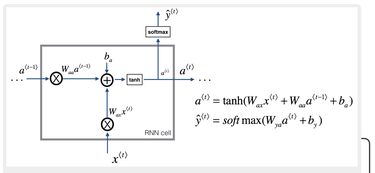
- Iterate through the data loader

```
src, tgt = next(iter(train_dataloader))
print(f"Source batch shape: {src.size()}")
print(f"Target batch shape: {tgt.size()}")

Source batch shape: torch.Size([64, 10])
Target batch shape: torch.Size([64, 10])
```

Build the Neural Network

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

Using cuda device
```

check gpu availability



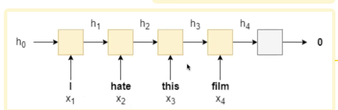
$$a^{\langle t\rangle} = \tanh(W_{aa}x^{\langle t\rangle} + W_{aa}a^{\langle t-1\rangle} + b_a)$$
$$\hat{y}^{\langle t\rangle} = soft\max(W_{ya}a^{\langle t\rangle} + b_y)$$

W here is the learnable parameters --> we have three W here, one for a, one for x, and one for y

tanh is the non-linear activation function --> without this our model would only be a linear model

softmax --> change the ouput to be the prediction

- 1. initialization function:
  - set up **hidden_size and embedding_size**
  - self.embedding --> vocabulary --> vocab_size x embedding_size matrix --> each row represents a token
  - self.rnn --> the RNN model is already a built-in model in torch.nn, just directly use it --> specify the hidden size.
  - **batch_first = True** means the RNN layer **produces tensors with shape (batch size, sequence length, hidden size)**,
    - matching the output shape of the embedding layer and maintaining consistency in how data is processed through the network.
  - self.fc --> a linear layer to get the final **probablity distribution of words**
    - e.g., {'dog': 0.5, 'cat' : 0.2, 'run': 0.1,...} (after softmax would be something like this)
- 2.forward function
  - 1. get the word/token embeddings based on input word index **(x is the input data stream)**
    - e.g., I hate this film --> get their embedding
    - input these words embedding and the hidden states (hidden states is initialized as all 0 as it is h_0)
  - 2. input both the embedding and the hidden states to the rnn models --> output a embedding and hidden states
    - the output embedding is converted to a probability distrbution with self.fc
    - return the out and hidden states



- 3. initialize the hidden states
  - the original hidden states is just a batch size x hidden size all zeros matix

```
class* torch.nn.*
class WordLevelRecurrentLanguageModel(nn.Module):
    def __init__(self, vocab_size, embedding_size, hidden_size):
        super(WordLevelRecurrentLanguageModel, self).__init__()
        self.hidden_size = hidden_size
        self.embedding_size = embedding_size

        self.embedding = nn.Embedding(vocab_size, embedding_size)
        self.rnn = nn.RNN(embedding_size, hidden_size, batch_first=True)

        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden_states):
        embedded = self.embedding(x)
        out, hidden_states = self.rnn(embedded, hidden_states)

        # Reshaping the outputs to batch_size * seq_length * hidden_size
        out = out.reshape(out.shape[0], out.shape[1], self.hidden_size)
        out = self.fc(out)
        return out, hidden_states

    def init_hidden_states(self, batch_size):
        hidden_states = torch.zeros(1, batch_size, self.hidden_size)
        return hidden_states
```

define the class

- 1x10 --> 10x**100** --> 1x50 — y = wx + b
  - 1000 parameters learn
  - don' t have much data --> use less parameters
  - have a lot of data --> use more parameters

```
vocab_size = len(vocab)
embedding_size = 64
hidden_size = 64
model = WordLevelRecurrentLanguageModel(vocab_size, embedding_size, hidden_size).to(device)
```

- get vocabulary size and set up hyperparameters
- put the model to device

```
Layer: embedding.weight | Size: torch.Size([9922, 64]) | Values : tensor([[-1.3751, -2.5271, -0.5333, -0.2699, -0.1728, -1.1404, 0.1339, 0.8066, 0.3057,
          0.4195, -0.3195, -0.3195, -0.3195, -0.3195, -0.3195, -0.3195, -0.3195,
          -0.1856, -1.7595, -0.1400, -1.7215, -0.3873, -1.1371, -1.5767, 0.5566, 1.5873,
          1.1670, -0.7892, -1.8447, -0.5843, 1.1300, 1.1150, 1.1400, -0.3515,
          0.3013, 1.1841, 0.8106, 2.1184, -2.8833, 0.6139, -0.3002, 0.9291,
          0.6400, 0.2008, -1.8441, -1.0377, -0.8332, 1.0565, -1.5195, 0.3406,
          -2.8806, -0.5473, -0.3873, 0.2429, -1.1353, -0.5735, 1.7751, -0.3119,
          [-0.8139, -0.3195, -0.3195, -0.3195, -0.3195, -0.3195, -0.3195, -0.3195,
          -0.8939, -1.6742, -1.4177, 1.2605, -1.8908, -0.6139, 1.8196, -1.8645,
          -0.7582, 1.4995, -1.1871, 1.6851, 1.4339, -0.6218, 1.3392, 1.1149,
          1.7017, 0.1854, -0.1786, 0.8918, 1.1156, 0.8858, -0.6314, 0.5645,
          0.4851, 1.1419, -0.6225, 0.8611, 0.1192, -2.3138, 0.1890, 1.4812,
          0.5838, -0.3585, 0.7752, 0.1407, -0.5386, -0.7119, 0.3428, 0.8908,
          0.1395, -1.6202, 0.3775, -0.9766, -0.3195, -0.3195, -0.4961, 1.3417,
          -0.3195, -1.5195, -0.3195, -0.3195, -0.3195, -0.7206, -0.4125, -0.1311],
         device='cuda:0', grad_fn=<Slice>)]
```

word embedding

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")

Output exceeds the size limit. Open the full output data in a text editor
Model structure: WordLevelRecurrentLanguageModel(
  (embedding): Embedding(9922, 64)
  (rnn): RNN(64, 64, batch_first=True)
  (fc): Linear(in_features=64, out_features=9922, bias=True)
)
```

print out the model structure and model weight

```
x = train_dataset[0][0].unsqueeze(0).to(device)
hidden_states = model.init_hidden_states(1).to(device)
logits, states = model(x, hidden_states)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted tokens: {y_pred}")


Predicted tokens: tensor([ 8588,  5640,  6703,  4291, 13858,    65,  9588, 23000, 17162, 17772],
       device='cuda:0')
```

use the untrained mdoel to obtain prediction

- get one training sample out from train dataset and put on gpu
- initialize the hidden states
- get the prediction (logits)
- use softmax to get the prediction probability
- get the predicted 10 tokens

Presented with xmind