Tutorial 8 - Recurrent Neural Network

Outline

- Announcement
- Suggestions about using activation function on the final output layer
- Recurrent Neural Network & LSTM

Announcement

- **NO** tutorial session next week!
- Eric's office hours:
  - March 14th (Thursday) 6:30-7:30 pm
  - March 19th (Tuesday) 6:30-7:30 pm
  - Zoom: https://berkeley.zoom.us/j/7510266955
- HW8 is released today and due next Thursday (which is the ugrad mid-term date, please start early on this HW).

Suggestions about using activation functions

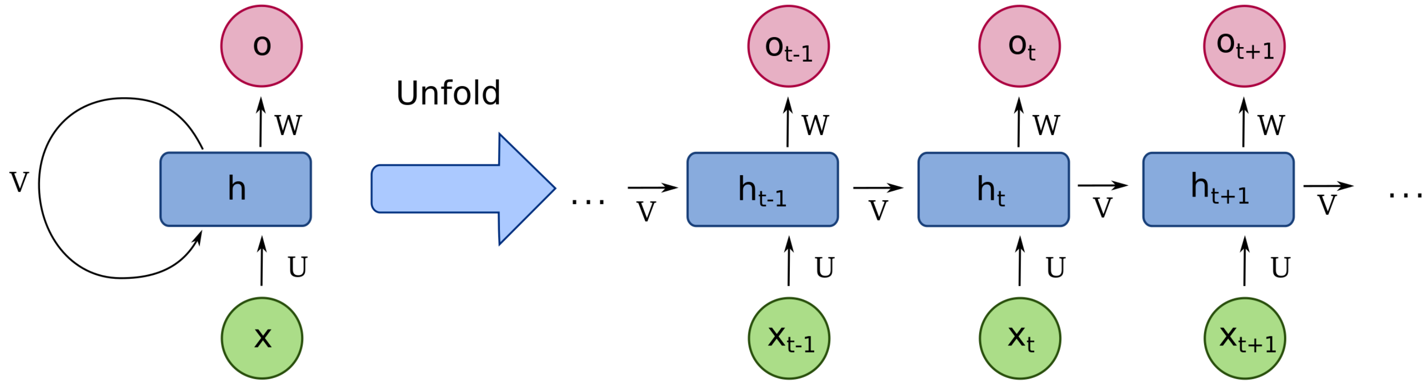
In most cases, the output layer is not being activated because the activation function will shrink the output range, which disable the model fit to data out of the range. For example, tanh will give output between -1 and 1, so if the targets range from $(-2, 2)$, the model will fail to learn.

But if the targets are probabilities, it's better to use Sigmoid or Softmax, which will enforce an output value in $(0, 1)$.

Recurrent Neural Network

RNN is a series of architecutres that is designed for sequential data, such as audio and text.

Vanilla RNN



- Inputs:
  - $\boldsymbol{X}(X_1, X_2, \cdots, X_t)$
  - h_0
- Feed forward:

$$h_t = \sigma(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

$$y_t = \sigma(h_t W_{oh}^T + b_{oh})$$

- PyTorch: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html#torch.nn.RNN

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.preprocessing import OneHotEncoder

import torch
import torch.nn as nn
```

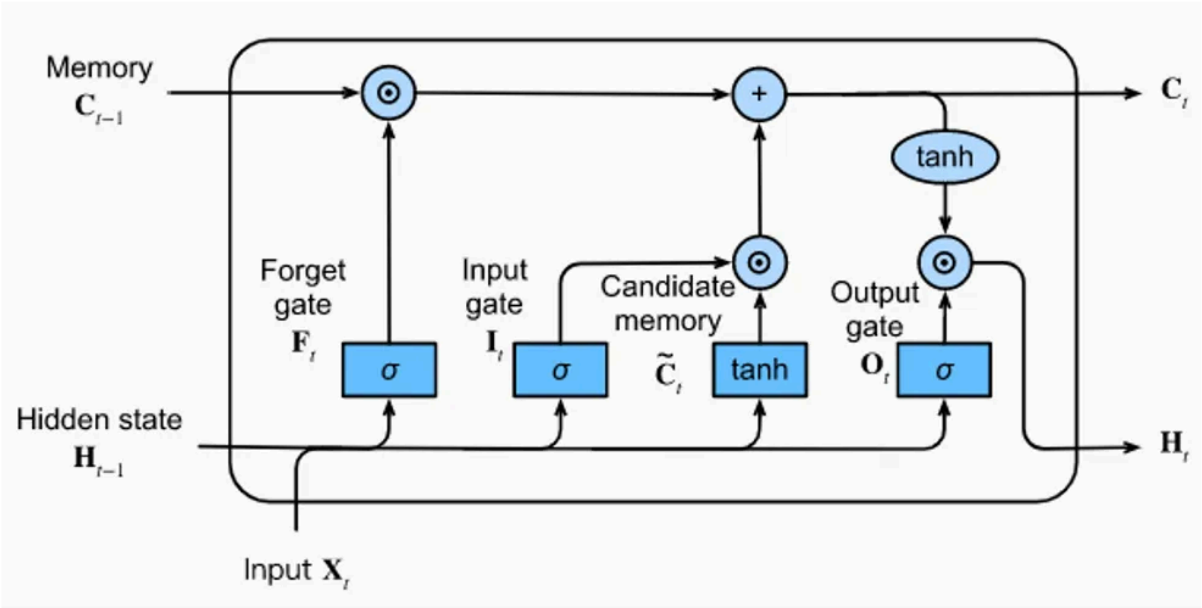
```
from torch.utils.data import Dataset, DataLoader


from rdkit import Chem
from rdkit import RDLogger
RDLogger.DisableLog("rdApp.*")
```

In [ ]:
```
# nn.RNN(input_dim, hidden_dim, num_layers)
rnn = nn.RNN(5, 3, 1, batch_first=True)

# input shape: (n_batch, n_seq, input_dim)
inputs = torch.rand(1, 2, 5)

# h0 shape: (n_layers, n_batch, hiden_dim)
h0 = torch.rand(1, 1, 3)

# output(h1,...,ht), ht
output, ht = rnn(inputs, h0)

print(output)
print(ht)
```

```
tensor([[[ 0.8481,  0.1570, -0.8677],
         [ 0.9528, -0.5074, -0.6681]]], grad_fn=<TransposeBackward1>)
tensor([[[ 0.9528, -0.5074, -0.6681]]], grad_fn=<StackBackward0>)
```

In [ ]:
```
# without explicitly setting h0
output, ht = rnn(inputs)
```

LSTM: Long-short Term Memory



- Inputs:
  - $\boldsymbol{X}(X_1, X_2, \cdots, X_t)$
  - h_0
  - c_0
- Feed forward:

$$
\begin{aligned}
i_t &= \mathrm{sigmoid}(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \mathrm{sigmoid}(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \mathrm{sigmoid}(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

- PyTorch: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html#torch.nn.LSTM

In [ ]:
```
lstm = nn.LSTM(5, 3, 1, batch_first=True)

# input shape: (n_batch, n_seq, input_dim)
inputs = torch.rand(1, 2, 5)

# hidden shape: (n_layers, n_batch, hiden_dim)
h0 = torch.rand(1, 1, 3)
c0 = torch.rand(1, 1, 3)

# output: h1, ... ht
# ht, ct
output, (ht, ct) = lstm(inputs, (h0, c0))

print(output)
print(ht)
print(ct)
```

```
tensor([[[ 0.1054, -0.0214,  0.0259],
         [ 0.0977,  0.0432, -0.0806]]], grad_fn=<TransposeBackward0>)
tensor([[[ 0.0977,  0.0432, -0.0806]]], grad_fn=<StackBackward0>)
tensor([[[ 0.2718,  0.0986, -0.1345]]], grad_fn=<StackBackward0>)
```

Generate SMILES strings using RNN

Data pre-processing:

- Add starting/ending tokens
    - `SOS` : Start Of Sequence
    - `EOS` : End Of Sequence
- One-hot Encoding
- Padding

In [ ]:
```
def load_smiles(path):
    with open(path) as f:
        smiles = f.read().split('\n')
```

```
    return smiles

smiles = load_smiles("Datasets/ani_smiles_clean.txt")
print(smiles[:5])
```

```
['C', 'N', 'O', 'CC', 'CN']
```

Padding: `"C=CC#N" -> ['SOS, 'C', '=', 'C', 'C', '#', 'N', 'EOS']`

In [ ]:
```
def pad_start_end_token(smiles):
    padded = []
    for smi in smiles:
        padded.append(["SOS"] + list(smi) + ["EOS"])
    return padded


padded_smiles = pad_start_end_token(smiles)
print(padded_smiles[:5])
```

```
[['SOS', 'C', 'EOS'], ['SOS', 'N', 'EOS'], ['SOS', 'O', 'EOS'], ['SOS', 'C', 'C', 'EOS'], ['SOS', 'C', 'N', 'EOS']]
```

Vocabulary: unique tokens

In [ ]:
```
vocab = np.unique(np.concatenate(padded_smiles))
print(vocab.shape)
```

```
(17,)
```

In [ ]:
```
class SmilesDataset(Dataset):
    def __init__(self, smiles, vocab):

        self.vocab = np.array(vocab, dtype=str).reshape(-1, 1)

        # One-hot encoding
        self.encoder = OneHotEncoder()
        self.encoder.fit(self.vocab)

        self.data = [
            torch.tensor(
                self.encoder.transform(np.array(s).reshape(-1, 1)).toarray(), # transform data
                dtype=torch.float
            ) for s in smiles
```

```
        ]

        # Padding: nn.utils.rnn.pad_sequence
        # shape: (n_samples, n_sequence, n_tokens)
        self.data = nn.utils.rnn.pad_sequence(self.data, batch_first=True)

        self.X = self.data[:, :-1, :]
        self.y = self.data[:, 1:, :]

    def __len__(self):
        return int(self.data.shape[0])

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]


data = SmilesDataset(padded_smiles, vocab)
input_size = data.vocab.shape[0] # should be 17
data.data
```

Out[ ]:
```
tensor([[[0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.]],

        [[0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.]],

        [[0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.]],

        ...,

        [[0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.]],

        [[0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.]],
```

```
        [[0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.]]])
```

Define Model

In [ ]:

```
class VanillaRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers=1):
        super().__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, input_size)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x, h=None):
        # rnn
        out, h = self.rnn(x, h)
        # fc
        out = self.fc(out)
        # softmax
        out = self.softmax(out)
        return out, h

    def init_hidden(self, batch_size):
        return torch.zeros(self.num_layers, batch_size, self.hidden_size)
```

Trainer

Training: try to predict the output tokens given inputs.

For example, a valid SMILES is `['SOS', 'C', 'N', 'EOS']`. Give model `['SOS', 'C', 'N']`, and try to let the model output `['C', 'N', 'EOS']`. In this way, the model can learn some information about probability distribution of the output tokens given inputs.

In [ ]:
```
class Trainer:
    def __init__(self, model, opt_method, learning_rate, batch_size, epoch, l2):
        self.model = model
        if opt_method == "sgdm":
            self.optimizer = torch.optim.SGD(model.parameters(), learning_rate, momentum=0.9)
        elif opt_method == "adam":
            self.optimizer = torch.optim.Adam(model.parameters(), learning_rate, weight_decay=l2)
        else:
            raise NotImplementedError("This optimization is not supported")

        self.epoch = epoch
        self.batch_size = batch_size


    def train(self, train_data, draw_curve=True):
        self.encoder = train_data.encoder

        train_loader = DataLoader(train_data, batch_size=self.batch_size, shuffle=True)
        train_loss_list, train_acc_list = [], []

        loss_func = nn.CrossEntropyLoss()
        for n in tqdm(range(self.epoch), leave=False):
            self.model.train()
            epoch_loss, epoch_acc = 0.0, 0.0
            for X_batch, y_batch in train_loader:
                batch_importance = y_batch.shape[0] / len(train_data)

                # batch outputs
                y_pred, _ = self.model(X_batch)

                # loss func
                batch_loss = loss_func(y_pred, y_batch)

                self.optimizer.zero_grad()
                batch_loss.backward()
                self.optimizer.step()

                # record accuracy
                batch_acc = torch.sum(torch.argmax(y_pred, axis=-1) == torch.argmax(y_batch, axis=-1)) / y_batch.shap
```

```
                epoch_acc += batch_acc.detach().cpu().item() * batch_importance
                epoch_loss += batch_loss.detach().cpu().item() * batch_importance

            train_acc_list.append(epoch_acc)
            train_loss_list.append(epoch_loss)

        if draw_curve:
            x_axis = np.arange(self.epoch)
            fig, axes = plt.subplots(1, 2, figsize=(10, 4))
            axes[0].plot(x_axis, train_loss_list, label="Train")
            axes[0].set_title("Loss")
            axes[0].legend()
            axes[1].plot(x_axis, train_acc_list, label='Train')
            axes[1].set_title("Accuracy")
            axes[1].legend()

    def sample(self, num_seq=10):
        self.model.eval()
        seqs = []
        with torch.no_grad():
            for _ in tqdm(range(num_seq), leave=False):
                chars = ['SOS']
                hidden = self.model.init_hidden(1)
                while chars[-1] != 'EOS':
                    input_encoding = self.encoder.transform(np.array([chars[-1]]).reshape(-1, 1)).toarray()
                    input_encoding = torch.tensor(input_encoding, dtype=torch.float).reshape(1, 1, -1)
                    out, hidden = self.model(input_encoding, hidden)

                    prob = out.detach().numpy().flatten()
                    prob /= np.sum(prob)

                    index = np.random.choice(self.model.input_size, p=prob)
                    out_encoding = np.zeros((1, self.model.input_size))
                    out_encoding[0, index] = 1.0
                    char = data.encoder.inverse_transform(out_encoding).flatten().tolist()[0]
                    chars.append(char)
                seqs.append(''.join(chars[1:-1]))
        return seqs


def validate(seq):
```

```
    num = len(seq)
    unique = set(seq)
    valid = []
    for s in unique:
        mol = Chem.MolFromSmiles(s)
        if mol is not None:
            valid.append(s)

    print(f"Number of unique SMILES: {len(unique)}")
    print(f"Number of valid & unique SMILES: {len(valid)}")
    return valid
```

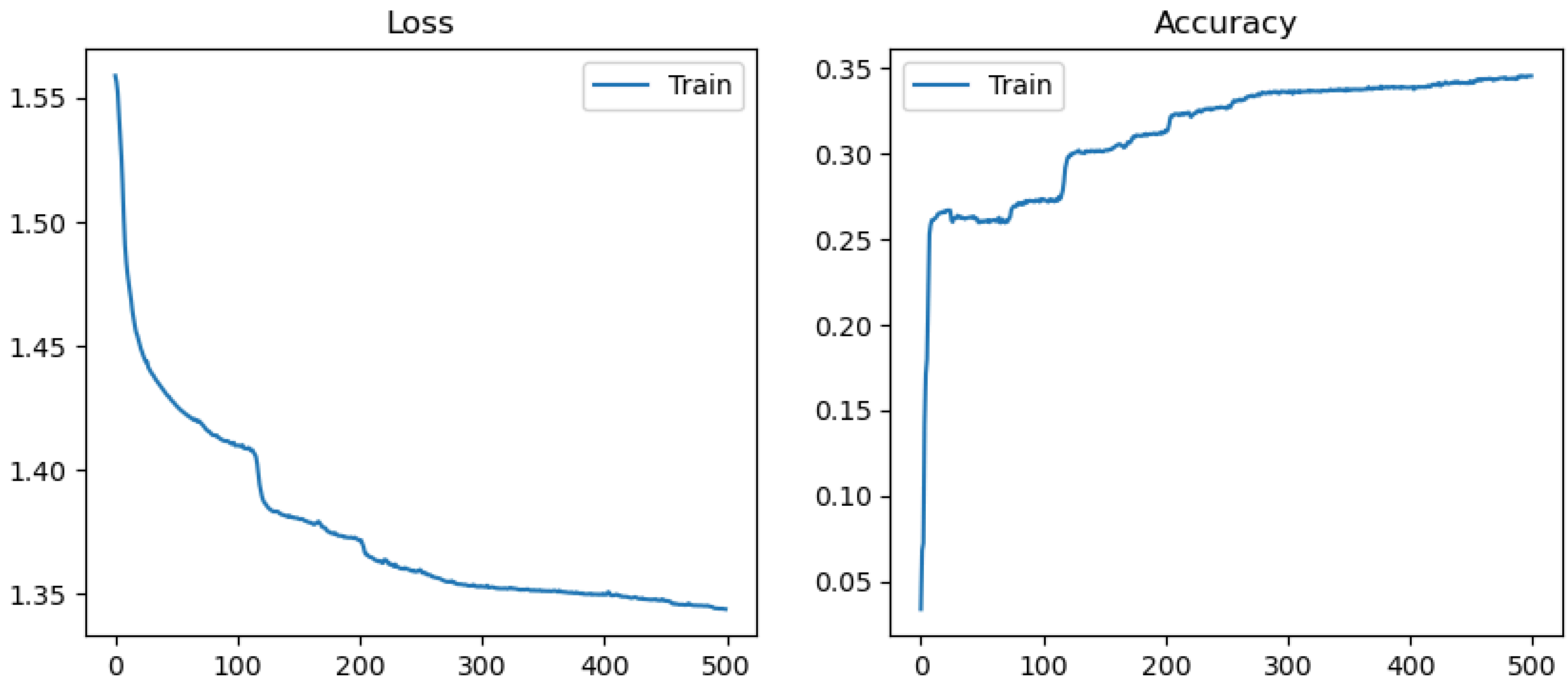
In [ ]:

```
model = VanillaRNN(input_size, 32, 1)
trainer = Trainer(model, "adam", 1e-3, 128, 500, 1e-5)
trainer.train(data)
seqs = trainer.sample(1000)
validate(seqs)
```

```
Number of unique SMILES: 15
Number of valid & unique SMILES: 9
```

Out[ ]:

```
['C=CCC=O',
 'C=CCC=C',
 'C=CC=O',
 'C=CC=COC',
 'C=CCC=CO',
 'C=CC=C',
 'C=CC(=O)C=O',
 'C=CC=CCC',
 'C=CC=CC=O']
```


Loss
Train
1.55
1.50
1.45
1.40
1.35
0
100
200
300
400
500
Accuracy
Train
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
100
200
300
400
500

In [ ]:

In [ ]: