Lab 5 Report:

Create Arthur Conan Doyle AI with RNN

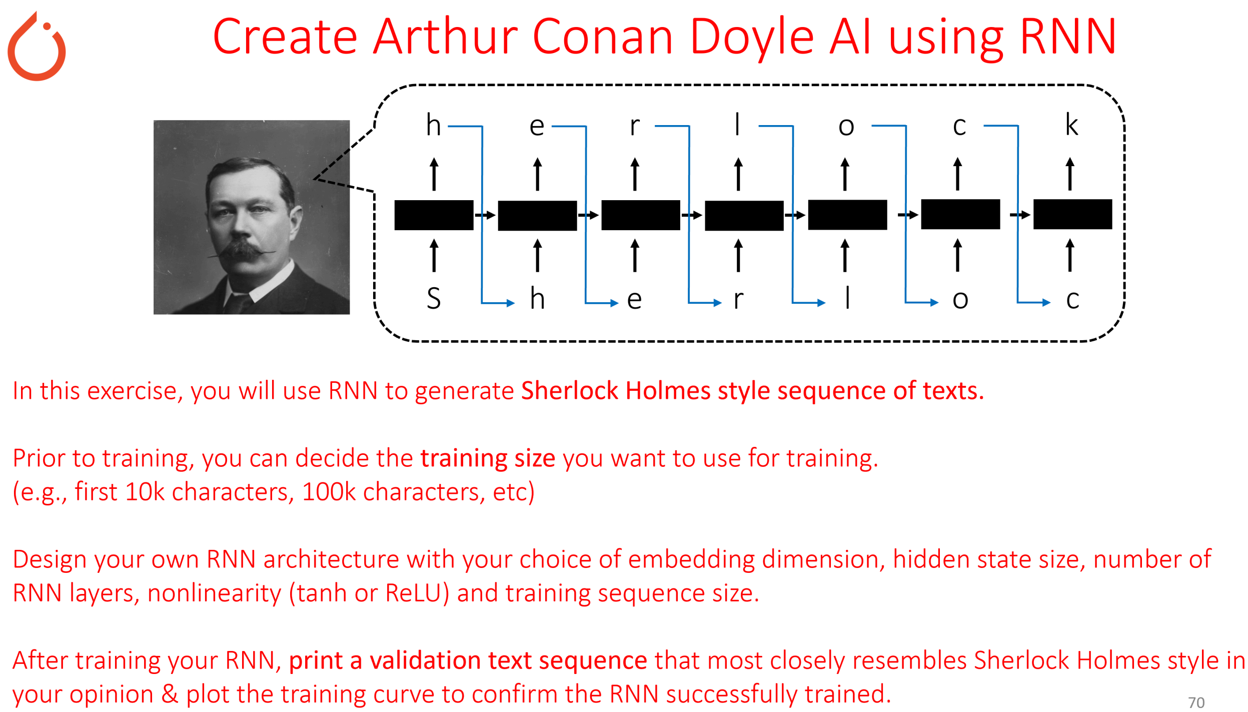
Name: Travis Hand

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import torch
from torch.distributions import Categorical
```

```
from IPython.display import Image # For displaying images in colab jupyter cell
```

```
Image('lab5_exercise.png', width = 1000)
```



Prepare Data

```
# You will train on the first N characters of the Sherlock Holmes book
# Pick the size of your training data, i.e. N
data_size_to_train = 10000

# Load the Sherlock Holmes data up to data_size_to_train

data = open('./sherlock.txt', 'r').read()[:data_size_to_train]

# Find the set of unique characters within the training data
characters = sorted(list(set(data)))
print(characters)

# total number of characters in the training data and number of unique characters
data_size, vocab_size = len(data), len(characters)

print("Data has {} characters, {} unique".format(data_size, vocab_size))
```

```
['\n', ' ', '!', '"', "'", '(', ')', ',', '-', '.', '1', '7', '8', ';', '?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J
Data has 10000 characters, 65 unique
```

```
# Use Python Dictionary to map the characters to numbers and vice versa

char_indices = {ch : i for i, ch in enumerate(characters)}
```

```
# Use the character_to_num dictionary to map each character in the training dataset to a number

char_indices = {ch : i for i, ch in enumerate(characters)}
idx_to_char = {i : ch for i, ch in enumerate(characters)}
print(char_indices)
print(idx_to_char)
```

```
{'\n': 0, ' ': 1, '!': 2, '"': 3, "'": 4, '(': 5, ')': 6, ',': 7, '-': 8, '.': 9, '1': 10, '7': 11, '8': 12, ';': 13, '?': 1
{0: '\n', 1: ' ', 2: '!', 3: '"', 4: "'", 5: '(', 6: ')', 7: ',', 8: '-', 9: '.', 10: '1', 11: '7', 12: '8', 13: ';', 14: '?
```

```
data = list(data)
for i, ch in enumerate(data):
    data[i] = char_indices[ch]

print(data[:20])
```

```
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 34]
```

Define Model

```
class CharRNN(torch.nn.Module):

    def __init__(self, num_embeddings, embedding_dim, input_size, hidden_size, num_layers, output_size):

        super(CharRNN, self).__init__()

        # RNN layer
        self.rnn = torch.nn.RNN(input_size, hidden_size, num_layers, nonlinearity='relu')

        # Embedding layer
        self.embedding = torch.nn.Embedding(num_embeddings, embedding_dim)

        # Decoder layer
        self.decoder = torch.nn.Linear(hidden_size, output_size)

        self.dropout = torch.nn.Dropout(0.25)

    def forward(self, input_seq, hidden_state):

        # Embed the input sequence
        embedded_seq = self.embedding(input_seq)

        # Pass the embedded sequence through the RNN
        output, hidden_state = self.rnn(embedded_seq, hidden_state)
        output = self.dropout(output)

        # Decode the output of the RNN to the desired output size
        decoded_output = self.decoder(output)

        # Deteach the hidden state to prevent backpropagation through time
        return decoded_output, hidden_state.detach()
```

Define Hyperparameters

```
# Fix random seed
torch.manual_seed(25)

# Define RNN network
rnn = CharRNN(num_embeddings=vocab_size, embedding_dim=1000, input_size=1000, hidden_size=512, num_layers=4, output_size=vocab_s

# Define learning rate and epochs
learning_rate = 0.0005
```

```
epochs = 100

# Size of the input sequence to be used during training and validation
# Split the training data into training and validation sets
training_sequence_len = 100
validation_sequence_len = 400

# Define loss function and optimizer
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate, weight_decay=1e-5)
device = torch.device("cpu")
# add .cuda() for GPU acceleration
if torch.cuda.is_available():
    rnn = rnn.cuda()
    loss_fn = loss_fn.cuda()
    device = torch.device("cuda")
```

Identify Tracked Values

```
# Tracking training loss per each input/target sequence fwd/bwd pass
training_loss = []
validation_accuracy = []
```

Train Model

```
import tqdm

data = torch.unsqueeze(torch.tensor(data), dim=1)

# Training Loop -----------------------------------------------------------------------------------------------------

for epoch in tqdm.trange(epochs):

    character_loc = np.random.randint(100) # Randomize the starting point of the training sequence
    iteration = 0 # Initialize iteration counter
    hidden_state = None # Reset hidden state at the beginning of each epoch

    while character_loc + training_sequence_len + 1 < data_size:
        input_seq = data[character_loc:character_loc + training_sequence_len] # Select the input sequence
        target_seq = data[character_loc + 1:character_loc + training_sequence_len + 1] # Offset input sequence by 1 for the targ

        output, hidden_state = rnn(input_seq, hidden_state) # Pass the input sequence through the RNN
        loss = loss_fn(torch.squeeze(output), torch.squeeze(target_seq)) # Calculate the loss

        training_loss.append(loss.item()) # Keep track of the training loss

        optimizer.zero_grad() # Zero out the gradients
        loss.backward() # Backpropagation
        optimizer.step() # Update the weights

        character_loc += training_sequence_len # Move to the next training sequence
        iteration += 1 # Increment the iteration counter

    # Sample and generate a text sequence after every epoch ----------------------------------------------------------------

    print("----------------------------------------")

    character_loc = 0
    hidden_state = None

    rand_index = np.random.randint(3000, 5000)
    input_seq = data[rand_index:rand_index + 1]

    with torch.no_grad():

        while character_loc < validation_sequence_len:
            output, hidden_state = rnn(input_seq, hidden_state) # Pass the input sequence through the RNN
            # output = torch.nn.functional.softmax(torch.squeeze(output), dim=0)

            # character_distribution = Categorical(logits=output)
            character_num = torch.argmax(output[0]) # Get the most probable character
```

```
        print(idx_to_char[character_num.item()], end="") # Print the character
        input_seq[0] = character_num.item() # Update the input sequence with the predicted character

        character_loc += 1

    print("\nAverage training loss for epoch {}: {}".format(epoch, np.mean(training_loss[-iteration:])))

    print("\n----------------------------------------")
```

```
  1%|          | 1/100 [00:04<06:44,  4.08s/it]----------------------------------------
he hat anthe he hath and the he he the hathe hame as anethe as and and and and the he anthe hore anthe he the hame anethate
Average training loss for epoch 0: 2.7456888309632888

----------------------------------------
  2%|▏         | 2/100 [00:07<06:25,  3.93s/it]----------------------------------------
 a as a a the he he the hat a a the he has a as the hat a and the he hore and he he he hat to the hame and and he has a and
Average training loss for epoch 1: 2.248498634858565

----------------------------------------
  3%|▎         | 3/100 [00:11<06:22,  3.94s/it]----------------------------------------
ure and he ham a as a a stast the has and as a as a as a a a and hat and and and hime hame hat a as a as a a and ham a as a
Average training loss for epoch 2: 2.064116699527008

----------------------------------------
  4%|▍         | 4/100 [00:15<06:13,  3.89s/it]----------------------------------------
e has and had to stand the stand the has and him a as a man the stand has and he ham the has a aster and he has and has and
Average training loss for epoch 3: 1.9322091738382976

----------------------------------------
  5%|▌         | 5/100 [00:19<06:06,  3.85s/it]----------------------------------------
ere of the stand the has and the strang a far the stand he has and where and the stand the stame as a lould he ham a and the
Average training loss for epoch 4: 1.8118386834558815

----------------------------------------
  6%|▌         | 6/100 [00:23<05:58,  3.81s/it]----------------------------------------
sk and what he ask the has and the has ask which whom and have the lation the had him the has and he has asked the has and h
Average training loss for epoch 5: 1.7196412550078497

----------------------------------------
  7%|▋         | 7/100 [00:26<05:52,  3.79s/it]----------------------------------------
han what the studieg which the studious and which which whom to studical the had the had astence to be as for to stand the s
Average training loss for epoch 6: 1.6189565447845844

----------------------------------------
  8%|▊         | 8/100 [00:30<05:46,  3.76s/it]----------------------------------------
for."

 "Nours of the studious and him to go not the studious of the studious and which of the prown to strice to be ford the prowe
Average training loss for epoch 7: 1.5153331088297295

----------------------------------------
  9%|▉         | 9/100 [00:34<05:41,  3.76s/it]----------------------------------------
urselof to meeticulars and entheression to studical outh of the studious and has astents the has asterangelf the has never o
Average training loss for epoch 8: 1.4193804372440686

----------------------------------------
 10%|█         | 10/100 [00:38<05:37,  3.75s/it]----------------------------------------
nts.

 "A has asked hablession."

 "I have the have the have the should not the have the have the have the lanched the sich he istentssession."

 "He is far the have the companion."

 "A should not he istents.
```

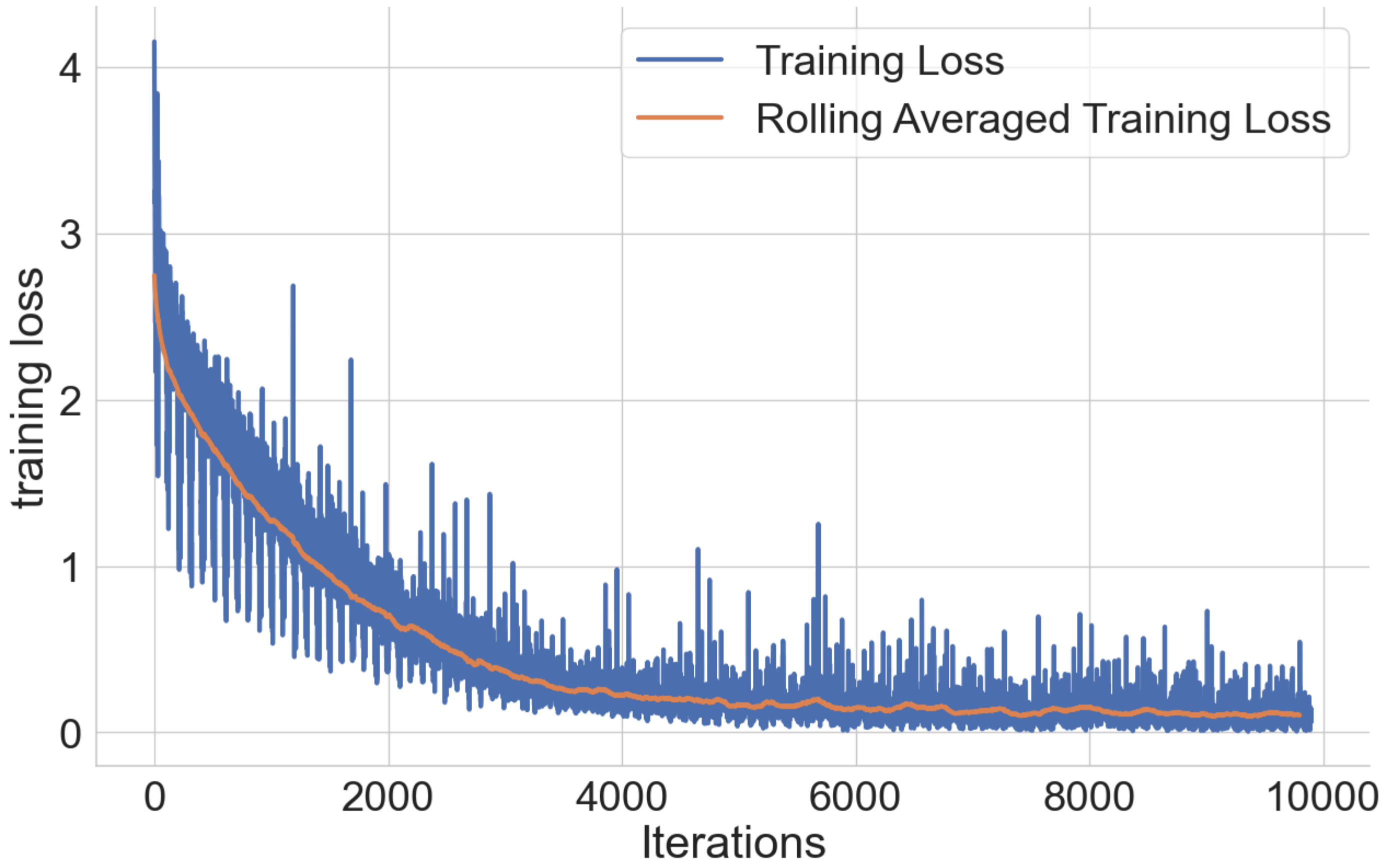
Visualize & Evaluate Model

```
# Import seaborn for prettier plot
import seaborn as sns

sns.set(style = 'whitegrid', font_scale = 2.5)


# Plot the training loss and rolling mean training loss with respect to iterations
# Feel free to change the window size
plt.figure(figsize = (15, 9))
```

```
plt.plot(training_loss, linewidth = 3, label = 'Training Loss')
plt.plot(np.convolve(training_loss, np.ones(100), 'valid') / 100,
         linewidth = 3, label = 'Rolling Averaged Training Loss')
plt.ylabel("training loss")
plt.xlabel("Iterations")
plt.legend()
sns.despine()
```




```
# Final test to evaluate the model performance -----------------------------------------------------------

character_loc = 0
hidden_state = None

rand_index = np.random.randint(2000, 5000)
input_seq = data[rand_index:rand_index + 1]

print("----------------------------------------")

with torch.no_grad():

    while character_loc < validation_sequence_len:
        output, hidden_state = rnn(input_seq, hidden_state)
        # output = torch.nn.functional.softmax(torch.squeeze(output), dim=0)
        # print(output.shape)
        # character_distribution = Categorical(logits=output)

        character_num = torch.argmax(output, dim=-1)

        print(idx_to_char[character_num.item()], end="")
        input_seq[0] = character_num.item()

        character_loc += 1
```

```
print("\n----------------------------------------")
```

```
----------------------------------------
bbey of mine, but now
 I hailed him with enthusiasm, and he has never taken out any systematic medical classes. His
 studies are very desultory and eccentric, but he has amassed a lot of
 out-of-the way knowledge which would astonish his professors."

 "Did you never ask him what he was going in for?" I asked.

 "No- he has never taken out any systematic medical classes. His
 studies are very desu
----------------------------------------
```