```python
##### DESCRIPTION #####
# Use to train a model on the specified dataset and hyperparameters.

import string
import random
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
import time
import sys
import os
from tqdm import tqdm

FILE_NAME = 'complete_sherlock_holmes.txt'
DATASET = 'Complete Sherlock Holmes'

##### HYPERPARAMETERS #####
CELL_TYPE = 'RNN' # DEFAULTS TO 'RNN'. Options: ['RNN', 'LSTM', 'GRU', 'RELU']
OPTIM_TYPE = 'Adam' # DEFAULTS TO 'Adam'. Options: ['Adam', 'ASGD', 'Adagrad','RMSprop']
HIDDEN_LAYERS = 1 # DEFAULT: 1
HIDDEN_SIZE = 100 # DEFAULT: 100

LEARNING_RATE = 0.005 # 0.005 for Adam, 0.05 for other optimizers
INPUT_SEQUENCE = 128 # DEFAULT: 128

TRAINING_ITERATIONS = 20000 # DEFAULT: 20000
INITIAL_SEQUENCE = '\n' # To use for eval

print('Running on',CELL_TYPE,OPTIM_TYPE,HIDDEN_LAYERS,HIDDEN_SIZE)

all_chars = string.printable
n_chars = len(all_chars)
file = open('./'+FILE_NAME).read()
file_len = len(file)

def get_random_seq():
    seq_len     = INPUT_SEQUENCE  # The length of an input sequence.
    start_index = random.randint(0, file_len - seq_len)
    end_index   = start_index + seq_len + 1
    return file[start_index:end_index]
def seq_to_onehot(seq):
    tensor = torch.zeros(len(seq), 1, n_chars)
    for t, char in enumerate(seq):
        index = all_chars.index(char)
        tensor[t][0][index] = 1
    return tensor
def seq_to_index(seq):
    tensor = torch.zeros(len(seq), 1)
    for t, char in enumerate(seq):
        tensor[t] = all_chars.index(char)
    return tensor
def get_input_and_target():
    seq    = get_random_seq()
    input  = seq_to_onehot(seq[:-1]) # Input is represented in one-hot.
    target = seq_to_index(seq[1:]).long() # Target is represented in index.
```

```
    return input, target

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('Device =', device)

class Net(nn.Module):
    def __init__(self):
        # Initialization.
        super(Net, self).__init__()
        self.input_size  = n_chars # Input size: Number of unique chars.
        self.hidden_size = HIDDEN_SIZE
        self.output_size = n_chars # Output size: Number of unique chars.

        # Ensures the size of the hidden layer stack does not exceed 3
        self.layers = HIDDEN_LAYERS
        if HIDDEN_LAYERS>3:
            self.layers=3

        # Create a rnn cell for the stack
        def create_cell(size_in, size_out):
            if CELL_TYPE=='LSTM':
                return nn.LSTMCell(size_in, size_out)
            elif CELL_TYPE=='GRU':
                return nn.GRUCell(size_in, size_out)
            elif CELL_TYPE=='RELU': # Not used in testing report
                return nn.RNNCell(size_in, size_out, nonlinearity='relu')
            else:
                return nn.RNNCell(size_in, size_out)

        self.rnn = create_cell(self.input_size, self.hidden_size)

        if HIDDEN_LAYERS>=2:
            self.rnn2 = create_cell(self.hidden_size, self.hidden_size)
        if HIDDEN_LAYERS>=3:
            self.rnn3 = create_cell(self.hidden_size, self.hidden_size)

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

    def forward(self, input, hidden, cell, hidden2=False, cell2=False, hidden3=False, cell3=False):
        # Forward function.
        #   takes in the 'input' and 'hidden' tensors,
        #   can also take in 'cell state' tensor if cell type is 'LSTM',
        #   takes additional hidden and cell state tensors for each layer
        if CELL_TYPE=='LSTM':
            hidden, cell = self.rnn(input, (hidden,cell))
            if self.layers>=2:
                hidden2, cell2 = self.rnn2(hidden, (hidden2,cell2))
            if self.layers>=3:
                hidden3, cell3 = self.rnn3(hidden2, (hidden3,cell3))
        else:
            hidden = self.rnn(input, hidden)
            if self.layers>=2:
                hidden2 = self.rnn2(hidden, hidden2)
            if self.layers>=3:
                hidden3 = self.rnn3(hidden2, hidden3)

```

```python
        # Linear transformation (fully connected layer) to the output
        if self.layers==3:
            output = self.fc(hidden3)
            return output, hidden, cell, hidden2, cell2, hidden3, cell3
        elif self.layers==2:
            output = self.fc(hidden2)
            return output, hidden, cell, hidden2, cell2
        else:
            output = self.fc(hidden)
            return output, hidden, cell
    def init_hidden(self):
        # Initial hidden state.
        return torch.zeros(1, self.hidden_size).to(device)
    def init_cell(self):
        # Initial cell state.
        return torch.zeros(1, self.hidden_size).to(device)

net = Net()
net.to(device)

# Training step function
def train_step(net, opt, input, target):
    seq_len = input.shape[0]
    hidden = net.init_hidden() # Initial hidden state
    cell = net.init_cell() # Initial cell state
    if HIDDEN_LAYERS >=2:
        hidden2 = net.init_hidden()
        cell2 = net.init_cell()
    if HIDDEN_LAYERS >=3:
        hidden3 = net.init_hidden()
        cell3 = net.init_cell()

    net.zero_grad()
    loss = 0 # Initial loss.

    for t in range(seq_len): # For each one in the input sequence
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(input[t], hidden, cell, hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(input[t], hidden, cell, hidden2, cell2)
        else:
            output, hidden, cell = net(input[t], hidden, cell)
        loss += loss_func(output, target[t])

    loss.backward() # Backward.
    opt.step() # Update the weights.

    return loss / seq_len

# Evaluation step function
def eval_step(net, init_seq=INITIAL_SEQUENCE, predicted_len=100):
    # Initialize the hidden state, input and the predicted sequence
    hidden        = net.init_hidden()
    cell          = net.init_cell()
    if HIDDEN_LAYERS >=2:
```

```
        hidden2 = net.init_hidden()
        cell2 = net.init_cell()
    if HIDDEN_LAYERS >=3:
        hidden3 = net.init_hidden()
        cell3 = net.init_cell()
    init_input    = seq_to_onehot(init_seq).to(device)
    predicted_seq = init_seq

    # Use initial string to "build up" hidden state.
    for t in range(len(init_seq) - 1):
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(init_input[t], hidde
cell, hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(init_input[t], hidden, cell, hidden2
cell2)
        else:
            output, hidden, cell = net(init_input[t], hidden, cell)
    # Set current input as the last character of the initial string.
    input = init_input[-1]

    # Predict more characters after the initial string.
    for t in range(predicted_len):
        # Get the current output and hidden state.
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(input, hidden, cell,
hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(input, hidden, cell, hidden2, cell2)
        else:
            output, hidden, cell = net(input, hidden, cell)

        # Sample from the output as a multinomial distribution.
        try:
            predicted_index = torch.multinomial(output.view(-1).exp(), 1)[0]
        except: # Added post to resolve errors with tensors containing 'inf'/'nan' values
            predicted_index = torch.multinomial(output.view(-1).exp().clamp(0.0,3.4e38), 1)
[0]
        # Add predicted character to the sequence and use it as next input.
        predicted_char  = all_chars[predicted_index]
        predicted_seq  += predicted_char
        # Use the predicted character to generate the input of next round.
        input = seq_to_onehot(predicted_char)[0].to(device)

    return predicted_seq


##### MAIN ALGORITHM #####

iters = TRAINING_ITERATIONS # Number of training iterations.

# The loss variables.
all_losses = []
# Initialize the optimizer and the loss function.
if(OPTIM_TYPE=='ASGD'):
    opt=torch.optim.ASGD(net.parameters(), lr=LEARNING_RATE)
if(OPTIM_TYPE=='Adagrad'): # Not used in testing results
```

```
    opt=torch.optim.Adagrad(net.parameters(), lr=LEARNING_RATE)
if(OPTIM_TYPE=='RMSprop'): # Not used in testing results
    opt=torch.optim.RMSprop(net.parameters(), lr=LEARNING_RATE)
else:
    opt = torch.optim.Adam(net.parameters(), lr=LEARNING_RATE)
loss_func = nn.CrossEntropyLoss()

# Training procedure.
start_time = time.time()
for i in tqdm(range(iters)):
    input, target = get_input_and_target() # Fetch input and target.
    input, target = input.to(device), target.to(device) # Move to GPU memory.
    loss = train_step(net, opt, input, target)   # Calculate the loss.
    all_losses.append(loss)

end_time = time.time()
total_time = end_time - start_time

# Calculates summary of losses
i_half = int(len(all_losses)*0.5)
i_quart = int(len(all_losses)*0.75)
loss_avg = np.sum(np.array(all_losses))/len(all_losses)
loss_avg_half = np.sum(np.array(all_losses[i_half:]))/len(all_losses[i_half:])
loss_avg_quart = np.sum(np.array(all_losses[i_quart:]))/len(all_losses[i_quart:])
loss_list=[elem.item() for elem in [loss_avg,loss_avg_half,loss_avg_quart]]
rolling_losses=[]
losses_copy = [i.item() for i in all_losses]
for i in range(len(losses_copy)):
    temp=losses_copy[np.max((i-100, 0)):i+1]
    rolling_losses.append(np.sum(temp)/len(temp))

plt.xlabel('iters')
plt.ylabel('loss')
plt.hlines(loss_list,0,len(losses_copy)-1,['red','orange','green'],'dashed')
plt.plot(rolling_losses)
plt.ylim(0,5)

print('Avg loss: {}'.format(loss_avg))
print('Avg loss last half: {}'.format(loss_avg_half))
print('Avg loss last quarter: {}'.format(loss_avg_quart))
print()
print('Training time: {} sec'.format(total_time))
print('{} min | {:.3f} hr'.format(total_time/60, total_time/3600))

# Creates a folder path to save training results to
PATH = 'Results'
for folder in [DATASET, CELL_TYPE, OPTIM_TYPE, HIDDEN_LAYERS, HIDDEN_SIZE]:
    folder=str(folder)
    if not os.path.isdir(PATH+'/'+folder):
        os.mkdir(PATH+'/'+folder)
    PATH+='/'+folder
PATH+='/'

print()
print('Results saved to: {}'.format(PATH))
model_path = PATH+'model.pt' # Saves model parameters

```

```
torch.save(net.state_dict(), model_path)

# Sequence of all 20000 loss values
file = open(PATH+'all_losses.txt','w')
file.write(' '.join([str(elem) for elem in losses_copy]))
file.close()

# A 5000 char sample generated after training
file = open(PATH+'sample.txt','w')
file.write(eval_step(net, predicted_len=5000))
file.close()

# Information on training
file = open(PATH+'info.txt','w')
file.write('Iterations: {}\n\n'.format(TRAINING_ITERATIONS))
file.write('Dataset: {}\nInput Size: {}\nLearning Rate: {}\n\n'.format(DATASET, INPUT_SEQUENCE, LEARNING_RATE))
file.write('Cell Type: {}\nOptimizer: {}\nHidden Layers: {}\nHidden Size: {}\n\n'.format(CELL_TYPE, OPTIM_TYPE, HIDDEN_LAYERS, HIDDEN_SIZE))
```

```python
##### DESCRIPTION #####
# Use to recall a previously trained model to keep generating more text if desired.

import string
import torch
import torch.nn as nn
import sys
import os

DATASET = 'Complete Sherlock Holmes'

##### HYPERPARAMETERS #####
# These must be the same as a previously trained model present in the files
CELL_TYPE = 'RNN'
OPTIM_TYPE = 'Adam'
HIDDEN_LAYERS = 1
HIDDEN_SIZE = 100

##### GENERATION PARAMETERS #####
# Alter the length of the new generated sequence and it's initial sequence
PREDICTION_LENGTH = 100
INITIAL_SEQUENCE = '\n'

print('Recalling model',CELL_TYPE,OPTIM_TYPE,HIDDEN_LAYERS,HIDDEN_SIZE)
print('Generating {} char, with init_seq: {}'.format(PREDICTION_LENGTH, INITIAL_SEQUENCE))

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('Device =', device)

all_chars = string.printable
n_chars = len(all_chars)

def seq_to_onehot(seq):
    tensor = torch.zeros(len(seq), 1, n_chars)
    for t, char in enumerate(seq):
        index = all_chars.index(char)
        tensor[t][0][index] = 1
    return tensor

class Net(nn.Module):
    def __init__(self):
        # Initialization.
        super(Net, self).__init__()
        self.input_size  = n_chars # Input size: Number of unique chars.
        self.hidden_size = HIDDEN_SIZE
        self.output_size = n_chars # Output size: Number of unique chars.

        # Ensures the size of the hidden layer stack does not exceed 3
        self.layers = HIDDEN_LAYERS
        if HIDDEN_LAYERS>3:
            self.layers=3

        # Create a rnn cell for the stack
        def create_cell(size_in, size_out):
            if CELL_TYPE=='LSTM':
                return nn.LSTMCell(size_in, size_out)
```

```
            elif CELL_TYPE=='GRU':
                return nn.GRUCell(size_in, size_out)
            elif CELL_TYPE=='RELU': # Not used in testing report
                return nn.RNNCell(size_in, size_out, nonlinearity='relu')
            else:
                return nn.RNNCell(size_in, size_out)

        self.rnn = create_cell(self.input_size, self.hidden_size)

        if HIDDEN_LAYERS>=2:
            self.rnn2 = create_cell(self.hidden_size, self.hidden_size)
        if HIDDEN_LAYERS>=3:
            self.rnn3 = create_cell(self.hidden_size, self.hidden_size)

        self.linear = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, cell, hidden2=False, cell2=False, hidden3=False, cell3=False):
        # Forward function.
        #   takes in the 'input' and 'hidden' tensors,
        #   can also take in 'cell state' tensor if cell type is 'LSTM',
        #   takes additional hidden and cell state tensors for each layer
        if CELL_TYPE=='LSTM':
            hidden, cell = self.rnn(input, (hidden,cell))
            if self.layers>=2:
                hidden2, cell2 = self.rnn2(hidden, (hidden2,cell2))
            if self.layers>=3:
                hidden3, cell3 = self.rnn3(hidden2, (hidden3,cell3))
        else:
            hidden = self.rnn(input, hidden)
            if self.layers>=2:
                hidden2 = self.rnn2(hidden, hidden2)
            if self.layers>=3:
                hidden3 = self.rnn3(hidden2, hidden3)

        # Linear transformation (fully connected layer) to the output
        if self.layers==3:
            output = self.linear(hidden3)
            return output, hidden, cell, hidden2, cell2, hidden3, cell3
        elif self.layers==2:
            output = self.linear(hidden2)
            return output, hidden, cell, hidden2, cell2
        else:
            output = self.linear(hidden)
            return output, hidden, cell
    def init_hidden(self):
        # Initial hidden state.
        return torch.zeros(1, self.hidden_size).to(device)
    def init_cell(self):
        # Initial cell state.
        return torch.zeros(1, self.hidden_size).to(device)

def eval_step(net, init_seq='\n', predicted_len=100):
    # Initialize the hidden state, input and the predicted sequence
    hidden        = net.init_hidden()
    cell          = net.init_cell()
    if HIDDEN_LAYERS >=2:
```

```
        hidden2 = net.init_hidden()
        cell2 = net.init_cell()
    if HIDDEN_LAYERS >=3:
        hidden3 = net.init_hidden()
        cell3 = net.init_cell()
    init_input    = seq_to_onehot(init_seq).to(device)
    predicted_seq = init_seq

    # Use initial string to "build up" hidden state.
    for t in range(len(init_seq) - 1):
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(init_input[t], hidde
cell, hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(init_input[t], hidden, cell, hidden2
cell2)
        else:
            output, hidden, cell = net(init_input[t], hidden, cell)
    # Set current input as the last character of the initial string.
    input = init_input[-1]

    # Predict more characters after the initial string.
    for t in range(predicted_len):
        # Get the current output and hidden state.
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(input, hidden, cell,
hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(input, hidden, cell, hidden2, cell2)
        else:
            output, hidden, cell = net(input, hidden, cell)

        # Sample from the output as a multinomial distribution.
        try:
            predicted_index = torch.multinomial(output.view(-1).exp(), 1)[0]
        except: # Added post to resolve errors with tensors containing 'inf'/'nan' values
            predicted_index = torch.multinomial(output.view(-1).exp().clamp(0.0,3.4e38), 1)
[0]
        # Add predicted character to the sequence and use it as next input.
        predicted_char  = all_chars[predicted_index]
        predicted_seq  += predicted_char
        # Use the predicted character to generate the input of next round.
        input = seq_to_onehot(predicted_char)[0].to(device)

    return predicted_seq

PATH = 'Results'
for folder in [DATASET, CELL_TYPE, OPTIM_TYPE, HIDDEN_LAYERS, HIDDEN_SIZE]:
    folder=str(folder)
    PATH+='/'+folder
PATH+='/model.pt'

net=Net()
net.to(device)

net.load_state_dict(torch.load(PATH))
net.eval()
```

```
print(eval_step(net, init_seq=INITIAL_SEQUENCE, predicted_len=PREDICTION_LENGTH))
```

Analysis of Results

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import string
```

Calculations about expected metrics

Here I have taken the dataset and determined what metrics may be expected out of some randomly picked 5000 character sequence to compare with the generated sample from each model.

In [2]:
```
file = open('./complete_sherlock_holmes.txt').read()
file_len = len(file)
```

In [3]:
```
words = re.split(' |\n',file)
word_list = [w for w in words if len(w)>0]
n1 = 5000*len(word_list)/file_len
print('Expected words per 5000 char: {}'.format(n1))
```

```
Expected words per 5000 char: 849.9241978679597
```

This would be the expected "word count" out of some 5000 character sequence. That is, the number of separate sequences of alphanumeric chacters and symbols, upper or lowercase, separated by spaces or new lines.

In [4]:
```
#string.printable[:95]
file2 = ''.join([i for i in file if i in string.printable[:95]])
n2 = len(file2)/file_len*5000
print('Expected characters w/ spaces per 5000 char: {}'.format(n2))
file3 = ''.join([i for i in file if i in string.printable[:94]])
n3 = len(file3)/file_len*5000
print('Expected characters w/o spaces per 5000 char: {}'.format(n3))
```

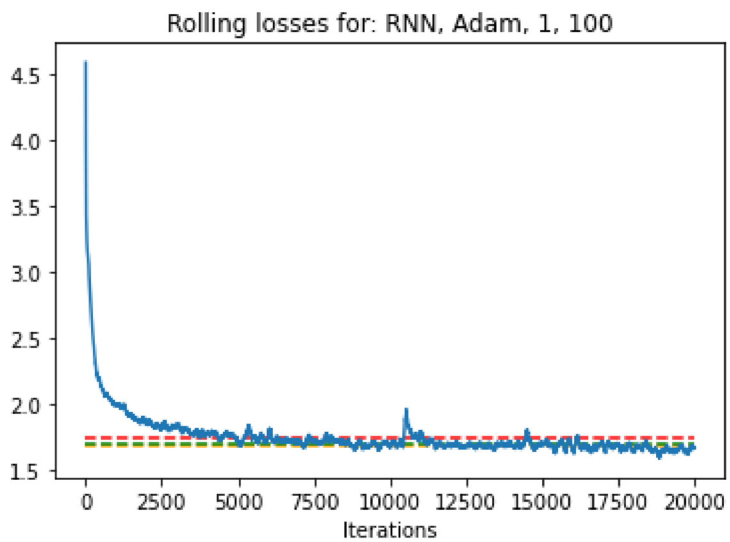
```
Expected characters w/ spaces per 5000 char: 4901.287202110336
Expected characters w/o spaces per 5000 char: 3732.427788378544
```

Characters with spaces would be the 5000 characters minus any format characters such as new line ('\n'), and without spaces would also subtract the number of space characters.

Tracking of Losses

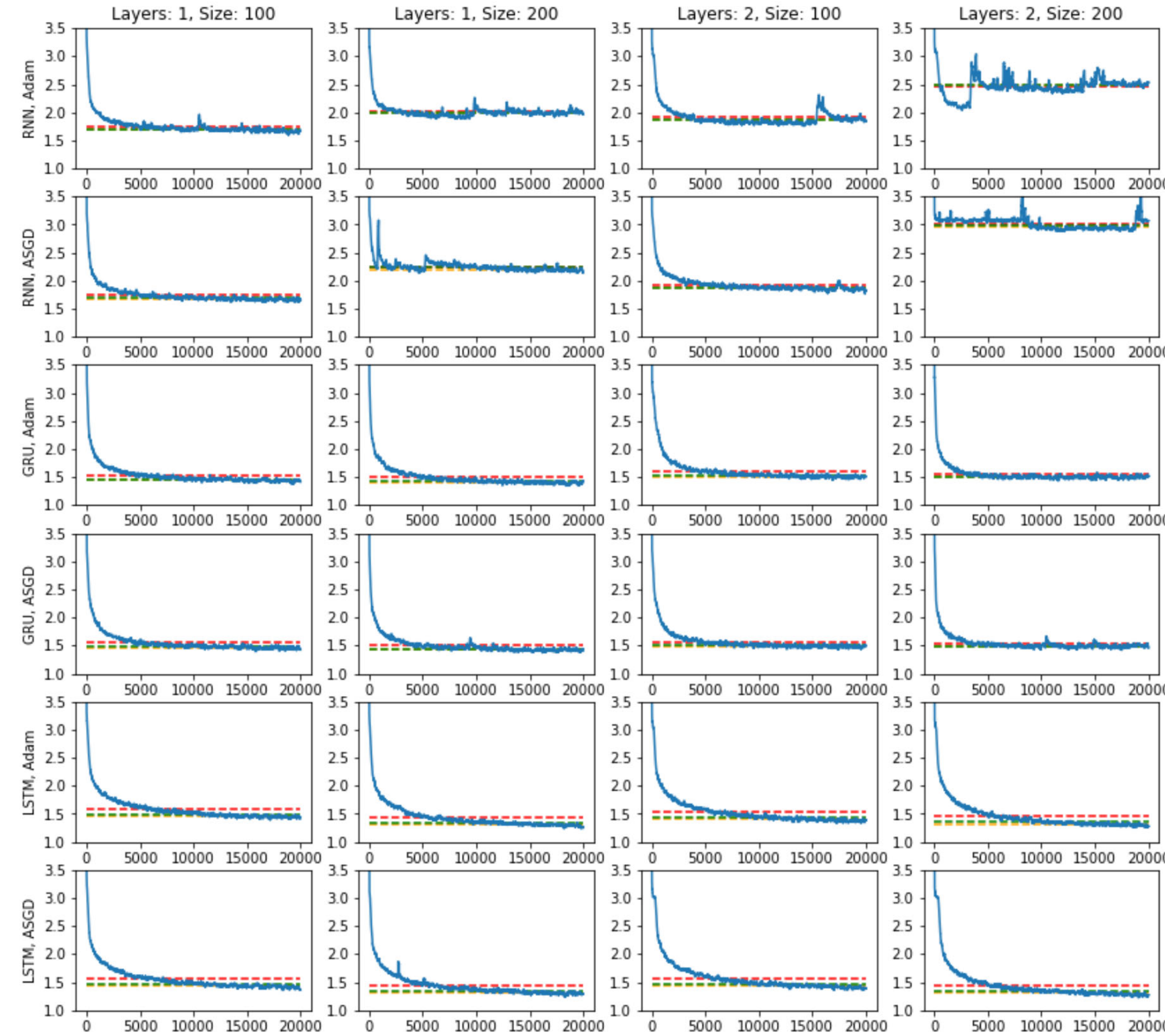
For each model I tracked the losses across all iterations in order to make sure that the model could converge after 20000 iterations. In order to smooth out the graph and large jumps in the loss, I took the rolling loss from the previous 100 iterations and plotted this sequence of losses along with horizontal lines showing the average loss across all iterations, across the last half (10000 iterations), and last quarter (5000 iterations). Shown in dotted red, orange, and green lines respectively.

In [5]:
```
file = open('./Results/Complete Sherlock Holmes/RNN/Adam/1/100/all_losses.txt').r
losses = [float(i) for i in file.split(' ')]
rolling_losses=[]
for i in range(len(losses)):
    temp=losses[np.max((i-100, 0)):i+1]
    rolling_losses.append(np.sum(temp)/len(temp))
avg = np.sum(losses)/len(losses)
avg_half = np.sum(losses[10000:])/len(losses[10000:])
avg_quart = np.sum(losses[5000:])/len(losses[5000:])
plt.plot(rolling_losses)
plt.hlines([avg,avg_half,avg_quart],0,len(losses)-1,['red','orange','green'],'das
plt.title('Rolling losses for: RNN, Adam, 1, 100')
plt.xlabel('Iterations')

plt.savefig('Images/loss_graph.png')
plt.show()
```



(Above) Example plot showing the rolling losses while training the model with cell type: RNN, optimizer: Adam, 1 hidden layer and layer size of 100.

In [6]:
```
plt.figure(figsize=(14,13))
sub_count = 1
for cell_type in ['RNN','GRU','LSTM']:
    for optim in ['Adam','ASGD']:
        for numlayers in ['1','2']:
            for size in ['100','200']:
                file_name = './Results/Complete Sherlock Holmes/'
                file_name += '{}/{}/{}/{}/all_losses.txt'.format(cell_type, optir
                file = open(file_name).read()
                losses = [float(i) for i in file.split(' ')]
                rolling_losses=[]
                for i in range(len(losses)):
                    temp=losses[np.max((i-100, 0)):i+1]
                    rolling_losses.append(np.sum(temp)/len(temp))

                avg = np.sum(losses)/len(losses)
                avg_half = np.sum(losses[10000:])/len(losses[10000:])
                avg_quart = np.sum(losses[5000:])/len(losses[5000:])

                plt.subplot(6,4,sub_count)
                plt.plot(rolling_losses)
                plt.hlines([avg,avg_half,avg_quart],0,len(losses)-1,['red','orang
                plt.ylim(1.0,3.5)
                if sub_count <= 4:
                    plt.title('Layers: {}, Size: {}'.format(numlayers, size))
                if sub_count%4 == 1:
                    plt.ylabel('{}, {}'.format(cell_type, optim))

                sub_count+=1
plt.savefig('Images/all_loss_graphs.png')
```



(Above) Rolling losses for all models

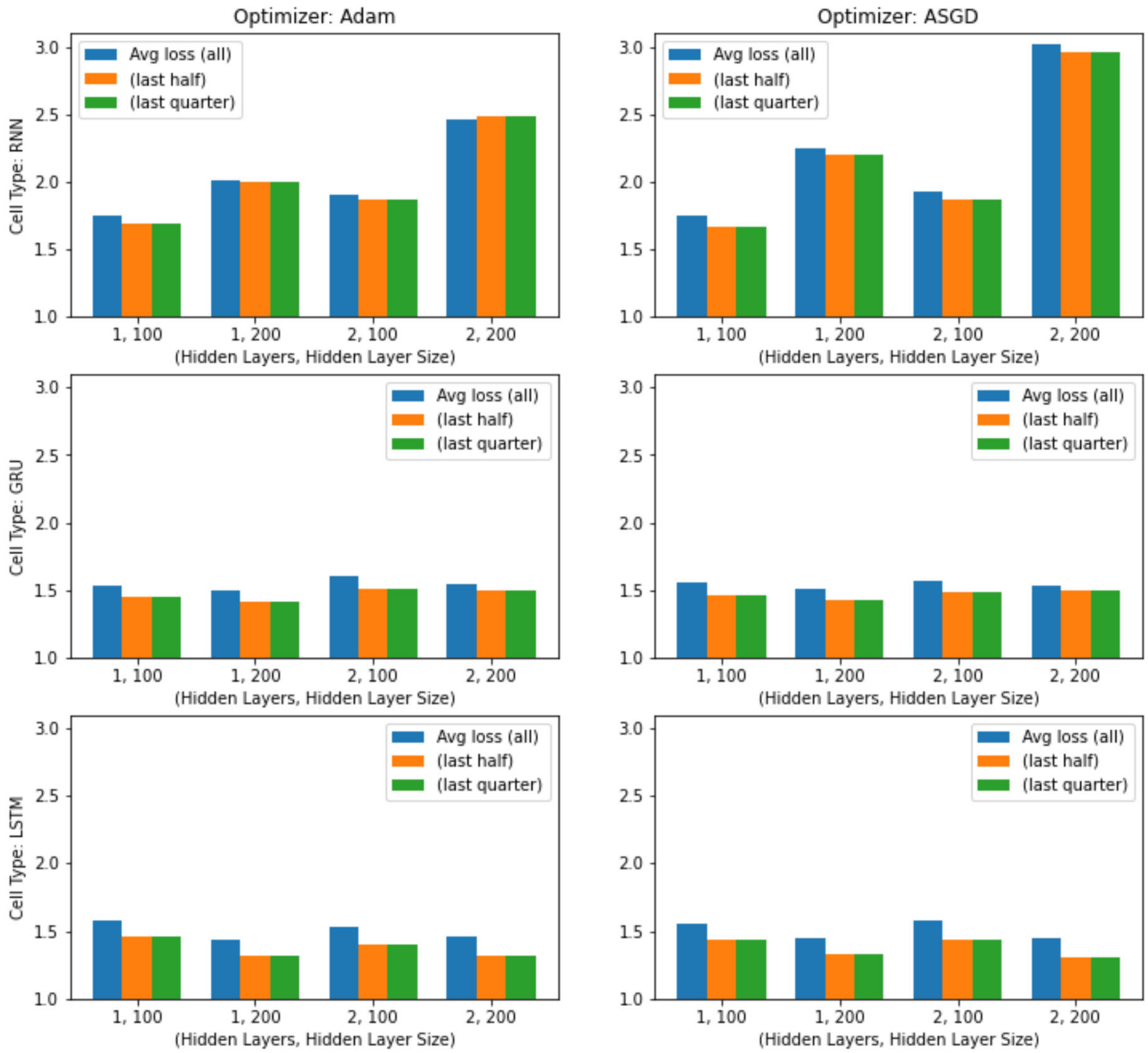
Table of performance results for each model

In [7]:
```
df = pd.read_csv('results.csv', sep=',')
df
```

Out[7]:

	Cell Type	Optimizer	# Hidden Layers	Hidden Layer Size	Avg Loss	last half	last quarter	training time (min)	Sample char count w/o spaces	Sample char count w/ spaces
0	RNN	Adam	1	100	1.753240	1.690466	1.673080	29.160343	3744	4892
1	RNN	Adam	1	200	2.011810	2.003207	1.994535	29.708095	3864	4892
2	RNN	Adam	2	100	1.908531	1.872844	1.929789	43.287523	3769	4881
3	RNN	Adam	2	200	2.457533	2.482899	2.530376	48.069504	4062	4912
4	RNN	ASGD	1	100	1.745751	1.673216	1.662270	43.538487	3785	4916
5	RNN	ASGD	1	200	2.252878	2.204309	2.192678	29.156702	3690	4867
6	RNN	ASGD	2	100	1.932660	1.863559	1.854473	67.390463	3546	4848
7	RNN	ASGD	2	200	3.019652	2.958157	2.986661	45.852272	3898	4960
8	GRU	Adam	1	100	1.530559	1.448299	1.436921	27.283551	3713	4910
9	GRU	Adam	1	200	1.494480	1.410895	1.401998	41.744309	3726	4918
10	GRU	Adam	2	100	1.609544	1.515976	1.507607	44.035200	3649	4868
11	GRU	Adam	2	200	1.546415	1.498690	1.502236	63.860197	3710	4855
12	GRU	ASGD	1	100	1.552805	1.468210	1.456479	41.326073	3715	4890
13	GRU	ASGD	1	200	1.505433	1.428692	1.419809	29.475967	3080	4642
14	GRU	ASGD	2	100	1.568220	1.488219	1.484442	66.975609	3653	4884
15	GRU	ASGD	2	200	1.537782	1.493338	1.489142	43.401833	3631	4893
16	LSTM	Adam	1	100	1.581827	1.464306	1.449594	26.358542	3807	4918
17	LSTM	Adam	1	200	1.435656	1.317489	1.297824	28.450239	3708	4912
18	LSTM	Adam	2	100	1.533304	1.403713	1.384826	59.830156	3715	4907
19	LSTM	Adam	2	200	1.458838	1.320205	1.305445	43.398384	3698	4895
20	LSTM	ASGD	1	100	1.554531	1.434153	1.417843	28.503249	3555	4854
21	LSTM	ASGD	1	200	1.443373	1.324596	1.304677	27.946114	3767	4910
22	LSTM	ASGD	2	100	1.575714	1.433288	1.413820	58.016687	3772	4912
23	LSTM	ASGD	2	200	1.447734	1.305508	1.286445	60.512118	3764	4913

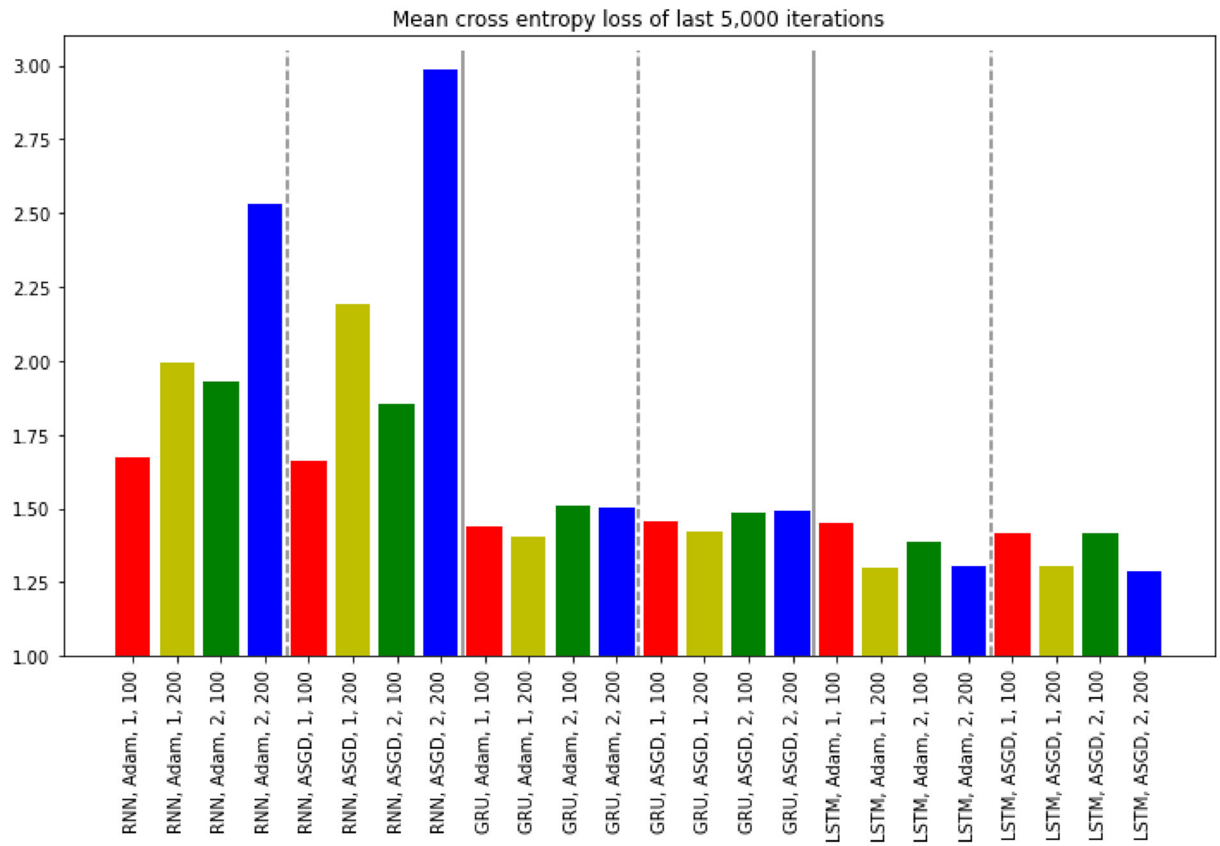
In [8]:
```
plt.figure(figsize=(12,15))
sub_count = 1
for cell_type in ['RNN','GRU','LSTM']:
    for optim in ['Adam','ASGD']:
        df_sub = df
        df_sub = df_sub[df_sub['Cell Type']==cell_type]
        df_sub = df_sub[df_sub['Optimizer']==optim]
        df_sub = df_sub[['Avg Loss','last half','last quarter']]
        losses = df_sub.to_numpy()
        plt.subplot(4,2,sub_count)
        avgs = losses[:,0]
        halfs = losses[:,1]
        quarts = losses[:,2]
        ind = np.arange(4)
        width = 0.25
        plt.bar(ind, avgs, width, label='Avg loss (all)')
        plt.bar(ind + width, halfs, width, label='(last half)')
        plt.bar(ind + 2*width, halfs, width, label='(last quarter)')
        plt.ylim(1.0,3.1)
        if sub_count%2 == 1:
            plt.ylabel('Cell Type: {}'.format(cell_type))
        plt.xlabel('(Hidden Layers, Hidden Layer Size)')
        if sub_count <=2 :
            plt.title('Optimizer: {}'.format(optim))
        plt.xticks(ind + width, ('1, 100', '1, 200', '2, 100', '2, 200'))
        plt.legend(loc='best')
        sub_count+=1
plt.savefig('Images/loss_comparison.png')
```



(Above) Bar chart comparison of average loss per all losses, last half losses, and last quarter losses.

In [9]:
```
labels = []
for cell_type in ['RNN','GRU','LSTM']:
    for optim in ['Adam','ASGD']:
        for numlayers in ['1','2']:
            for size in ['100','200']:
                labels.append('{}, {}, {}, {}'.format(cell_type, optim, numlayers
```

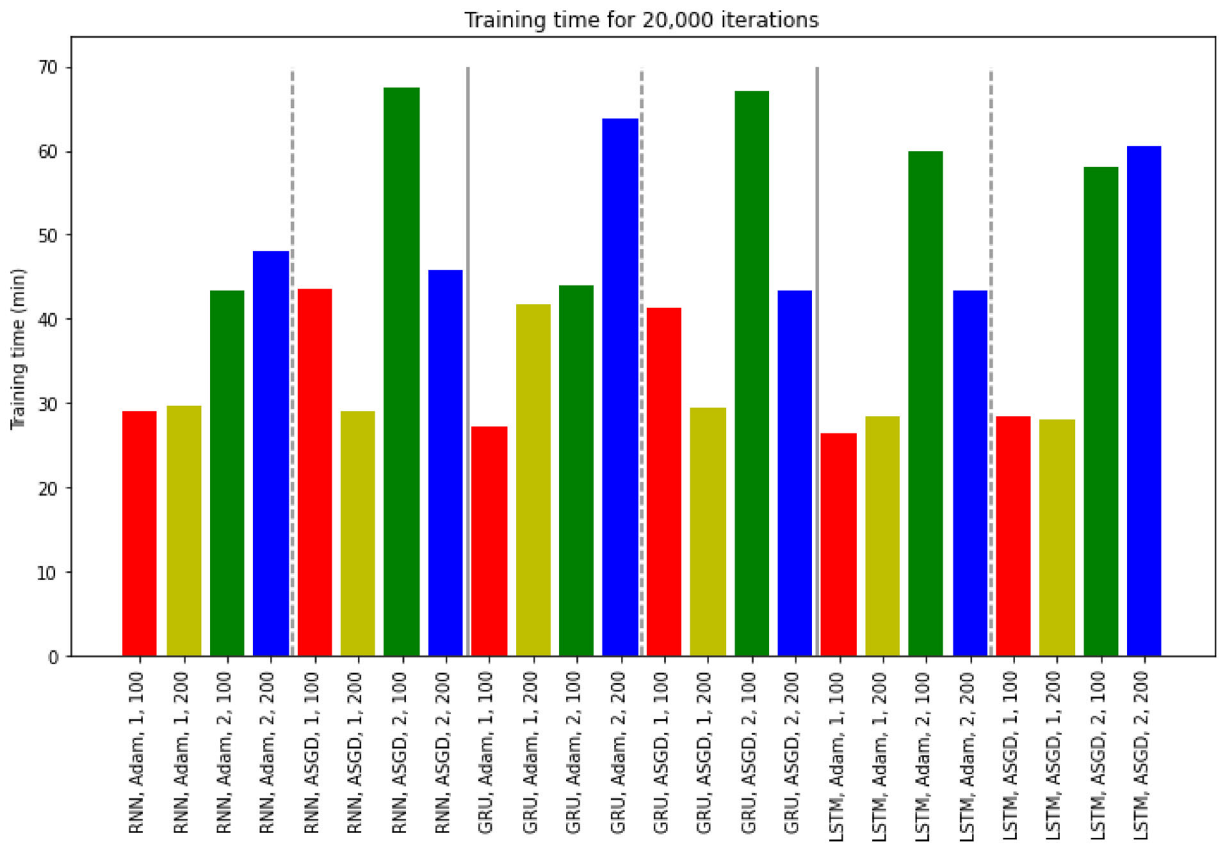
In [10]:
```
losses = df['last quarter'].to_numpy()
plt.figure(figsize=(10,7))
plt.bar(np.arange(len(losses)),losses,color=['r','y','g','b'],tick_label=labels)
plt.xticks(rotation='vertical')
plt.ylim(1.0,3.1)
plt.vlines([7.5,15.5], 1.0, 3.05, 'gray')
plt.vlines([3.5,11.5,19.5], 1.0, 3.05, 'gray','dashed')
plt.title('Mean cross entropy loss of last 5,000 iterations')
plt.tight_layout()
plt.savefig('Images/loss_last_quarter.png')
plt.show()

print('Lowest cross entropy loss: LSTM, ASGD, 2, 200. ({})'.format(df['last quart
```




```
Lowest cross entropy loss: LSTM, ASGD, 2, 200. (1.286444902)
```

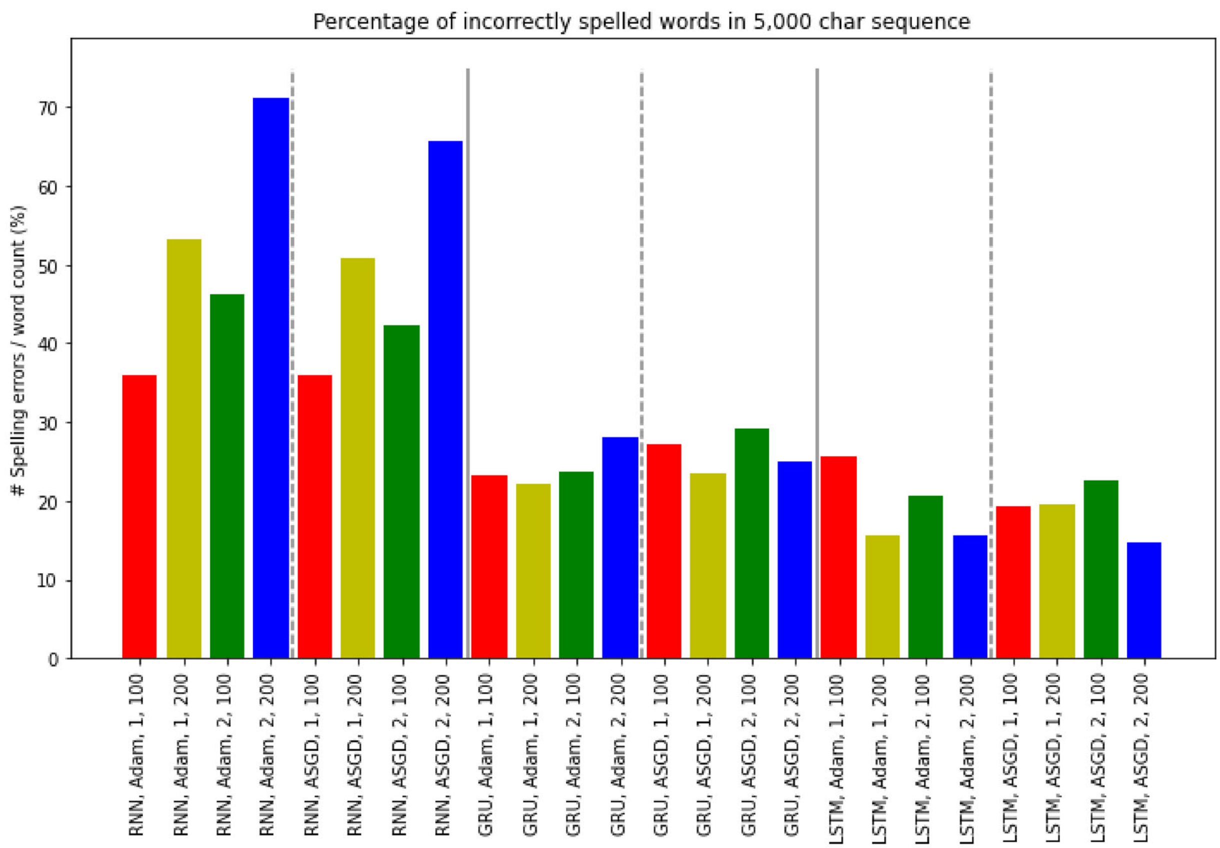
(Above) Comparison of last quarter cross entropy loss across all models.

In [11]:
```
times = df['training time (min)'].to_numpy()
plt.figure(figsize=(10,7))
plt.bar(np.arange(len(times)),times,color=['r','y','g','b'],tick_label=labels)
plt.xticks(rotation='vertical')
plt.vlines([7.5,15.5], 0, 70, 'gray')
plt.vlines([3.5,11.5,19.5], 0, 70, 'gray','dashed')
plt.ylabel('Training time (min)')
plt.title('Training time for 20,000 iterations')
plt.tight_layout()
plt.savefig('Images/training_time.png')
plt.show()

print('Shortest training time: LSTM, Adam, 1, 100. ({} min)'.format(df['training
```




```
Shortest training time: LSTM, Adam, 1, 100. (26.35854196 min)
```

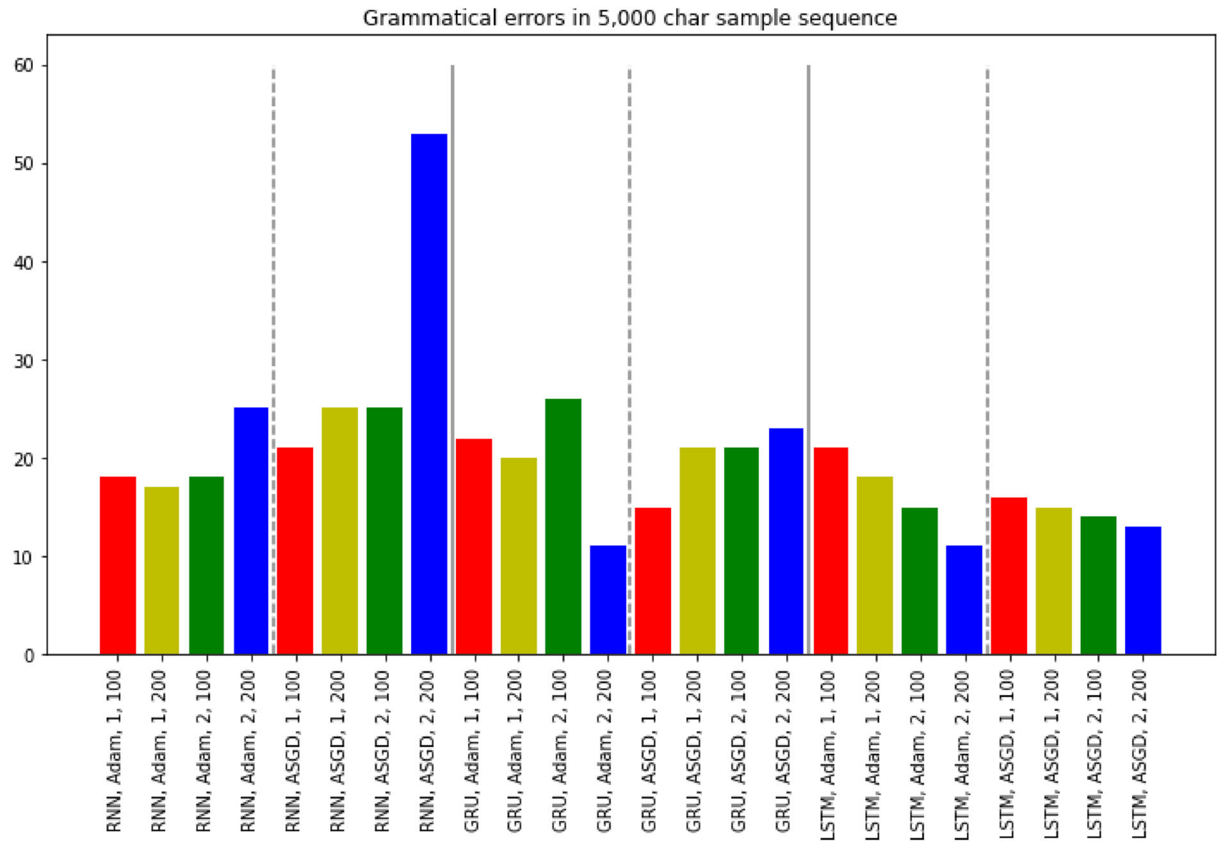
In [12]:
```
errors = df['% spelling errors'].to_numpy()*100
plt.figure(figsize=(10,7))
plt.bar(np.arange(len(times)),errors,color=['r','y','g','b'],tick_label=labels)
plt.xticks(rotation='vertical')
plt.vlines([7.5,15.5], 0, 75, 'gray')
plt.vlines([3.5,11.5,19.5], 0, 75, 'gray','dashed')
plt.ylabel('# Spelling errors / word count (%)')
plt.title('Percentage of incorrectly spelled words in 5,000 char sequence')
plt.tight_layout()
plt.savefig('Images/spelling_percentage.png')
plt.show()

print('Smallest percentage of incorrectly spelled words: LSTM, ASGD, 2, 200. ({}%
```




```
Smallest percentage of incorrectly spelled words: LSTM, ASGD, 2, 200. (14.71610
66%)
```

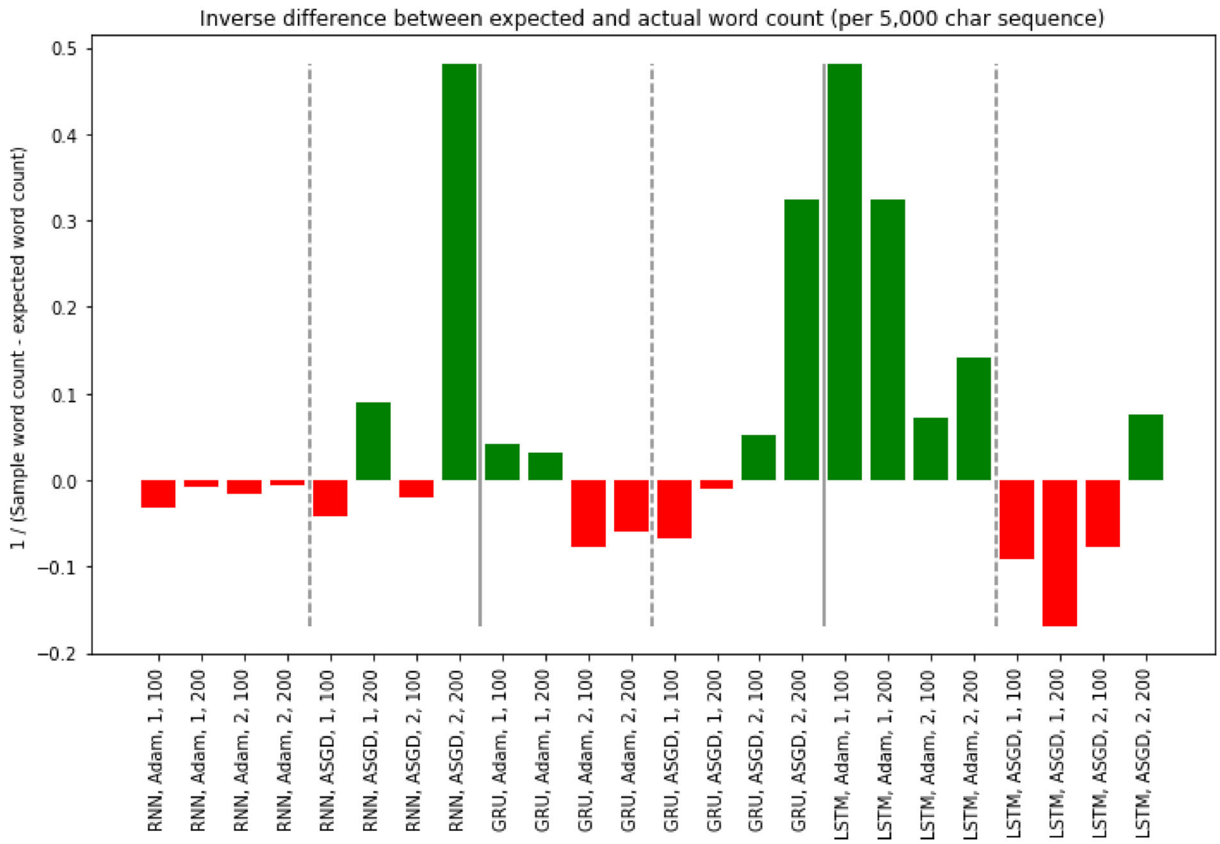
In [13]:

```
gerrors = df['grammer errors'].to_numpy()
plt.figure(figsize=(10,7))
plt.bar(np.arange(len(times)),gerrors,color=['r','y','g','b'],tick_label=labels)
plt.xticks(rotation='vertical')
plt.vlines([7.5,15.5], 0, 60, 'gray')
plt.vlines([3.5,11.5,19.5], 0, 60, 'gray','dashed')
plt.title('Grammatical errors in 5,000 char sample sequence')
plt.tight_layout()
plt.savefig('Images/grammar_errors.png')
plt.show()

print('Fewest grammatical errors: GRU, Adam, 2, 200. ({})'.format(df['grammer err
```

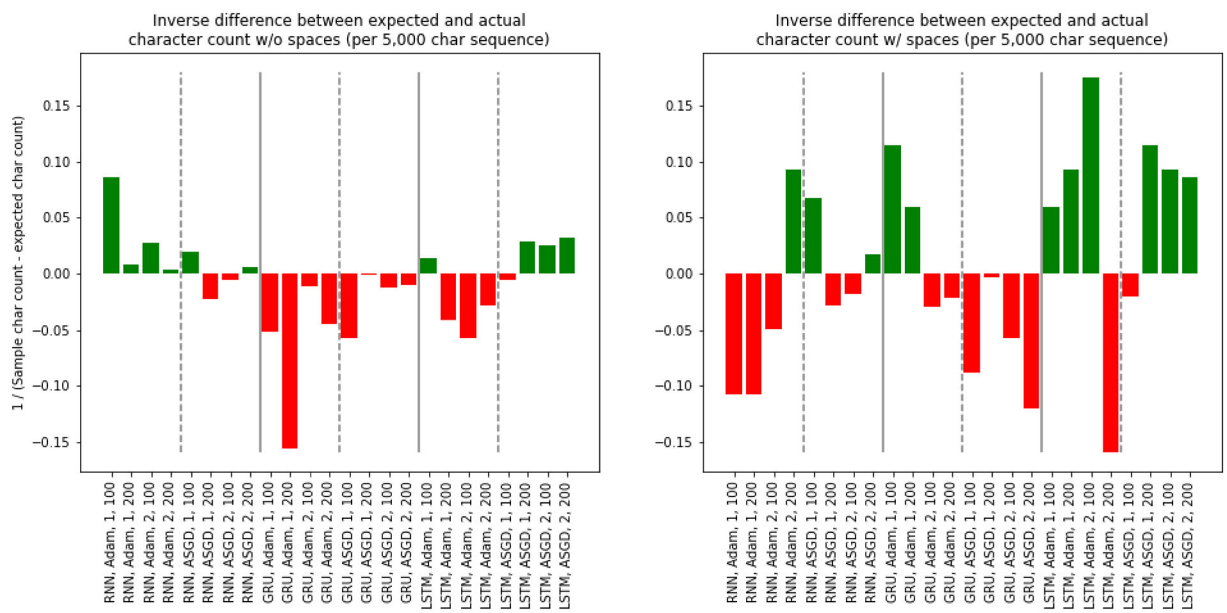



```
Fewest grammatical errors: GRU, Adam, 2, 200. (11)
```

In [14]:
```
data = (df['Sample word count'].to_numpy() - n1)**-1
plt.figure(figsize=(10,7))
colors=['g' if i>=0 else 'r' for i in data]
plt.bar(np.arange(len(times)),data,color=colors,tick_label=labels)
plt.xticks(rotation='vertical')
plt.vlines([7.5,15.5], data.min(), data.max(), 'gray')
plt.vlines([3.5,11.5,19.5], data.min(), data.max(), 'gray','dashed')
plt.ylabel('1 / (Sample word count - expected word count)')
plt.title('Inverse difference between expected and actual word count (per 5,000 c
plt.tight_layout()
plt.savefig('Images/word_count_comparison.png')
plt.show()
```



In [15]:
```
plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
data = (df['Sample char count w/o spaces'].to_numpy() - n3)**-1

colors=['g' if i>=0 else 'r' for i in data]
plt.bar(np.arange(len(times)),data,color=colors,tick_label=labels)
plt.xticks(rotation='vertical')
plt.vlines([7.5,15.5], -0.16, 0.18, 'gray')
plt.vlines([3.5,11.5,19.5], -0.16, 0.18, 'gray','dashed')
plt.ylabel('1 / (Sample char count - expected char count)')
plt.title('Inverse difference between expected and actual\ncharacter count w/o sp

plt.subplot(1,2,2)
data = (df['Sample char count w/ spaces'].to_numpy() - n2)**-1

colors=['g' if i>=0 else 'r' for i in data]
plt.bar(np.arange(len(times)),data,color=colors,tick_label=labels)
plt.xticks(rotation='vertical')
plt.vlines([7.5,15.5], -0.16, 0.18, 'gray')
plt.vlines([3.5,11.5,19.5], -0.16, 0.18, 'gray','dashed')
plt.title('Inverse difference between expected and actual\ncharacter count w/ spa

plt.savefig('Images/char_count_comparison.png')
plt.show()
```



In [ ]: