Hw06

November 10, 2021

1 Exploring RNNs

In this assignment, you will be asked to modify the notebook which we went over in class exploring the use of RNNs.

We begin by donwloading and unziping the dataset.

[2]:
```
!wget https://download.pytorch.org/tutorial/data.zip
```

```
--2021-11-10 01:20:00--  https://download.pytorch.org/tutorial/data.zip
Resolving download.pytorch.org (download.pytorch.org)... 13.226.14.112,
13.226.14.36, 13.226.14.84, ...
Connecting to download.pytorch.org (download.pytorch.org)|13.226.14.112|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 2882130 (2.7M) [application/zip]
Saving to: data.zip

data.zip            100%[===================>]   2.75M  15.0MB/s    in 0.2s

2021-11-10 01:20:00 (15.0 MB/s) - data.zip saved [2882130/2882130]
```

[3]:
```
!unzip data.zip
```

```
Archive:  data.zip
   creating: data/
  inflating: data/eng-fra.txt
   creating: data/names/
  inflating: data/names/Arabic.txt
  inflating: data/names/Chinese.txt
  inflating: data/names/Czech.txt
  inflating: data/names/Dutch.txt
  inflating: data/names/English.txt
  inflating: data/names/French.txt
  inflating: data/names/German.txt
  inflating: data/names/Greek.txt
  inflating: data/names/Irish.txt
```

```
  inflating: data/names/Italian.txt
  inflating: data/names/Japanese.txt
  inflating: data/names/Korean.txt
  inflating: data/names/Polish.txt
  inflating: data/names/Portuguese.txt
  inflating: data/names/Russian.txt
  inflating: data/names/Scottish.txt
  inflating: data/names/Spanish.txt
  inflating: data/names/Vietnamese.txt
```

Loading and Formatting the Data We provide helper functions for loading the data, and store it as a dictionary with entries for each nationality. Data is split into training and test as well.

[4]:
```
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os
import unicodedata
import string
import torch

# This function returns the path of the files in the dataset
def findFiles(path): return glob.glob(path)

# Specifying list of characters
all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

# This function converts unicode to ascii
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

# Function for reading afile and splitting into lines
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

# Specifying percentage of data used for training
perTrain = 0.9

# Build the category_lines dictionary, a list of names per language
category_lines_train = {}
category_lines_test = {}
all_categories = []
```

```
# Loading all the data
for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = readLines(filename)
    category_lines_train[category] = lines[0:int(perTrain*len(lines))]
    category_lines_test[category] = lines[int(perTrain*len(lines)):]

# Specifying the number of categories
n_categories = len(all_categories)

# Find letter index from all_letters, e.g. "a" = 0
def letterToIndex(letter):
    return all_letters.find(letter)

# Turn a line into a <line_length x 1 x n_letters>,
# or an array of one-hot letter vectors
def lineToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor
```

Helper functions for converting the words to tensors. A simple one-hot encoding is used for the characters.

[5]:
```
import random

# Converting a prediction to category labels
def categoryFromOutput(output):
    top_n, top_i = output.topk(1)
    category_i = top_i[0].item()
    return all_categories[category_i], category_i

# Random selection of an entry in a list
def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

# Randomly selecting a category and then a name from the list
def randomTrainingExample():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines_train[category])
    category_tensor = torch.tensor([all_categories.index(category)],␣
 ↪dtype=torch.long)
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor
```

```
# Helper function for tracking time ellapsed
def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

Plot Graph

[6]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker


def plot_training_loss(all_iterCt, all_losses):
    plt.figure()
    plt.plot(all_iterCt,all_losses)
    plt.xlabel('Iteration')
    plt.ylabel('Training Loss')
    plt.title("Baseline model with the standard gradient descent")
    plt.show()
```

Test Model Performance(Accuracy) Functions

[7]:
```
# Keep track of correct guesses in a confusion matrix
confusion = torch.zeros(n_categories, n_categories)

# Just return an output given a line
def evaluate(line_tensor, model):
    hidden = model.initHidden()

    for i in range(line_tensor.size()[0]):
        output, hidden = model(line_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
def accuracy(model, all_categories):
    for category in all_categories:
        for line in category_lines_test[category]:
            line_tensor = lineToTensor(line)
            output = evaluate(line_tensor, model)
            guess, guess_i = categoryFromOutput(output)
            category_i = all_categories.index(category)
            confusion[category_i][guess_i] += 1
```

```
    # Normalize by dividing every row by its sum
    accuracy = 0
    for i in range(n_categories):
        confusion[i] = confusion[i] / confusion[i].sum()
        accuracy += confusion[i][i]
    accuracy /= n_categories

    # Displaying the average accuracy
    print('Average Macro Accuracy = {:.2f}\n'.format(accuracy))

    return confusion

def test_plot(confusion, all_categories):
    # Set up plot
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(confusion.numpy())
    fig.colorbar(cax)

    # Set up axes
    ax.set_xticklabels([''] + all_categories, rotation=90)
    ax.set_yticklabels([''] + all_categories)

    # Force label at every tick
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()
```

Train model performance(Accuracy) Function

[8]:
```
# Keep track of correct guesses in a confusion matrix
confusion = torch.zeros(n_categories, n_categories)

# Go through a bunch of examples and record which are correctly guessed
def train_accuracy(model, all_categories):
    for category in all_categories:
        for line in category_lines_train[category]:
            line_tensor = lineToTensor(line)
            output = evaluate(line_tensor, model)
            guess, guess_i = categoryFromOutput(output)
            category_i = all_categories.index(category)
            confusion[category_i][guess_i] += 1

    # Normalize by dividing every row by its sum
    accuracy = 0
    for i in range(n_categories):
```

```
        confusion[i] = confusion[i] / confusion[i].sum()
        accuracy += confusion[i][i]
    accuracy /= n_categories

    # Displaying the average accuracy
    print('Average Macro Train Accuracy = {:.2f}\n'.format(accuracy))
```

Train Function

[9]:
```
import torch.nn as nn

# Specifying our criterion to match the last prediction layer for the model
criterion = nn.CrossEntropyLoss()

import time
import math

# Specifying the number of gradient decent
n_iters = 50000

# Specifying some other variables for display of the output
print_every = 5000
plot_every = 1000

# Function performing one step of backpropagation
def train(category_tensor, line_tensor, optimizer_type, model):
    hidden = model.initHidden()
    model.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = model(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    if optimizer_type=="Standard":
      for p in model.parameters():
          p.data.add_(p.grad.data, alpha=-0.005)
    elif optimizer_type=="Adam":
      optimizer.step()

    return output, loss.item()
```

1.1 [Task 1] Using Adam Optimizer [30 pts]

We will be replacing the implementation of standard gradient descent in the baseline model by a call of the Adam optimizer. Do the following:

1. [12 pts] Train the baseline model with the standard gradient descent and a version using Adam optimizer. Plot the learning curves for both approaches. Train both models for only 50,000 iterations.
2. [12 pts] Evaluate the performance of both models on the test set.
3. [6 pts] Comment on the performance of both methods on the test set, and the shape of their learning curves.

1.1.1 Task 1.1

[12 pts] Train the baseline model with the standard gradient descent and a version using Adam optimizer. Plot the learning curves for both approaches. Train both models for only 50,000 iterations.

Model Architecture

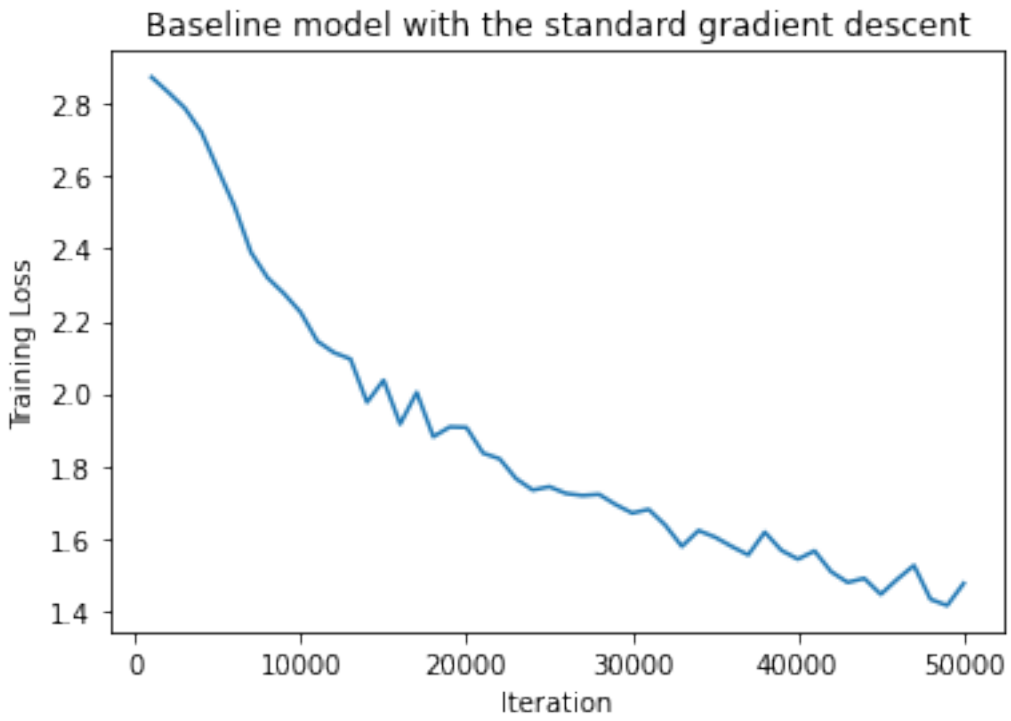
[15]:
```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.g = nn.Linear(input_size + hidden_size, hidden_size)
        self.f = nn.Linear(input_size + hidden_size, output_size)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.g(combined)
        output = self.f(combined)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

Baseline model with the standard gradient descent

[18]:
```
# Keep track of losses for plotting
current_loss = 0
all_losses_baseline_1 = []
all_iterCt_baseline_1 = []

# Getting the start time
start = time.time()

n_hidden = 128
```

```
rnn_baseline_1 = RNN(n_letters, n_hidden, n_categories)

for iter in range(1, n_iters + 1):
    # Training using a random name and accumulating the loss
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor, "Standard",␣
 ↪rnn_baseline_1)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '' if guess == category else ' (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100,
            timeSince(start), loss, line, guess, correct))

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses_baseline_1.append(current_loss / plot_every)
        all_iterCt_baseline_1.append(iter)
        current_loss = 0


plot_training_loss(all_iterCt_baseline_1,all_losses_baseline_1)

train_accuracy(rnn_baseline_1, all_categories)
```

```
5000 10% (0m 6s) 3.1350 Serafim / Italian  (Portuguese)
10000 20% (0m 12s) 0.2861 Karkampasis / Greek
15000 30% (0m 18s) 2.5371 Pherigo / Portuguese  (Italian)
20000 40% (0m 24s) 2.8370 Kwang  / Arabic  (Korean)
25000 50% (0m 31s) 2.3612 Aitken / Dutch  (Scottish)
30000 60% (0m 37s) 0.8114 Coelho / Portuguese
35000 70% (0m 43s) 2.2121 Amato / Japanese  (Italian)
40000 80% (0m 49s) 0.3863 Rossini / Italian
45000 90% (0m 56s) 0.9226 D'cruz / Spanish  (Portuguese)
50000 100% (1m 2s) 2.2313 Kopp / Czech  (German)
```



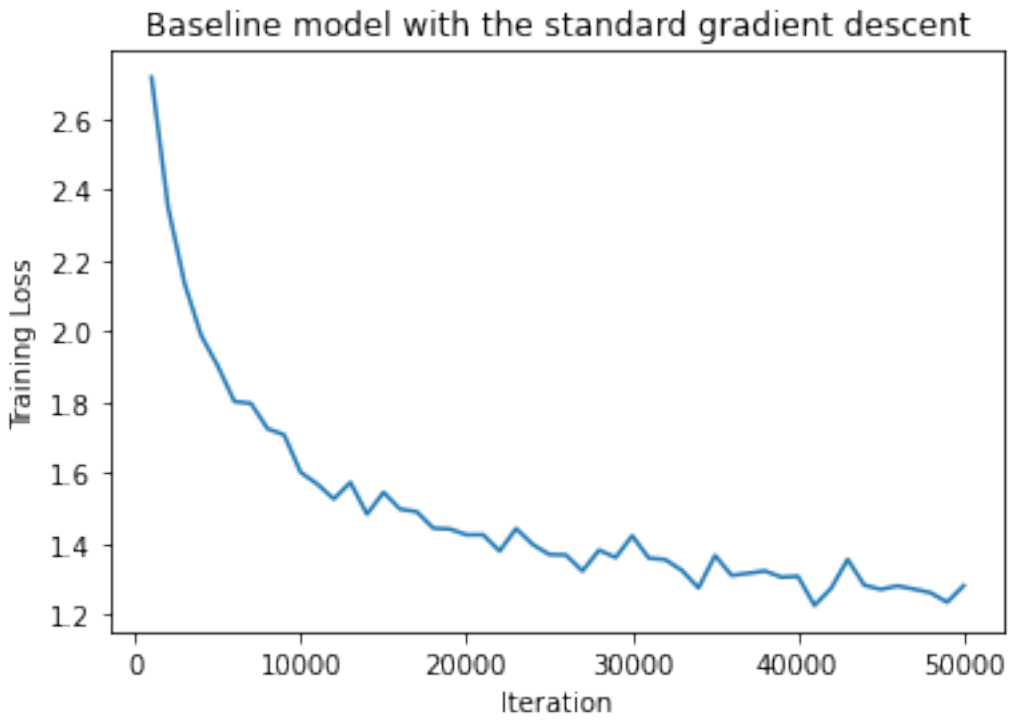

```
Average Macro Train Accuracy = 0.54
```

Adam optimizer

[16]:
```
# Keep track of losses for plotting
current_loss = 0
all_losses_adam_1 = []
all_iterCt_adam_1 = []

# Getting the start time
start = time.time()

n_hidden = 128
rnn_adam_1 = RNN(n_letters, n_hidden, n_categories)

optimizer = torch.optim.Adam(rnn_adam_1.parameters(), lr=1e-3)

for iter in range(1, n_iters + 1):
    # Training using a random name and accumulating the loss
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor, "Adam", rnn_adam_1)
    current_loss += loss
```

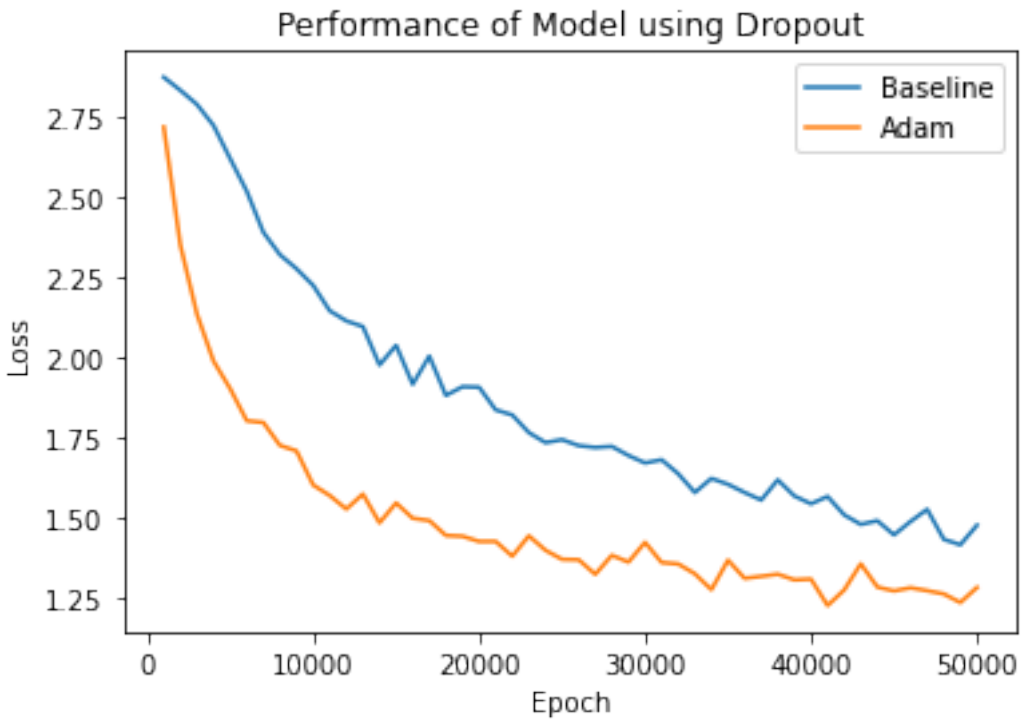
```
    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '' if guess == category else ' (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100,
            timeSince(start), loss, line, guess, correct))

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses_adam_1.append(current_loss / plot_every)
        all_iterCt_adam_1.append(iter)
        current_loss = 0


plot_training_loss(all_iterCt_adam_1,all_losses_adam_1)

train_accuracy(rnn_adam_1, all_categories)
```

```
5000 10% (0m 8s) 1.8230 Jedynak / Czech  (Polish)
10000 20% (0m 16s) 0.0366 Thao / Vietnamese
15000 30% (0m 24s) 0.2021 Sook / Korean
20000 40% (0m 32s) 0.3502 Horiatis / Greek
25000 50% (0m 40s) 0.4421 Narvaez / Spanish
30000 60% (0m 49s) 1.4873 Cernochova / Spanish  (Czech)
35000 70% (0m 57s) 0.5241 Awad / Arabic
40000 80% (1m 5s) 1.2559 Lachance / French
45000 90% (1m 13s) 0.1666 Min / Chinese
50000 100% (1m 22s) 0.6116 Piatek / Polish
```




```
Average Macro Train Accuracy = 0.59
```

[19]:
```
plt.plot(all_iterCt_baseline_1, all_losses_baseline_1, all_iterCt_adam_1,␣
 ↪all_losses_adam_1)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(['Baseline','Adam'])
plt.title("Performance of Model using Dropout")
plt.show()
```



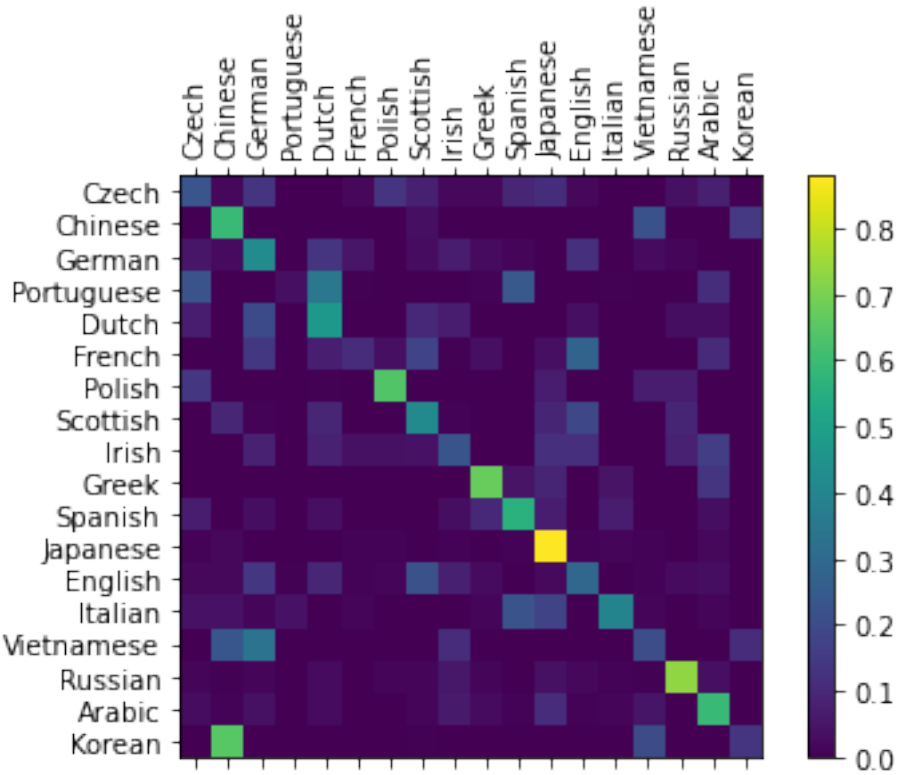
1.1.2 Task 1.2

Evaluate the performance of both models on the test set.

Performance of baseline model on test set

[20]:
```
baseline_confusion=accuracy(rnn_baseline_1, all_categories)
test_plot(baseline_confusion, all_categories)
```

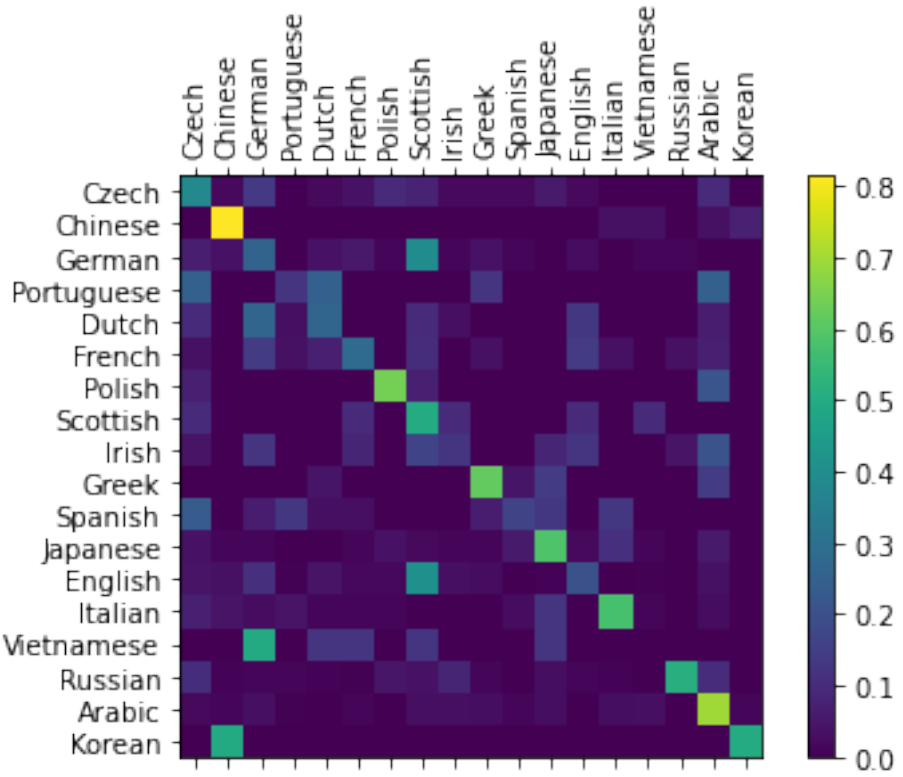
```
Average Macro Accuracy = 0.42
```



Performance of model using Adam optimizer on test set

[35]:
```
adam_confusion=accuracy(rnn_adam_1, all_categories)
test_plot(adam_confusion, all_categories)
```

```
Average Macro Accuracy = 0.40
```



1.1.3 Task 1.3

[6 pts] Comment on the performance of both methods on the test set, and the shape of their learning curves. Train the baseline model with the standard gradient descent :

- Train Accuracy= 54%
- Test Accuracy= 42%
- Train Accuracy-Test Accuracy=12%

Train the model using Adam optimizer:

- Train Accuracy= 59%
- Test Accuracy= 40%
- Train Accuracy-Test Accuracy=19%

Observations:

- Model uing Standard gradient descent is better than the model using Adam optimizer as the Test Accuray is more with Standard gradient descent model.

- Model using Adam optimizer have steeper learning curve.
- Both the model are overfitting but comparitively it seems Model using Adam optimizer is overfitting more as difference between training accuracy and test accuracy is larger than the baseline model.
- When we plot the test accuracy at each epoch along with the train accuracy we will get more insights about where exactly the overfitting has started.

1.2 [Task 2] Implementing a More Complex RNN [30 pts]

Replace the linear layer g in the baseline model by a two-layer fully connected neural network with ReLU activation. The new subnetwork should implement:

$$h^{(t)} = g(c^{(t)}) = \sigma\left(W_2 \cdot \sigma\left(W_1 \cdot c^{(t)} + b_1\right) + b_2\right),$$

where σ is a ReLU activation, and (W_k, b_k) are the parameters for a linear layer. Then, answer the following:

1. [12 pts] Compare the learning curves for the baseline trained with Adam and this more complex model trained with Adam as well. Train both models for only 50,000 iterations.
2. [12 pts] Evaluate the performance of both models on the test set.
3. [6 pts] Comment on the performance of both methods on the test set, and the shape of their learning curves.

1.2.1 Task 2.1

[12 pts] Compare the learning curves for the baseline trained with Adam and this more complex model trained with Adam as well. Train both models for only 50,000 iterations.

Model Architecture Subnetwork:

$$h^{(t)} = g(c^{(t)}) = \sigma\left(W_2 \cdot \sigma\left(W_1 \cdot c^{(t)} + b_1\right) + b_2\right),$$

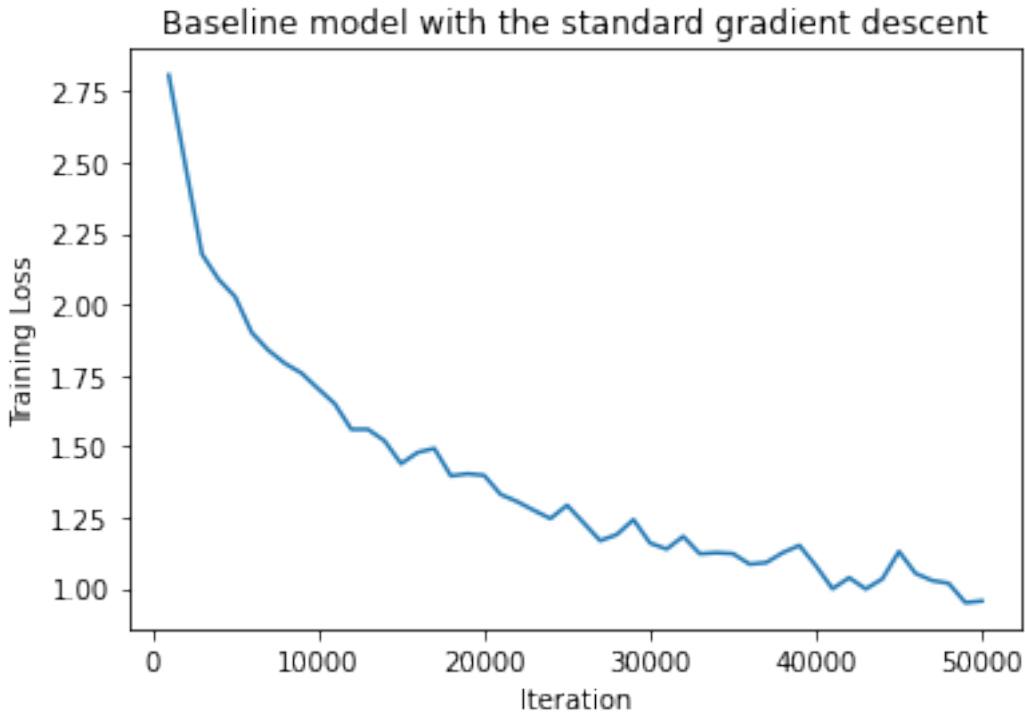
[22]:
```
import torch.nn as nn
import torch.nn.functional as F
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, hidden_size_2 ,output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.g = nn.Linear(input_size + hidden_size, hidden_size_2)
        self.g_new = nn.Linear(hidden_size_2, hidden_size)
        self.f = nn.Linear(input_size + hidden_size, output_size)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = (F.relu(self.g(combined)))
        hidden = (F.relu(self.g_new(hidden)))
        output = self.f(combined)
```

```
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

[23]:
```
# Keep track of losses for plotting
current_loss = 0
all_losses_complex_2 = []
all_iterCt_complex_2 = []

# Getting the start time
start = time.time()

n_hidden = 128
n_hidden2= 156
rnn_complex_2 = RNN(n_letters, n_hidden, n_hidden2, n_categories)

optimizer = torch.optim.Adam(rnn_complex_2.parameters(), lr=1e-3)

for iter in range(1, n_iters + 1):
    # Training using a random name and accumulating the loss
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor, "Adam", rnn_complex_2)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '' if guess == category else ' (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100,
            timeSince(start), loss, line, guess, correct))

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses_complex_2.append(current_loss / plot_every)
        all_iterCt_complex_2.append(iter)
        current_loss = 0



plot_training_loss(all_iterCt_complex_2,all_losses_complex_2)

train_accuracy(rnn_complex_2, all_categories)
```

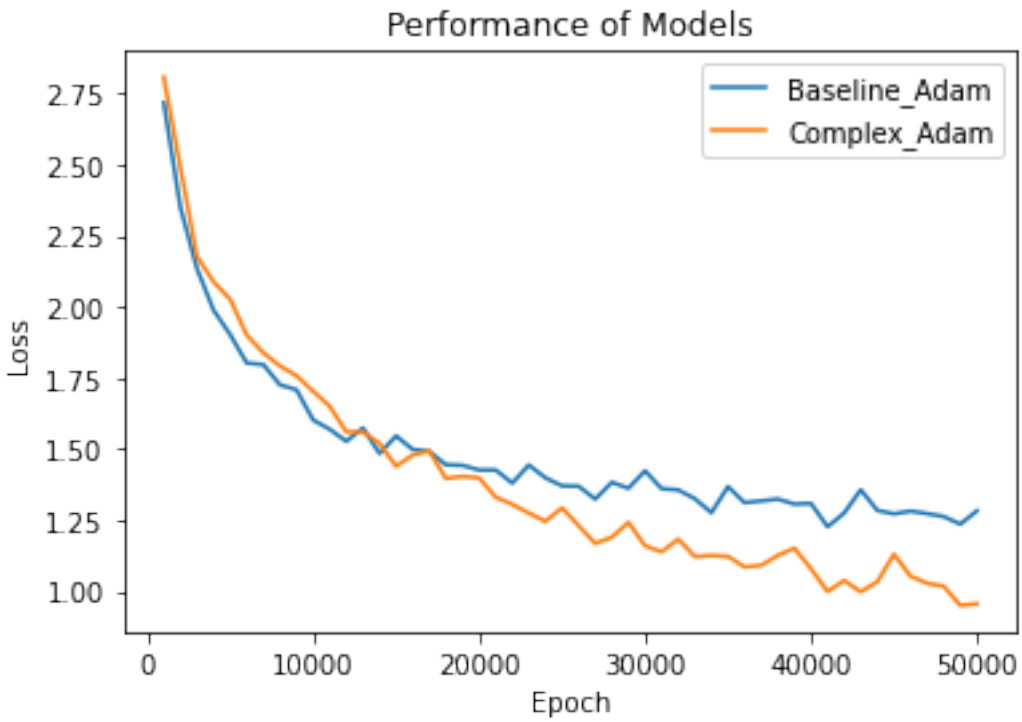
```
5000 10% (0m 16s) 1.4805 Jin / Korean  (Chinese)
10000 20% (0m 33s) 0.9102 Thi / Korean  (Vietnamese)
15000 30% (0m 50s) 1.1277 Hwang / Chinese  (Korean)
20000 40% (1m 6s) 3.4136 Gagnon / Russian  (French)
```

```
25000 50% (1m 23s) 1.4750 Aweryanoff / Arabic  (Russian)
30000 60% (1m 40s) 0.7789 Netsch / Czech
35000 70% (1m 57s) 3.3165 Johnstone / French  (Scottish)
40000 80% (2m 14s) 0.6108 Salamanca / Spanish
45000 90% (2m 31s) 0.2197 Serafini / Italian
50000 100% (2m 48s) 1.8024 Rodrigues / French  (Portuguese)
```




```
Average Macro Train Accuracy = 0.68
```

[24]:
```
plt.plot(all_iterCt_adam_1, all_losses_adam_1,␣
 ↪all_iterCt_complex_2,all_losses_complex_2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(['Baseline_Adam','Complex_Adam'])
plt.title("Performance of Models")
plt.show()
```



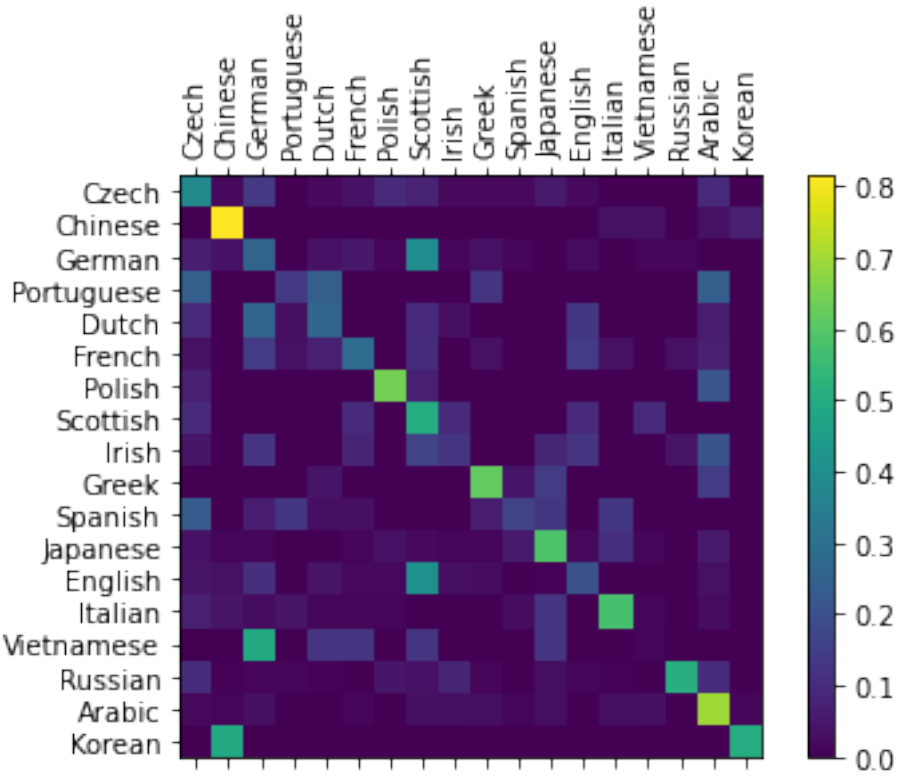
1.2.2 Task 2.2

[12 pts] Evaluate the performance of both models on the test set.

Performance of baseline model trained using Adam optimizer on the test set

[34]:
```
adam_confusion=accuracy(rnn_adam_1, all_categories)
test_plot(adam_confusion, all_categories)
```

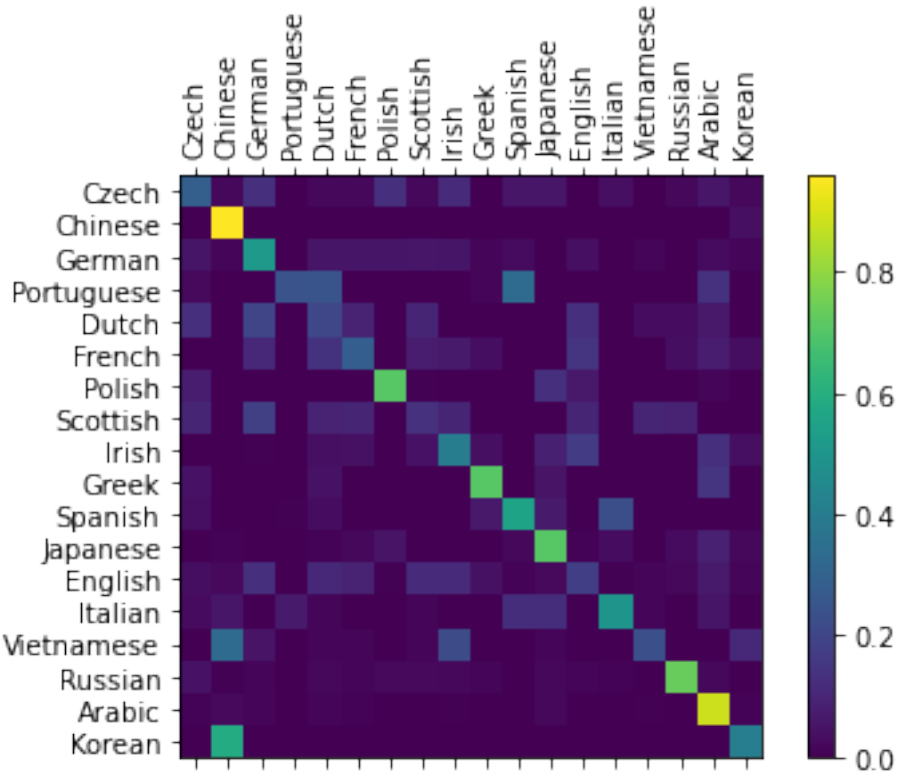
```
Average Macro Accuracy = 0.41
```



Performance of Complex model trained using Adam optimizer on the test set

[26]:
```
complex_confusion=accuracy(rnn_complex_2, all_categories)
test_plot(complex_confusion, all_categories)
```

```
Average Macro Accuracy = 0.48
```



1.2.3 Task 2.3

3. [6 pts] Comment on the performance of both methods on the test set, and the shape of their learning curves.

Train the baseline model with Adam Optimizer :

- Train Accuracy= 54%
- Test Accuracy= 42%
- Train Accuracy-Test Accuracy=12%

Train the complex model using Adam optimizer:

- Train Accuracy= 68%
- Test Accuracy= 48%
- Train Accuracy-Test Accuracy=20%

Observations:

- More complex model trained with Adam optimizer performs better than the baseline model trained using Adam optimizer as the Test Accuray is more with the complex model.

- Performance graph showing training loss of both the models starts at same point and with iterations complex model loss decreses more.
- Complex model trained with Adam optimizer have steeper learning curve.
- Both the model are overfitting but comparitively it seems Complex Model using Adam optimizer is overfitting more as difference between training accuracy and test accuracy is larger than the baseline model using adam optimizer.
- When we plot the test accuracy at each epoch along with the train accuracy we will get more insights about where exactly the overfitting has started.

1.3 [Task 3] Using LSTM [40 pts]

Replace the custom-built RNN for a standard LSTM layer. This may require you to do some significant changes to the network class and training functions.

1. [16 pts] Compare the learning curves for the baseline trained with Adam and the LSTM model trained with Adam as well. Train both models for only 50,000 iterations.
2. [16 pts] Evaluate the performance of both models on the test set.
3. [8 pts] Comment on the performance of both methods on the test set, and the shape of their learning curves.

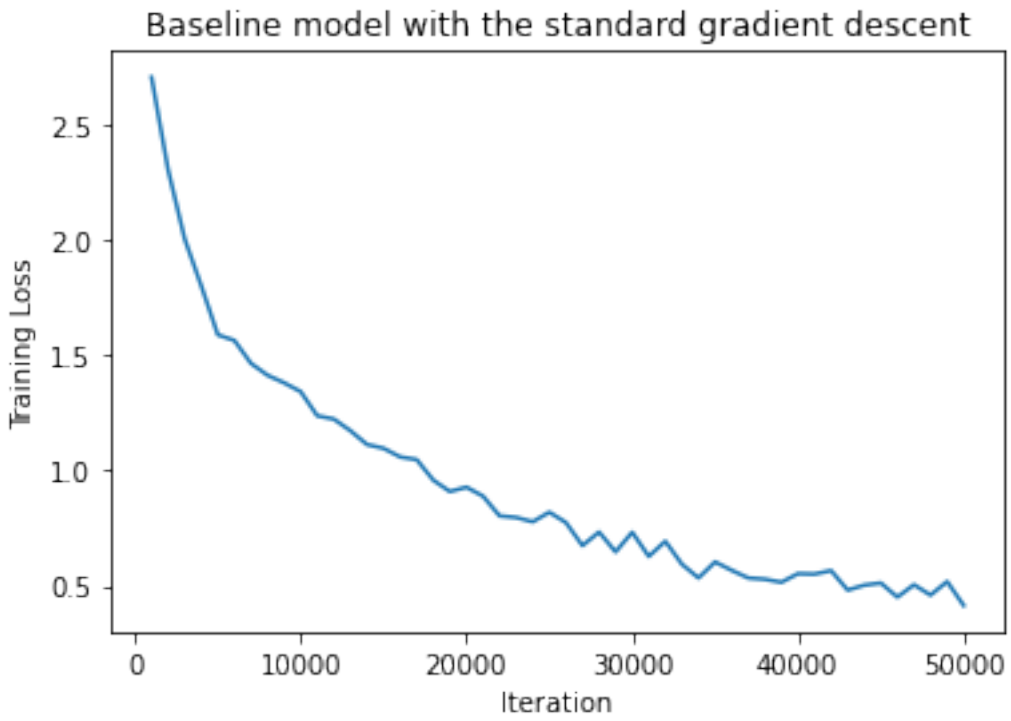
1.3.1 Task 3.1

[ ]:
```
### TO DO - Enter Your Code here... You are welcome to add more cell if needed
```

[10]:
```
import torch.nn as nn

class LSTM_model(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM_model, self).__init__()
        self.hidden_size = hidden_size
        self.lstm_cell = nn.LSTM(input_size, hidden_size)
        self.g = nn.Linear(hidden_size, output_size)


    def forward(self, input, hidden):
        #print(input.view(1, 1, -1))
        out, hidden = self.lstm_cell(input.view(1, 1, -1), hidden)
        #print(out)
        output = self.g(hidden[0])
        return output.view(1, -1), hidden


    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size), torch.zeros(1, 1, self.
↪hidden_size)
```

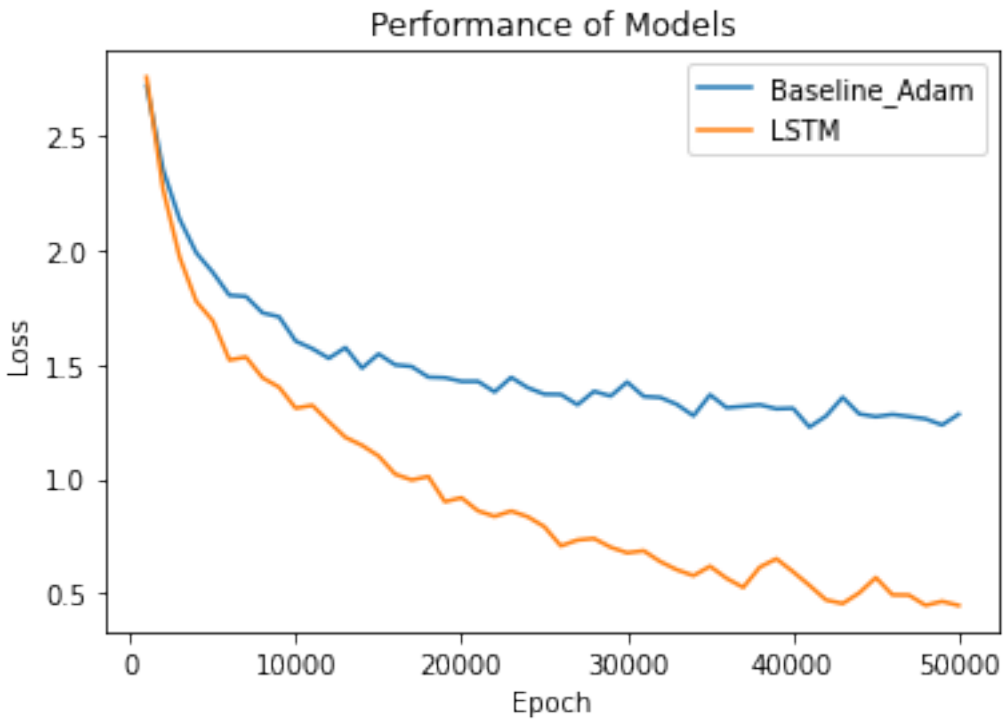
[32]:
```
# Keep track of losses for plotting
current_loss = 0
all_losses_lstm_3 = []
all_iterCt_lstm_3 = []

# Getting the start time
start = time.time()

n_hidden = 128
lstm = LSTM_model(n_letters, n_hidden, n_categories)

optimizer = torch.optim.Adam(lstm.parameters(), lr=1e-3)

for iter in range(1, n_iters + 1):
    # Training using a random name and accumulating the loss
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor, "Adam", lstm)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '' if guess == category else ' (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100,
            timeSince(start), loss, line, guess, correct))

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses_lstm_3.append(current_loss / plot_every)
        all_iterCt_lstm_3.append(iter)
        current_loss = 0

plot_training_loss(all_iterCt_lstm_3,all_losses_lstm_3)


train_accuracy(lstm, all_categories)
```

```
5000 10% (0m 22s) 1.3784 Rigley / English
10000 20% (0m 45s) 0.3336 Dioli / Italian
15000 30% (1m 8s) 1.2530 An / Chinese  (Vietnamese)
20000 40% (1m 32s) 2.2933 Aalsburg / Russian  (Dutch)
25000 50% (1m 55s) 0.2975 Rivera / Spanish
30000 60% (2m 18s) 0.2154 Rios / Spanish
35000 70% (2m 41s) 0.4190 Favager / French
40000 80% (3m 3s) 0.4321 Tso / Chinese
45000 90% (3m 26s) 0.0016 Bursinos / Greek
50000 100% (3m 49s) 3.4879 Gro / Chinese  (German)
```




```
Average Macro Train Accuracy = 0.86
```

[27]:
```
plt.plot(all_iterCt_adam_1, all_losses_adam_1,␣
 ↪all_iterCt_lstm_3,all_losses_lstm_3)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(['Baseline_Adam','LSTM'])
plt.title("Performance of Models")
plt.show()
```



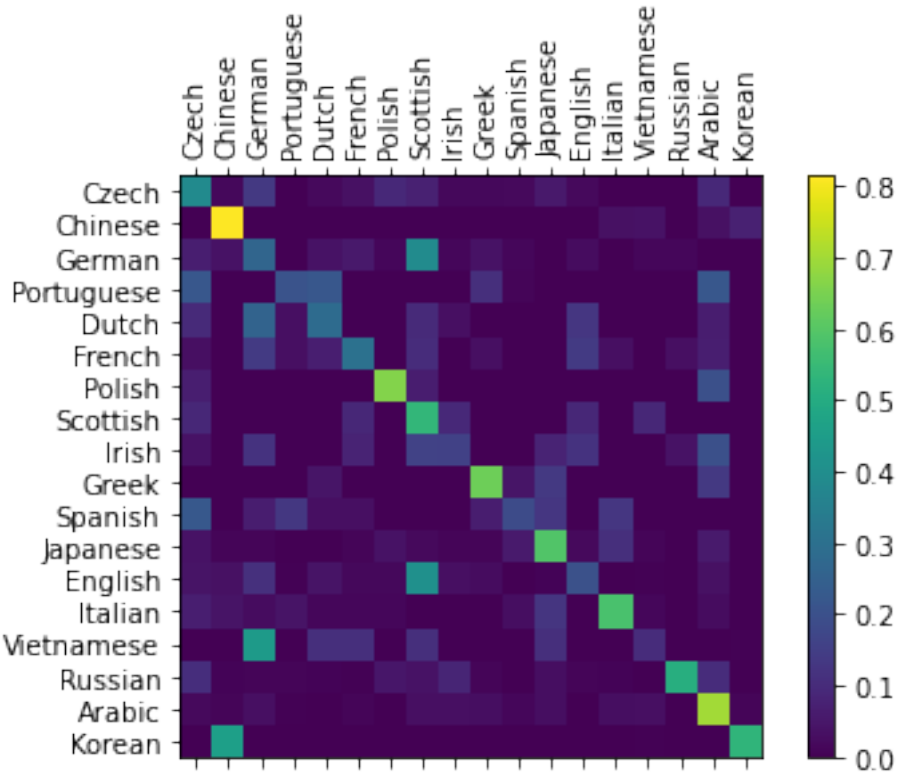
1.3.2 Task 3.2

[16 pts] Evaluate the performance of both models on the test set.

Performance of baseline model trained using Adam optimizer on the test set

[33]:
```
adam_confusion=accuracy(rnn_adam_1, all_categories)
test_plot(adam_confusion, all_categories)
```

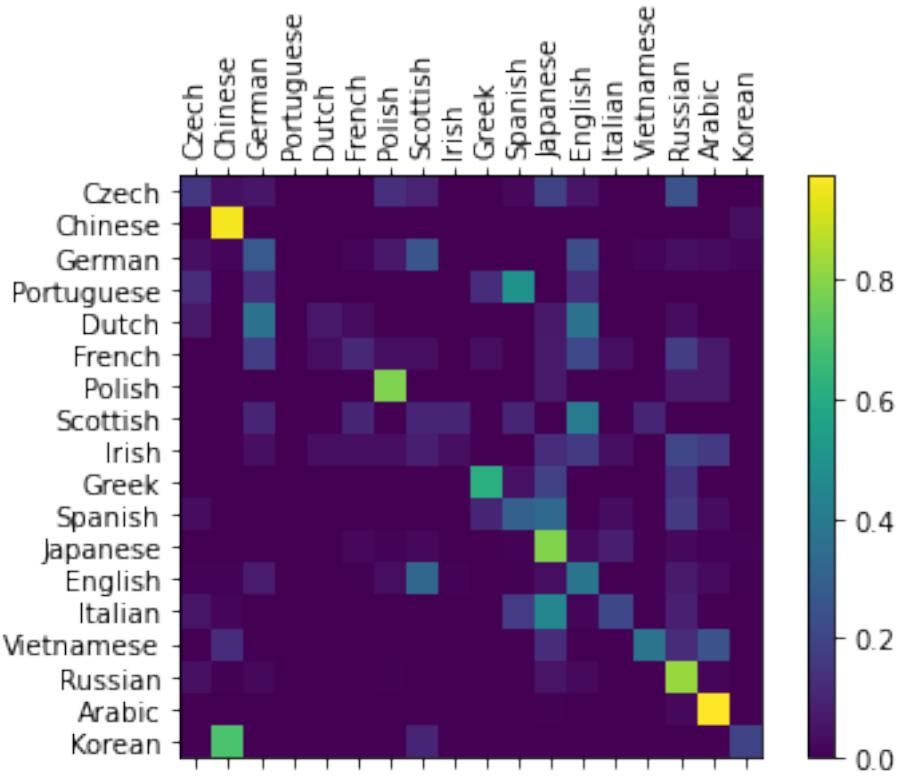
```
Average Macro Accuracy = 0.43
```



Performance of LSTM model trained using Adam optimizer on the test set

[31]:
```
lstm_confusion=accuracy(lstm, all_categories)
test_plot(lstm_confusion, all_categories)
```

```
Average Macro Accuracy = 0.40
```



1.3.3 Task 3.3

[8 pts] Comment on the performance of both methods on the test set, and the shape of their learning curves.

Train the baseline model with Adam Optimizer :

- Train Accuracy= 54%
- Test Accuracy= 43%
- Train Accuracy-Test Accuracy=11%

LSTM model trained with Adam optimizer:

- Train Accuracy= 86%
- Test Accuracy= 40%
- Train Accuracy-Test Accuracy=46%

Observations:

- Baseline model trained with Adam optimizer better than the LSTM model trained using Adam optimizer as the Test Accuray is more with the baseline model.

- Performance graph showing training loss of both the models starts at same point and with iterations LSTM training loss decreses more.
- LSTM model trained with Adam optimizer have steeper learning curve.
- Both the model are overfitting but comparitively LSTM Model using Adam optimizer is more overfitted as difference between training accuracy and test accuracy is much larger than the baseline model using adam optimizer.
- This shows that the LSTM model is more specialized to training data and its unable to generalize well to the new data, resulting in a decrease in test accuracy.
- When we plot the test accuracy at each epoch along with the train accuracy we will get more insights about where exactly the overfitting has started

[36]:
```
!apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-latex-recommended
# Convert Notebook to PDF
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono fonts-texgyre
  javascript-common libcupsfilters1 libcupsimage2 libgs9 libgs9-common
  libijs-0.35 libjbig2dec0 libjs-jquery libkpathsea6 libpotrace0 libptexenc1
  libruby2.5 libsynctex1 libtexlua52 libtexluajit2 libzzip-0-13 lmodern
  poppler-data preview-latex-style rake ruby ruby-did-you-mean ruby-minitest
  ruby-net-telnet ruby-power-assert ruby-test-unit ruby2.5
  rubygems-integration t1utils tex-common tex-gyre texlive-base
  texlive-binaries texlive-latex-base texlive-latex-extra texlive-pictures
  texlive-plain-generic tipa
Suggested packages:
  fonts-noto apache2 | lighttpd | httpd poppler-utils ghostscript
  fonts-japanese-mincho | fonts-ipafont-mincho fonts-japanese-gothic
  | fonts-ipafont-gothic fonts-arphic-ukai fonts-arphic-uming fonts-nanum ri
  ruby-dev bundler debhelper gv | postscript-viewer perl-tk xpdf-reader
  | pdf-viewer texlive-fonts-recommended-doc texlive-latex-base-doc
  python-pygments icc-profiles libfile-which-perl
  libspreadsheet-parseexcel-perl texlive-latex-extra-doc
  texlive-latex-recommended-doc texlive-pstricks dot2tex prerex ruby-tcltk
  | libtcltk-ruby texlive-pictures-doc vprerex
The following NEW packages will be installed:
  fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono fonts-texgyre
  javascript-common libcupsfilters1 libcupsimage2 libgs9 libgs9-common
  libijs-0.35 libjbig2dec0 libjs-jquery libkpathsea6 libpotrace0 libptexenc1
  libruby2.5 libsynctex1 libtexlua52 libtexluajit2 libzzip-0-13 lmodern
  poppler-data preview-latex-style rake ruby ruby-did-you-mean ruby-minitest
  ruby-net-telnet ruby-power-assert ruby-test-unit ruby2.5
```