AAI-511-Final-Project_NathanE_ChrisW_PaulT

August 11, 2023

Name : *Christopher J. Watson , Nathan Edwards , Paul Thai*

School : *Marcos School of Engineering, University of San Diego*

Class : *AAI 511 – Neural Networks and Learning*

Assignment : *MSAAI Final Project*

Date : 8/15/2023

0.0.1 Libraries

---

[1]:
```
# Libraries

# File and Data Handling
import os
import time

# Data Visualization and Numerical Computing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# MIDI Processing
import mido

# Machine Learning and Deep Learning
from torch.utils.data import Dataset, DataLoader
import torch
from torch import nn, optim

# Machine learning Metrics and Evaluations
```

```
from sklearn.metrics import accuracy_score, precision_score, f1_score,␣
↪confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import f1_score

# Enable Cuda Development
print('GPU Available: ', torch.cuda.is_available())
if torch.cuda.is_available():
    print('GPU Name: ', torch.cuda.get_device_name(torch.cuda.current_device()))
    device = torch.device("cuda:" + str(torch.cuda.current_device()))
```

```
C:\Users\chris\anaconda3\envs\tensorflow\lib\site-packages\tqdm\auto.py:21:
TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
GPU Available:  True
GPU Name:  NVIDIA GeForce RTX 4080
```

0.0.2 Classes

---

[2]:
```
# Define what the data is.
class Dataset(Dataset):

    def __init__(self, sequences, labels):
        self.sequences = sequences
        self.labels = labels

    def enum(self,y):
        # set composer list
        composers = ["bach", "bartok", "byrd", "chopin", "handel", "hummel",␣
↪"mendelssohn","mozart", "schumann"]
        #print(y)
        for i in range(len(composers)):
            if composers[i] == y:
                slot = i
        out = [0] * 9
        out[slot] = 1

        return out

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

    def __getitem__(self, idx):
        sequence = self.sequences[idx]
```

```
        label = self.labels[idx]
        label = self.enum(label)


        sequence = torch.tensor(sequence)
        label = torch.tensor(label)
        # enable cuda cores
        if torch.cuda.is_available():
            label = label.to(device)
            #print('GPU Tensor Enabled(Labels):', label.is_cuda)
            sequence = sequence.to(device)
            #print('GPU Tensor Enabled(Sequences):', sequence.is_cuda)
        sequence = sequence[None, :, :]
        return sequence, label
```

0.0.3 Helper Functions

---

[3]:

```
# Gets directory listing
def get_children(a_dir):
    dirs = []
    files = []
    for name in os.listdir(a_dir):
        if os.path.isdir(os.path.join(a_dir, name)):
            dirs.append(name)
        else:
            files.append(name)
    return [dirs,files]

# Creates tables from files in directory
def create_files_table(top_level, out_file):
    temp_comps = []
    temp_songs = []
    temp_paths = []

    composer_names, songs = get_children(top_level)

    for composer in composer_names:
        temp_path = top_level + '/' + composer
        temp, songs = get_children(temp_path)
        for song in songs:
            if song != '.DS_Store':
                temp_comps.append(composer)
                temp_paths.append(temp_path + '/' + song)
                temp_songs.append(song.split(".")[0])
```

```python
    temp_dict = {'Composers': temp_comps, 'Songs': temp_songs, 'Paths':␣
↪temp_paths}

    table = pd.DataFrame.from_dict(temp_dict)

    table.to_csv('./' + out_file + '.csv',index=False)

    return table


# Function to extract the notes played in a MIDI file with timestamps
def extract_notes_with_meta(midi_filepath):
    notes = {}
    midi = mido.MidiFile(midi_filepath)
    max_time = 0
    time_counter = 0
    for track in midi.tracks:
        time_counter = 0
        for msg in track:
            time_counter += msg.time
            max_time = max(max_time,time_counter)
            if msg.type == 'note_on':
                if msg.velocity != 0:  # Ensure it's a Note On event
                    notes[msg.note] = notes.get(msg.note, []) + [(msg.velocity,␣
↪time_counter, 1)]  # 1 represents Note On
            elif msg.type == 'note_off':
                notes[msg.note] = notes.get(msg.note, []) + [(msg.
↪velocity,time_counter, 0)]  # 0 represents Note Off

    return notes, max_time

# Creates a single sequence from a song as a sample for training
def create_single_sequences(notes, start, tick_count, seq_count):
    VEL = 0
    TM = 1
    ON = 2

    temp_keys = notes.keys()

    seq =  [[0] * 128]* seq_count
    seq = np.array(seq)

    for x in temp_keys:
        temp_note = np.array(notes[x])
        time_store = 0
        for i in range(start, seq_count+start):
            temp_vel = 0
```

```
            for t in range(time_store,temp_note[:,TM].size):
                if (temp_note[t,TM]>tick_count*i):
                    break
                else:
                    if temp_note[t,ON] == 1:
                        if temp_note[t,VEL] > temp_vel:
                            temp_vel = temp_note[t,VEL]
                time_store = t
            seq[i-start,x] = temp_vel
    return seq

# Sequences all songs in a table
def sequence_songs(df_songs, tick_count, seq_count, jiggle_on=False):
    # temp variables
    labels = []
    sequences = []

    # shake variable
    shake_amount = [0]

    # jiggle the content of the data
    if jiggle_on:
        shake_amount = [0, int(seq_count/2)]

    # main data loop
    for j in shake_amount:
        for song in df_songs.iterrows():
            temp_count = tick_count
            song = song[1]
            notes, max_time = extract_notes_with_meta(song['Paths'])
            if max_time / tick_count < seq_count:
                temp_count = int((max_time / seq_count)*.8)
            for i in range(int((max_time-j)/(seq_count*temp_count))):
                sequences.append(create_single_sequences(notes, i*seq_count+j,␣
↪temp_count, seq_count))
                labels.append(song['Composers'])

    return labels, sequences
```

0.0.4 Preprocessing

---

[4]:
```
# get all the data paths for the dataset
devpath = './Composer_Dataset/NN_midi_files_extended/dev/'
testpath = './Composer_Dataset/NN_midi_files_extended/test/'
trainpath = './Composer_Dataset/NN_midi_files_extended/train/'
```

```python
# create tables for each set
dev_table = create_files_table(devpath, 'dev_table')
test_table = create_files_table(testpath, 'test_table')
train_table = create_files_table(trainpath, 'train_table')
```

[6]:
```python
# This was the various samplings used in conjuction with transfer learning
#1labels, sequences = sequence_songs(train_table, 200, 128,jiggle_on=True)
labels, sequences = sequence_songs(train_table, 300, 128,jiggle_on=True)
#2labels, sequences = sequence_songs(train_table,  128, 128,jiggle_on=True)
```

[7]:
```python
# This was the various samplings used in conjuction with transfer learning
#1labels_test, sequences_test = sequence_songs(test_table, 200,␣
↪128,jiggle_on=True)
labels_test, sequences_test = sequence_songs(test_table, 300,␣
↪128,jiggle_on=True)
#2labels_test, sequences_test = sequence_songs(test_table, 128,␣
↪128,jiggle_on=True)
```

[8]:
```python
# Show all the composers
myset = set(labels_test)
myset
```

[8]:
```
{'bach',
 'bartok',
 'byrd',
 'chopin',
 'handel',
 'hummel',
 'mendelssohn',
 'mozart',
 'schumann'}
```

[10]:
```python
# Show all composers in the table
train_table.Composers.unique()
```

[10]:
```
array(['bach', 'bartok', 'byrd', 'chopin', 'handel', 'hummel',
       'mendelssohn', 'mozart', 'schumann'], dtype=object)
```

[11]:
```python
# Creating the dataset for the dataloader
dataset = Dataset(sequences, labels)
dataset_test = Dataset(sequences_test, labels_test)

# Creating data loader
batch_size = 1

data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
test_loader = DataLoader(dataset_test, batch_size=batch_size, shuffle=False)
```

[12]:
```
dataset[0][0].shape
```

[12]: torch.Size([1, 128, 128])

[13]:
```
len(test_loader)
```

[13]: 334

0.0.5 Model Creation

---

[14]:
```
# Model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 4, kernel_size=2, padding=1)
        self.conv2 = nn.Conv2d(4, 8, kernel_size=2, padding=1)
        self.conv3 = nn.Conv2d(8, 16, kernel_size=2, padding=1)
        self.conv4 = nn.Conv2d(16, 32, kernel_size=2, padding=1)

        self.soft = nn.Softmax(dim=1)

        self.pool = nn.MaxPool2d(kernel_size=2)

        self.LSTM = nn.LSTM(8, 300, 7, batch_first=True)

        self.fc1 = nn.Linear(2400, 1200)
        self.fc2 = nn.Linear(1200, 600)
        self.fc3 = nn.Linear(600, 100)
        self.fc4 = nn.Linear(100, 9)

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = self.pool(nn.functional.relu(self.conv3(x)))
        x = self.pool(nn.functional.relu(self.conv4(x)))

        x =  x[:,0,:,:]
        h0 = torch.zeros(7, x.size(0), 300)
        c0 = torch.zeros(7, x.size(0), 300)
        if torch.cuda.is_available():
            h0 = h0.to(device)
            #print('GPU Tensor Enabled(labels):', labels.is_cuda)
            c0 = c0.to(device)
            #print('GPU Tensor Enabled(inputs):', inputs.is_cuda)
```

```
        x, _ = self.LSTM(x, (h0, c0))

        x = torch.flatten(x, 1)

        x = nn.functional.relu(self.fc1(x))
        x = nn.functional.relu(self.fc2(x))
        x = nn.functional.relu(self.fc3(x))
        x = self.fc4(x)
        # return nn.functional.sigmoid(x)
        return self.soft(x)
```

[15]:
```
# Create model object
model = CNN()
if torch.cuda.is_available():
    model.cuda()

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adagrad(model.parameters(),lr=0.001)
```

[18]:
```
# This was the score of our best model which can be imported through the model␣
↪load function
best_model = 0.7245685124246504
```

[16]:
```
# load state of previous model
model.load_state_dict(torch.load('best-model.pt'))
```

[16]: <All keys matched successfully>

0.0.6 Training

---

[17]:
```
def Test(dataloader, best_model):
    # Evaluating the model
    model.eval()

    real = []
    pred = []
    counter = 0
    # Disable gradient computation to save memory
    with torch.no_grad():
        for inputs, labels in dataloader:

            # Forward pass
            outputs = model(inputs.float())
            _, predicted = torch.max(outputs.data, 1)
```

```
            outputs = outputs.tolist()
            labels = labels.tolist()

            pred_labal = outputs[0].index(max(outputs[0]))

            real_label = labels[0].index(max(labels[0]))

            real.append(real_label)
            pred.append(pred_labal)

    # scoring metric
    score = f1_score(pred, real, average = "weighted")

    Enable best save
    if score > best_model:
        print(classification_report(real, pred))
        best_model = score
        torch.save(model.state_dict(), 'model_something.pt')

    return score, best_model
```

[23]:
```
# Training loop
num_epochs = 3
prev = time.time()
cur_model = 0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in data_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()

        inputs = inputs.to(torch.float)
        labels = labels.to(torch.float)
        if torch.cuda.is_available():
            labels = labels.to(device)
            inputs = inputs.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
```

```
    # Print the stats loss for the epoch
    epoch_loss = running_loss / len(data_loader)
    cur_model, best_model = Test(test_loader, best_model)
    now = time.time()
    print(f"Epoch [{epoch+1}/{num_epochs}] Loss: {epoch_loss:.4f} Time-Taken:␣
 ↪{now-prev}s F1-Score: {cur_model} Best-Score: {best_model}")
    prev = now

# we never once reached this statement except to show of this notebook at the␣
 ↪end.
print("Training complete!")
```

```
Epoch [1/3] Loss: 0.0210 Time-Taken: 32.40267491340637s F1-Score:
0.6959843236492259 Best-Score: 0.7175077123970345
              precision    recall  f1-score   support

           0       1.00      0.91      0.95        58
           1       0.33      0.40      0.36        10
           2       0.86      1.00      0.92        30
           3       0.87      0.69      0.77        48
           4       0.39      0.39      0.39        18
           5       0.85      0.81      0.83        62
           6       0.59      0.42      0.49        40
           7       0.66      0.84      0.74        44
           8       0.35      0.50      0.41        24

    accuracy                           0.73       334
   macro avg       0.66      0.66      0.65       334
weighted avg       0.75      0.73      0.73       334

Epoch [2/3] Loss: 0.0206 Time-Taken: 29.49116611480713s F1-Score:
0.7245685124246504 Best-Score: 0.7245685124246504
Epoch [3/3] Loss: 0.0205 Time-Taken: 30.71710443496704s F1-Score:
0.700046913635001 Best-Score: 0.7245685124246504
Training complete!
```

0.0.7 Evaluation and Results

---

[24]:
```
# Evaluating the model
model.eval()

prediction_list = list()
labels_list = list()

real = []
```

```python
pred = []
counter = 0
# Disable gradient computation to save memory
with torch.no_grad():
    for inputs, labels in test_loader:

        # Forward pass
        outputs = model(inputs.float())
        _, predicted = torch.max(outputs.data, 1)

        #print(outputs)

        outputs = outputs.tolist()
        labels = labels.tolist()

        pred_labal = outputs[0].index(max(outputs[0]))


        #print(pred_labal)

        #print(labels)

        real_label = labels[0].index(max(labels[0]))

        real.append(real_label)
        pred.append(pred_labal)



print(classification_report(real, pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.90      0.95        58
           1       0.29      0.40      0.33        10
           2       0.83      1.00      0.91        30
           3       0.76      0.73      0.74        48
           4       0.53      0.44      0.48        18
           5       0.91      0.77      0.83        62
           6       0.54      0.38      0.44        40
           7       0.61      0.82      0.70        44
           8       0.23      0.29      0.25        24

    accuracy                           0.70       334
   macro avg       0.63      0.64      0.63       334
weighted avg       0.72      0.70      0.71       334
```

Considering how many times the model was loaded and retrained with different variations of the train data, sometimes the results varied per run slightly. We never reached an end conclusion of how accurate our model could get because we were limited on training time but these are some of our demonstrated results. Our best scores had an F1-Score of .724 and an accuracy of 73%. We consider this to be adequate results and our thoughts for improvements are documented in the final report