Ονοματεπώνυμο: Χαριτούδης Απόστολος-Ανδρέας

Μάθημα: Επεξεργασία Σημάτων Φωνής και Ήχου

Α.Μ.: π17178

Εξάμηνο: 8ο

Πανεπιστήμιο Πειραιώς (Πα.Πει)

1ο θέμα

Λογική σχεδίασης του υλοποιημένου μοντέλου:

Η λογική σχεδίασης του μοντέλου ASR (automated speech recognition) συστήματος ήταν μέσω ενός καλά οργανωμένου dataset όπου κάθε λέξη της αγγλικής γλώσσας θα ήταν δική του κλάση με ακουστικά αρχεία εύρους από εκατοντάδες έως χιλιάδες, τα οποία θα αντιστοιχούσαν στην κλάση αυτή. Η δομή της dataset θα πρέπει να είναι της μορφής:

Όνομα_Αρχείου

|

Audio

{fold1…….foldn με n το συνολικό πλήθος των λέξεων το αγγλικού λεξιλογίου βάση του Wikipedia:
https://en.wikipedia.org/wiki/List_of_dictionaries_by_number_of_words
το 2016 υπολογίζονται ότι υπήρχαν 520,000

Από την άλλη έχουμε αυτό το άρθρο :
https://englishlive.ef.com/blog/language-lab/many-words-english-language/

Το οποίο επισημαίνει ότι το συνολικό πλήθος είναι ακόμα μεγαλύτερο. Όπως και να έχει θα έχουμε πάνω 500 χιλιάδες κλάσεις, 1 κλάση για κάθε λέξη.

[Κάθε κλάση μέσα θα έχει από δεκάδες έως και χιλιάδες ακουστικά αρχεία της ίδιας λέξης ώστε το μοντέλο να μπορεί να ξεχωρίζει την ίδια λέξη με τους διαφορετικούς τρόπους προφοράς, καθώς όλοι έχουμε ιδιοτροπίες προφοράς στον προφορικό λόγο. Γενική ιδέα είναι να μειωθεί η πιθανότητα σφάλματος μεταξύ λέξεων με παρόμοιο τρόπο προφοράς.

|

Metadata

{metadata_file_name.csv

|

ReadMeFile.txt

|

Source_of_Audio_fiels_contributors.txt

Όπως φάνηκε παραπάνω θα πρέπει να υπάρχουν 4 αρχεία μέσα στον φάκελο, τα 2 από αυτά δεν είναι απαραίτητα αλλά θα βοηθήσουν την αναγνώριση όσων συνέφεραν στο ακουστικό υλικό, καθώς και μια μικρή επεξήγηση στο readme file το οποίο θα επεξηγεί σύντομα όλα τα απαραίτητα.

Δομή των αρχείων :

Το CSV είναι της δομής :

* slice_file_name:

Το όνομα του αρχείου ήχου. Το όνομα λαμβάνει την ακόλουθη μορφή: [audioID]-[classID]-[occurenceID]-[sliceID] .wav, όπου:

[audioID] = το audio ID της ηχογράφησης από την οποία έχει ληφθεί αυτό το απόσπασμα (φέτα)

[classID] = ένα αριθμητικό αναγνωριστικό της κλάσης ήχου)

[occurenceID] = ένα αριθμητικό αναγνωριστικό για τη διάκριση διαφορετικών εμφανίσεων του ήχου μέσα στην αρχική εγγραφή

[sliceID] = ένα αριθμητικό αναγνωριστικό για να διακριθούν οι διαφορετικές φέτες που έχουν ληφθεί από το ίδιο περιστατικό

* audioID:

Το audio ID της ηχογράφησης από την οποία έχει ληφθεί αυτό το απόσπασμα (φέτα)

* start

Ο χρόνος έναρξης της φέτα στην αρχική ηχογράφηση του αρχείου

* end:

Ο χρόνος λήξης της φέτα στην αρχική ηχογράφηση του αρχείου

* saliance:

Μια (υποκειμενική) εκτίμηση του ήχου. 1 = πρώτο πλάνο, 2 = φόντο.

*fold:

Ο αριθμός (1-n) στον οποίο έχει εκχωρηθεί αυτό το αρχείο.

* classID:

Αριθμητικό αναγνωριστικό της κατηγορίας ήχου: Με σημείο έναρξης το 0 και τέλος n με βάση το πόσες λέξεις καταχωρηθούν.

* class:

Το όνομα της τάξης: Κάθε λέξη του αγγλικού λεξιλογίου από το a έως το z

Σημείωση: Ίσως να ήταν προτιμότερο να είχα προσθέσει και την αλφαβητική ταξινόμηση για την μείωση φόρτους ως "raw" data και για μια καλύτερη οργάνωση. Δηλαδή να υπήρχε μια υπερκλάση όλων των γραμμάτων, να ξεκίναγε ως εξής :superfold1-26 για τα 26 γράμματα του αγγλικού αλφαβήτου και μετά εντός κάθε superfold ,fold-superfoldnumber-1-m με μ το ολικό πλήθος των λέξεων που ξεκινάνε με το γράμμα της superfold στη οποία ανήκει και το superfoldnumber το νούμερο της superfold ώστε να αποφευχθούν τυχών σφάλματα καθώς superfold θα έχει έως ένα σημείο κοινή αρίθμηση, οπότε βάζοντας το νούμερο της superfold μετά fold, επιβιώνουμε σε ποια υπερκλάσης ανήκει. Αλλά δεν υλοποιήθηκε στον κώδικα αυτή εκδοχή οπότε θα είναι περιττή πληροφορία. Να σημειωθεί ότι η εργασία δεν περιέχει δεδομένα εκπαίδευσης καθώς τα προτεινόμενα δεδομένα από το OpenSLR δεν βρισκόντουσαν στην μορφή που επιθυμούσα και θα έπαιρνε πάρα πολύ χρόνο να τα κάνω slice όλα για να είναι κάθε λέξη ξεχωριστά, με βάση την λογική υλοποίησης μοντέλου που ακολούθησα.

Κώδικας γράφτηκε στο Jupyter Notebook σε γλώσσα python:

```
import math, random
import torch
import torchaudio
from torchaudio import transforms
from IPython.display import Audio
import pandas as pd
from pathlib import Path
from torch.utils.data import DataLoader, Dataset, random_split
import torch.nn.functional as F
```

```
from torch.nn import init



# Prepare training data from Metadata file
# In the 'filename' between then '' goes the filename with the dataset
download_path = Path.cwd()/''

# Read metadata file
#if that exist in csv formation 'folder'/'csv file'
metadata_file = download_path/''/''
df = pd.read_csv(metadata_file)
df.head()

# Construct file path by concatenating fold and file name
#needs to exist /fold/arxeia
df['relative_path'] = '/fold' + df['fold'].astype(str) + '/' + df['slice_file_name'].astype(str)

# Take relevant columns
#every word  has its own "class"-category so that no word will be mixed and create inconsistence in this model
df = df[['relative_path', 'classID']]
df.head()

class AudioUtil():

  # Load an audio file. Return the signal as a tensor and the sample rate

  @staticmethod
  def open(audio_file):
    sig, sr = torchaudio.load(audio_file)
    return (sig, sr)
 # Convert the given audio to the desired number of channels

  @staticmethod
```

```
 def rechannel(aud, new_channel):
  sig, sr = aud

  if (sig.shape[0] == new_channel):
   # Nothing to do
   return aud

  if (new_channel == 1):
   # Convert from stereo to mono by selecting only the first channel
   resig = sig[:1, :]
  else:
   # Convert from mono to stereo by duplicating the first channel
   resig = torch.cat([sig, sig])

  return ((resig, sr))
 # Since Resample applies to a single channel, we resample one channel at a time, if we have different frequencies
 @staticmethod
 def resample(aud, newsr):
  sig, sr = aud

  if (sr == newsr):
   # Nothing to do
   return aud

  no_channels = sig.shape[0]
  # Resample first channel
  resig = torchaudio.transforms.Resample(sr, newsr)(sig[:1,:])
  if (no_channels > 1):
   # Resample the second channel and merge both channels
   retwo = torchaudio.transforms.Resample(sr, newsr)(sig[1:,:])
   resig = torch.cat([resig, retwo])

  return ((resig, newsr))
```

```
# Pad  the signal to a fixed length 'max_ms' in milliseconds
@staticmethod
def pad_trunc(aud, max_ms):
  sig, sr = aud
  no_rows, sig_len = sig.shape
  max_len = sr//1000 * max_ms

  if (sig_len > max_len):
    # Truncate the signal to the given length
    sig = sig[:,:max_len]

  elif (sig_len < max_len):
    # Length of padding to add at the beginning and end of the signal
    pad_begin_len = random.randint(0, max_len - sig_len)
    pad_end_len = max_len - sig_len - pad_begin_len

    # Pad with 0s
    pad_begin = torch.zeros((no_rows, pad_begin_len))
    pad_end = torch.zeros((no_rows, pad_end_len))

    sig = torch.cat((pad_begin, sig, pad_end), 1)

  return (sig, sr)

# Shifts the signal to the left or right by some percent.

@staticmethod
def time_shift(aud, shift_limit):
  sig,sr = aud
  _, sig_len = sig.shape
  shift_amt = int(random.random() * shift_limit * sig_len)
  return (sig.roll(shift_amt), sr)
```

```
# Generate a Spectrogram


  @staticmethod
  def spectro_gram(aud, n_mels=64, n_fft=1024, hop_len=None):
    sig,sr = aud
    top_db = 80

    # spec has shape [channel, n_mels, time], where channel is mono, stereo etc
    spec = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n_mels=n_mels)(sig)

    # Convert to decibels
    spec = transforms.AmplitudeToDB(top_db=top_db)(spec)
    return (spec)

  @staticmethod
  def spectro_augment(spec, max_mask_pct=0.1, n_freq_masks=1, n_time_masks=1):
    _, n_mels, n_steps = spec.shape
    mask_value = spec.mean()
    aug_spec = spec

    freq_mask_param = max_mask_pct * n_mels
    for _ in range(n_freq_masks):
      aug_spec = transforms.FrequencyMasking(freq_mask_param)(aug_spec, mask_value)

    time_mask_param = max_mask_pct * n_steps
    for _ in range(n_time_masks):
      aug_spec = transforms.TimeMasking(time_mask_param)(aug_spec, mask_value)

    return aug_spec
# Audio/Speech Dataset

class Audioo(Dataset):
  def __init__(self, df, data_path):
```

```
    self.df = df
    self.data_path = str(data_path)
    self.duration = 5000
    self.sr = 8000
    self.channel = 2
    self.shift_pct = 0.4
# Number of items in dataset

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



  # Get i'th item in dataset
  def __getitem__(self, idx):
    # Absolute file path of the audio file - concatenate the audio directory with
    # the relative path
    ##TODO declare data_path value its not working
    audio_file = self.data_path + self.df.loc[idx, 'relative_path']
    # Get the Class ID
    class_id = self.df.loc[idx, 'classID']

    aud = AudioUtil.open(audio_file)
    # Some sounds have a higher sample rate, or fewer channels compared to the
    # majority. So make all sounds have the same number of channels and same
    # sample rate. Unless the sample rate is the same, the pad_trunc will still
    # result in arrays of different lengths, even though the sound duration is
    # the same.
    reaud = AudioUtil.resample(aud, self.sr)
    rechan = AudioUtil.rechannel(reaud, self.channel)

    dur_aud = AudioUtil.pad_trunc(rechan, self.duration)
    shift_aud = AudioUtil.time_shift(dur_aud, self.shift_pct)
    sgram = AudioUtil.spectro_gram(shift_aud, n_mels=64, n_fft=1024, hop_len=None)
```

```
    aug_sgram = AudioUtil.spectro_augment(sgram, max_mask_pct=0.1, n_freq_masks=2, n_time_masks=2)

    return aug_sgram, class_id

myds = Audioo(df, data_path)

# Random split of 90:10 between training and validation
no_items = len(myds)
no_train = round(no_items * 0.9)
no_val = no_items - no_train
train_ds, val_ds = random_split(myds, [no_train, no_val])

# Create training and validation data loaders
train_dl = torch.utils.data.DataLoader(train_ds, batch_size=16, shuffle=True)
val_dl = torch.utils.data.DataLoader(val_ds, batch_size=16, shuffle=False)
# Audio Classification Model

class AudioClassifier (nn.Module):
    # ----------------------------
    # Build the model architecture
    # ----------------------------
    def __init__(self):
        super().__init__()
        conv_layers = []

        # First Convolution Block with Relu and Batch Norm. Use Kaiming Initialization
        self.conv1 = nn.Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(8)
        init.kaiming_normal_(self.conv1.weight, a=0.1)
        self.conv1.bias.data.zero_()
        conv_layers += [self.conv1, self.relu1, self.bn1]
```

```
    # Second Convolution Block
    self.conv2 = nn.Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    self.relu2 = nn.ReLU()
    self.bn2 = nn.BatchNorm2d(16)
    init.kaiming_normal_(self.conv2.weight, a=0.1)
    self.conv2.bias.data.zero_()
    conv_layers += [self.conv2, self.relu2, self.bn2]

    # Second Convolution Block
    self.conv3 = nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    self.relu3 = nn.ReLU()
    self.bn3 = nn.BatchNorm2d(32)
    init.kaiming_normal_(self.conv3.weight, a=0.1)
    self.conv3.bias.data.zero_()
    conv_layers += [self.conv3, self.relu3, self.bn3]

    # Second Convolution Block
    self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    self.relu4 = nn.ReLU()
    self.bn4 = nn.BatchNorm2d(64)
    init.kaiming_normal_(self.conv4.weight, a=0.1)
    self.conv4.bias.data.zero_()
    conv_layers += [self.conv4, self.relu4, self.bn4]

    # Linear Classifier
    self.ap = nn.AdaptiveAvgPool2d(output_size=1)
    self.lin = nn.Linear(in_features=64, out_features=10)

    # Wrap the Convolutional Blocks
    self.conv = nn.Sequential(*conv_layers)



def forward(self, x):
```

```
        # Run the convolutional blocks
        x = self.conv(x)

        # Adaptive pool and flatten for input to linear layer
        x = self.ap(x)
        x = x.view(x.shape[0], -1)

        # Linear layer
        x = self.lin(x)

        # Final output
        return x

# Create the model
speechMod = AudioClassifier()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
speechMod = speechMod.to(device)
# Check that it is on Cuda
next(speechMod.parameters()).device
# Training Loop

def training(model, train_dl, num_epochs):
  # Loss Function, Optimizer and Scheduler
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
  scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001,
                                    steps_per_epoch=int(len(train_dl)),
                                    epochs=num_epochs,
                                    anneal_strategy='linear')

  # Repeat for each epoch
  for epoch in range(num_epochs):
    running_loss = 0.0
```

```
correct_prediction = 0
total_prediction = 0

# Repeat for each batch in the training set
for i, data in enumerate(train_dl):
    # Get the input features and target labels, and put them on the GPU
    inputs, labels = data[0].to(device), data[1].to(device)

    # Normalize the inputs
    inputs_m, inputs_s = inputs.mean(), inputs.std()
    inputs = (inputs - inputs_m) / inputs_s

    # Zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    scheduler.step()

    # Keep stats for Loss and Accuracy
    running_loss += loss.item()

    # Get the predicted class with the highest score
    _, prediction = torch.max(outputs,1)
    # Count of predictions that matched the target label
    correct_prediction += (prediction == labels).sum().item()
    total_prediction += prediction.shape[0]

# Print stats at the end of the epoch
no_batches = len(train_dl)
```

```
    avg_loss = running_loss / no_batches
    acc = correct_prediction/total_prediction
    print(f'Epoch: {epoch}, Loss: {avg_loss:.2f}, Accuracy: {acc:.2f}')
    print('Finished Training')
no_epochs=2   #Can take higher or lower value
training(speechMod, train_dl, no_epochs)
# Inference

def inference (model, val_dl):
  correct_prediction = 0
  total_prediction = 0

  # Disable gradient updates
  with torch.no_grad():
    for data in val_dl:
      # Get the input features and target labels
      inputs, labels = data[0].to(device), data[1].to(device)

      # Normalize the inputs
      inputs_m, inputs_s = inputs.mean(), inputs.std()
      inputs = (inputs - inputs_m) / inputs_s

      # Get predictions
      outputs = model(inputs)

      # Get the predicted class with the highest score
      _, prediction = torch.max(outputs,1)
      # Count of predictions that matched the target label
      correct_prediction += (prediction == labels).sum().item()
      total_prediction += prediction.shape[0]

  acc = correct_prediction/total_prediction
  print(f'Accuracy: {acc:.2f}, Total items: {total_prediction}')
```

```
# Run inference on trained model with the validation set
inference(speechMod, val_dl)
##ToDo Call to predict-"decipher"a give audio file.
```

Αρχικά, βρίσκουμε την τοποθεσία του CSV αρχείου, μετά το διαβάζουμε και παίρνουμε το class id και το relative path. Το class id είναι η αριθμητική ταξινόμηση των λέξεων και το relative path, είναι το path που ¨φτιάξαμε¨ συνενώνοντας το όνομα του αρχείου και το fold. Έπειτα, διαβάζουμε τα ακουστικά αρχεία μορφής wav με την χρήση torchaudio, pytorch. Σε περίπτωση που έχουμε αρχεία mono ή stereo θα μεταβούμε στις απαραίτητες ενέργειες, ώστε να μετατρέψουμε το mono σε stereo με το να διπλασιάσουμε το mono με τον εαυτό του για να το μετατρέψουμε σε stereo. Και αυτό, καθώς πιστεύω θα είναι πιο εύκολο για το μοντέλο μας να αναμένει ότι όλα τα στοιχεία θα έχουν τις ίδιες διαστάσεις, για αυτό μετατρέπει τα μονοφωνικά αρχεία σε στερεοφωνικά, διπλασιάζοντας το πρώτο κανάλι στο δεύτερο. Επειδή όμως μπορεί να υπάρξει διαφορά μεγεθών στον ρυθμό δειγματοληψίας από αρχείο σε αρχείο προβαίνουμε στις απαραίτητες ενέργειες, ώστε όλα τα ακουστικά αρχεία να έχουν ρυθμό δειγματοληψίας 8000Hz. Πιο συγκεκριμένα, τυποποιούμε και μετατρέπουμε όλους τους ήχους στον ίδιο ρυθμό δειγματοληψίας. Στην συνέχεια, αλλάζουμε το μέγεθος όλων των δειγμάτων ήχου για να έχουν το ίδιο μήκος λαμβάνοντας υπόψη μας ότι μέσος όρος άρθρωσης λέξεων του αγγλικού λεξιλογίου είναι 4.7 δευτερόλεπτα, οπότε εμείς θα το κάνουμε 5, είτε επεκτείνοντας τη διάρκεια του, γεμίζοντάς το με σιωπή, είτε περικόπτοντάς το. Μετά από αυτές τις διεργασίες, κάνουμε augment στο ακατέργαστο ηχητικό σήμα εφαρμόζοντας Time Shift για να μετατοπίσουμε τον ήχο προς τα αριστερά ή τα δεξιά κατά ένα τυχαίο ποσό. Στη συνέχεια μετατρέπουμε τον ενισχυμένο ήχο σε φασματογράφημα Mel, καθώς με βάση το τρόπο υλοποίησης που ακολούθησα, ίσως να είναι ο καταλληλότερος τρόπος εισαγωγής δεδομένων ήχου στο μοντέλο αυτό.

Μετά από αυτόν τον γύρο επαύξησης, θα το ξανακάνουμε το augment, μόνο που τώρα θα το κάνουμε στο ήδη επαυξημένο αρχείο και όχι στο αρχικό. Χρησιμοποιήθηκε η τεχνική SpecAugment, η οποία κάνει τα εξής:

- Μάσκα συχνότητας, στην οποία καλύπτεται τυχαία μια σειρά διαδοχικών συχνοτήτων προσθέτοντας οριζόντιες ράβδους στο φασματογράφημα.
- Μάσκα χρόνου, στην οποία εκτός αποκλείουμε τυχαία τις χρονικές περιόδους από το φασματογράφημα χρησιμοποιώντας κάθετες γραμμές.

Τώρα που έχουμε ορίσει όλες τις συναρτήσεις μετασχηματισμού πριν από την επεξεργασία, θα ορίσουμε ένα προσαρμοσμένο αντικείμενο Pytorch Dataset. Για να τροφοδοτήσουμε τα δεδομένα μας στο μοντέλο με Pytorch, χρειαζόμαστε δύο αντικείμενα:

- Ένα προσαρμοσμένο αντικείμενο Dataset που χρησιμοποιεί όλο τον ήχο που μετασχηματίζεται για την προ-επεξεργασία ενός αρχείου ήχου και προετοιμάζει ένα στοιχείο δεδομένων κάθε φορά.
- Και ένα ενσωματωμένο αντικείμενο DataLoader που χρησιμοποιεί το αντικείμενο Dataset για την ανάκτηση μεμονωμένων στοιχείων δεδομένων και τα συσκευάζει σε μια δέσμη δεδομένων.

Στην συνέχεια, χρησιμοποιούμε το προσαρμοσμένο μας σύνολο Dataset για να φορτώσουμε τις δυνατότητες και τις ετικέτες από το πλαίσιο δεδομένων Pandas και να τα χωρίσουμε τυχαία σε αναλογία 90:10 σε σύνολα εκπαίδευσης και επικύρωσης. Στη συνέχεια, τα χρησιμοποιούμε για να δημιουργήσουμε τα προγράμματα DataLoaders εκπαίδευσης και επικύρωσης. Ο λόγος που έσπασα τα σύνολα σε αναλογία 90:10 είναι πως πιστεύω ότι μεγάλο σύνολο εκπαίδευσης επιφέρει καλύτερα αποτελέσματα. Όταν ξεκινήσει το στάδιο εκπαίδευσης, ο DataLoader, θα λάβει τυχαία μία παρτίδα χαρακτηριστικών εισόδου που περιέχει τη λίστα με τα ονόματα αρχείων ήχου και θα εκτελέσει τις μετατροπές ήχου προ-επεξεργασίας σε κάθε αρχείο ήχου. Θα λάβει επίσης μια παρτίδα από τις αντίστοιχες ετικέτες στόχου που περιέχουν τα αναγνωριστικά κλάσης. Έτσι θα εξάγει μια δέσμη δεδομένων εκπαίδευσης κάθε φορά, τα οποία μπορούν να τροφοδοτηθούν απευθείας ως είσοδο στο μοντέλο.

Η διαδικασία μετατροπής δεδομένων είναι η εξής. Αρχικά, ο ήχος από το αρχείο φορτώνεται σε έναν Numpy πίνακα σχήματος (no_channels, no_samples). Δεδομένου ότι τα κανάλια και οι ρυθμοί δειγματοληψίας κάθε ήχου είναι διαφορετικοί, οι δύο επόμενοι μετασχηματισμοί επαναπροσδιορίζουν τον ήχο σε ένα τυπικό 8kHz και σε ένα τυπικό 2 καναλιών. Λαμβάνοντας υπόψη ότι ορισμένα κλιπ ήχου μπορεί να είναι περισσότερο ή λιγότερο από 5 δευτερόλεπτα διάρκειας τυποποιούμε επίσης τη διάρκεια του ήχου σε σταθερό μήκος που να ισούται με 5 δευτερόλεπτα. Η αύξηση δεδομένων Time Shift αλλάζει τώρα τυχαία κάθε δείγμα ήχου προς τα εμπρός ή προς τα πίσω. Τα σχήματα είναι αμετάβλητα. Ο ενισχυμένος ήχος μετατρέπεται τώρα σε φασματογράφημα Mel, με αποτέλεσμα ένα σχήμα (no_channels, Mel freq_bands, time_steps). Η αύξηση δεδομένων SpecAugment εφαρμόζει τώρα τυχαία μάσκες χρόνου και συχνότητας στα φασματογράμματα Mel. Τα σχήματα παραμένουν αμετάβλητα. Γνωρίζοντας ότι τα δεδομένα μας αποτελούνται πλέον από εικόνες Spectrogram, δημιουργούμε μια

αρχιτεκτονική ταξινόμησης CNN (Convolutional Neural Network) για την επεξεργασία τους. Η αρχιτεκτονική έχει τέσσερα συνελικτικά μπλοκ που δημιουργούν τους χάρτες χαρακτηριστικών. Αυτά τα δεδομένα στη συνέχεια αναδιαμορφώνονται στη μορφή που χρειαζόμαστε, ώστε να μπορούν να εισαχθούν στο επίπεδο γραμμικού ταξινομητή, το οποίο τελικά εξάγει τις προβλέψεις για τις n κλάσεις με n το συνολικό πλήθος των αγγλικών λέξεων.

Ο περιληπτικός τρόπος λειτουργίας του CNN είναι ο εξής. Αρχικά, μια δέσμη εικόνων εισάγεται στο μοντέλο με σχήμα (batch_sz, no_channels, Mel freq_bands, time_steps). Κάθε επίπεδο CNN εφαρμόζει τα φίλτρα του για να αυξήσει το βάθος της εικόνας, δηλαδή αριθμό καναλιών. Το πλάτος και το ύψος της εικόνας μειώνονται καθώς εφαρμόζονται οι πυρήνες και τα βήματα. Τέλος, αφού περάσουμε από τα τέσσερα επίπεδα CNN, παίρνουμε τους χάρτες των χαρακτηριστικών εξόδου. Αυτό συγκεντρώνεται και ισοπεδώνεται σε συγκεκριμένο σχήμα και στη συνέχεια εισάγεται στο γραμμικό επίπεδο, το οποίο εξάγει μία βαθμολογία πρόβλεψης ανά κλάση.

Στην φάση εξάσκησης, ορίζουμε τις λειτουργίες του optimizer, loss και scheduler για να διαφοροποιήσουν δυναμικά το ρυθμό μάθησης καθώς προχωρά η εξάσκηση, κάτι που συνήθως επιτρέπει τη σύγκλιση της εξάσκησης σε λιγότερες epochs. Εξασκούμε το μοντέλο για αρκετές epochs, επεξεργάζοντας μια δέσμη δεδομένων σε κάθε επανάληψη. Παρακολουθούμε μια απλή μέτρηση ακρίβειας που μετρά το ποσοστό των σωστών προβλέψεων.

Τέλος, στον κώδικα δεν υπάρχει η μέθοδος που θα ζητήσει κάποιο αρχείο για να το ακούσει και να το εμφανίσει σε γραπτή μορφή, και οπότε δεν μπορούμε να είμαστε σίγουροι κατά πόσο αποτελεσματικό είναι το μοντέλο αυτό όταν καλείται να λειτουργήσει εκτός του πεδίου εκμάθησης ή εκπαίδευσης.

2^{ο} θέμα:

Στο ίδιο source2021 συμπιεσμένο φάκελο βρίσκεται και το csv αρχείο με τους ήχους που ζητήθηκαν να αναγνωριστούν. Ήμουν στην ομάδα 150.