FstMuse

A deep learning project

Dionigi Rodriguez , Saad Rahman

13/02/23

INTRODUCTION

Building this project we wanted to experiment and explore an innovative way to handle audio data. We did this by building three models with increasing levels of difficulty and obtaining different levels of success with them.

We were inspired by a paper from Hareesh Bahuleyan (University of Waterloo, ON, Canada) on using the mel spectrograms of songs to extract features and work with it in terms of Neural Network data, by exploiting a number of Deep Learning techniques that are available for image recognition/creation .

MEI SPECTROGRAM

The mel spectrogram of an audio file is a visual representation of the melody of that file in terms of change in the frequencies. To actually compute it we used the short-time Fourier Transformation and then the Mel scale Transformation. Luckily to code all of this just two Tensorflow functions are needed.

```
def toMelSpectroGram(self,mels=128):
    np.random.seed(0)  # Set a specific seed for NumPy
    lb.display.__random_state__ = np.random.RandomState(0)
    self.mSpGram= tfio.audio.melscale(tfio.audio.spectrogram(input=self.rawData,nfft=2048,window=2048,stride=512),mels=mels,rate=self.frequency,fmin=0,fmax=8000)
```

This assures us a visual representation of the audio data and, even more, a representation of the melody of the songs, which is the key feature on which we want to train our models.

DATASET

We used the FMA (free music archive) dataset to train all of the models. It's an open source free music library of 30 or more second clips, it also offers a variety of useful scripts to do feature extraction and such, but we wanted to work on the data ourselves.

That is why in the DataPipeline.py and parseDataset.py files we wrote all of the methods we would need to load, parse, normalize and access our data. In particular the whole pipeline is summarized in the preProcess() function which can be called upon a path where the files are and will take care of all the preprocessing needed to feed the data to the models.

```
def preProcess(folderPath,genres,split=0.2,dims=(1291, 128)):
    nCores=8

    #Load the data
    trackList=multiProcessLoad(folderPath,n=nCores) #Load the data

    #process the data
    trackList=oneEncode(trackList,genres)
    processed=multiProcessTask(process,trackList,n=nCores)

    #split the data
    sP=int(len(processed)*split)
    random.shuffle(processed)
    trainingX=[normalizeDim(dims,elem.mSpGram) for elem in processed[sP:]]
    trainingY=[elem.oneHotLabel for elem in processed[sP:]]
    validationX=[normalizeDim(dims,elem.mSpGram) for elem in processed[:sP]]
    validationY=[elem.oneHotLabel for elem in processed[:sP]]
    data=[trainingX,trainingY,validationX,validationY]
    return data
```

It's important to highlight that we used three different partitions of the dataset to train the different models due to different demands in normalization and number of samples, to better suit the individual needs.

FIRST MODEL

The first model(named ToyModel) that we built and trained, to see if the project could be viable on a larger scale, is a CNN that classifies 30 second clips in three classes: Folk, Rock, Hip-Hop.

```
class toyClassifier:
    def __init__(self,dim,classes):
        self.shapes=dim
        self.loaded=None


        self.model = models.Sequential()
        self.model.add(layers.Conv2D(32, (9, 9), activation='relu', input_shape=(dim[0], dim[1], 1)))#,kernel_regularizer=regularizers.l2(reg)))
        self.model.add(layers.Conv2D(64, (3, 3), activation='relu'))#,kernel_regularizer=regularizers.l2(reg)))
        self.model.add(layers.MaxPooling2D((2, 2)))
        self.model.add(layers.Conv2D(128, (2, 2), activation='relu'))


        self.model.add(layers.Flatten())

        self.model.add(layers.Dense(32, activation='relu'))#,kernel_regularizer=regularizers.l2(reg)))
        self.model.add(layers.Dense(classes, activation='softmax'))

        self.model.compile(optimizer='adam',
                           loss='categorical_crossentropy',
                           metrics=['accuracy'])
        return
```

After a lot of trial and error we arrived at this structure, a simple CNN with three convolutional layers and one dense layer. The first part is composed of two Convolutional layers and a Max Pooling layer, to ensure attention and reduce overfitting. The second part is composed of just one convolutional layer. For all the activation functions we always used "Relu" which is considered best practice when working with images and for the output layer a three dimensional "Softmax" function, to

ensure correct probabilities. Regarding the loss function we used “categorical cross-entropy” as it is best practice for one-hot encoded, multi class, classification problems.

We had initial problem with severe overfitting and after trying different regularization techniques such as l2 regularization and insertion of dropout layers at 0.2 and 0.5 percent values, we managed to overcome this problem by simplifying the model (as seen above) and normalizing the dataset (ensuring the same amount of data for all of the classes).

We trained the model on just under 300 samples for 10 epochs and managed the following accuracy scores:

```
8/8 [==============================] - ETA: 0s - loss: 9.2505e-05 - accuracy: 1.0000
8/8 [==============================] - 54s 7s/step - loss: 9.2505e-05 - accuracy: 1.0000
- val_loss: 1.1722 - val_accuracy: 0.8571
<keras.src.callbacks.History object at 0x00000172160E92D0>

[Done] exited with code=0 in 599.147 seconds
```

so a 100% accuracy on the training data an 85% accuracy on the validation data and an additional 80% accuracy on the test set observed afterwards.

FSTMUSE

Our second model (named fstMuse) is a more complex classifier model, it’s a CNN that uses the same techniques as the previous one to classify 7 major genres (Pop, Jazz, Rock, Folk, Hip-hop, Electronic, Punk).

This model is composed of 5 convolutional layers and one dense layer.

The first part is composed of two Convolutional layers and a Max Pooling layer, just like the previous model.

The second part is composed of two convolutional layers which are more dense for features extraction from the melSpectrogram.

The third part is just one Convolutional layer. For all the activation functions we always used “Relu” which is considered best practice when working with images and for the output layer a three dimensional “Softmax” function, to ensure correct probabilities. Regarding the loss function we used “categorical cross-entropy” as it is best practice for

one-hot encoded, multi class, classification problems.

```
class fstMuse:
    def __init__(self,dim,classes):
        self.shapes=dim
        self.loaded=None
        reg=0
        #print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))
        self.model = models.Sequential()
        self.model.add(layers.Conv2D(32, (9, 9), activation='relu', input_shape=(dim[0], dim[1], 1)))#,ke
        self.model.add(layers.Conv2D(64, (3, 3), activation='relu'))#,kernel_regularizer=regularizers.l2(
        self.model.add(layers.MaxPooling2D((2, 2)))

        self.model.add(layers.Conv2D(64, (9, 9), activation='relu'))
        self.model.add(layers.Conv2D(128, (3, 3), activation='relu'))
        self.model.add(layers.MaxPooling2D((4, 4)))

        #self.model.add(layers.Conv2D(64, (9, 9), activation='relu'))
        #self.model.add(layers.Conv2D(128, (3, 3), activation='relu'))
        #self.model.add(layers.MaxPooling2D((2, 2)))

        #self.model.add(layers.Dropout(0.2))
        self.model.add(layers.Conv2D(128, (2, 2), activation='relu'))


        #self.model.add(layers.MaxPooling2D((2, 2)))


        self.model.add(layers.Flatten())

        self.model.add(layers.Dense(32, activation='relu'))#,kernel_regularizer=regularizers.l2(reg)))
        #self.model.add(layers.Dense(16, activation='relu'))
        self.model.add(layers.Dense(classes, activation='softmax'))

        self.model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
        return
```


We trained the model on 2.400 samples for 10 epochs and managed the following accuracy scores

```
46/46 [==============================] - ETA: 0s - loss: 1.2541 - accuracy: 0.5109
46/46 [==============================] - 912s 20s/step - loss: 1.2541 - accuracy: 0.5109 - val_loss: 1.7108 -
val_accuracy: 0.3540
```

51% on the training set and 35% on the validation 33% accuracy on the test set observed afterwards.

GAN

Building the Gan (named toyGan) was the most ambitious part of this project and it was made to experiment on how this representation of sound data could translate into generative networks. It was particularly challenging because of the huge amount of data and time that would normally take to build and successfully train something like this, nonetheless we tried to make the best with the limited resources available to us.

The first part was building the generative and the discriminator models to then combine.

For the generative part we landed on this structure, which takes as input a random noise and through two upsampling blocks and three convolutional blocks expands and

modifies the random noise leaving us with an image of desired dimensions.

```
    return
def generator(self):
    self.gen= models.Sequential()
    # Takes in random values and reshapes it to 7x7x128
    # Beginnings of a generated image
    self.gen.add(layers.Dense(323*32*128, input_dim=128))
    self.gen.add(layers.LeakyReLU(0.2))
    self.gen.add(layers.Reshape((323,32,128)))

    # Upsampling block 1
    self.gen.add(layers.UpSampling2D())
    self.gen.add(layers.Conv2D(128, 5, padding='same'))
    self.gen.add(layers.LeakyReLU(0.2))


    # Upsampling block 2
    self.gen.add(layers.UpSampling2D())
    self.gen.add(layers.Conv2D(128, 5, padding='same'))
    self.gen.add(layers.LeakyReLU(0.2))

    # Convolutional block 1
    self.gen.add(layers.Conv2D(128, 4, padding='same'))
    self.gen.add(layers.LeakyReLU(0.2))

    # Convolutional block 2
    self.gen.add(layers.Conv2D(128, 4, padding='same'))
    self.gen.add(layers.LeakyReLU(0.2))

    # Conv layer to get to one channel
    self.gen.add(layers.Conv2D(1, 4, padding='same', activation='sigmoid'))


    return "gen built"
```

The general structure of the upsampling block is: Upsampling layer, convolutional layer and a LeakyRelu layer. The convolutional block is just a convolutional layer and a LeakyRelu layer.

Regarding the discriminator we started with the toyModel architecture changing the classes to 0 or 1 but quickly found out that it wasn't yielding good results because of the lack of convergence of the model's losses, in fact the generator one would shrink to 0 and the discriminator would blow up to exponential numbers. We then landed on a more complex architecture which assured convergence of the losses.

```
def discriminator(self):
    self.disc = models.Sequential()

    self.disc.add(layers.Conv2D(32, 5, input_shape = (1291,128,1)))
    self.disc.add(layers.LeakyReLU(0.2))
    self.disc.add(layers.Dropout(0.4))

    self.disc.add(layers.Conv2D(64, 5))
    self.disc.add(layers.LeakyReLU(0.2))
    self.disc.add(layers.Dropout(0.4))

    self.disc.add(layers.Conv2D(128, 5))
    self.disc.add(layers.LeakyReLU(0.2))
    self.disc.add(layers.Dropout(0.4))

    self.disc.add(layers.Conv2D(256, 5))
    self.disc.add(layers.LeakyReLU(0.2))
    self.disc.add(layers.Dropout(0.4))

    self.disc.add(layers.Flatten())
    self.disc.add(layers.Dropout(0.4))
    self.disc.add(layers.Dense(1, activation='sigmoid'))


    return "disc built"
```

This architecture is more complex and is composed of four convolutional block, of one conv layer, one LeakyRelu layer and one dropout layer, to combat overfitting, and an output block which uses a flat layer and an output layer using a “sigmoid” activation function to ensure proper probabilities.

We then had to construct the actual GAN so we built the following training loop which takes in the data, builds the proper labels for them, creates the noise to input the generator and then trains the models by feeding the generated data and the real data into the discriminator.

```
def train_step(self,batch):
    real =batch
    fake= self.gen(tf.random.normal((1,128)),training=False)
    #print(fake)
    #print(fake.shape)
    fake=DP.normalizeDim2(self.dim,fake)
    print(fake.shape)
    #print(type(real)[0][0])
    #print(real.shape)
    #new_shape = (-1, 1292, 128, 1)  # Use -1 to automatically infer the batch size
    #reshapedFake = tf.reshape(fake, new_shape)

    with tf.GradientTape() as Gtape:
        yHreal=self.disc(real[0][0][0],training=True)
        yHfalse=self.disc(fake,training=True)
        yHrf=tf.concat([yHreal,yHfalse],axis=0)

        yRF=tf.concat([tf.zeros_like(yHreal), tf.ones_like(yHfalse)],axis=0)

        realNoise=0.15*tf.random.uniform(tf.shape(yHreal))
        falseNoise= -0.15*tf.random.uniform(tf.shape(yHfalse))
        yRF+= tf.concat([realNoise,falseNoise],axis=0)

        dLoss=self.discLoss(yRF,yHrf)
    Gdisc=Gtape.gradient(dLoss,self.disc.trainable_variables)
    self.discOpt.apply_gradients(zip(Gdisc,self.disc.trainable_variables))

    with tf.GradientTape() as dTape:
        newIm=self.gen(tf.random.normal((1,128)), training=True)
        newIm=DP.normalizeDim2(self.dim,newIm)
        preds= self.disc(newIm,training=False)
        gLoss=self.genLoss(tf.zeros_like(preds),preds)
    Ggen=dTape.gradient(gLoss,self.gen.trainable_variables)
    self.genOpt.apply_gradients(zip(Ggen,self.gen.trainable_variables))

    return {"discLoss": dLoss, "genLoss":gLoss}
```

We trained for 200 epochs on 500 samples of mixed genres, we managed to get the generator loss and the discriminator loss to converge to low values but unfortunately we didn’t manage to generate proper audio clips.

It’s important to notice that the toyGan only produces matrices (that represent the mel spectrogram of an hypothetical song), to actually obtain a sound bite, we first need to apply the inverse of the short-time fourier transform and then choose a frequency to reconstruct the main audio file. All of this is done inside the generate() method of our GAN.

TESTING TOOL

To make things more interesting and actually give a way to test the model in a more “human friendly” way we built a small python based GUI to interact with the models.

To make sure it works it's necessary to download the models from the link in the readme.md in the repository and put them in the "/models" folder.

Then to run in it's sufficient to run the gui.py file, insert a 30 second clip and let the magic happen!

A little rough looking but better than writing a lot of code for every test!

REFERENCES

FMT api/database: https://github.com/mdeff/fma

The paper that inspired us: https://arxiv.org/abs/1804.01149