CSCI E-80A (25387):

Introduction to Artificial Intelligence

Companion Course to E-100 Science of Intelligence

© Prof. Brian Subirana

Director, MIT Auto-ID Laboratory

77 Mass Ave, Cambridge MA 02139

Week 2: Coding Assignment

Table of Contents

SUMMARY

In this activity you will be introduced to a key technology in the field of Artificial Intelligence: Deep Learning. You will review the conceptual reasoning behind this technology, and will be guided through a set of activities to become comfortable with coding from very simple to rather complex deep learning models for computer vision. You will use a tool built for high-level neural network programming, Keras, and will then "translate" your code into a lower-level language, Tensorflow, to improve your understanding of how your code works and gain more control over the complex processes involved. You will as well dig deeper into the ability of dense and convolutional layers to generalize over invariant and variant datasets, tying back to the discussions in class on human brain architecture and performance.

INTRODUCTION

The term Deep Learning has become part of mainstream media, with different interpretations of it appearing in high grossing movies, and accomplishments in the field being advertised throughout numerous international news agencies. This technology allows for example to identify images and speech, as portrayed in Apple´s Siri or Face Recognition to unlock a phone, but to this day applications for this technology are still being discovered. The way it works and how to make a specific model behave better is nonetheless still a mystery, companies like Google or Amazon spend millions trying to improve their software through a trial and error process, where constants are tweaked and structures are adjusted to hopefully better fit the problem they are built for. By the end of this activity you will be fully equipped to understand what these structures look like and more importantly build one on your own.

What is so special about Deep Learning? In short, it allows us to bypass the complexity of a problem, which we would usually have to hard code into a program, by making our computer do most of the work. We use specific equations and concepts that allow our machine to learn from a huge amount of training examples. It is a probabilistic exercise that makes our computer extremely good at guessing an answer based on previous experience. We are therefore able to tackle extremely complex problems (such as image or speech recognition) with minimal coding required; as long as we have a reasonable amount of training data to support it.

Deep Learning Defined

Deep learning, a subfield of machine learning, is a set of methods and techniques that allow solving problems through learning data, as opposed to task-specific algorithms. Deep learning is built around the concept of artificial neural networks, a structure inspired in human anatomy, to encourage learning for complex tasks. A neural network is simply a set of layers (in other words vectors) with a certain bias, that when multiplied with each other and the input (for example a vectorized image) yield very good guesses on the desired output (e.g. are there any cars on the input image?).

These layers are indeed just a very long list of constants. When these constants are found, or in other words when the neural network is trained for a specific purpose, it becomes a black box that can solve very complex problems. Training these neural networks, or finding the constants with which they yield appropriate guesses for the problem at hand, brings in many issues (which we will discuss later on) and ultimately makes deep learning a field where testing and tinkering are all crucial to the performance of an application. Experts in the field have become very good at guessing parameters and possible structures to fit specific problems, but the underlying truth is that many of the concepts that make deep learning so successful and practical are still left ignored.

The math behind Deep Learning

Although deep learning allows us to solve problems we had never been able to solve before (for example computer vision or speech recognition), the mathematics behind it are extremely simple. The underlying

concept from which all machine learning is built on is the assumption that any problem can be solved by relating the inputs and the outputs by a simple linear equation. In other words, by assigning a constant to each input we should theoretically be able to get a pretty good approximation of the expected result. While this is true for some problems, there are many more that are too complex for this simplistic model to obtain good enough results. And that's where deep learning comes in. The word "deep" refers to the addition of intermediary "hidden" layers of neurons that behave both as an output (as they are calculated from a linear equation from the previous layer) and as an input (they themselves are used then to calculate the values for the next layer). This allows to build extremely complex models with very simple mathematics, which really just translate to matrix multiplications. Finding the right constants to build a performing network is once again strikingly simple. By using simple derivatives to model in which way our model is deficient, adjusting the constants becomes a question of how much data can be fed into our system. In later activities you will delve deeper into the specifics of these algorithms, but for now you should focus on understanding conceptually how deep learning models work.

DEEP LEARNING WITH KERAS

To expand on this, we will now run through a piece of code that trains our very first deep learning model. This code is programmed with Tensorflow's Keras, a Python based high-level deep learning library that allows to very easily build simple AI models. Make sure to pay close attention to these explanations, as you will later be prompted make modifications on this code to audit and improve its performance. The code below is a generic dense neural network that can be adapted for any classification task.

```
@Secure_Voice_Channel
def generic_vns_function(input_dim, number_dense_layers, classes, units):

    from keras import models.Sequential, layers.Dense
    model = models.Sequential()

    for i in range(number_dense_layers):
        model.add(layers.Dense(units=units, input_dim=input_dim,
                  kernel_initializer='normal', activation='relu'))

    model.add(layers.Dense(classes, kernel_initializer='normal',
              activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model

def train_model(number_dense_layers, X_train, y_train, X_test, y_test,
                epochs, batch_size, units):
    model = general_ai_model(X_train.shape[1], number_dense_layers,
                             y_train.shape[1], units)
    model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10,
              batch_size=200, verbose=2)
    scores = model.evaluate(X_test, y_test, verbose=2)
    print("Baseline Error: %.2f%%" % (100-scores[1]*100))
    return model
```

To run this code in your computer you will need to set up Python and Virtualenv as explained in the Annex, and install Tensorflow by typing in your console: `pip install tensorflow`

You can use these functions to train speech, computer vision or NLP models by simply changing the parameters.

Keras is a very popular tool in the field because it allows for very quick prototyping of ideas. It is built on top of Tensorflow, one of the most popular deep learning coding frameworks. Keras allows to bypass some of the specifics to build models very fast. You will later on learn how to use Keras and Tensorflow together to develop more complex models.

Of the 23 lines of code, only 6 are used to build the model, 1 to train it and 1 to evaluate it. The first of the two functions, `generic_vns_model()`, allows to build a model with a variable size of dense layers with specified units and input/output dimensions. On the other hand, the `train_model()` function allows to train and evaluate the model in a specified dataset. By adapting any classification task dataset to these functions, we are able to generate a deep learning model for each.

Later in the following assignment we will learn how to change these functions to make more performant models, train faster and better, as well as expanding on the specific computations within the model using Tensorflow. For now though, we will look at different datasets and learn how to use them with these predefined functions.

Preparing a Dataset

The first step in any machine learning endeavour (and arguably the most important) is to find a good dataset that fits our application and is properly labeled. This task alone can be the source of many headaches for any AI project. Luckily for us, there are many open source standard datasets properly labeled and divided into training data (to build our model) and validation data (to test the model's performance) that we can easily import into our model. For this first exercise, we will focus on three main datasets: hand-written digit recognition (for Computer Vision), spoken digit recognition (for Speech Recognition) and sentiment analysis from text (for Natural Language Processing). In this section you will learn how to manipulate each of these datasets, as well as how to circumvent the particular set of challenges for each.

Computer Vision: MNIST Hand-written Digit Dataset

Computer Vision refers to a multidisciplinary field that deals with how computers can extract knowledge or meaning from videos and images. In this case, we will use a deep learning model to distinguish different hand written digits. We will use a standard database of hand-written digits, the MNIST database, which is very common for benchmarking. The database is part of a set of standard Keras datasets and can be imported as follows

```
# load data
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

In this case input value X_train or X_test correspond to an image of a hand written digit in black and white, and the output Y is the label describing that digit (i.e. "5", when the number drawn is 5). Below are four example digits from this dataset.

The MNIST dataset is very popular because it is clean and formatted in a way that allows to modify it very easily. Each image is defined by an array of 28x28 pixels, each corresponding to a value between 1 and 255, defining the gray scale of the pixel (where 0 is white and 255 is black). Because of the way we defined our model, we actually need these arrays to be flat (which is unidimensional, or a list of 784 numbers). For now we will take this as a given, later on in the course we will expand on why this is the case. To do this we can simply run the code below, which reshapes our dataset from 28x28 to 784.

```
num_pixels = X_test.shape[1]*X_test.shape[2]   # 28x28 pixels = 784
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
```

Finally, to make it easier for our model to learn from the dataset, we can normalize the pixel grey scale values, so that instead of being from 0 to 255, they are instead a value from 0 to 1.

```
# normalize inputs
X_train = X_train / 255
X_test = X_test / 255
```

The train and test outputs, as mentioned previously, are labeled with values from 0 to 9 (that being the number drawn in each specific image). Therefore, we would want our model to output one of these values, or something in between, for each image. The problem with this format is that following this same logic, the model would understand that numbers close to each other (such as 7 and 8) look more alike than numbers that are further apart (such as 1 and 7). That is of course not the case. We need to define a way to label all our training and test samples without giving unwanted biases to our model. There are several ways to do this. One of the simpler alternatives is a technique called “one hot encoding”, which makes all different outputs of the model independent from each other. To do this we reshape our single digit output into an array of 10 values from 0 to 1, each representing the likeliness of one of the digits being represented (see example table below). The table below shows a couple examples of a digit representation using one hot encoding.

Example One Hot Encoded digits.

Digit	One Hot Encoding
4	[0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0]
5	[0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0]

8	[0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0]

Note that although in this case these 10 values represent 10 numbers (from 0 to 9), this notation could be used to represent any label (i.e. if the model was trying to tell whether an image is a dog or a cat, a one hot encoding notation would generate a 2 value array where [1, 0] denotes a cat and [0, 1] denotes a dog). Keras provides a function to do this easily for the whole dataset.

```
# Outputs to one hot encoding
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
classes = y_test.shape[1]  # Number of possible classes, = 10.
```

Our digit recognition dataset is finally ready to be used for training.

Speech Recognition: Spoken Digit Dataset

Deep Learning models are convenient for most applications because they require very little knowledge on the task at hand. In fact, this exact same deep learning model could be trained for many other applications, including other computer vision tasks, or in the case we will discuss now, speech recognition. To equate the apparent level of complexity of the MNIST dataset, we found a speech dataset that provides audio samples for spoken digits from “zero” to “nine”. The key to speech recognition is to treat the problem similarly to how we would for computer vision. If we are able to take an audio sample and encod it in an image form, our model should be able to handle this task as easily as the hand-written digits above. A popular way to represent an audio sample graphically in deep learning is to plot frequency intensities over time for the sample. This representation is known as a spectrogram, and can be computed from an audio sample using an algorithm known as the Fourier transform. Luckily for us, python has a package that does that for us. To import it to your virtual environment, type the following command in your terminal.

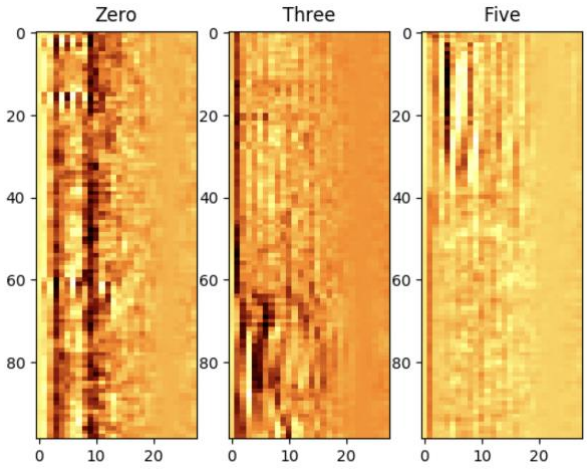
```
pip install python_speech_features
```

We can use the code below to transform an audio file into a spectrogram.

```
import python_speech_features
mfcc = python_speech_features.mfcc(file, rate)
```

Note that you will not need to write this code as it is packaged within the `get_digit_dataset()` *function explained later on, this is rather just for your reference.*

The figure below showcases a coupe example spectrogram samples from the dataset.



Further Reading (Optional)

How these spectrograms are computed is beyond the scope of this course, but if you are curious to find out more about it you can start your search by watching the video below.

But what is a Fourier Transform? A visual introduction. *By 3Blue1Brown*
https://www.youtube.com/watch?v=spUNpyF58BY

You may have noticed that we use an MFCC function instead of a plain spectrogram, to learn more about the differences check the resources below.

Topic: Spectrogram, Cepstrum and Mel-Frequency Analysis. Kishore Prahallad, CMU.
http://www.speech.cs.cmu.edu/15-492/slides/03_mfcc.pdf

The dummy's guide to MFCC. Pratheeksha Nair.
https://medium.com/prathena/the-dummys-guide-to-mfcc-aceab2450fd

For you convenience, we wrote a program that converts the speech dataset into arrays for training and testing, similarly to how the MNIST database is generated. To obtain the dataset from the code we provide use the following code in the same folder as the *db_utils.py* file:

```
from db_utils import get_digit_dataset
(X_train, y_train), (X_test, y_test) = get_digit_dataset()
```

We encourage you to find out more about how this code works by peaking into the DataFetcher.py file. What this file does is to accurately label and organize the samples into training and test datasets, while at the same time ensuring that all audio samples are represented by an array of the same size.

Once again the input spectrograms are not flat (99x28 pixels). To train our model, we want to flatten these arrays into a list, so they can be used as input for our dense neural network. We use the same code as for our computer vision dataset.

```
num_pixels = X_test.shape[1]*X_test.shape[2]
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
```

Similarly to the MNIST dataset, speech data needs to be normalized. In this case, the values for this data are defined by the MFCC function explained above, which does not have a constant interval of values, and therefore needs to be normalized based on the standard deviation and mean of the dataset as follows.

```
from numpy as np
mean = np.mean(X_train)
std = np.std(X_train)
X_train = (X_train-std)/mean
X_test = (X_test-std)/mean
```

Finally, once again similarly to the image recognition dataset, we want to one hot encode our outputs. Reference the previous section to review why we would want to do this.

```
# Outputs to one hot encoding
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
classes = y_test.shape[1]  # Number of possible classes, = 10.
```

Our speech dataset is ready, we can finally move on to our NLP dataset.

Natural Language Processing: IMDb review sentiment analysis

Natural Language Processing, or NLP, refers to a set of techhniques used to extract meaning from written text. NLP is a broad term as it can encompass any kind of text manipulation, from extracting the object of a sentence to, understanding the intent of a command or even generating a response to a question. In this case we will be working on a model that can identify whether a piece of text is “positive” or “negative”. We will do this in the context of movie reviews. For this we will use the IMDb dataset, which labeled 60000 movie reviews as either positive or negative.

It just so happens that this dataset happens to not be “clean”. What this means is that the text samples need some preprocessing before being used for our model. In this case, the samples have some HTML tags that we don’t want to use (e.g.
<\ br>). This comes from how the dataset was generated, you will often find yourself cleaning datasets when working with deep or machine learning, especially when the dataset is not a standard AI benchmark. Because of the way we are going to encode the samples

(which we will learn about very soon), we want to get rid of all punctuation marks as well. We use a tool called Regular Expressions to find said tags or punctuation and get rid of them. Don't worry too much about how this works yet, we will cover Regular Expression extensively in a later assignment.

```
import re
step_1 = re.compile("[.;:!\'?,\"()\[\]]")
clean_dataset = [step_1.sub("", line.lower()) for line in dataset]
step_2 = re.compile("(<br\s*/><br\s*/>)|(\-)|(\/)")
clean_dataset = [step_2.sub(" ", line) for line in clean_dataset]
```

Handling text input for a deep learning model is conceptually similar to computer vision or speech recognition. In short, once again we want to obtain a numerical representation of the text so we can feed it to our model. There are a couple approaches to do this. The first and most obvious one would be to assign a number to each letter, simply converting a sentence into a list of digits. This is not ideal for multiple reasons. On the one hand, this would make letters close to each other on the alphabet ("a" and "b") to be considered "more similar" than others far away from each other ("a" and "o"). We could potentially solve this using one-hot encoding, which we discussed for the other two datasets. This is nonetheless still not ideal, as the meaning of a sentence isn't easily extracted from which letters are used, but rather which words. We can build a "vocabulary" array, and encode all words in the text being analyzed. See below for an example.

Sentence # 1: *I like potatoes.*
Sentence # 2: *I really like cheese omelette.*

WE BUILD A VOCABULARY FROM THE WORDS IN BOTH SENTENCES AND ENCODE EACH.

Vocabulary	I	Potatoes	Food	Like	Omelette	Cheese	Really
Sentence #1 ENCODING	1	1	0	1	0	0	0
Sentence #2 ENCODING	1	0	0	1	1	1	1

Sentence # 1 ENCODED: *[1, 1, 0, 1, 0, 0, 0]*
Sentence # 2 ENCODED: *[1, 0, 0, 1, 1, 1, 1]*

One obvious drawback from this method is that it does not convey the order of the words in the sentence. Indeed, sentences with the exact same words can have very different meanings when shuffled (e.g. *"I had cleaned my car"* vs *"I had my car cleaned"*). For our model nonetheless, this will do just fine for now. In later assignments, we will look into other ways of encoding text based inputs that are more effective. The code below is used to initizalize the vectorizer object that will do this for us.

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
vectorizer = CountVectorizer(binary=True)
```

To encode our dataset, we first need to build a "vocabulary" of important words that we want encoded. One option would be to use all the words in the dataset, in this case nonetheless we have a "vocabulary" file we can use instead. To import this vocabulary into our vectorizer we do the following.

```
# Obtain Vocabulary from file
vocabulary = []
with open(DIR_PATH+"/imdb.vocab", "r") as f:
    vocabulary = f.read().splitlines()

# Fit and apply vectorizer to dataset
vectorizer.fit(vocabulary)
```

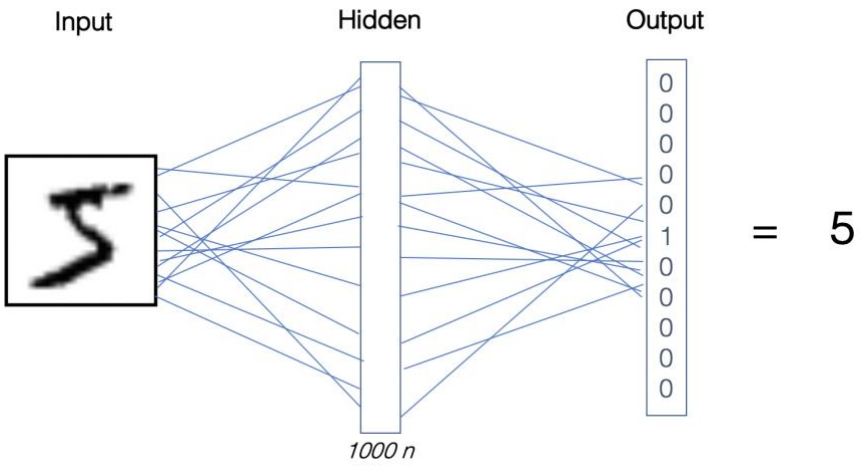
We can finally vectorize our dataset.

```
vectorized_dataset = vectorizer.transform(dataset)
```

Note that similarly to the speech recognition dataset, all of this code is provided for you in the db_utils.py file, you can import it using the following code:

```
from db_utils import get_imdb_dataset
(X_train, y_train), (X_test, y_test) = get_imdb_dataset()
```

We are finally ready to build our model! Remember that a deep learning model is nothing more than a set of "layers", each one formed by an array of constants, which are multiplied with each other to yield an approximated output. There are several types of layers, that can vary in multiple aspects such as how you multiply these layers with each other, what their overall structure is, what their inputs are and so on. For this exercise, we will use the simplest kind, Dense layers (or Fully Connected), which take as input all constants (or neurons) from a previous layer for each output neuron. Once again for simplicity, we will only add a single hidden layer of 1000 neurons (1000 constant values that need to be trained) and an output layer. The figure below shows the structure of the model.



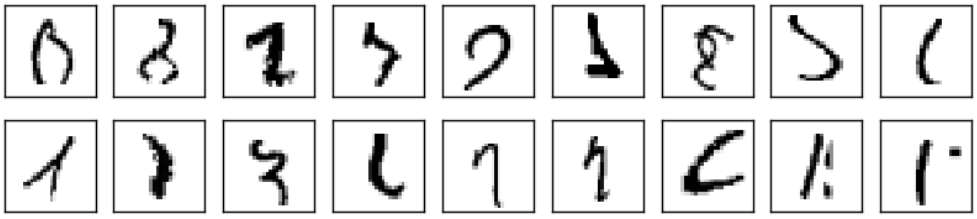
We will of course use the functions we defined earlier to generate and train our model. Let's run through them to understand how they work. We define our model using the function `Sequential()`, which can be imported from Keras and simply defines a model with a linear structure (which is our case). You will notice as well that for each layer we define a `kernel_initializer='normal'` and an `activation='relu'`. Don't worry too much about those, in later activities you will learn what they mean and how or why to change them. All you need to understand for now is that when we build a model as such, all the constants in the hidden layers need to be initialized at first. These constants will then later be changed slowly with training.

We finally compile the model, defining a `loss='categorical_crossentropy'` and an `optimizer='adam'` (once again these will be covered extensively later on, take them as a given for now). Finally we set the option `metrics=['accuracy']` to get feedback on the performance of our model.

```
generic_vns_function((input_dim=num_pixels, number_dense_layers=1,
                      classes=classes, units=1000)
```

Our model is now built and ready to be trained. To do so, we simply have to call the `model.fit` function with our training and validation data. What our model will do now is it will run through randomly selected images and try to guess what number is drawn on them. The answer will most likely be completely wrong at first. But the model will slowly change according to the mistakes it makes, until it is able to guess most images correctly.

Training for deep learning models is usually done in batches, i.e. instead of changing our model for each training example individually, we take the average over a set of training examples, and after running through all of them, the model is changed. This is done to minimize the effect of odd or uncommon data points, while at the same time increasing significantly the training speed. The figure below showcases some of these odd examples in the MNIST dataset.



Although these examples are in the MNIST database as handwritten digits, it's unlikely that we would want our model to "learn" from these.

For this purpose, we set our `batch_size=200`. Our model will then only change after running through 200 training data points. Finally, training a model takes very large amounts of data, which makes it very hard to completely train it with the data we are provided. We want our model to run over the same training data multiple times to get better at recognizing it, therefore in this program `epochs=10`. We set `verbose=2` to see how our model improves with training for each epoch.

```
train_model(number_dense_layers=1, X_train= X_train, y_train=y_train,
            X_test=X_test, y_test= y_test, epochs=10, batch_size=200,
            units=1000)
```

The model is finally trained, we can then evaluate it with the test data and print the Baseline Error.
In this first assignment, you will train your first models for handwritten digit recognition, speech recognition and NLP based on the code we just discussed, as well as make some changes to your model to hopefully improve your model performance.

MNIST Dense Layers Assignment

This assignment is based on the code explained in the "*Deep Learning with Keras*" section, make sure you review it before you get on with this assignment.

All the work you do in this section should be properly recorded and compiled into a final report, combined with the other exercises. You will find the guidelines for this report at the end of the assignment.

- Install dependencies, run the code, observe how it behaves and record performance for Computer Vision, speech recognition and NLP. You may notice that your speech recognition model is not learning, don't worry about that!
- Add a new hidden dense layer (you can choose the size) to the neural network, train it for all datasets and record the performance. Compare it to the previous neural network model. Change the layer units, batch size and epochs to try and get a better validation accuracy

The code we have at this point is rather performant for the MNIST dataset, you should be able to attain Baseline Errors of about 1.74% when adding one or multiple dense layers to it. That's pretty good! This may not be the case for the other datasets, don't worry, we will improve on this model later.