Artificial Neural Networks with Keras

Hunter Anderson
Stephen Shin

June 15, 2023

1 Introduction

This is an exploration into Artificial Neural Networks. The project begins by uncovering the origins of ANNs and how they developed over time. We examine the 'artificial neuron' and observe how this computational structure evolved into more complex forms. Ultimately we arrive at the 'multi-layer perceptron' and similar variants. We also take a peek into how the computations work for each model. Next, we apply three different ML models to the housing data-set used in class. The models are, (1) a decision tree classifier (which we have seen already), (2) a perceptron, and (3) a simple neural network architecture. We then assess the relative performance of these three ML models. To see the techniques we have learned in-action, we move to a more advanced classification problem which requires a larger network to produce an effective solution. Given a data-set of faces and corresponding age-labels, we seek to train a neural network that accurately classifies the age of a person given a picture of their face. This third and final task comprises the majority of work in this project.

2 Motivation

One of the hottest topics in machine learning today is how we can condition computers to think and make decisions for themselves. With AI making its impact heavily known in recent news, there is no better time to introduce ourselves in one of its essential components. Today we are making an attempt to dive into one of the more interesting networks, Artificial Neural Networks (ANNs). ANNs essentially simulate how a human brain takes input and processes information. Mimicking the concept of neurons in the brain, we can make this network "learn" through trial and error using mathematical functions and algorithms. We find ourselves especially focused on image classification, which is why we have decided to work on a model that takes in images of peoples' faces and determine an age range.

3 Research Questions and Answers

1. What is a neural network and how did it come to be? Specifically, what were some of the early proto-types/inspirations for neural networks? What does a modern neural network look like? What are some of the mathematics behind such a model and how do we get a model to fit a data-set?

    - Artificial neurons originated as a way to compute logical propositions, with networks (many neurons composed together) being capable of computing arbitrarily complex logical propositions. Then, for numerical computation, the threshold logic unit (TLU) was created. These were stacked on top of one another to create the perceptron. The perceptron was then layered to create the multi-layer perceptron. Layers then become more complex and include different activation functions, sizes, training strategies, etc. to amount to the current state-of the art.

2. How does can we construct a data-pipeline, starting with raw images and text files (as in the Audience data-set for age classification) and ending in a machine-learning interpret-able format? How do we send this data to the model?

- Image files take up a significant amount of storage, especially since the data-set that we will be using has image resolution (816x816). The pipeline we designed converts the image into a smaller (256x256) resolution and stores each image in a ndarray (from numpy package). This can then be fed one batch at a time to the ML model.

3. How do we (1) create a neural network architecture for image-classification and (2) how do we train it on a data-set?

    - The Tensorflow.Keras library contains all the tools for creating such a neural-network. The basis of these neural networks is the keras.Model() class, which has components that allow you to build nearly any architecture that you could want. The model.compile, model.fit, model.evaluate, and model.predict functions are the ones which we will be using to create, train, and make predictions with the model.

4 Data

4.1 Housing Data (Sample)

The housing data-set is one from CSE163, it can be found here. Briefly, the features are beds, bath, price, year built, sqft, price per sqft, and elevation. These features are all numerical, which is perfect for plugging in to our model (no one-hot encoding required). The label is the city, where there are two categories: San Francisco and New York.

4.2 Face Image Data

Image classification is a very common task in machine learning, and there are all kinds of image classification problems. The problem we will be tackling is the classification of an individual's age (into different age ranges) based on a photo of their face. The data used for this project can be found here. This data-set is a collection of 26,580 images from personal photo-albums released to the public, with 2,284 different subjects. The authors of this project provide a version of the data where faces are aligned, centered, and cropped, which allows us to avoid the problem of facial recognition. There is still a significant amount of noise in the dataset after the modification, and the authors note that this reflects a real-world sample of a random face photo. The classification falls into 8 categories:

Category:	0	1	2	3	4	5	6	7
Age Range:	(0-2)	(4-6)	(8-13)	(15-20)	(25-32)	(38-43)	(48-53)	(60-)

Where I use these numerical values (0-7) to represent the outputs in the machine learning model. More detail about the data is provided in the link. Explicit download instructions are also provided in the linked website, please go there to access the data-set and learn more about it. The files I use are *aligned.tar.gz (1.9G)* for the images and *fold_0_data.txt* - *fold_4_data.txt* for the age-labels corresponding to the images.

5 Method

1. Download and install the software packages required for these tasks. This includes Jupyter notebooks, sklearn, matplotlib, and tensorflow.keras. This also includes the data that has its own section above.
2. Figure out how to use confusion, f1, precision, and various other methods from sklearn.metrics. Just familiarize ourselves with the concepts and function capabilities.
3. Research deeply about neural networks, train them on our desired data sets, and evaluate the model to see how well it is performing

6 Challenge Goals

- New Library - We were required to familiarize ourselves with the keras library and additional functions from sklearn to complement it.
- Machine Learning - Applying neural networks to a well-known dataset and comparing different neural net sizes. Neural networks is not a simple topic. It was a challenge in order to dive into such a complex category of machine learning. Not only do we need to understand it, we have to understand it enough to teach it. A big portion of this project is being able to take many hours of research on this topic and being able to clearly and effectively present it to a larger audience.

7 The Project

A Note on Methods and Results: This project does not lend itself to being categorized into separate methods and results. The methods for this project are non-trivial and it is unlikely that they could be reproduced accurately without seeing the code. In fact, the best way to explain the methods for this project is to show the code directly. Moreover, the code is tightly interlinked with the 'results' section, as producing the code was a part of our research questions. Instead, the results are organized by each component of the project. The components are:

- Understanding neural networks.
- Applying neural networks to a toy-problem (housing data).
- Image classification on face-data.

The topics will explored in the above order. They all lead into one another and the 'results' will be discussed within each section, for each section.

7.1 Neural Networks

7.1.1 Artificial Neurons

Artificial neurons arose from propositional logic, where it was discovered that a neural network can compute any logical proposition. The figure below demonstrates a neural network for a single logical argument. The neuron C requires at least two inputs for activation, where activated means True, and an inactive is False. 1

7.1.2 The Perceptron

The next step up in complexity is a neuron that takes numerical inputs, rather than binary True/False values. We call these *threshold logic units* (TLU). The perceptron is a composition of TLUs. The output of the TLU depends on its weights and the step function applied to it. First, lets examine the weighted sum z that the step function $step(z)$ takes:

$$z = w_0 + \mathbf{x} \cdot \mathbf{w} = w_0 + \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = w_0 + \sum_{i=1}^{n} x_i w_i$$

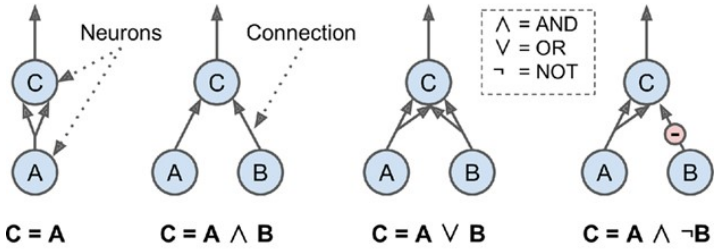

Figure 1: This is an artificial neuron configuration representing basic logic propositions.

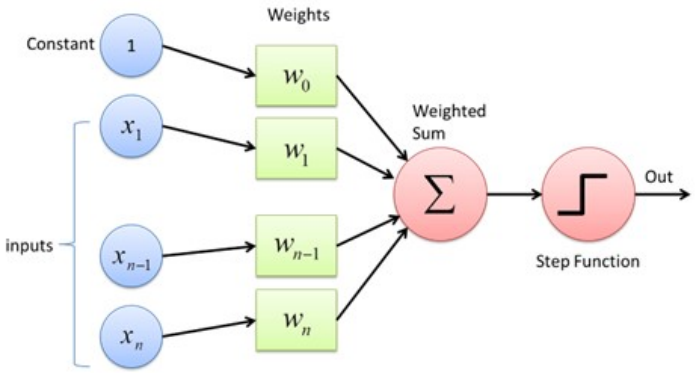

Figure 2: A threshold logic unit, computes a weighted sum of inputs then applies a step function.

This equation corresponds to the weighted sum in Figure 2. To get the output of the TLU, we must apply the step function to this weighted sum. The two most common step functions are the heaviside function and the sign function. For $z \in \mathbb{R}$ the functions are:

$$\text{heaviside}(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

$$\text{sgn}(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ +1 & z > 0 \end{cases}$$

Thus, given inputs $\mathbf{x}$ and weights $\mathbf{w}$, a single threshold logic unit has output $\text{step}(\mathbf{x} \cdot \mathbf{w})$.
The Perceptron is a layer of m TLUs, each with output $step(\mathbf{x} \cdot \mathbf{w}_j)$, so the total output is $step(\mathbf{x} \cdot \mathbf{W})$ where $\mathbf{W}$ is the weight-matrix and $w_{i,j}$ denotes the weight for the ith component (x_i) for the jth TLU, which is at $(row, col) = (i, j)$. Now imagine all of the input data, that is many vectors $\mathbf{x}$, each corresponding to a single instance. Each $\mathbf{x}$ is a row of our data-set, so we can represent the entire dataset as a matrix $\mathbf{X}$. Then the output of our perceptron can be expressed succinctly as

$$\phi(\mathbf{XW} + \mathbf{b}) = \phi\left(\mathbf{X}\begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,m} \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}\right)$$

Where ϕ is an activation function. A visual can be observed in Figure 3.
Thus, given some input data $\mathbf{X}$ we can compute a number... next we need to compute the *right* number. That is, how do we come up with the right weight matrix? Lets start with computing the new weight for a single perceptron, letting the output of $step(\mathbf{x} \cdot \mathbf{w}_j) = y_j$. Then for each j, we can update all weights $w_{i,j}$ according to the perceptron learning rule:

$$w'_{i,j} = w_{i,j} + \eta(y_j - \hat{y_j})x_i$$

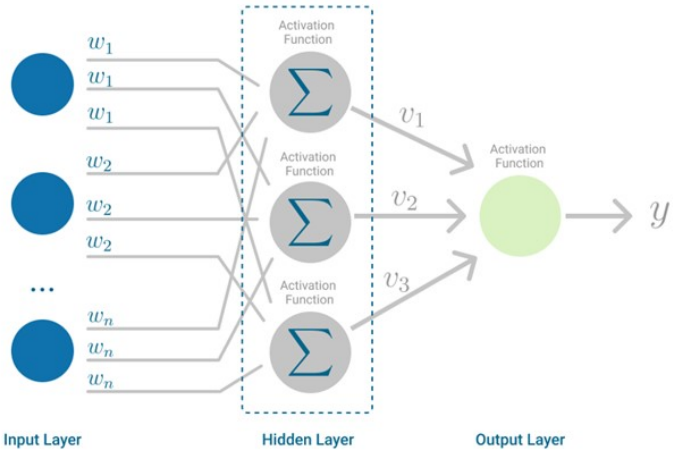

Figure 3: Perceptron with n input neurons and one output, the activation function in our case is the step function.

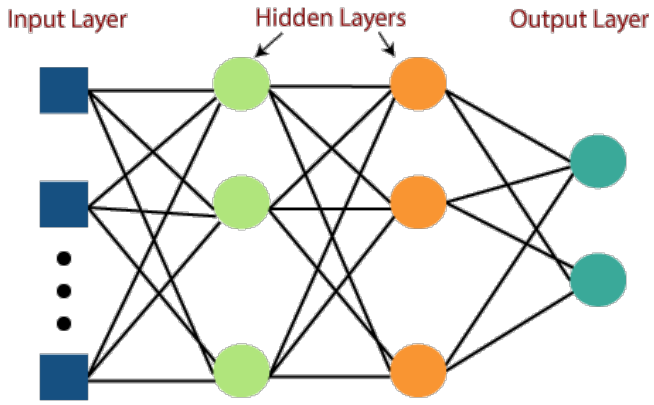

Figure 4: Visualization of multi-layer Perceptron

7.1.3 Multilayer Perceptrons

We can get a heuristic understanding of MLPs via Figure 4. However, the mathematics behind the training of such a network is beyond the scope of this document. The important thing is that we have an intuitive understanding of how this MLP behaves based on the single perceptron. More modern neural network architectures involve different kinds of layers (including convolutions) and different activation functions that we apply to the outputs. We leave out these details for now, but we will come back to them when creating a neural network.

7.2 Neural Networks with Keras: Application on Housing Data

Keras is a package used for creating and training many kinds of artificial neural networks in python. The package allows flexibility to create nearly any kind of neural network that you want. Keras is far too robust for our purposes, so we will not explore the package in any more depth than our project requires. Lets first give a brief overview of how to create, compile, and train a neural network in Keras. We create the model one 'layer' at a time, while specifying activation functions for the neurons in that layer. The following code shows us how to do this for a very simple NN architecture that takes in 7 input categories and returns one output in the form of a sigmoid activation function.

```
inputs = tf.keras.Input(shape=(7,))
x = tf.keras.layers.BatchNormalization()(inputs)
x = tf.keras.layers.Dense(4, activation=tf.nn.relu)(x)
outputs = tf.keras.layers.Dense(1, activation="sigmoid")(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Listing 1: Code to create a model as a sequence of layers

The next step is to compile our model, where we have to specify the parameters optimizer, loss, and metrics. Optimizer tells the model what algorithm it should use for updating its weights during each training cycle. The loss function is a metric of how bad our predictions were (do not have time to go into the details). The metrics function specifies what metric we will use for evaluating the quality of our predictions. Typically we just set this to accuracy, which is a metric common in all machine learning models. Accuracy is simply the percentage of correct predictions our model made. The code to compile the model with these parameters is below:

```
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
```

Listing 2: Code to compile the model

The final step is to train the model on the data. The parameters we must specify are x,y, and epoch. x is the features for the training set. y is the labels for the training set. epoch is an integer representing the number of weight updates/steps to do (each weight update trains on all of x, but many updates are needed to get the model to fit a dataset).

```
model.fit(x=X_train, y=y_train, epochs=35)
```

Listing 3: Code to train the model

With this, we have all the basic tools to create a neural network in Keras. Lets dive into an application.

7.2.1 ML on Housing Data

Lets create a model for the the housing dataset that we worked with in CSE163. This is a classification problem to determine which state a house is in based on 7 numerical parameters. The code below fully creates a Decision Tree, Perceptron, and small Neural Network and trains these models on the housing dataset. It also computes an accuracy score for each model. The comments in the code describe what is happening at each step. The output of the code can be found on the following page. An explanation of the output follows.

```
# Lets compare the performance of a perceptron vs a decision tree vs a neural network
    on the housing data given in lecture
housing_df = pd.read_csv("homes.csv")
X_house = housing_df.loc[:, housing_df.columns != 'city']
y_house = housing_df['city'].copy()
# Let NY and SF be 0 and 1 respectively.
ny_mask = y_house == 'NY'
sf_mask = y_house == 'SF'
y_house.loc[ny_mask] = 0
y_house.loc[sf_mask] = 1
y_house = y_house.astype(int)

# Splitting into train and test sets
```

```
X_house_train_all, X_house_test, y_house_train_all, y_house_test = train_test_split(
    X_house, y_house, test_size=0.8)
# Splitting into train and validation sets
X_house_train, X_house_valid, y_house_train, y_house_valid = train_test_split(
    X_house_train_all, y_house_train_all, test_size=0.8)


# Declaring models
#Tree
tree = DecisionTreeClassifier()
#Perceptron
perceptron = Perceptron()
#Neural Network
inputs = tf.keras.Input(shape=(7,))
x = tf.keras.layers.BatchNormalization()(inputs)
x = tf.keras.layers.Dense(4, activation=tf.nn.relu)(x)
outputs = tf.keras.layers.Dense(1, activation="sigmoid")(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss="binary_crossentropy",
    metrics=["accuracy"],
)

# Fitting models on dataset
tree.fit(X_house_train, y_house_train)
perceptron.fit(X_house_train, y_house_train)
model.fit(x=X_house_train, y=y_house_train, epochs=40, validation_data=(X_house_valid,
     y_house_valid))


predict_tree = tree.predict(X_house_test)
predict_percept = perceptron.predict(X_house_test)
predict_nn = model.predict(X_house_test)
nn_binary_pred = []
for i in predict_nn:
    if i < 0.5:
        nn_binary_pred.append(0)
    else:
        nn_binary_pred.append(1)


print("Accuracy score for Tree:", accuracy_score(y_house_test, predict_tree))
print("Accuracy score for Perceptron:",accuracy_score(y_house_test, predict_percept))
print("Accuracy score for NN:", accuracy_score(y_house_test, nn_binary_pred))
```

Listing 4: Code for making predictions on Housing Data

Output (start)
Epoch 1/40 1/1 [==============================] - 1s 603ms/step - loss: 0.6046 - accuracy: 0.8421 - val_loss: 166.2892 - val_accuracy: 0.5570
Epoch 2/40 1/1 [==============================] - 0s 38ms/step - loss: 0.6026 - accuracy: 0.7895 - val_loss: 118.3111 - val_accuracy: 0.5570
⋮
Epoch 40/40 1/1 [==============================] - 0s 48ms/step - loss: 0.5321 - accuracy: 0.7895 - val_loss: 20.1257 - val_accuracy: 0.5570

Accuracy score for Tree: 0.766497461928934
Accuracy score for Perceptron: 0.5304568527918782
Accuracy score for NN: 0.46446700507614214
Output (end)
Lets start by analyzing the output of this block of code. The first part of the output is training information for the neural network. Our model undergoes 40 epochs of training, that is 40 weight updates for the parameters in the network. Each epoch trains on the entire data-set, but we need to do many weight updates to produce a model that fits our data. Each epoch produces a string of statistics about that training epoch. The first one is loss, which is a way of measuring how 'bad' a

model's prediction was. There are many different kinds of loss functions, the one we specify in our model is 'binary crossentropy'. Next is accuracy, perhaps the most straightforward and useful statistic that we can generate. This is simply $\frac{\#\text{correct predictions}}{\#\text{predictions}}$, or the percentage of correct predictions our model made. Then we have the same statistics for the validation data-set. This is primarily used to avoid overfitting, the thought is that you want to stop training when the accuracy for the validation data-set starts decreasing. Once the last epoch is reached, the model is finished training on the data and we can move on to model validation.

At the end of the output section, we can see the accuracy scores for the different ML models on our test set. However, we quickly notice poor relative performance from the Perceptron and Neural Network. Lets explore why this is. The perceptron likely underperformed because it is nearly identical to a linear regression model, which is not suited for a task with this many features. The Neural Network's poor performance is more interesting as well as more difficult to analyze. Neural networks can fail to converge for countless different reasons. Usually, getting a model to fit the data is a combination of creating the correct structure for the neural network, having good input data, and training for the right amount of time with the right hyper parameters. We have very little experience working with Neural Networks, but it is very likely that there is a way to apply a NN that results in very high accuracy for this problem. We did not have time to explore this any further or train a more accurate model, as all of our time was invested in the face image part of the project. The point of this exercise is simply to show that we can construct, train, and make predictions with a neural network.

file_name	user_id	original_image	face_id	age	gender	x	y	dx	dy	tilt_ang	fiducial_yaw_angle	fiducial_score
114841417@N06	12068804204_085d553238_o.jpg	481	(60, 100)	f	1141	780	975	976	0	0	118	
114841417@N06	12078357226_5fdd9367de_o.jpg	483	(4, 6)	f	1788	341	306	306	-10	0	168	
114841417@N06	12019067874_0e988248af_o.jpg	483	(4, 6)	f	3	183	932	777	-115	0	27	
114841417@N06	12077009614_2490487d2a_o.jpg	484	45	f	258	133	1734	1734	15	0	11	
114841417@N06	12059865494_dace7a1325_o.jpg	485	13	f	1346	294	1001	1001	10	0	102	
114841417@N06	12061744626_215481e333_o.jpg	486	(15, 20)	f	1735	729	287	287	-10	0	89	
114841417@N06	12061744626_215481e333_o.jpg	483	(4, 6)	f	2033	852	242	242	-10	0	82	
114841417@N06	12059875396_f5c3a70550_o.jpg	488	(15, 20)	f	344	1083	752	753	-85	0	103	
114841417@N06	12060036015_e7c827be8d_o.jpg	486	(15, 20)	f	210	231	544	545	-75	0	81	
114841417@N06	12076982073_3d7cfa797b_o.jpg	491	45	m	411	688	1676	1677	-100	0	64	
114841417@N06	12076982073_3d7cfa797b_o.jpg	492	(15, 20)	m	74	221	982	982	-70	0	62	
114841417@N06	12101541316_35e1f0b9a1_o.jpg	493	(15, 20)	m	1216	1391	388	389	-95	0	118	
114841417@N06	12101541316_35e1f0b9a1_o.jpg	490	(8, 12)	m	1370	661	357	357	-85	0	79	
114841417@N06	12095094203_507149205c_o.jpg	490	(8, 12)	m	426	84	688	688	-85	0	35	
114841417@N06	12056392065_2b8d9a757a_o.jpg	489	35	f	227	0	1754	1445	-95	0	61	
114841417@N06	12059434533_676e110ed5_o.jpg	490	(8, 12)	m	0	334	633	626	-90	0	21	
114841417@N06	12076963753_eb16e6ffd6_o.jpg	489	35	f	449	186	828	774	185	0	38	
114841417@N06	12019511266_45314bb42b_o.jpg	485	13	f	639	871	637	638	-85	0	91	
114841417@N06	12059995593_632d96aca7_o.jpg	485	13	f	242	300	412	412	-100	0	88	
114841417@N06	12060970675_f9f9180169_o.jpg	486	(15, 20)	f	1735	729	287	287	-10	0	89	
114841417@N06	12060970675_f9f9180169_o.jpg	483	(4, 6)	f	2033	852	242	242	-10	0	82	
114841417@N06	12059995564_3e347b45bd_o.jpg	483	(4, 6)	f	27	47	1727	1728	-95	0	85	
114841417@N06	12101763726_1d11b44823_o.jpg	490	(8, 12)	m	981	1043	612	612	-10	0	38	
114841417@N06	12101763726_1d11b44823_o.jpg	494	23	f	2218	772	542	542	5	0	89	
114841417@N06	12101763726_1d11b44823_o.jpg	485	13	f	1566	790	529	529	-15	0	104	
114841417@N06	12077824396_e5c732d4fe_o.jpg	491	45	m	1778	865	829	829	-10	0	101	
114841417@N06	12077824396_e5c732d4fe_o.jpg	482	(48, 53)	m	1437	0	644	583	10	0	72	
114841417@N06	12085238366_58ba30f728_o.jpg	496	(60, 100)	m	1931	954	523	523	10	0	35	
114841417@N06	12085238366_58ba30f728_o.jpg	489	35	f	933	967	491	491	-5	0	76	

Figure 5: Text data file format

7.3 Age Classification of Human Faces

In this section we tackle the problem of human age-classification. Given only a photo of an individual's face, we want to create a deep convolutional neural network that can accurately predict the age range of that individual.

7.3.1 Data Aggregation

After downloading the data we find ourselves with two sets of files.

- A folder of folders of face images
- A folder of text files, each text file containing data about a portion of the images

Python gives us many tools that allow us to put this data into a more agreeable format. The target format is an ndarray (a numpy array) for each image and a corresponding label (0-7) that marks the age range of that image. There is one deficiency in the provided text files, which is that the file folders do not have a column name. Thus, for each text file we add 'file_name ' to the beginning so when opening the text file it looks like Figure 5. We can then convert each text file to a DataFrame and concatenate them.

```
# Convert text to CSV
face_df1 = pd.read_csv(img_labels_0_file_path, index_col=False,
                       sep="[ \t]",engine='python')
face_df2 = pd.read_csv(img_labels_1_file_path, index_col=False,
                       sep="[ \t]", engine='python')
face_df3 = pd.read_csv(img_labels_2_file_path, index_col=False,
                       sep="[ \t]", engine='python')
face_df4 = pd.read_csv(img_labels_3_file_path, index_col=False,
                       sep="[ \t]", engine='python')
face_df5 = pd.read_csv(img_labels_4_file_path, index_col=False,
                       sep="[ \t]", engine='python')
frames = [face_df1, face_df2, face_df3, face_df4, face_df5]
face_df = pd.concat(frames, ignore_index=True)
```

Listing 5: Code to convert text files into data-frame

Next we refine the DataFrame to drop null values, include only the columns that we want, and convert the age from the string ordered pair to an integer. For example (4, 6) becomes 4.

```
face_df.drop_duplicates(inplace = True)
df = face_df[['file_name', 'user_id', 'face_id']].copy()
df = df.replace(to_replace='None', value=np.nan).dropna()
df['min_age'] = df['face_id'].apply(lambda s:  int(s.strip(punctuation)))
```

```
df = df[['file_name', 'user_id', 'min_age']].copy()
df.drop_duplicates(inplace=True)
```

Listing 6: Code to clean data

The next section of code is *really* ugly, so I will not include it here. In the next section we go through each image that we downloaded and find the corresponding age from our DataFrame. Then we convert the image to a ndarray and rescale it to (128x128). Next we put the image in an array x and the age category in another array y. After we have done this for all images in a folder, we save these arrays to access them at a later time (cannot store them in memory because we are dealing with multiple gigabytes of information). Then we repeat this process for each folder (the images are spread across 160+ different folders) until all images are converted and saved.

7.3.2 Building The Model

In order to build the model we have to specify all layers, as in the housing problem. However, we are not clever enough to come up with our own convolutional neural network, nor do we have time to experiment with what might work. Thus, we copy the Xception network from the Keras documentation. The only difference between the specified model and our model is the input shape of our images and the number of classes which we are predicting. A visualization of the neural network architecture we are using can found in Figure 8.

```
model = make_model(input_shape=(128, 128, 3), num_classes=8)
```

Listing 7: Using method call to make a Neural Network

The next part of building the model is compiling it. As previously mentioned we must specify optimizer, loss, and metrics. The loss function 'sparse_categorical_crossentropy' computes percentages for each different class in the output and its prediction is simply the class with the highest percentage (the percentages total to 1, this is a probabilistic output). The optimizer is again far too complicated to cover (and I don't understand it). The only metric is the same accuracy that we have seen in all our ML problems.

```
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

Listing 8: Compile the model

With this, the model is correctly configured and ready to be trained.

7.3.3 Training the Model

This part deserves its own paper and months of research. We only had two weeks. The majority of our time in this project was spent training the model, since getting a neural network to converge is not a straightforward task. It involves lots of trial and error and should involve much more experience than we have. But enough complaining, let's start at the beginning:

Configuring Tensorflow for GPU

A Neural Network of this size takes a *very* long time to train on your traditional CPU or at-home laptop. It would have taken multiple weeks to train this model on my laptop, but I was able to reduce training time significantly. Firstly, why does training take so long? Well, simply because there are a lot of computations to do. Lets make some simple observations. It takes around 50 epochs to train the model and each epoch is training on approximately 15,000 images. Each time we train on an image, we have to update 2,738,240 parameters (this is how many parameters our model has). And that is assuming that your model will converge on your first attempt. Take it from me, this kind of computation takes a long time. We did not realize how long this computation would take until we got to this step and started training. Initially, it took an entire minute to train the model on a single image, we quickly realized this would not work and searched for a solution. The first step we took was to reduce image-size from (816x816) to (128x128), which sped up computation significantly, however this was still nowhere near what we needed to finish our model in time. To speed up training time even further I had to figure out how to train the model on a GPU. The graphics computations for

videogames involve a lot of linear algebra, or large quantities of simple mathematical computations. It turns out that training a ML model isn't much different in this regard. However, getting the software and configuring my computer to train on GPU was no easy task. I installed a linux subsystem for windows and reinstalled python and all of the python-packages on Linux. Then I installed CUDA for linux, which allows TensorFlow to do computations on the GPU. After all the sofware was installed I was able to run Jupyter notebooks on the Linux subsystem, but still use the Windows GUI to write and run code. More importantly, the software configuration allowed TensorFlow to run on the GPU. This reduced training time from an estimated two weeks, to just under a day. With the software headache behind us, we are ready to train the model.

Defining an Epoch

Usually one would train on batches, but the computation proved to be too large (I ran out of RAM each time I tried to train on more than a few images at a time), so I had to define my own epoch. I utilized the model.train_on_batch() function to train one image at a time. This is perhaps not as efficient, but it guarantees that my program will not crash due to a MemoryError.

```
################ ONE EPOCH ####################
def one_epoch(np_dir, model):
    curr_metrics = 0
    # Traverse all folders
    for numpy_folder in os.listdir(np_dir):

        # get ndarray paths
        full_np_path = os.path.join(np_dir, numpy_folder)
        x_path = full_np_path + "//features.npy"
        y_path = full_np_path + "//labels.npy"

        # load ndarrays
        x = load(x_path)
        y = load(y_path)

        # train_on_batch for each image
        for i in range(len(x)):
            # Reset metrics at the beginning of each epoch
            if (i == 0 and fldrs == 0):
                curr_metrics = model.train_on_batch(
                    x = numpy.array([x[i]]),
                    y = numpy.array([y[i]]),
                    reset_metrics=True,
                    return_dict=True,
                )
            else:
                curr_metrics = model.train_on_batch(
                    x = numpy.array([x[i]]),
                    y = numpy.array([y[i]]),
                    reset_metrics=False,
                    return_dict=True,
                )

        # save the model
        model.save(model_file_loc + "//mdl.h5")
```

Listing 9: Using train_on_batch() to train one image at a time

Now all that remains is to train the model. While this step didn't take much of my time, it certainly took a lot of the computer's time. My computer was running for about a week straight, trying different configurations for input data (image size, batch size, epochs, etc). Each time I tweaked something slightly, but the model failed to converge 5 or 6 times during training. On the final attempt, I was able to get the Neural Network to converge and achieve an almost 90% accuracy on the data-set. It turns out the step-size was not large enough, causing training to take many more epochs than I thought necessary. The following code runs a lot of epochs, unfortunately I did not have time to configure a validation data-set to know when a good time to stop training might be.

```
np_dir = "/mnt/d/.../Face_image_numpy_frontal_small"
epoch_count = 100
for i in range(epoch_count):
    print("Current Epoch:", i)
```

```
5     one_epoch(np_dir, model)
```

Listing 10: Using function to train model

I terminated the program after 69 epochs, as I feared the model may be overfitting the training data.

Output (start)
Current Epoch: 0
Metrics for epoch: 'loss': 1.3936882019042969, 'accuracy': 0.5458398461341858
⋮
Metrics for epoch: 'loss': 0.3309441804885864, 'accuracy': 0.879825234413147
Current Epoch: 67
Metrics for epoch: 'loss': 0.3168116509914398, 'accuracy': 0.8862154483795166
Current Epoch: 68
Metrics for epoch: 'loss': 0.3162821829319, 'accuracy': 0.8879759907722473
Output (end)
Above is some of the output during training, as you can see the accuracy increased from 54% in the first iteration to 88% in the last iteration. Since we were so worried about creating an accurate model, we used all of the data for training (oops!). We did not have time to fix this mistake and unfortunately we have no real test set to validate our findings. Truly, we are astonished that we even managed to create a model that would converge on the training set.

Figure 6: Predicted age: (15-20) years old

Figure 7: Predicted age: (0-3) years old

7.3.4 Testing the Model

Our only test set is Hunter and his roomate, who took selfies and put them into the model. The accuracy for this test set is 0.50. Hunter and his roomate are both 20 years old.
The following code was used to predict the age of each person.

```
# Test an Image of My Face
import imageio
import cv2
my_img_path = "/mnt/d/.../myFace.jpg"
i1_big = imageio.v2.imread(my_img_path)
i2_small = cv2.resize(i1_big, (128, 128))
prediction = model(numpy.array([i2_small]))
result_category = np.argmax(prediction)
```

Listing 11: Using function to train model

With a prediction as erroneous as the one made for Hunter's roomate, it is not likely that this model is very accurate. However, this does go to show a proof of concept. With more time, data, and experience training neural networks, we think it is likely that one could devise a model that predicts an individual's age range with $\geq .80$ accuracy.

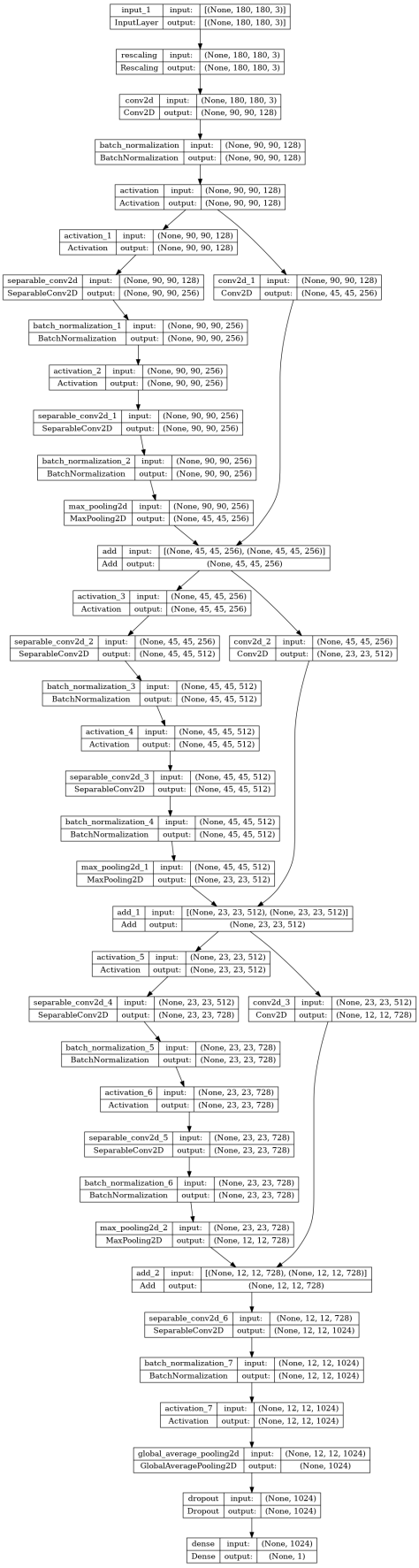

Figure 8: Visualization of the Xception network used for image classification

8 Impact and Limitations

8.1 Limitations

As mentioned in section 7.3, time severely limited the quality of a model that we were able to make. The model we created should not be used outside of test purposes. The fact that no validation set or test set was used on our model means that we have absolutely no way of knowing how accurate the model will be on data that it hasn't seen before. Moreover, there are likely biases in the data-set that will become more apparent when testing a model created from the data. No data-set is perfect, and while this face-image data-set is incredibly noisy and diverse, there are bound to be some hidden biases or underrepresented groups of people for which age classification will not be correct. The performance of any model created from this data-set is limited by the quality of the data-set itself.

The potential impacts for our results are limitless. Like we've mentioned, AI has been making its way into our everyday life, and there is a lot to improve on. The opportunities created from these networks are endless, and it all comes down to understanding the models behind them. When it comes to the pure concept of neural networks and this type of machine learning, we believe there is no room for any bias that may impact our results. However, any projects that may stem from conclusions that we've made in this report may very easily lead to possible bias and reinforcing stereotypes. An example that could help put this into perspective is an instance mentioned during class. While we have been processing images of faces to classify an age group, controversies arise when a model is trained to identify possible criminals or terrorists. When training a model to identify possible negative connotations based on looks and features, its extremely important to understand the type of data that is fed to that model. When given data that is clearly biased, the model is then trained and condition to adhere to that bias. A limitation that these results have is that they cannot look at the larger picture past the data given. Here underlies a very dangerous implication of how neural networks can be used when in the context of public policy and criminal justice findings. We believe that the conclusions made from this project should be used to open a door to further innovations towards neural networks, instead of promoting these models as a fail-proof system for classifications.

9 Work Plan Evaluation

We set up a schedule with specific dates and times for when to call over zoom. We generally had an objective for each zoom call, such as manipulating the data set or learning specific package. Our proposed work plan seemed to be somewhat accurate through most of the project. We ultimately ran into issues processing data sets and converting them into NumPy arrays, as well as training our model. This was a task that took longer than expected, and we should have allocated more time for this section.

10 Testing

Testing that our methods worked happened in small chunks of code within a Jupyter notebook. We tested all of our data-frames and numpy images simply by printing and comparing them. We tested many of our computations by doing small computations and ensuring everything worked on a small scale before using all of our data. Here is one code snippet that tests whether we can find the age for a given image (when aggregating data in an ML-usable form)

```
# TESTING ----------------- that functions work
refined_df = df.loc[df['file_name'] == "100003415@N08"]
user_id_loc = refined_df.loc[refined_df['user_id'] == "9523333835_c7887c3fde_o.jpg"]
print(type(user_id_loc['min_age']))
z = user_id_loc['min_age'].to_string(index=False)
print(z)
user_id_loc['min_age']

print("9523333835_c7887c3fde_o.jpg" in set(refined_df['user_id']))
```

Listing 12: Code used to find age of an image

After testing it on a few random images, we used the same methods to find the corresponding age of an image for every single image in our dataset. Of course, we still ran into errors along the way but we had to fix them as they popped up. This project doesn't lend itself well to 'testing' because there are no methods with explicit specification and expected behavior. It's more of a - cobble together functions and try things until you get the output that you want - kind of project.

11 Collaboration

The only contributing members of the following project are Hunter Anderson and Stephen Shin.
We would like to credit the authors of the data sets that we've used.

1. Housing Data - Stephanie Yee and Tony Chu
2. Face Image Data - Open University of Israel and Audience

We would also like to thank the authors of the Keras API for being so thorough in their specification. Finally, we would like to thank modern search engines for allowing quick and easy access to solutions for common programming predicaments.