An Introduction to Artificial Neural Networks

By Shabeel Kandi, Narendra Bhalla, Akeel Ahamed

Contents

Hello World 3

Introduction to Neural Networks 3

Background and development 3

The Model 4

Supervised learning Model 5

Supervised Learning Model with Example 6

Logistic Regression 6

Loss (error) function: 8

Cost function 8

Image Recognition Example **Error! Bookmark not defined.**

Applications **Error! Bookmark not defined.**

Limitations: How the Artificial differs from the Biological 13

Conclusion 14

References 14

Hello World

In this brief report, we hope to give you an intro into the world of Artificial Neural Networks(ANNs) through a supervised learning framework (if you don't know what Supervised learning is yet, not to worry). Thereafter, we hope to apply this framework to show you how we can correctly identify pictures of cats using a logistic regression algorithm, which is used in classification problems under supervised learning.

Introduction to Neural Networks

Artificial Neural Networks are computing systems inspired by the biological neural networks, that process information the same way that biological nervous systems such as our brain does. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems.

An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. ANNs progressively improve their performance by adjusting parameters in its layers.

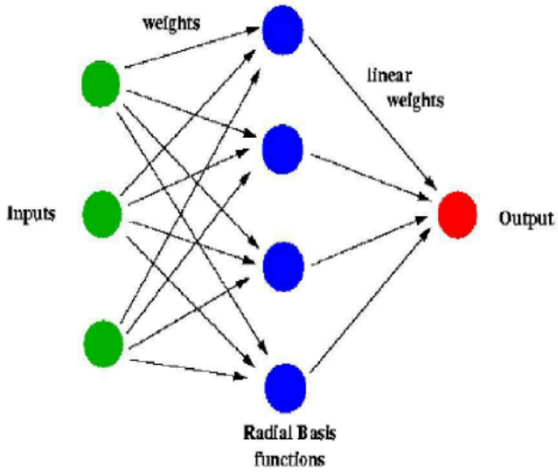


Figure 1: Block Diagram of Artificial Neural Network

Background and development

The field of neural networks is simultaneously the oldest and the newest in the discipline of machine learning.

In 1943, McCulloch-Pitts produced a model of the neuron that is still used today in artificial neural networking. This was incorporated into the model of a perceptron (an algorithm to decide if an input belongs to a specific class or not) put forward by Rosenblatt (1957), but his model lacked applicability.

In 1960, **ADALINE** (Adaptive Linear Element) was developed to recognize binary patterns so that it could predict the next bit, after reading streaming bits from a phone line. **MADALINE** (Many Adaline) was the first neural network applied to a real-world problem, using an adaptive filter that eliminates echoes on phone lines (Clabaugh, C et al., 2000). Still, these algorithms worked similar to linear regression, and could not establish their relevance in the field of AI. Things were going nowhere, which is what Marvin Minsky and Seymore Papert expressed through their 1969 book *Perceptrons*, which was used to discredit ANN research.

The Backpropagation algorithm was made popular in 1986 by David Rumelhart, Geoffrey Hinton, and Ronald Williams. Although the method was discovered before, their publication stood out because the idea is stated concisely and clearly, as stated by Yann Lecun in a retrospective (Widrow, B. & Lehr, M., 1990). As a student of Machine Learning it is clearly visible that their description is essentially identical to the way the concept is still explained in textbooks and AI classes. Thus, we will attempt to explain how it works in this report, and, making use of these ideas, show how a supervised learning algorithm can be used to solve a simple logistic regression problem.
First, let us look at the basic model of an ANN.

The Model

Single Neural Network

To describe the model of a neural network, we start with a simple example of a Single Neural Network. Given data about the size of houses on the real estate market, we can fit a function that will predict their price. The input, which is the size of the house (x), is passed through the "neuron" which implements the ReLU function (Rectified Linear Unit defined as max{0, x}) and spits out the output (house price).

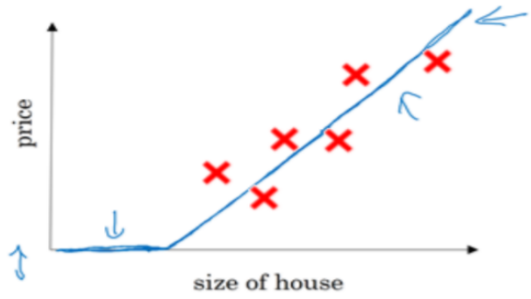


Figure 2: Housing Price Prediction

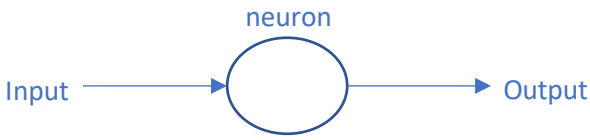


Figure 2a. Single Neural Network

Multiple Neural Network

The price of a house can be affected by other features such as size, number of bedrooms, zip code and wealth. The role of a neural network is to predict the price(y) from the inputs(x) we feed into it. The hidden units in between will be automatically generated.

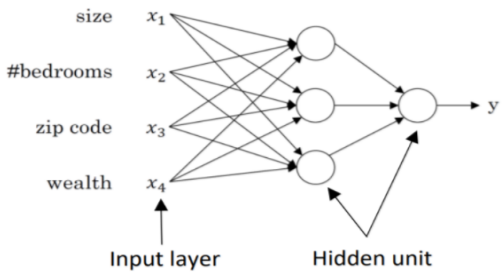
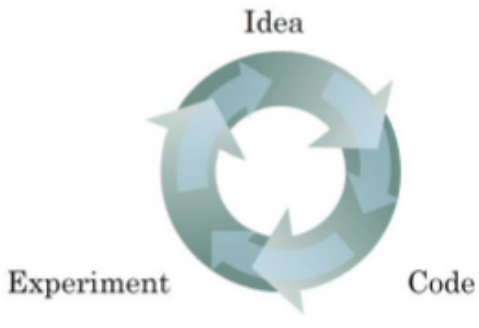


Figure 3: Housing price prediction with multiple inputs

Training a Neural Network

The process of training a neural network is iterative:



- We first have an idea that we would like to implement
- We try it out using code.
- The experiment tells us how well our neural network does and based on how it performed, we revisit the idea and try to improve it.

This process can take a lot of time and affect our productivity. Thus, faster computing helps to iterate and improve our new algorithm. We will now apply ANNs to solve a supervised learning problem.

Figure 4: Training Model

Supervised learning Model

Supervised learning is where we have a dataset and already know what our correct output should look like and we use an algorithm to learn the mapping function from the input to the output. i.e. Y = f(x).

It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process of her students. The teacher already knows the correct answer, and only tells the students how to reach it by means of an algorithm. The student then iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance.

Supervised learning problems are categorized into "**regression**" and "**classification**" problems. In a regression problem, we are trying to predict results with a **continuous** output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results with a **discrete** output. In other words, we are trying to map input variables into discrete categories.

Input(x)	Output(y)	Application
Home features	Price	Real Estate
Ad, user info	Click on ad? (0/1)	Online Advertising
Image	Object (1….1000)	Photo tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Image, Radar info	Position of other cars	Autonomous driving

Table 1: Examples of supervised learning along with their applications

Supervised Learning Model with Example

Our project revolves around a binary classification example where the result is a discrete value output (whether the image is a 'cat' or 'non-cat').

The goal is to train a classifier where the input is an image, represented by a feature vector, and predicts whether the corresponding label is 1 or 0. In this case, it predicts whether the input is a "cat" image (1) or a "non-cat" image (0).

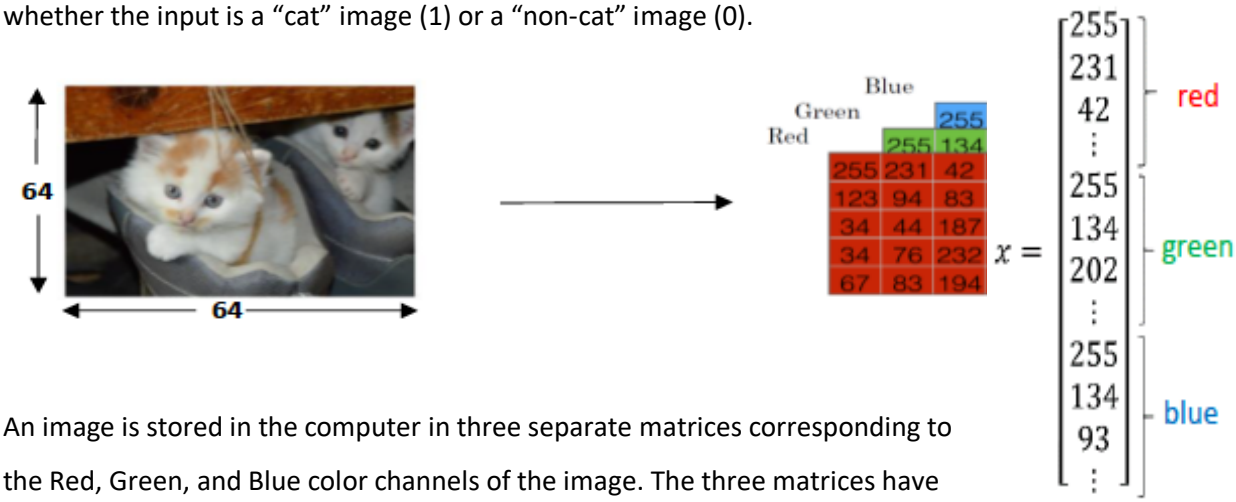


Figure 5: Feature vector x(for **one** image)

An image is stored in the computer in three separate matrices corresponding to the Red, Green, and Blue color channels of the image. The three matrices have the same size as the image; the resolution of the cat image is 64 X 64 pixels and thus, the three matrices (RGB) are 64 X 64 each.

The value in each cell in the matrix represents the pixel intensity. These values are then transformed into a **feature vector x,** of n-dimensions (where n is the number of pixel intensity values represented by the RGB matrices).

Logistic Regression

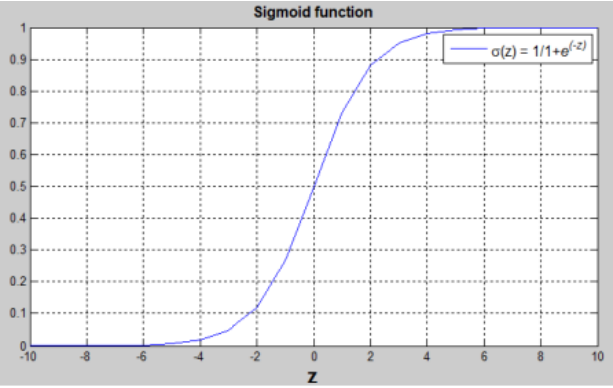
Logistic regression, in simple terms, is a learning algorithm for binary classification problems. In this case specifically, it's goal is to predict the outcome of a training image i.e. if we want to classify pictures of cats, a cat picture will have the value of 1 and 0 otherwise. In Mathematical terms:

$$\text{Given } x, \hat{y} = P(y = 1|x), where\ 0 \leq \hat{y} \leq 1$$

The parameters used in Logistic regression are:

- The input feature vectors: $x \in \mathbb{R}^{n_x}$, where n_x is the number of features (64 X 64 X 3 in this case)
- The training label: y ∈ 0,1
- The weights: $w \in \mathbb{R}^{n_x}$, where n_x is the number of features
- The bias: b ∈ $\mathbb{R}$

- The output: $\hat{y} = \sigma(w^t x + b)$
- Sigmoid function: s = $\sigma(w^t x + b) = \sigma(z) = \frac{1}{1+e^{-z}}$



$(w^t x + b)$ is a linear function (ax + b). However, since this can be anywhere in the range of negative infinity to positive infinity, we need a way to restrict the values to be in the range we want to work with [0,1]. Thus, the sigmoid function is used.

Observe that:

- • If z is a large positive number, then $\sigma(z)$ = 1
- • If z is small or large negative number, then $\sigma(z)$ = 0
- • If z = 0, then $\sigma(z)$ = 0.5

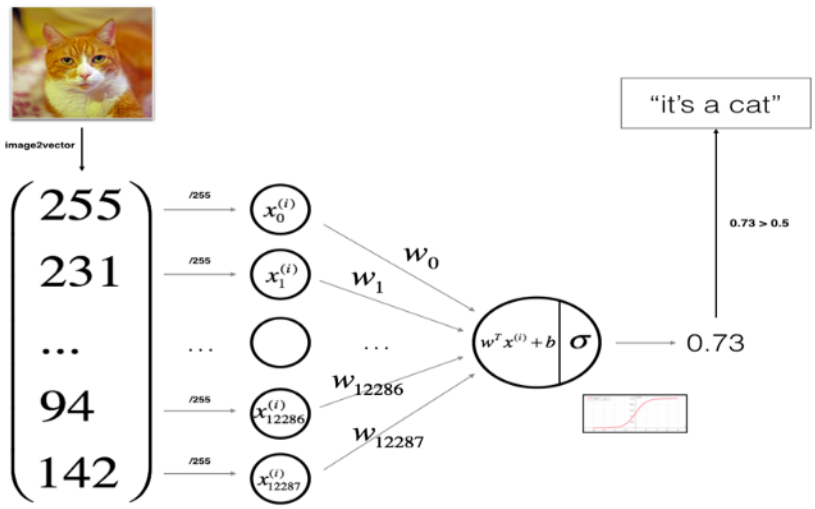
Recap:

$x^{(i)}$ the i-th training example

$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$

Given $\{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\}$, *we want* $\hat{y}^{(i)} \approx y^{(i)}$

Let us now look at the General Architecture of the learning Algorithm:



The computations in our Logistic Regression model can be thought of as a one-layer Neural Network. They are organised as a series of **forward propagation steps** where we compute the output of the neural network for each training example, followed by **backward propagation steps** which compute the gradients/ derivatives by which to update parameters w,b(more on this later). We do this m times (where m represents the number of training examples we are using) and finally, after the last iteration, we will end up with the optimal values of w, b which we use on the **test set** images to compute the output i.e. whether they are "cats" (1) or "non-cats" (0).

Loss (error) function:

Forward Propagation also goes one step further and computes the loss function which measures the discrepancy between the prediction ($\hat{y}^i$) and the desired output (y^i). In other words, the loss function computes the error for a **single** training example image. In the above example, since the image is a cat, $y^1 = 1$ and $\hat{y}^1 = 0.73$, thus the error for this image is 1-0.73 = 0.27

$L(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$ Not used as the function is **not convex**

$L(\hat{y}^{(i)}, y^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$ The loss function used is **Convex**

- If $y^{(i)} = 1$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(\hat{y}^{(i)})$ where $\log(\hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 1
- If $y^{(i)} = 0$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(1 - \hat{y}^{(i)})$ where $\log(1 - \hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 0

Cost function

Backward Propagation updates the parameters w,b using a fancy algorithm called the Gradient Descent algorithm(its workings are explained in the following section). In order to do this, we first need to define the cost function.

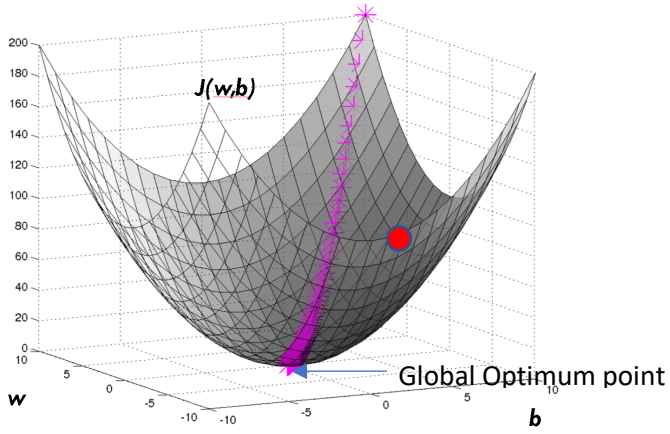
The cost function is the average of the loss function of the entire training set.

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[(y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$$

In order to find the optimal value of (w,b) that minimises the cost, let us now look at the Gradient Descent Algorithm.

Gradient Descent Algorithm

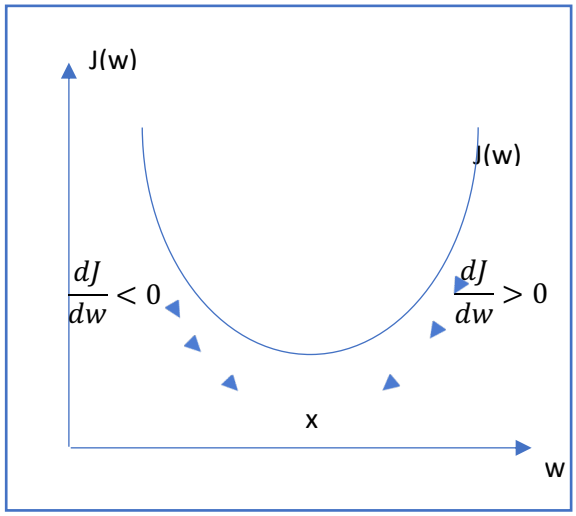
- For gradient descent, we first initialise the parameter values of w, b to some random initial value (denoted by the red dot for example).
  Note: We usually initialise at zero when coding.
- Since the cost function is convex (by definition), no matter where we initialise, we will end up at roughly the same optimal value.
- What one step of gradient descent does is it starts at that initial point and **takes a step in the steepest downhill direction**. This means that we can even end up at the "global optimum" point (the point which gives optimal w,b) even after just one step.



To better understand the gradient descent algorithm, let us reduce the system to be two dimensional :

Suppose we are trying to minimize the cost function J(w) with respect to w; thus, we want to find the optimal w.

Note the formula for w: **w=** $\boldsymbol{w - \alpha dw}$ (where dw is the gradient of the cost function with respect to w, α *is the learning rate*).



- If we are initialising from the **right** of the minimum point(x), we get a **positive gradient** and thus the new value of w will be smaller . Thus, the system will converge to the global optimum from the right.
- If we are initialising from the **left** of the minimum point , we get a **negative gradient** and thus the new value of w will also be smaller . Thus, the system will converge to the global optimum from the left.

Once we have performed several iterations of forward and backward propagation steps e.g. 100 or 1000 , we will be sure to end up with the parameters w,b that minimize the cost function. We then

use those minimized values on our test set images to see how well our model performed. We will show you three images that we randomly selected from the internet as part of our test set:

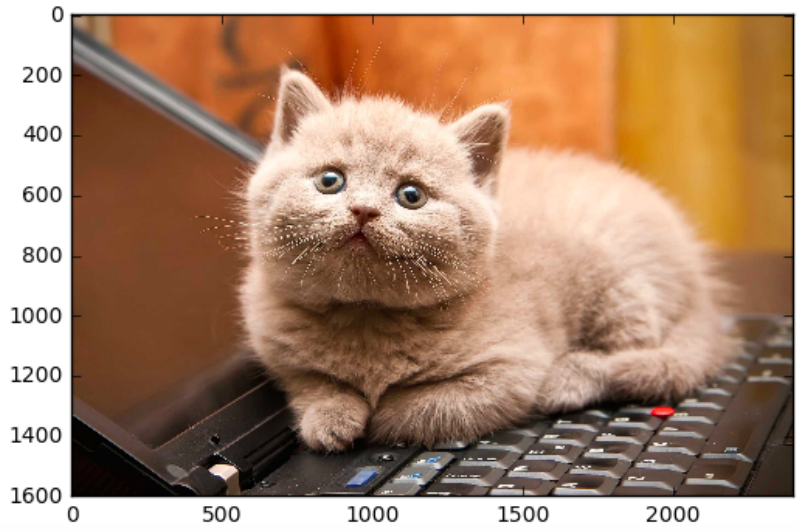
Image 1

```
# Insert IMAGE NAME
my_image = "laptop_cat.jpg"   # change this to the name of our image file

# We preprocess the image to fit our algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a \"" +
      classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
y = 1.0, your algorithm predicts a "cat" picture.
```



As you can see, our algorithm correctly predicted a “cat” image. Let us test it on an image that is not a cat.

Image 2

```
# Insert IMAGE NAME
my_image = "sir_trump.jpg"   # change this to the name of our image file

# We preprocess the image to fit our algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a \"" +
      classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
y = 0.0, your algorithm predicts a "non-cat" picture.
```

Well, Mr. Trump certainly is not a cat! How about something a little more confusing:

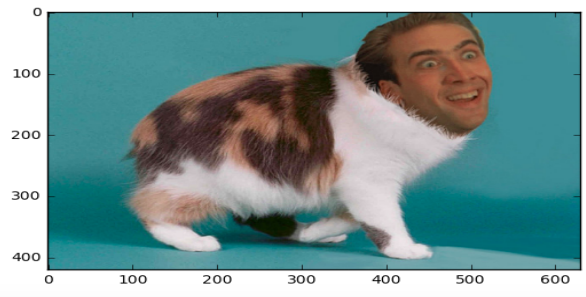
Image 3

```
# Insert IMAGE NAME
my_image = "cat_cage.jpg"   # change this to the name of our image file

# We preprocess the image to fit our algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a \"" +
      classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
y = 0.0, your algorithm predicts a "non-cat" picture.
```



Nice! Our algorithm wasn't confused by thinking that Mr Cage ironically transformed himself into a cat.

Note: Even though these images were identified correctly, our model is only 70% accurate on the test set.

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, learning_rate = 0.005,
          print_cost = False)
```

```
{'w': array([[ 0.19033591],
       [ 0.12259159]]), 'b': 1.9253598300845747}
{'w': array([[ 0.19033591],
       [ 0.12259159]]), 'b': 1.9253598300845747}
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

This means that there are instances where it could go wrong. Some ways that it could be improved are by using a linear activation function(such as the ReLU), increasing the number of iterations, reducing overfitting (where our network adapted too much to the training data, to the point where it now performs poorly for data that is not already known) etc.

Other ANN Architectures

In this section, we'll talk about some basic architectures in use today.

1. Convolutional Neural Networks: They are a class of deep, feed-forward artificial neural networks, with their most common applications in image recognition (invented 1989, by LeCun). In a CNN, neurons focus on parts or particular aspects of an image and then info is pooled to the next layer and the next until it has the image and then recognizes it). CNNs use a different kind of multilayer perceptrons designed to require minimal processing. They are also used for natural language processing, drug discovery and also playing checkers!

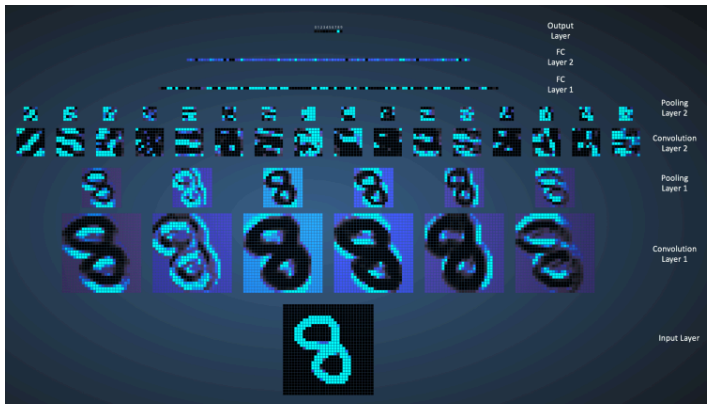


Figure 5: CNN Working

2. Recurrent Neural Networks (RNNs): This is a class of artificial neural network where the output later is reconnected to the neural network, thus forming a directed cycle. As a result, it exhibits dynamic temporal behaviour. Unlike feedforward neural networks, RNNs can make use of their internal memory to process arbitrary sequences of inputs. This

Figure 6: RNN Fun Applications. This RNN was trained on Romantic Novels, and asked for its opinion on various pictures.

makes them applicable to unsegmented or connected tasks such as handwriting or speech recognition.

3. Generative Adversarial Networks (GANs):

These are a class of artificial intelligence algorithms used in unsupervised machine learning, implemented by a system of two neural networks contesting with each other in a zero-sum game framework: the discriminator and the generator. The generator looks at real data and tries to replicate it, and the discriminator compares this information with the real data and decides whether it's a fake or an original. This process continues until the generator becomes a master forger and can fool the discriminator. This has applications in Text to Image synthesis, Improving image resolution & molecule generation among many others. Discovered by Ian Goodfellow et al. in 2014, there is a lot of research going on using this technique due to its vast applicability in AI.

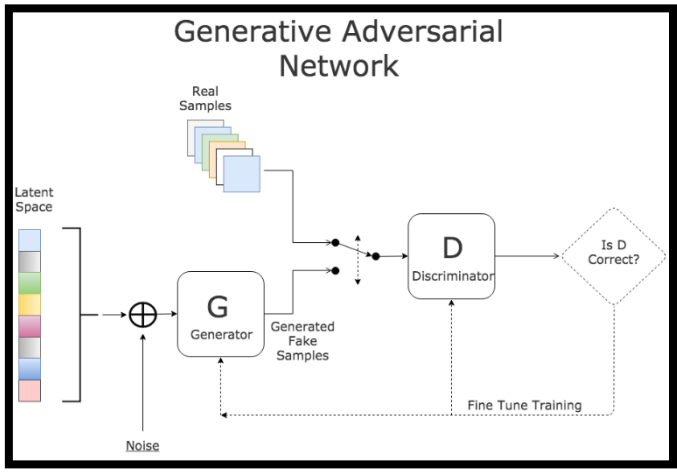


Figure 7: Basic GAN

Limitations: How the Artificial differs from the Biological

Artificial Neural Networks (ANNs) are mathematical models inspired by the workings of the brain, and resemble biological neural networks to varying extents. There have always been two aims of Neural Network research: The first goal, understanding the brain, is associated with neuroscience, and is primarily basic research. The second goal, emulation of the brain, is linked with computer science and artificial intelligence, and is primarily applied science. The technological usefulness of neural networks aligns reasonably well with the same spectrum - AI researchers and computer scientists use the highly simplified artificial neural networks (brain emulating), but have little use for complex spiking neural networks (brain understanding). Hence, there are many types of ANNs and there are various ways of comparing them to Biological Neural Networks (BNNs), but we'll focus on how they are different, to set apart the technology from the reality.

1. ANNs work **synchronously** (at the same time) while BNNs don't. This is a difficult property to model, and hence hasn't come into practice just yet. It may also be one of the major reasons why ANNs and BNNs function so differently in other respects as well.

2. Single biological neurons are *slow*, but standard neurons in ANNs are **much faster**. Thus, many ANNs can do tasks which are too complex for humans, in just mere seconds. On the flip side, there are many things that a neuron in the brain can do that ANNs don't even possess the capability to handle (for example, interpreting a complex visual scene).
3. BNNs have very *complicated topologies*, while ANNs in most cases are organized in **simple structures**. In addition, the largest ANNs only have **several million parameters**, while BNNs possess *many billions* of them.
4. BNNs use *very little power* compared to ANNs, which **require powerful processors** to work effectively.
5. BNNs *don't usually start or stop learning* and are continuously learning for very short time periods. ANNs on the other hand, have different training and evaluation phases, i.e. they are **either learning or being tested**.
6. Finally, ANNs are **highly specialized**. There are different ANNs proficient at different tasks, while we can do all of them and more using our BNNs.

It's worth pointing out that all models are approximations of reality. As George E. P. Box has said: "All models are wrong, but some are useful." ANNs are proof that this statement holds true.

Conclusion

Artificial Intelligence emerged as a field in 1956, and is growing today with every passing day. Andrew Ng, one of the pioneers of Deep Learning, has said: "AI is like the new electricity. At the rate at which it is growing, we will need as many AI scientists as we currently have electricians." Artificial Neural Networks are responsible for putting the I in AI, and are hence central to the growth of this field. The popular current applications of this field include self-driving cars, facial recognition & virtual personal assistants. However, the future possibilities are practically endless, thanks to the AI effect, which states: "AI is everything that hasn't been done yet." And we all know, there will always be a lot of that.

References

- https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html
- https://wiki.eecs.yorku.ca/course_archive/2011-12/F/4403/_media/ann_-_math_model.pdf

- http://www.marekrei.com/blog/neural-networks-part-1-background/
- Neural Networks & Deep Learning Course by Andrew Ng, on Coursera: https://www.coursera.org/learn/neural-networks-deep-learning.
- McCulloch, S. & Pitts, W. (1943). “A logical calculus of the ideas immanent in nervous activity.” *The Bulletin of Mathematical Biophysics* 5.4: 115-133.
- Rosenblatt, F. (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review* 65.6 : 386.
- Supervised & Unsupervised Machine Learning Algorithms: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/
- Clabaugh, C et al. (2000) Neural Networks – History [online] available at https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/index.html (Accessed November 10, 2017)
- Kurenkov, A (2015). A Brief History of Neural Networks and Deep Learning : Parts 1 to 4 [online] available at http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning/ (Accessed November 09, 2017)
- Widrow, B., & Lehr, M. (1990). *30 years of adaptive neural networks: perceptron, madaline, and backpropagation*. Proceedings of the IEEE.
- Perez, C. (2017). Why we should be Deeply Suspicious of Backpropagation [online] available at https://medium.com/intuitionmachine/the-deeply-suspicious-nature-of-backpropagation-9bed5e2b085e(Accessed November 10, 2017)
- Lavrenko, V. (2015). Neural Networks 1: a 3-minute history [online] available at https://www.youtube.com/watch?v=jZYz0EUPYBI (Accessed November 27, 2017)
- Schmidhuber, J. et al. (2009). "A Novel Connectionist System for Improved Unconstrained Handwriting Recognition" (PDF). IEEE Transactions on Pattern Analysis and Machine Intelligence. Available at http://people.idsia.ch/~juergen/tpami_2008.pdf (Accessed November 09, 2017)
- Various Authors (no date). Recurrent Neural Networks [online] available at https://en.wikipedia.org/wiki/Recurrent_neural_network (Accessed November 09, 2017)
- Various Contributors (2013). Artificial Neural Networks, a Wikibook, published by Wikipedia [online] available at: https://en.wikibooks.org/wiki/Artificial_Neural_Networks/History (Accessed November 08, 2017)
- Deeplearning4j-Skymind (2017). History of Deep Learning [online] available at https://www.youtube.com/watch?v=n6XSDA3kfEw (Accessed November 27, 2017)
- John, Y. (2014). How similar are the functioning of artificial neural networks and the human brain? [online] available at: https://www.quora.com/How-similar-are-the-functioning-of-artificial-neural-networks-NN-and-that-of-the-human-brain-How-so (Accessed November 27, 2017)

Image Credits:

- Figures 1: Structure : https://blogs.mathworks.com/loren/2015/08/04/artificial-neural-networks
- Figures 2,3,4: https://www.coursera.org/specializations/deep-learning
- Figure 5: CNNs- https://ujwlkarn.files.wordpress.com/2016/08/conv_all.png?w=748
- Figure 6: RNNs - https://medium.com/@samim/generating-stories-about-images-d163ba41e4ed
- Figure 7: GANs- https://www.linkedin.com/pulse/gans-one-hottest-topics-machine-learning-al-gharakhanian/

Appendix

Step 1: Import Packages
- [numpy](www.numpy.org) is the fundamental package for scientific computing with Python.
- [h5py](http://www.h5py.org) is a common package to interact with a dataset that is stored on an H5 file.
- [matplotlib](http://matplotlib.org) is a famous library to plot graphs in Python.
- [PIL](http://www.pythonware.com/products/pil/) and [scipy](https://www.scipy.org/) are used here to test your model with your own picture at the end.

```
In [7]: import numpy as np
        import matplotlib.pyplot as plt
        import h5py
        import scipy
        from PIL import Image
        from scipy import ndimage
        from lr_utils import load_dataset
        %matplotlib inline # displays plots directly on the notebook rather than a separate pop-up box
```

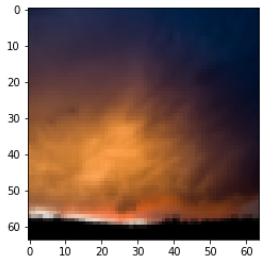
Step 2: Load Data
We have a dataset ("data.h5") containing:
- a training set of m_train = 209 images labeled as cat (y=1) or non-cat (y=0)
- a test set of m_test = 50 images labeled as cat or non-cat
- each image is of shape (num_px=64, num_px=64, 3) where 3 is for the 3 channels (RGB). Thus, each image is square (height = 64) and (width = 64).

Note: We use 64 by 64 pixels just to demonstrate this example, but the dimensions of an image can take any size between 0-255(the maximum value of a pixel channel).
We first load the data by running the following code.

```
# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()
# The following code is printed for clarity so you know the shapes of the data we are working with
print(train_set_x_orig.shape)
print(train_set_y.shape)
print(test_set_x_orig.shape)
print(test_set_y.shape)
print(classes)

# Print an example from the training set
for i in range(1):
    plt.figure(i)
    plt.imshow(train_set_x_orig[i])
```

```
(209, 64, 64, 3)
(1, 209)
(50, 64, 64, 3)
(1, 50)
[b'non-cat' b'cat']
```



The following code will reshape the datasets:

```
# Reshape the training and test examples

### START CODE HERE ### (≈ 2 lines of code)
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0],-1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0],-1).T
### END CODE HERE ###

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

```
train_set_x_flatten shape: (12288, 209)
train_set_y shape: (1, 209)
test_set_x_flatten shape: (12288, 50)
test_set_y shape: (1, 50)
sanity check after reshaping: [17 31 56 22 33]
```

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.
Note : 12288 = pixel_dimension_of_image * 3(red, green, blue colour channels) = 64*64*3.

One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

Lets Standardise our dataset:

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

Sigmoid Function:
Applying it in code in Python:

```
def sigmoid(z):
    s = 1/(1+np.exp(-z))
    return s # returns the value of s whenever called
```

For example:

```
print ("sigmoid([0, -0.7]) = " + str(sigmoid(np.array([0,2]))))
```

```
sigmoid([0, -0.7]) = [ 0.5          0.88079708]
```

Step 3: Initialise Parameters(w,b)

We now initialise w,b to be a vector of zeros using the np.zeros() function :

```
def initialize_with_zeros(dim):
    w = np.zeros((dim,1))
    b = 0

    assert(w.shape == (dim, 1)) #  See if the statement asserted is true; if false it
    # generates an error
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

Step 4: Forward and Backward Propagation

Now that our parameters are initialized, we can do the "forward" and "backward" propagation steps for learning the parameters.

Note:

Forward Propagation:

- We get X
- Compute $A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, \ldots, a^{(m-1)}, a^{(m)})$
- Then calculate the cost function: $J = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

Here are the two formulas we will be using:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X(A - Y)^T \qquad (1)$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \qquad (2)$$

```python
def propagate(w, b, X, Y):
    """
    Arguments Definition:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, number of examples)

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b
    """

    m = X.shape[1] # this sets m to be the shape of the 2nd dimension of the X vector i.e. no. of examples

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T,X) +b)              # compute activation
    cost = -(np.dot(Y,np.log(A).T) + np.dot(1-Y,np.log(1-A).T))/m    # compute cost

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = np.dot(X,(A-Y).T)/m
    db = np.sum(A-Y)/m

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

Step 5: Optimise :

Now, we need to update w,b (to minimise the cost function J) using the gradient descent algorithm explained previously. For a parameter θ, the update rule is $\theta = \theta - \alpha\, d\theta$, where α is the learning rate.

```python
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    """
    Argument Definitions:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, number of examples)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps
    """
    costs = [] # initialise costs to be the empty vector

    for i in range(num_iterations):


        # Cost and gradient calculation (≈ 1-4 lines of code)
        grads, cost = propagate(w,b,X,Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule (≈ 2 lines of code)
        w = w - learning_rate*dw
        b = b - learning_rate*db

        # Record the costs
        if i % 100 == 0:
            costs.append(cost) # append: updates the list by adding an element to the list

        # Print the cost every 100 training examples
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}   # dictionary containing the weights w and bias b

    grads = {"dw": dw,
             "db": db}  # dictionary containing the gradients of the weights and bias with respect to the cost function

    return params, grads, costs
```

Step 6: Predict Labels for Training Set X

There are two steps to computing predictions:

- Calculate $Y = A = \sigma(w^T X + b)$
- Convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5); store the predictions in a vector Y_prediction.

```
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression parameters (w, b)
    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/1) for the examples in X
    '''
    m = X.shape[1] #number of examples
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    # Compute vector "A" predicting the probabilities of a cat being present in the picture
    A = sigmoid(np.dot(w.T,X) + b)

    for i in range(A.shape[1]):

        # Convert probabilities A[0,i] to actual predictions p[0,i]
        if A[0,i]<=0.5:
            Y_prediction[0,i]=0
        else:
            Y_prediction[0,i]=1

    assert(Y_prediction.shape == (1, m))
    print(params)
    return Y_prediction
```

Step 7: Merging all functions into a model

- We now merge all the functions we have previously built and combine it into a single function called “model”. We then use the model function to predict our test set labels.
- Training accuracy is close to 100%. This is a good sanity check: our model is working and has high enough capacity to fit the training data. Test accuracy is around 70%. It is actually not bad for this simple model, given the small dataset we used and that logistic regression is a linear classifier (but we used a non-linear activation function).

```
def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):
    """
    Builds the logistic regression model by calling the function you've implemented previously
    Arguments:
    X_train -- training set represented by a numpy array of shape (num_px * num_px * 3, m_train)
    Y_train -- training labels represented by a numpy array (vector) of shape (1, m_train)
    X_test -- test set represented by a numpy array of shape (num_px * num_px * 3, m_test)
    Y_test -- test labels represented by a numpy array (vector) of shape (1, m_test)
    num_iterations -- hyperparameter representing the number of iterations to optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used in the update rule of optimize()
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """
    # initialize parameters with zeros (≈ 1 line of code)
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent (≈ 1 line of code)
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost = False)

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples (≈ 2 lines of code)
    Y_prediction_test = predict(w,b, X_test)
    Y_prediction_train = predict(w,b, X_train)

    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
    #abs(Y_prediction_train - Y_train) gives you a vector of how different your predictions
    #are from the actual ground truth labels on the training set samples --
    # for example, if your neural network made perfect predictions, then Y_prediction_train
    #would be equal to Y_train, then the resulting vector would be all zeros.
    #Taking the mean() gives you the Average Error over all the training set samples.
    #Multiplying by 100 gives you the average error as a percentage.
    #But, since you want the accuracy and not the error, you flip it around by subtracting
    #the error from 100. (accuracy % = 100 - error %)
```

```
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

We now run the following code to train our model(note the number of iterations performed):

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, learning_rate = 0.005,
          print_cost = False)
```

```
{'w': array([[ 0.19033591],
       [ 0.12259159]]), 'b': 1.9253598300845747}
{'w': array([[ 0.19033591],
       [ 0.12259159]]), 'b': 1.9253598300845747}
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

Step 7: Test with our own image

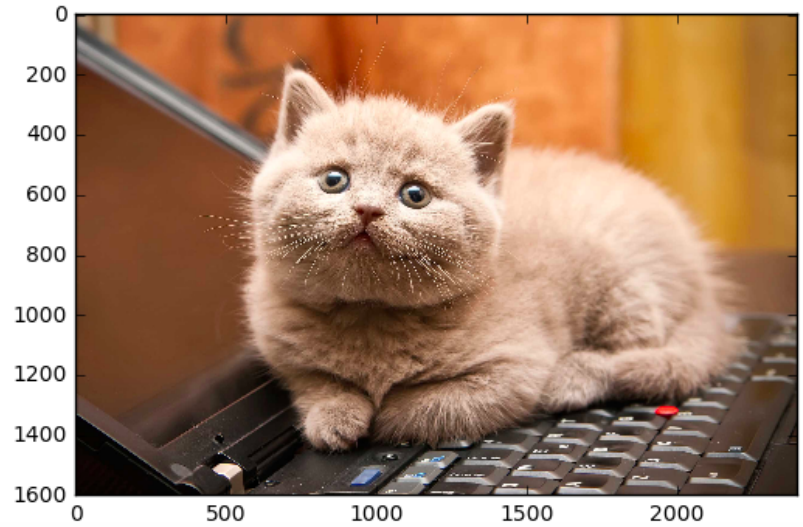
Finally, and perhaps the most exciting part of this simulation, is to test our model with any image of our choice.

```
# Insert IMAGE NAME
my_image = "laptop_cat.jpg"   # change this to the name of our image file

# We preprocess the image to fit our algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a \"" +
      classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
y = 1.0, your algorithm predicts a "cat" picture.
```




```
# Insert IMAGE NAME
my_image = "cat_cage.jpg"   # change this to the name of our image file

# We preprocess the image to fit our algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a \"" +
      classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
y = 0.0, your algorithm predicts a "non-cat" picture.
```

```
# Insert IMAGE NAME
my_image = "sir_trump.jpg"   # change this to the name of our image file

# We preprocess the image to fit our algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a \"" +
      classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
y = 0.0, your algorithm predicts a "non-cat" picture.
```