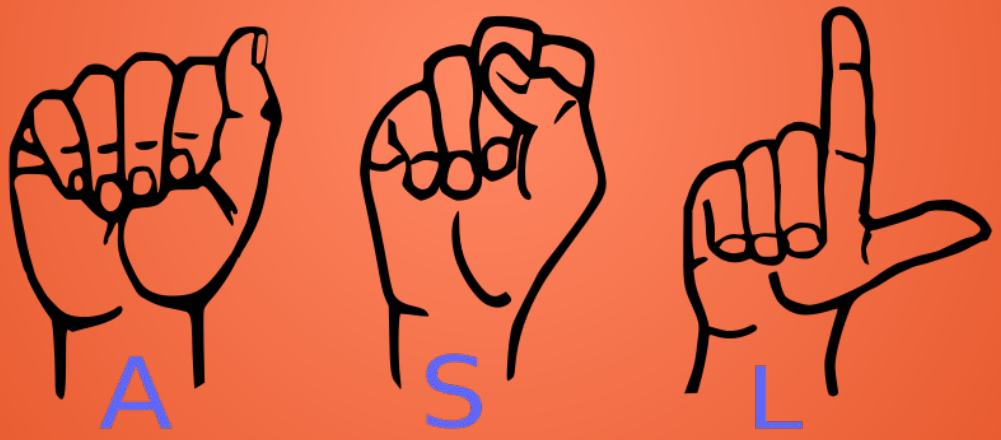
SignText

Hunter College - CS499 - Major Capstone Spring 2020 - MVP Demo



Presented by: Kelvin Basdeo, Kat Chonka, Steni Stephan

May 18, 2020

Product Definition

- Problem: Deaf & Hard of Hearing people lack technological tools that allow them to communicate with a native interface (without text or speech).
- Vision: To make technology more accessible.
- Strategy: Create a web application that translates ASL into written text, built from a machine learning model.
- Goal: Having a functional & accurate interface.
- Our final project is composed of a fully functional UI integrated with a machine learning model to translate ASL to text.

Demo

Tools, Technologies & Sources

Machine Learning

Web Framework

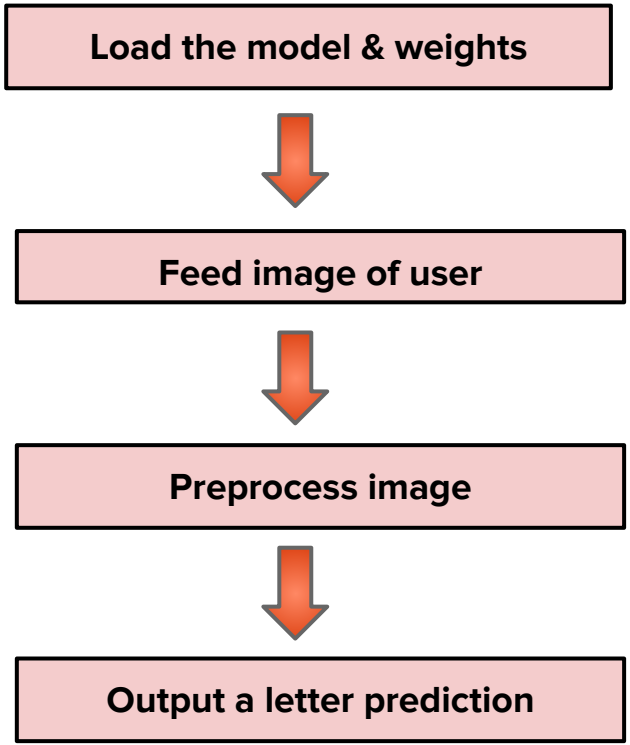
Dataset & ML Model from Kaggle

Front-End: User Interface

Product Architecture & Overview

- Front-End
  - Built with React-Bootstrap
  - Takes images of the user every 3 seconds using a timer and sends them to the back-end.
  - Uses library Axios for HTTP request handling.
  - Displays prediction response in the text box.
- Back-End
  - Built with Flask
  - Receives images from the front end in the form of a base-64 encoded string.
  - Decodes the image, saves it, stores the path.
  - Passes the path to a prediction script.
  - Prediction script returns a response corresponding to the letter that the image most closely resembles.

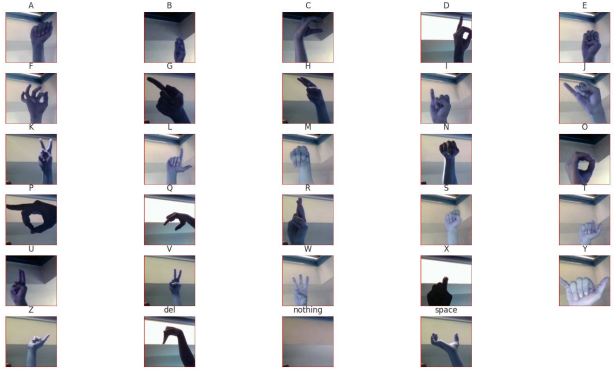
Prediction script methodology:



Our ML Model

- Dataset contains **87,000** total images.
  - *"Classifying Images of the ASL Alphabet using Keras"* by Dan Rasband
  - 29 "classes"/groups of photos. 26 of them are for the letters A-Z and then there are 3 more for 'SPACE', 'DELETE', and 'NOTHING'.
- Our ML model was trained on the dataset ➔ adapted from Kaggle.
- Slim-CNN type model was chosen:
  - CNNs are proven to work for image recognition.
  - Slim-cnn is a very recent (2019) development where the model is at least 87% smaller (300kb) than other comparable neural networks, making it easier to share & deploy. [1]

Plotted Sample of Each Class

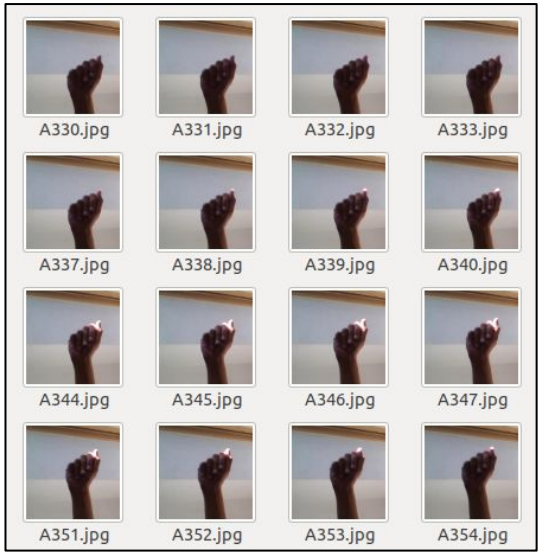


[1] Ankita Sharma and Hassan Foroosh. Slim-CNN: A Light-Weight CNN for Face Attribute Prediction. (July 2019) https://arxiv.org/abs/1907.02157

Challenges & Lessons Learned

- Inaccurate predictions:
  - Training data was too similar / lacked variance
  - Images are all in similar backgrounds, minor differences
  - Background noise in user-generated pictures
    - Face, body, wall-color, etc
- Improving the model:
  - Use a more varied dataset ➔ different backgrounds, positions, lighting
  - Prior image processing to separate the hand from the rest of the background
    - But we would have to work with live video feed & not image screenshots ➔ larger challenges

Sample of our unvaried dataset:

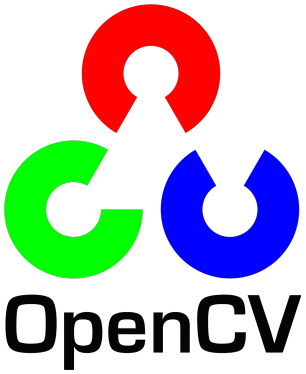


Questions?

OpenCV

Computer vision: deals with how computers can gain high-level understanding from digital images or videos

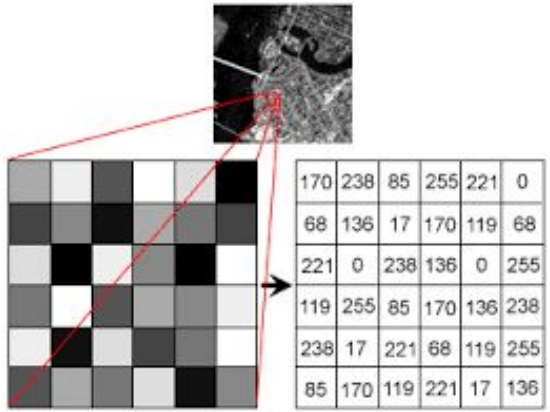
- “Open Source Computer Vision”
- Library of programming functions for computer vision
- Applications & uses:
  - Image processing & analysis
  - Video processing & analysis
  - Feature extraction
  - Object detection
  - Motion analysis
  - and more!
- Supports Python, C++ and other languages



OpenCV: How does it work?

- How do computers process images?
  - Pixel values!
  - Each number represents the pixel intensity at that particular location.
  - Color images have multiple values for a single pixel. These values represent the intensity of respective channels – Red, Green and Blue channels for RGB images.

- OpenCV for Python has NumPy support
  - **NumPy** is a highly optimized library for numerical operations on arrays.
  - All the Python OpenCV array structures are converted to-and-from Numpy arrays. (As per official OpenCV documentation)



OpenCV: Basic Image Manipulation

Reading & Displaying Images

```
# Python Code:
import cv2

path = '/path/to/bird.jpg'

# Read in the image:
img = cv2.imread(path)

# Displaying the image in a window:
cv2.imshow('Bird on a fence',  img)

# The following line will keep the image
# open on a window until key pressed
cv2.waitKey(0)

cv2.destroyAllWindows()
```

Sample image: **bird.jpg**

OpenCV: Basic Image Manipulation

Color Conversion & Writing Images

```
# Python Code:
import cv2

path = '/path/to/bird.jpg'

# Read in the image:
img = cv2.imread(path, 0)

# Write to directory:
cv2.imwrite('graybird.jpg', img)
```

graybird.jpg

- Flag ***0*** tells the program to read the image in as grayscale.
- No flag (as previously used) reads in the image in BGR (Blue-Green-Red) format
- Other flags are available.

OpenCV: Connecting to Video/Webcam

```
import cv2

# Create a video capture object
cap = cv2.VideoCapture(0)

# Loop until the user presses esc
while True:
    # Capture frame by frame
    ret, frame = cap.read()
    # Display back
    cv2.imshow('Video Feed', frame)

    if cv2.waitKey(1) == 27:
        break
cap.release()
cv2.destroyAllWindows()
```

Argument 0 = input is coming from the webcam

Alternatively, you can pass the name of a video file in parenthesis.
Ex: cam = cv2.VideoCapture('myvideo.mp4')

Opens a window on your computer titled 'Video Feed' with your webcam data/frames.

OpenCV: Summary & Applications

- Image Processing:
  - Image rotation, translation, resizing, etc.
  - Color manipulation
  - Segmentation
  - Finding contours (edges)
- Motion detection
- Object Recognition
- Neural Networks
- Machine Learning

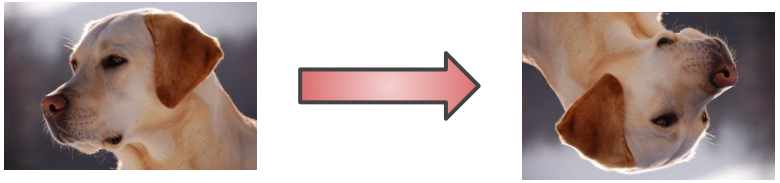
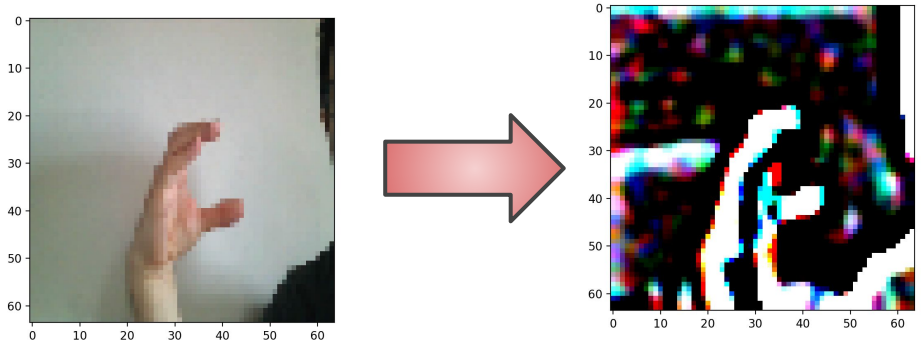


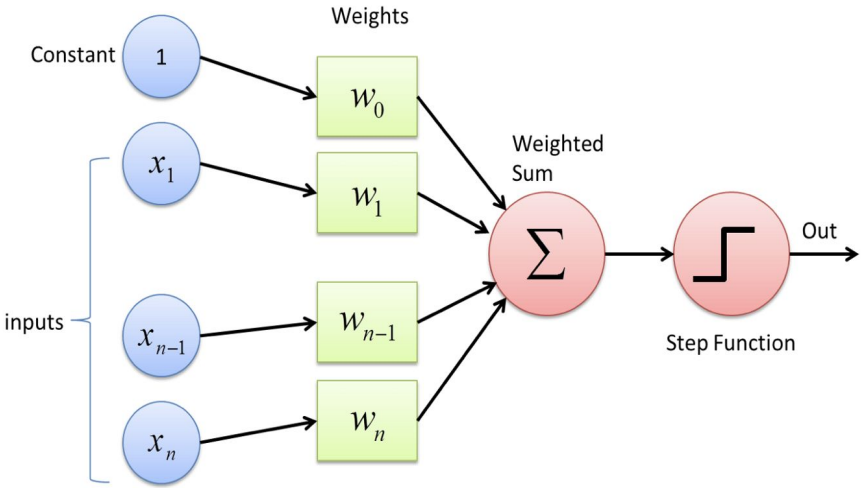
Image Rotation



LaPlacian Edge Detection

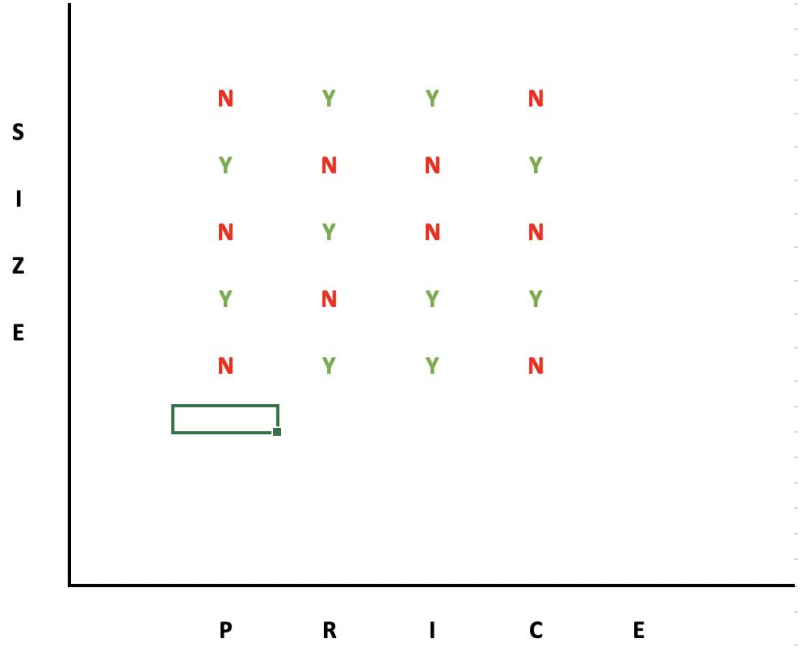
Perceptron: What is Perceptron?

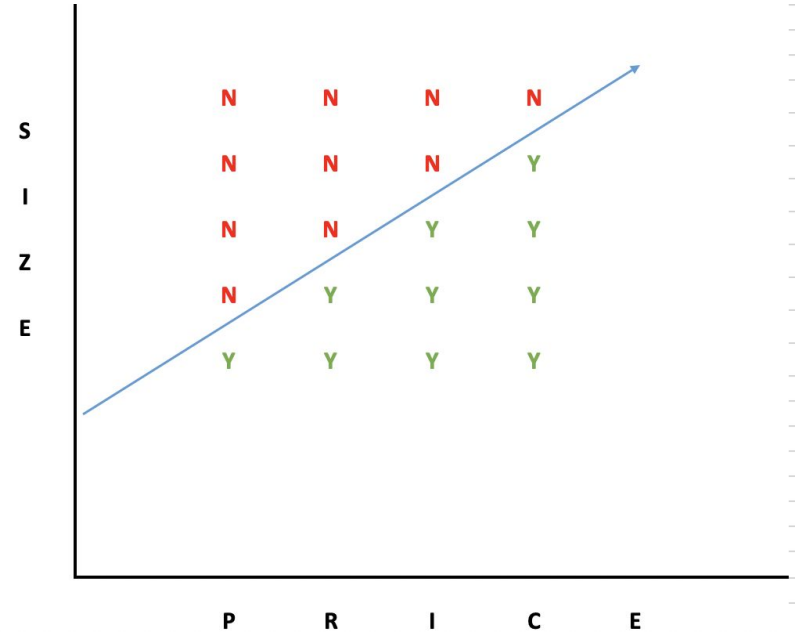
- Machine Learning Algorithm
- Binary Classifier
- Works when data is linearly separable
- 4 Parts:
  - Input Values
  - Weights & Bias
  - Net Sum
  - Activation Function



Perceptron: Linearly Separable Data

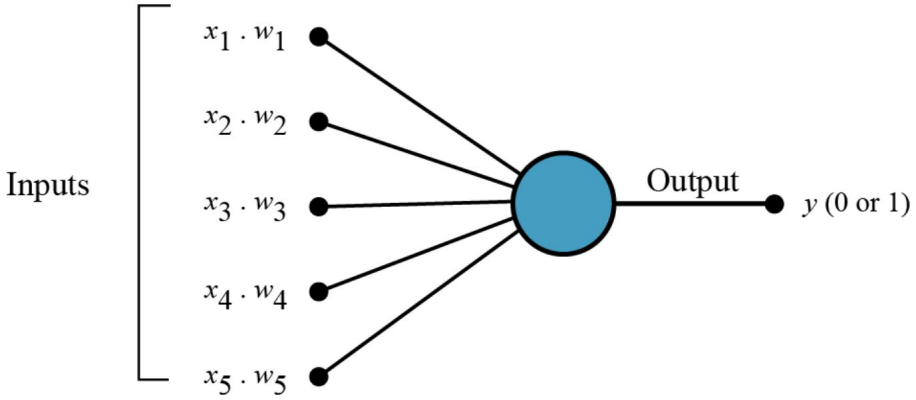
- What is linearly separable data?

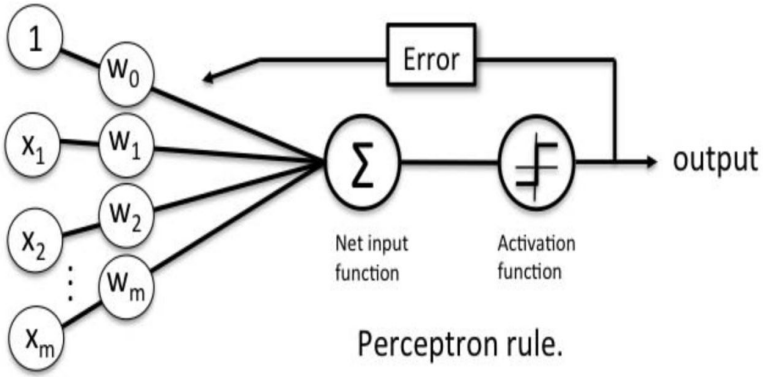




Perceptron: First Step

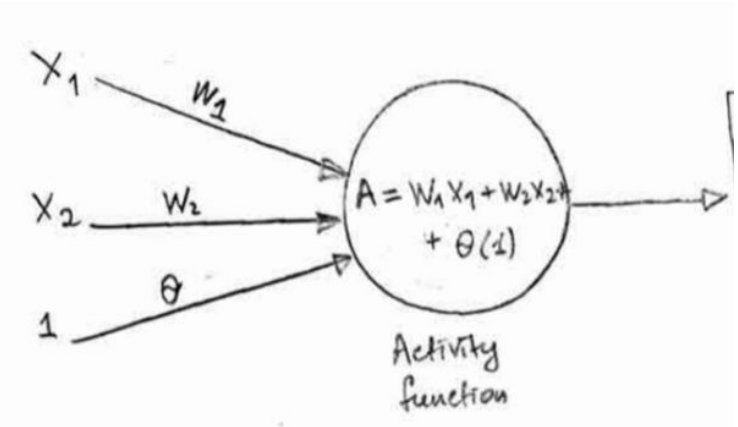
- Step 1: All of the inputs **(x)** are multiplied by their weights **(w)**.
  - Weights are used to account for any bias in the dataset.
  - *How the Perceptron Learns:* Iteratively updates the weights until the activity function produces the correct slope & orientation of the boundary line.





Perceptron: Second Step

- Step 2: Add all the multiplied values **(weighted sum)**.
  - Activity Function = K = $(w_i * x_i) + (1*\theta)$
  - Activity function defines the line that will bisect the data
  - The bias term $(1*\theta)$ defines the vertical placement of the line
    - Allows the activation function to be shifted to the left or right, to better fit the data.



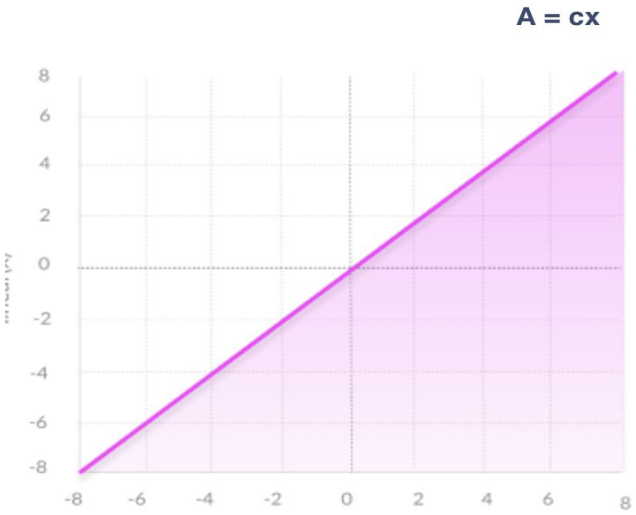
$$e_j = d_j - y_j$$

e_j - error; d_j - desired output; y_j - actual output

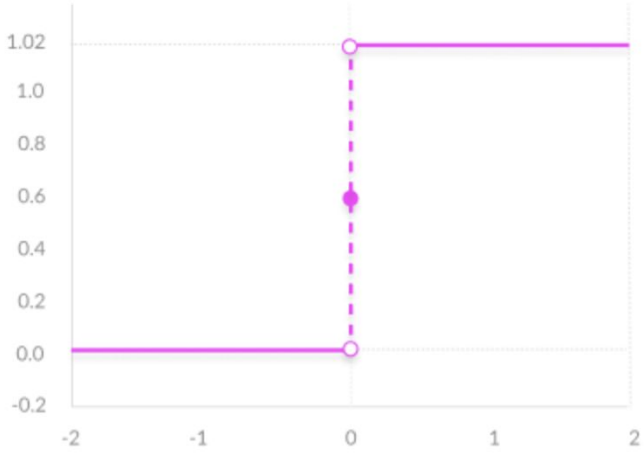
Perceptron: Final Step

- Step 3: Apply the weighted sum to the correct activation function.
  - Activation functions are mathematical equations used to determine the outputs of neural networks

Linear Activation Function

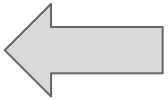


Binary Step Function

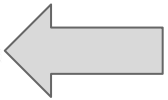


Perceptron: Code Example

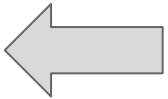
```
# Define a perception function, returns weights
def perceptron(data, amt_samples, outputfile):

    weights = np.zeros(3)    # 2 features + 1 = 3, one is for bias
    epochs = amt_samples     # iterations
    lr = 0.2                 # learning rate

    for i in range(epochs):
        item, label = choice(data)
        result = np.dot(weights, item)     # compute the dot product

        activation = None
        if result < 0:
            activation = 0
        else:
            activation = 1

        margin = label - activation

        weights += lr * margin * item

        # Write to outfile
        outputfile.write(str(round(weights[0], 1)))
        outputfile.write(",")
        outputfile.write(str(round(weights[1], 1)))
        outputfile.write(",")
        outputfile.write(str(round(weights[2], 1)))
        outputfile.write("\n")

    return weights
```



Input Values, Weights/Bias



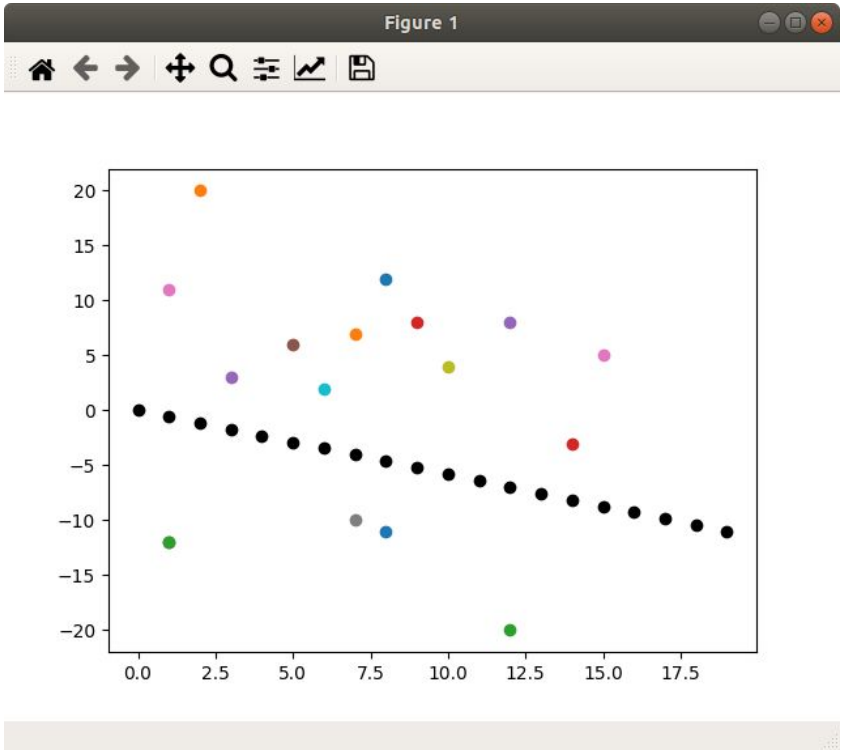
Weights*Input



Activation Function (Binary Step)

Perceptron: Decision Boundary

An example of the boundary line:



Uses final weights to calculate the equation of the line:

```
w1 = weights[0]
w2 = weights[1]
b = weights[2]      #bias

x_intercept = (0, -b / w2)
y_intercept = (-b / w1, 0)
m = -(b / w2) / (b / w1)
```

Perceptron: Importance

- Both Perceptron & Gradient Descent are the building blocks of modern Neural Networks *trained with backpropagation*.
  - Backpropagation: "backward propagation of errors"
  - Calculates the gradient of the error function with respect to the neural network's weights.
  - The “backwards” part of the name stems from the fact that the calculation of the gradient proceeds backwards through the network, with the gradient of the final layer of weights being calculated first and the gradient of the first layer of weights being calculated last”
- Further study into both Perceptron & Gradient Descent is important. Modern implementations take advantage of specialized GPUs to further improve performance that we can apply to solve the complex problems of today’s world.

Further questions?

Thank You!

http://signtext.ue.r.appspot.com