Drones for Humanity

1.0

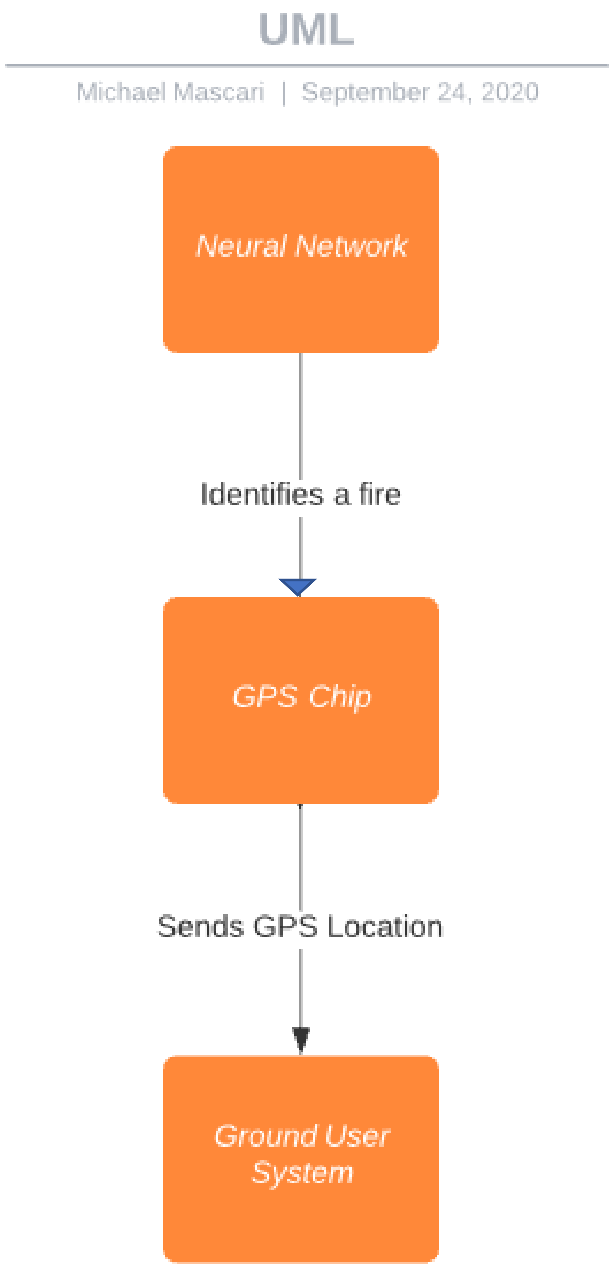
Design Document

September 2020

By: Michael Mascari

Table of Contents

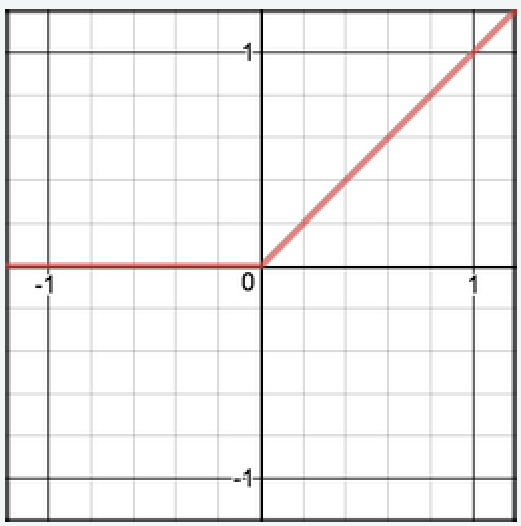
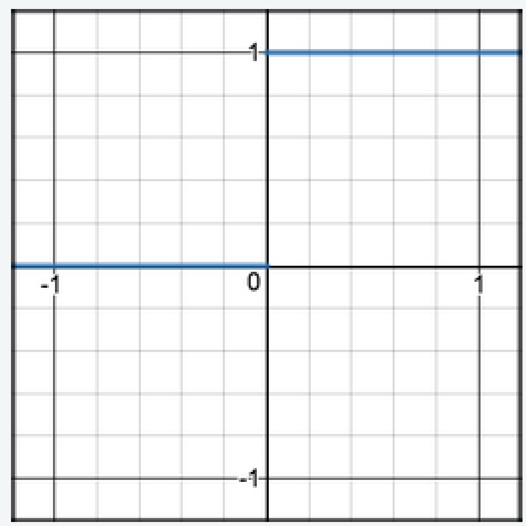
1. UML



There are no private/public variables passed between the classes. There will be output from functions, which are labeled on the directional arrows. The Neural Network is planned to be one class in its entirety.

2. Class Functionality

a) Neural network will have 3 main methods, predict, computeLoss, and fit. These are the three required methods for taking in data, forming it to a neural network, and predicting if new data is similar to old data. Among those methods will also be some helper methods like an activation function, which lets the neurons have a fractional weight to them instead of 1/0. Then possibly a derivative of the activation method will be needed if backpropagation will be a part of the CNN. The most likely candidate for activation function is the ReLu activation function.

Function	Derivative
$R(z) = \begin{Bmatrix} z & z > 0 \\ 0 & z <= 0 \end{Bmatrix}$	$R'(z) = \begin{Bmatrix} 1 & z > 0 \\ 0 & z < 0 \end{Bmatrix}$
	

```
def relu(z):
  return max(0, z)
```

```
def relu_prime(z):
  return 1 if z > 0 else 0
```

Credit: ml-cheatsheet.com

b) The GPS Chip's class is very straight forward. Only two methods in its class. First it is told to open a channel with the ground user system and make sure everything is okay then wait. Second, once a fire is detected it will send a signal through the channel it opened earlier.

c) The Ground User System is very similar to the GPS Chip class. It listens on the channel that GPS Chip Class opens waiting for a signal then prints a message to the ground user if a message is received.

3. Pseudocode

```
class NeuralNetwork:

  # input a vector [a, b, c, ...] with the number of nodes in each layer
  def __init__(self, layers, alpha=0.01):
    # list of weight matrices between layers
    self.W = []

    # network architecture will be a vector of numbers of nodes for each layer
    self.layers = layers
    # learning rate
    self.alpha = alpha

    # initialize the weights between layers (up to the next-to-last one) as normal random variables
    for i in np.arange(0, len(layers) - 2):
      self.W.append(np.random.randn(layers[i] + 1, layers[i + 1] + 1))

    # initialize weights between the last two layers (we don't want bias for the last one)
    self.W.append(np.random.randn(layers[-2] + 1, layers[-1]))

  def reLU(self, z):
    return max(z, 0)

  def reLUDerivative(self, z):
    if z > 0:
      return 1
    else:
      return 0

  # fit the model
  def fit(self, X, y, epochs=10000, update=1000, alpha=0.01, activation='E'):
    # add a column of ones to the end of X
    X = np.hstack((X, np.ones([X.shape[0], 1])))

    for epoch in np.arange(0, epochs):

      # feed forward, backprop, and weight update
      for (x, target) in zip(X, y):
        # make a list of output activations from the first layer
        # (just the original x values)
        A = [np.atleast_2d(x)]

        # feed forward
        for layer in np.arange(0, len(self.W)):
          # feed through one layer and apply sigmoid activation
          net = A[layer].dot(self.W[layer])
          out = self.elu(net, alpha)

          # add our network output to the list of activations
          A.append(out)

        # backpropagation
        error = A[-1] - target
```

```python
            D = [error * self.reLUDerivative(A[-1])]

            # loop backwards over the layers to build up deltas
            for layer in np.arange(len(A) - 2, 0, -1):
                delta = D[-1].dot(self.W[layer].T)
                delta = delta * self.reLUDerivative(A[layer])
                D.append(delta)


            # reverse the deltas since we looped in reverse
            D = D[::-1]

            # weight update
            for layer in np.arange(0, len(self.W)):
                self.W[layer] -= self.alpha * A[layer].T.dot(D[layer])

        if (epoch) % update == 0:
            loss = self.computeLoss(X, y)

    def predict(self, X, addOnes=True):
        # initialize data, be sure it's the right dimension
        p = np.atleast_2d(X)

        # add a column of 1s for bias
        if addOnes:
            p = np.hstack((p, np.ones([X.shape[0], 1])))

        # feed forward!
        for layer in np.arange(0, len(self.W)):
            p = np.dot(p, self.W[layer])

        return p

    def computeLoss(self, X, y):
        # initialize data, be sure it's the right dimension
        y = np.atleast_2d(y)

        # feed the datapoints through the network to get predicted outputs
        predictions = self.predict(X, addOnes=False)
        loss = np.sum((predictions - y) ** 2) / 2.0

        return loss
```

Hopefully, the CNN would look something similar to this.