Using a Feed Forward Neural Network to Computationally Predict 105mm Lag and Depression Angle Errors on Side Firing Gunships

Albert Young III

Abstract

This paper covers the development of a feed forward neural network trained to make predictions of the nominal lag and depression angles of the 105mm Howitzer on side-firing gunships over time. This tool is presented as a novel way of maintaining and updating gun errors induced by misalignments in gun/aircraft reference frames. A background is given on the implementation of the neural network, using the modern machine learning framework Tensorflow, as well as the process of training the network, and the model's accuracy.

Using a Feed Forward Neural Network to Computationally Predict 105mm Lag and Depression Angle Errors on Side Firing Gunships

Introduction

Being able to accurately engage an enemy target within close proximity of friendly forces, from thousands of feet away, with a non-guided munition is the expertise of side firing gunships. Any procedure or technique which could enhance the accuracy of a gunship's munitions would be a valued addition to the communities knowledge, and has the potential to save lives. To this end, this paper presents a machine learning algorithm capable of predicting corrections to the lag and depression angles of the 105mm Howitzer. These corrections are learned over time, and applied after several inputs into the system. Similar to manual procedure known as a tweak, this automatic learning algorithm has the potential to be able to simultaneously correct wind and gun errors, while maintaining a "memory" of previous inputs. The technique presented here is also amicable to other caliber projectile weapons, and to correcting sensor and inertial navigation unit discrepancies with the proper tooling.

Neural Networks

The field of machine learning has grown exponentially in the past twenty years. This field is dedicated to uprooting the traditional notion of programming a computer line by line. Instead of telling a computer *how* to solve a problem, a computer is presented with observational data and it computes *how* to solve it on its own. While computer learning has been a holy grail of sorts in the computer industry, it was not until Rumelhart, Hinton, and Williams (1986) published a seminal paper on a new algorithm that the field began to grow. During the last decade, thanks to the effects of large scale corporations and globalization, the amount of information available has grown faster than the speed of processors. The ability to train computers to work on data-sets and deduce patterns is vitally important to a variety of industries, and it's being deployed on a large scale by companies such as Google, Microsoft, and Facebook.

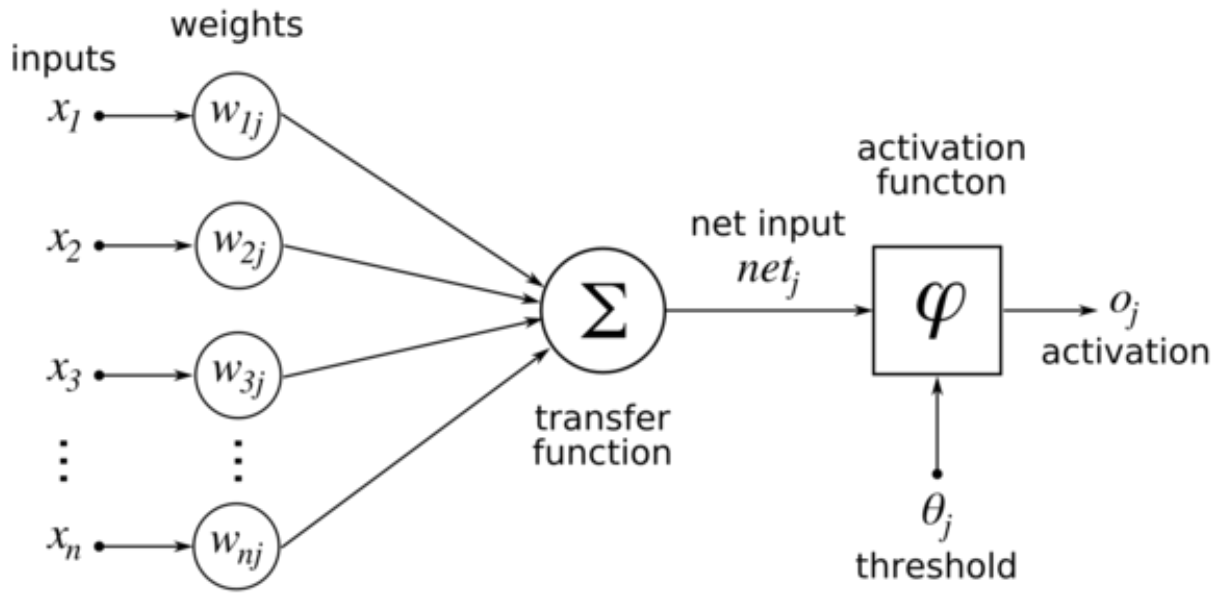


Figure 1. Overview of a single neuron in a neural network (Commons, 2016a)

The basic building block of a neural network is a neuron, depicted in Figure 1. A neuron consists of any number of inputs, weights to be applied to those inputs, a summation function, and an activation function. Both the inputs and outputs of a neuron can be densely or sparely connected to other neurons, building a neural network (Figure 2). The activation function can be selected at will, and different activation functions are better at different tasks. For this model, the sigmoid (1) and tanh (2) functions were selected.

$$Sigmoid(t) = \frac{1}{1 + e^{t}} \tag{1}$$

$$Tanh(t) = \frac{2}{1 + e^{-2t}} - 1 \tag{2}$$

Rumelhart et al. (1986)'s breakthrough was that these functions are fully differentiable, and that can be used to modify the weights of each neuron. After training data has been supplied, the network will output a value. That value can be compared against a known value for the given inputs and an error can be assigned. By using the chain rule, that error can be back-propagated through the network to correct weights of

each neuron (Bengio & Courville, 2016). Through a multitude of iterations of this process, the network will learn the proper output for a given input, *even if it has not seen that input before.*

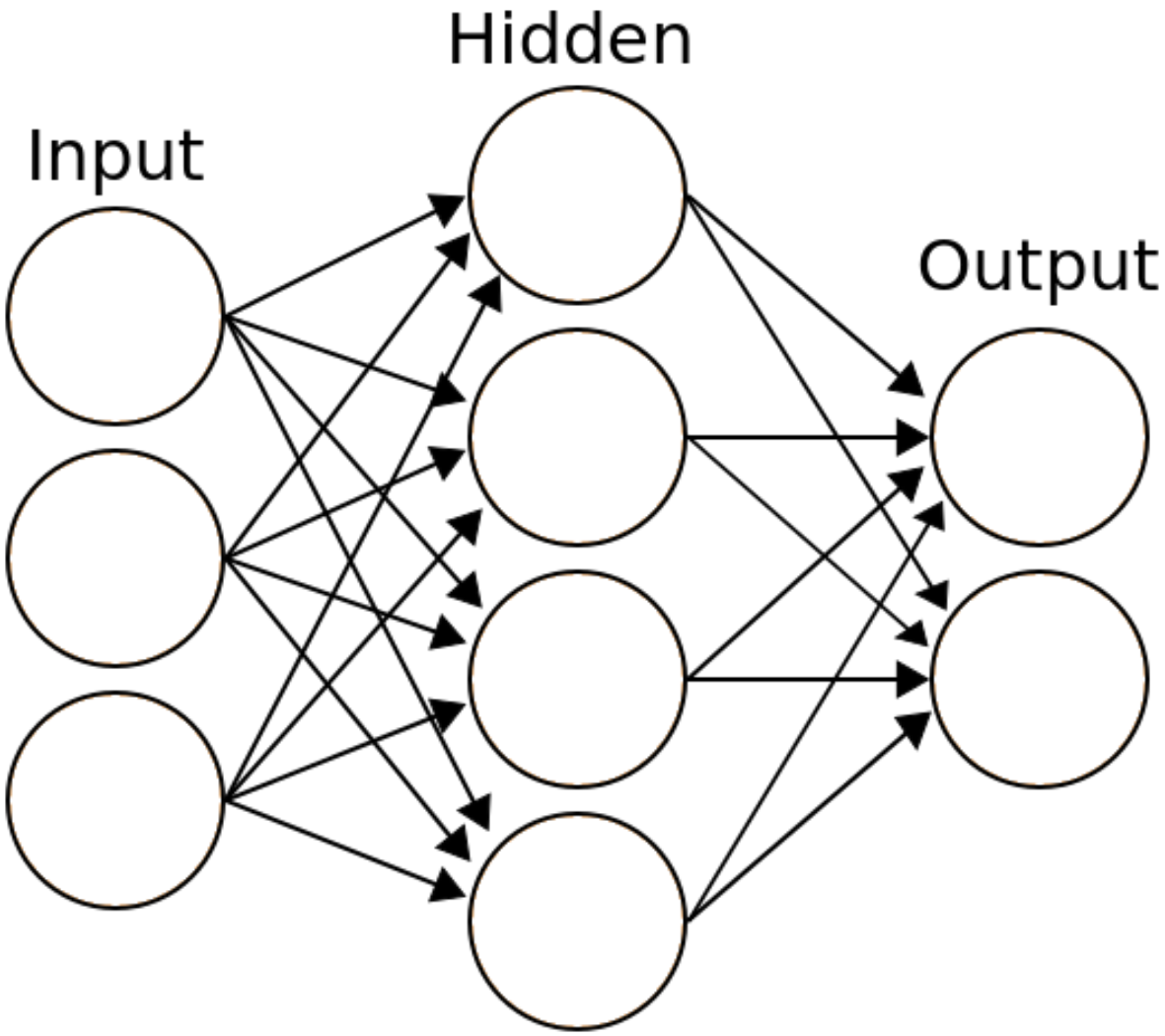


Figure 2. A fully connected neural network (Commons, 2016b)

Ballistic Modeling

The topic of exterior ballistics, or the effects after a projectile has left the barrel of the weapon, has been studied in depth for over one hundred years. McCoy (2012) dedicates the second chapter of his book to the mathematics behind the trajectory of a bullet in this

state. Most of the work studied assumes the weapon is fired either from a horizontal position, or into the air. There is few public works concerning the effects of firing a projectile from the attitudes required for side firing aircraft. The Aircrew Weapons Delivery Manual for the AC-130U describes the general algorithm it uses to compute the lag and depression angles, but without access to the proprietary code base owned by Boeing, there is no way to re-implement the procedure (United States Air Force, 2009). To complete the training of the neural network, a proper ballistic model needed to be developed from scratch.

Instead of using the china-lake algorithm, a iterative method has been developed. The ballistic model presented in appendix B can be summarized by the following procedure:

1. Nominal Parameters Entered From Ballistic Database
2. Calculate Gun Pointing Vector
3. Generate a Bullet Object
4. While Bullet's Z Value is Above Zero:
    1. Calculate the Speed of Sound Based on Altitude and Temperature
    2. Calculate Bullet Drag Vector Over Time Iteration
    3. Apply Gravity and Drag to Bullet Over Time Iteration
    4. Return New Bullet to Next Iteration

Tensorflow

Tensorflow is a programming system in which computations are composed of graphs, which can be distributed over many computer systems (Abadi et al., 2015). It is specifically designed for machine learning and can compute the derivative of its built in neural network components. This greatly simplifies the construction of neural networks.

The network used in this paper is composed of one input layer, two hidden layers consisting of 256 neurons each, and one output layer. Tensorflow allows the construction of the network through a series of procedural statements.

Methods

To complete training of the neural network, a data set was prepared using the ballistic model previously described. The data set consists of lag and depression angles with random errors added for a given set of nominals, as demonstrated in Table 1.

Generated Input		Actual Values		Expected Outcome	
Lag	Dep	Lag	Dep	Lag	Dep
-222	-400	-220	-410	2	10

Table 1

Example of Generated Sample Data

Once this data is generated, it can be feed into the learning algorithm presented in appendix C. This algorithm outputs corrected nominal lag and depression angles. The critical competent to the learning algorithm is the application of the proper cost function. This function informs the network of the error between the predicted and the actual values of the training data. Several standard functions were attempted, but a custom function similar to the least mean squares function allowed the network to train the fastest.

Results

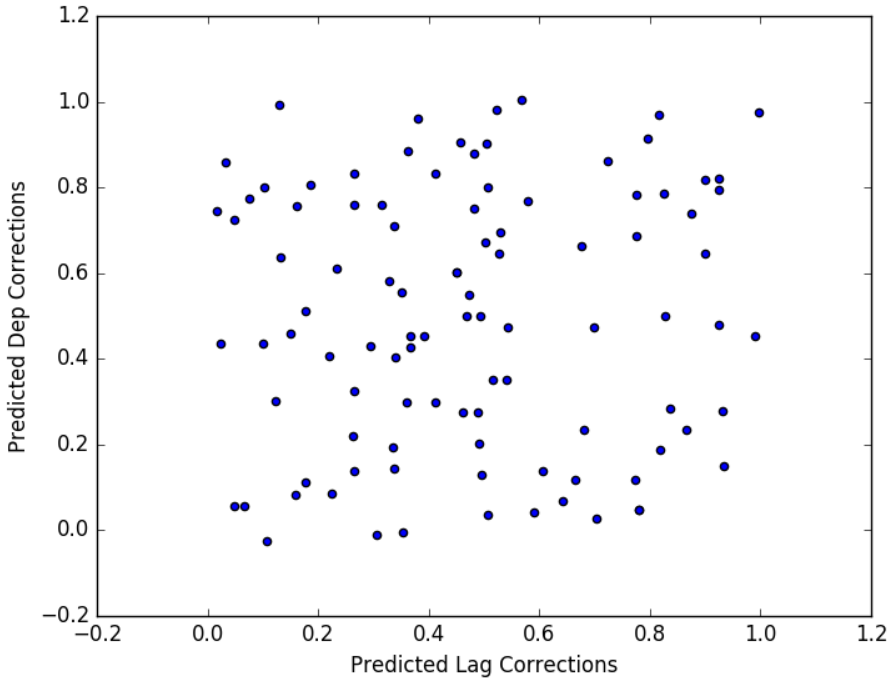
The neural network presented can achieve greater than 99.9 percent accuracy in predicting error corrections for a given set of learned lag and depression angles, as shown in Figure 3a and 3b. Note that the axis are scaled values, as the neural network only accepts values from zero to one. A scaling function is included in Appendix C.

The network was trained over 5000 iterations, and achieved the accuracy in less than three minutes on a low end laptop. The testing data shown in Figure 3a was never seen by the neural net, as the traing set was completely different from the testing set. This shows that the network could be trained in real time, with new values being added every flight.

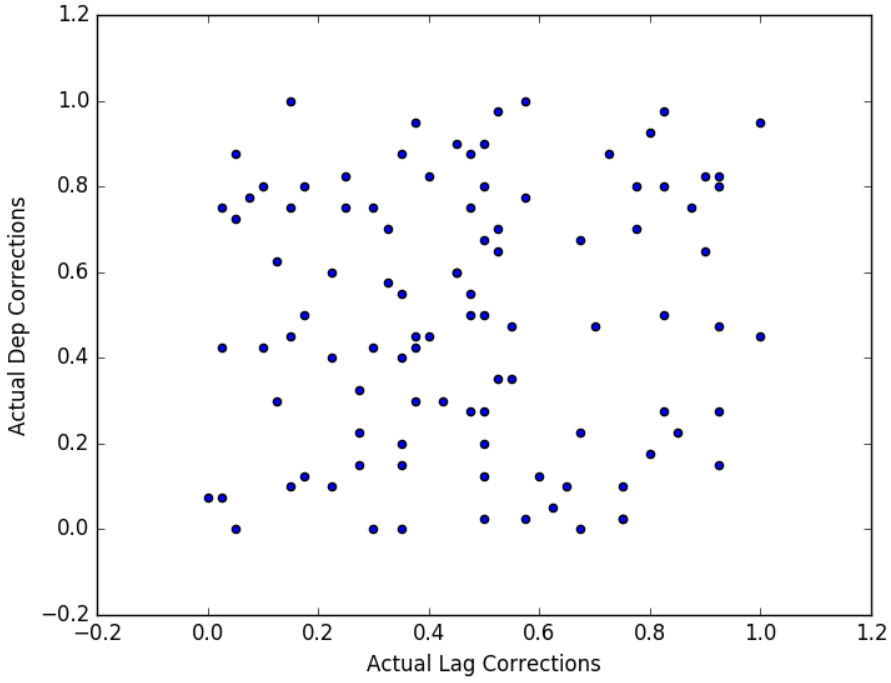
Conclusions

This paper has presented a method of predicting lag and depression angle errors on side firing gunships using feed forward neural networks. The method is complimentary to the existing methods used in operational gunships, with the added benefit of being able to "remember" correction values for a giving situation. This method does require a considerable larger amount of computing power than the current U-model gunship's mission computers can handle, a modest workstation laptop with only one dedicated graphical processing unit could speed up performance by 55%.

This work was conducted to show the viability of utilizing machine learning algorithms in side firing gunship applications, thus a relativity simple task was selected. This work can be expanded to not only automatically correct lag and depression angles, but to also correct for wind error. Work has already been started on accomplishing this goal. Also, the use of feed forward neural networks could be applied to sensor/gun reference frame corrections.



(a) The Output of the Fully Trained neural network



(b) The Desired Output

Figure 3. Results

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems.* Retrieved from `http://tensorflow.org/` (Software available from tensorflow.org)

Bengio, I. G. Y., & Courville, A. (2016). *Deep learning.* Retrieved from `http://www.deeplearningbook.org` (Book in preparation for MIT Press)

Commons, W. (2016a). *Artificialneuronmodel.* Retrieved from `\url{https://upload.wikimedia.org/wikipedia/commons/thumb/6/60ArtificialNeuronModel_english.png/600px-ArtificialNeuronModel_english.png}` ([Online; accessed July 1, 2016])

Commons, W. (2016b). *Artificialneuronmodel.* Retrieved from `\url{https://commons.wikimedia.org/w/index.php?curid=1496812}` ([Online; accessed July 1, 2016])

McCoy, R. L. (2012). *Modern exterior ballistics: the launch and flight dynamics of symmetric projectiles*. Schiffer Pub.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986, Oct 09). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533-536. Retrieved from `http://dx.doi.org/10.1038/323533a0` doi: 10.1038/323533a0

United States Air Force. (2009). *To 1c-130(a)u-34-cd-1.* United States Department of the Air Force. (Aircrew Weapons Delivery Manual Nonnuclear)

Appendix A

Copyright of Included Software

Because of the closed down nature of the current military procurement of software, and the negative effects it has on the operators who actually use said software, the following is released under the GNU Public License.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.

Appendix B

Ballistic Modeling Code

```
import Vector3d
import math

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

vec = Vector3d.Vector3D

projArea = 0.00864286
inital105Velocity = 1620.0

def rad2deg(rad):
    return math.degrees(rad)


def deg2rad(deg):
    return math.radians(deg)


def k2c(k):
    return k - 273.15


def c2k(c):
```

```
    return c + 273.15


def feet2meters(feet):
    return feet * 0.3048


def meters2feet(meters):
    return meters * 3.28084


def feet2knot(feet):
    return feet * 0.000164579


def knot2feet(knot):
    return knot * 6076.12


def knot2meters(knot):
    return knot * 1852.0


def meters2knots(meters):
    return meters * 0.000539957
```

```
def kph2kps(kph):
    return (kph / (60.0 * 60.0))


def pound2kg(pound):
    return pound * 0.453592


def deg2mil(deg):
    return deg * 17.778


def mil2deg(mil):
    return mil / 17.778


def mil2rad(mil):
    return deg2rad(mil2deg(mil))


def mils2deg(mils):
    return mils / 1000.0


def rad2mils(rad):
    return rad * 1000.0
```

```
def tempAtAltitudeSI(meters):
    # Calculate the temp at altitude in SI units, input meters
    # L = temp laps rate K/m T0 = sea level standard temp"
    return 288.15 - (0.0065 * meters)


def tempAtAltitudeC(meters):
    return k2c(tempAtAltitudeSI(meters))

def pressureAtAltitude(meters):
    # returns SI units in Pa
    L  =  0.0065     # K/m
    g  =  9.80665    # m/s^2
    p0 =  101325     # Pa
    h  =  meters     # m
    M  =  0.0289644 # kg/mol
    T0 =  288.15     # K
    R  =  8.31447    # J/(mol*K)

    return p0 / (math.pow((1 - ((L * h) / T0)), ((g* M) / (R * L))
        ))


def densityAtAltitudeMeters(meters):
    # returns SI units of kg/m^3
    p = pressureAtAltitude(meters)
```

```
    r = 287.058
    temp = tempAtAltitudeSI(meters)
    return (p / (r * temp))

class DragPoint:
    def __init__(self, x = 0.0, y = 0.0):
        self.x = x
        self.y = y


dragPoints =  [DragPoint( -1000 ,  0.122),
               DragPoint( 0.01  ,  0.122),
               DragPoint( 0.6   ,  0.122),
               DragPoint( 0.8   ,  0.122),
               DragPoint( 0.9   ,  0.135),
               DragPoint( 0.95  ,  0.191),
               DragPoint( 1.0   ,  0.336),
               DragPoint( 1.05  ,  0.400),
               DragPoint( 1.1   ,  0.413),
               DragPoint( 1.2   ,  0.397),
               DragPoint( 1.35  ,  0.382),
               DragPoint( 1.5   ,  0.370),
               DragPoint( 1.75  ,  0.350),
               DragPoint( 2.0   ,  0.330),
               DragPoint( 2.5   ,  0.285),
               DragPoint( 3.0   ,  0.255),
               DragPoint( 4.0   ,  0.210),
```

```
                DragPoint(5.0      ,   0.185)]


def interp(a, b, frac):
    # points A and B, frac between 0 and 1
    nx = a.x+(b.x-a.x)*frac
    ny = a.y+(b.y-a.y)*frac
    return DragPoint(nx,  ny)



def interpolate(imputmach):
    # returns drag coefficient
    arrayLength = len(dragPoints)
    if  imputmach <= dragPoints[0].x:
        return dragPoints[0].y
    if imputmach >= dragPoints[arrayLength - 1].x:
        return dragPoints[arrayLength - 1].y

    for i in range(arrayLength):
        x1 = dragPoints[i].x
        x2 = dragPoints[i+1].x

        if imputmach == x1 :
            return dragPoints[i].y
        if imputmach == x2 :
            return dragPoints[i + 1].y
        if (imputmach < x2) and (imputmach > x1):
```

```
            distance = x2 - x1
            percent = (imputmach - x1) / distance
            out = interp(dragPoints[i], dragPoints[i+1], percent)
            return out.y


def interpDrag(mach):
    return interpolate(mach)


def speedOfSound(c):
    # returns speed of sound in m/s
    return 331.3 + (0.606 * c)


def knots2mach(knots, c):
    # takes knots/second and converts to mach, returns mach
    return (knots / (meters2knots((speedOfSound(c)))))


def dragEquation(density, velocity, coeff):
    return 0.5 * density * (velocity * velocity) * projArea *
        coeff


def bulletDrag(alt, velocity):
    # "takes feet/sec and feet alt, returns drag force"
```

```
    metersAlt = feet2meters(alt)
    density  = densityAtAltitudeMeters(metersAlt)
    vel = feet2meters(velocity)
    temp = tempAtAltitudeC(metersAlt)
    return dragEquation(density, vel, (interpDrag( knots2mach(
        feet2knot(velocity), temp))))


def bulletDecelleration(feetAlt, FPSofBullet):
    return meters2feet((bulletDrag(feetAlt, FPSofBullet)) /
        15.01163948515)


# Bullet Construction and Manipulation


def setLagAndDepression(lag, depression):
    x = vec(1.0, 0.0, 0.0)
    y = vec(0.0, 1.0, 0.0)
    z = vec(0.0, 0.0, 1.0)
    vecLeft = vec(-1.0, 0.0, 0.0)
    vecLeftWithLag = vecLeft.clone().rotateZ( mil2rad(lag))
    vecWithDep = vecLeftWithLag.clone().rotateY( mil2rad(
        depression))
    vecWithNoseUp = vecWithDep.clone().rotateX( deg2rad(3.0))
    vecWithYaw = vecWithNoseUp.clone().rotateZ( deg2rad(0.0))
    return vecWithYaw
```

```
def addBankToGunAngle(bank, gpv):
    # gpv is a Vector3d
    return gpv.clone().rotateY( deg2rad(bank))

def rotateToAcHeading(heading, gunVec):
    z = vec(0.0, 0.0, 1.0)

    headingAngleWithoutZ = heading.clone().setZ(0).unit()
    headingAngle = headingAngleWithoutZ.angleTo(vec(1,0,0))
    if headingAngleWithoutZ.y < 0:
        headingAngle = ((deg2rad(180) - headingAngle) + deg2rad
            (180))

    gunWithoutZ = gunVec.clone().setZ(0).unit()
    gunAngle = gunWithoutZ.angleTo(vec(1,0,0))
    if gunWithoutZ.y < 0:
        gunAngle = ((deg2rad(180) - gunAngle) + deg2rad(180))
    gunVecRot = gunVec.clone().rotateZ( deg2rad(180))
    gunVecRot2 = gunVecRot.clone().rotateZ( (headingAngle +
        deg2rad(90)) )

    return gunVecRot2

class Bullet:
    def __init__(self):
```

```
        self.gpvWithBank = 0
        self.position = 0
        self.oldPosition = 0
        self.velocity = 0
        self.decelVec = 0
        self.mach = 0
        self.velocityAC = 0

    def __str__( self ):
        return self.toString()

    def toString(self):
        return ("gpvWithBank: " + str(self.gpvWithBank) + "\n" +
                "position:     " + str(self.position) + "\n" +
                "oldPosition: " + str(self.oldPosition) + "\n" +
                "velocity:     " + str(self.velocity) + "\n" +
                "decelVec:     " + str(self.decelVec) + "\n" +
                "mach:         " + str(self.mach) + "\n" +
                "VelocityAC:   " + str(self.velocityAC) + "\n")




def bulletMaker(heading, distance, y, height, lag, depression,
    TAS, bank):
    position = vec(distance, y, height)
    gunVecNoHeading = setLagAndDepression(lag, depression)
```

```
    gpvWithBank = addBankToGunAngle(bank, gunVecNoHeading)
    gunVec = rotateToAcHeading(heading, gpvWithBank)
    vector1 = gunVec.clone().multiplyScalar(inital105Velocity)
    vector2 = heading.clone().multiplyScalar(knot2feet(kph2kps(TAS
        )))

    velocity = vector1.clone().add(vector2)

    bul = Bullet()
    bul.gpvWithBank = gunVec
    bul.position = position
    bul.velocity = velocity
    bul.velocityAC = vector2
    return bul

def velocity(vel, acc, sec):
    return vel + (acc * sec)



def displacementTime(x0, v0, a, t):
    return x0 + (v0 * t) + (0.5 * a * (t * t))

def changeBulletFirst(bul, sec, windSpeed, windDir):

    windFPS = knot2feet(windSpeed)/3600 * sec
```

```
class WindVec:
    def __init__(self):
        self.x = math.sin(deg2rad(-windDir)) * windFPS
        self.y = -math.cos(deg2rad(-windDir)) * windFPS

windVec = WindVec()

pos = bul.position
x0 = pos.x
y0 = pos.y
z0 = pos.z

vel = bul.velocity
vx0 = vel.x
vy0 = vel.y
vz0 = vel.z

decellScaler = bulletDecelleration(z0 , vel.length())
decellVec = bul.gpvWithBank.clone().negate().multiplyScalar(
   decellScaler)

vx = velocity(vx0, decellVec.x, sec)
vy = velocity(vy0, decellVec.y, sec)
vz = velocity(vz0, (decellVec.z + -32.2), sec)

newPositionX = displacementTime(x0, (vx0+vx)/2.0, decellVec.x,
    sec) + windVec.x
```

```
    newPositionY = displacementTime(y0, (vy0+vy)/2.0, decellVec.y,
        sec) + windVec.y
    newPositionZ = displacementTime(z0, (vz0+vz)/2.0, (decellVec.z
        + -32.2), sec)

    newBullet = Bullet()
    newBullet.oldPosition = bul.position
    newBullet.gpvWithBank = bul.gpvWithBank
    newBullet.mach = knots2mach(feet2knot(vel.length()),
       tempAtAltitudeC(newPositionZ))
    newBullet.decelVec = decellVec
    newBullet.position = vec(newPositionX, newPositionY,
       newPositionZ)
    newBullet.velocity = vec(vx, vy, vz)
    newBullet.velocityAC = bul.velocityAC
    return newBullet


def changeBulletRest(bul, sec, windSpeed, windDir):
    windFPS = knot2feet(windSpeed)/3600 * sec

    class WindVec:
        def __init__(self):
            self.x = math.sin(deg2rad(-windDir)) * windFPS
            self.y = -math.cos(deg2rad(-windDir)) * windFPS

    windVec = WindVec()
```

```
pos = bul.position
x0 = pos.x
y0 = pos.y
z0 = pos.z

newDirection = bul.position.clone().sub(bul.oldPosition).unit
    ()

vel = bul.velocity
vx0 = vel.x
vy0 = vel.y
vz0 = vel.z

decellScaler = bulletDecelleration(z0 , vel.length())
if decellScaler < 0 :
    decellScaler = -decellScaler
    decellVec = newDirection.clone().negate().multiplyScaler(
        decellScaler)
else:
    decellVec = newDirection.clone().multiplyScalar(
        decellScaler)

vx = velocity(vx0, decellVec.x, sec)
vy = velocity(vy0, decellVec.y, sec)
vz = velocity(vz0, (decellVec.z + -32.2) , sec)
```

```
    newPositionX = displacementTime(x0, (vx0+vx)/2.0, decellVec.x,
        sec) + windVec.x
    newPositionY = displacementTime(y0, (vy0+vy)/2.0, decellVec.y,
        sec) + windVec.y
    newPositionZ = displacementTime(z0, (vz0+vz)/2.0, (decellVec.z
        + -32.2), sec)

    newBullet = Bullet()
    newBullet.oldPosition = bul.position
    newBullet.gpvWithBank = bul.gpvWithBank
    newBullet.mach = knots2mach(feet2knot(vel.length()),
       tempAtAltitudeC(newPositionZ))
    newBullet.decelVec = decellVec
    newBullet.position  = vec(newPositionX, newPositionY,
       newPositionZ)
    newBullet.velocity = vec(vx, vy, vz)

    return newBullet

def createBulletPathArray(heading, distance, y, height, lag,
   depression, TAS, bank, windSpeed, windDir):
    firstBul = bulletMaker(heading, distance, y, height, -lag,
       depression, TAS, bank)
    secondBul = changeBulletFirst(firstBul, 0.1, windSpeed,
       windDir)
    bulletContainer = []
    bulletContainer.append(firstBul)
```

```
    bulletContainer.append(secondBul)


    currentLocation = 2
    while True:
        newBullet = changeBulletRest(bulletContainer[
            currentLocation-1], 0.1, windSpeed, windDir)
        bulletContainer.append(newBullet)
        currentLocation = currentLocation + 1
        if ((bulletContainer[currentLocation-1]).position).z < 0.0
            :
            break

    return bulletContainer

def calcImpactPoint(bulletContainer):
    numberOfLines = len(bulletContainer)
    lastPoint = bulletContainer[numberOfLines - 1].position
    secondLastPoint = bulletContainer[numberOfLines - 2].position
    lineVec = lastPoint.clone().sub(secondLastPoint)
    x = -secondLastPoint.z / lineVec.unit().z
    impact = secondLastPoint.clone().add(lineVec.unit().
        multiplyScalar(x))
    return impact


def plotBulletArray(bc):
    x = []
```

```
y = []
z = []

for p in bc:
    x.append(p.position.x)
    y.append(p.position.y)
    z.append(p.position.z)

impact = calcImpactPoint(bc)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c='r', marker='o')
ax.scatter(impact.x, impact.y, impact.z, c='b', marker='x')

ax.quiver([bc[0].position.x, bc[0].position.x],
          [bc[0].position.y, bc[0].position.y],
          [bc[0].position.z, bc[0].position.z],
          [bc[0].gpvWithBank.x, bc[0].velocityAC.x],
          [bc[0].gpvWithBank.y, bc[0].velocityAC.y],
          [bc[0].gpvWithBank.z, bc[0].velocityAC.z],
          length = 100, pivot = 'tail')
plt.autoscale(False)
plt.show()
```

Appendix C

Neural Network Code

```
import FireControl as fc
import math
from scipy import optimize
from random import randint
import tensorflow as tf


# heading, distance, y, height, lag, depression, TAS, bank,
    windSpeed, windDir
heading = fc.vec(0,1,0)
distance = 7179
y = 0
height = 9000
lag = -222
dep = -411
tas = 190
bank = -24
windSpeed = 0
windDir = 0

lagMax = 12.0
lagMin = -392.0

depMax = -314.0
depMin = -628.0
```

```
bulletArray = fc.createBulletPathArray(heading, distance, y,
    height,
                                        lag, dep, tas, bank,
                                            windSpeed, windDir)

impactPoint = fc.calcImpactPoint(bulletArray)

def fBoth(input):
    ba = fc.createBulletPathArray(heading, distance, y, height,
                                    input[0], input[1], tas, bank,
                                        windSpeed, windDir)
    impact = fc.calcImpactPoint(ba)

    return [impact.x, impact.y]

lagFix, depFix = optimize.fsolve(fBoth, x0=[lag, dep])

eachMovementOfPlane = []

lagUsed = []
depUsed = []

lagDiff = []
depDiff = []
```

```
impactPoints = []
impactX = []
impactY = []

timeStep = []


for i in range(0, 1000, 1):

    lDiff = randint(-20, 20)
    dDiff = randint(-20, 20)
    bulletArray = fc.createBulletPathArray(heading, distance, y,
        height,
                                           lagFix + lDiff, depFix
                                              + dDiff, tas, bank,
                                              windSpeed, windDir)
    eachMovementOfPlane.append(bulletArray)
    lagUsed.append(lagFix + lDiff)
    depUsed.append(depFix + dDiff)
    lagDiff.append(lDiff)
    depDiff.append(dDiff)
    timeStep.append(i)
    impactPoints.append(fc.calcImpactPoint(bulletArray))
    impactX.append(impactPoints[i].x)
    impactY.append(impactPoints[i].y)
```

```
def scaleArrayDown(x):
    newX = []
    for element in x:
        newX.append((element - min(x)) / (max(x) - min(x)))
    return newX


def scaleBetween(unscaledNum, minDomain, maxDomain, minRange,
    maxRange):
    return (maxDomain - minDomain) * (unscaledNum - minRange) / (
        maxRange - minRange) + minDomain


def scale(x, minDomain, maxDomain, minRange, maxRange):
    newX = []
    for element in x:
        newX.append(scaleBetween(element,minDomain, maxDomain,
            minRange, maxRange))
    return newX


# This will be the input vectors to the neural net
scaledLagUsed = scale(lagUsed, 0.0, 1.0, lagMin, lagMax)
scaledDepUsed = scale(depUsed, 0.0, 1.0, depMin, depMax)

# These will be the correct output vectors
scaledLagDiff = scale(lagDiff,  0.0, 1.0, -20.0, 20.0)
```

```
scaledDepDiff = scale(depDiff,  0.0, 1.0, -20.0, 20.0)

trainLag = scaledLagUsed[100:len(scaledLagUsed)]
testLag = scaledLagUsed[0:100]

trainDep = scaledDepUsed[100:len(scaledDepUsed)]
testDep = scaledDepUsed[0:100]

trainLagDiff = scaledLagDiff[100:len(scaledLagDiff)]
testLagDiff = scaledLagDiff[0:100]

trainDepDiff = scaledDepDiff[100:len(scaledDepDiff)]
testDepDiff = scaledDepDiff[0:100]

inputData = []
outputData = []
testInput = []
testOutput = []

for i in range(0, len(testLag), 1):
    testInput.append([testLag[i], testDep[i]])
    testOutput.append([testLagDiff[i], testDepDiff[i]])

for i in range(0, len(trainLag), 1):
    inputData.append([trainLag[i], trainDep[i]])
    outputData.append([trainLagDiff[i], trainDepDiff[i]])
```

```
# Parameters
learning_rate = 0.001
training_epochs = 5000
batch_size = 100
display_step = 1

# Network Parameters
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 2 # Lag and Depression
n_output = 2 # Lag correction, depression correction

# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_output])


# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with tanh activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.tanh(layer_1)
    # Hidden layer with tanH activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2
        '])
    layer_2 = tf.nn.tanh(layer_2)
    # Output layer with linear activation
```

```
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_output]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_output]))
}

# Construct model
pred = multilayer_perceptron(x, weights, biases)

# Define loss and optimizer
cost = tf.reduce_mean(tf.abs(tf.square(tf.sub(pred , y))))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=
   learning_rate).minimize(cost)

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
```

```
sess = tf.InteractiveSession()


sess.run(init)


def predict(x, prediction):
    print "Predicted: ", [scaleBetween(prediction[0][0], -20.0,
       20.0, 0.0, 1.0),
                          scaleBetween(prediction[0][1], -20.0,
                             20.0, 0.0, 1.0)]
    print "Actual   : ", [scaleBetween(testOutput[x][0], -20.0,
       20.0, 0.0, 1.0),
                          scaleBetween(testOutput[x][1], -20.0,
                             20.0, 0.0, 1.0)]



# Training cycle
for epoch in range(training_epochs):
    avg_cost = 0.
    total_batch = int(len(inputData)/batch_size)
    # Loop over all batches
    for i in range(total_batch):
        batch_x = inputData[i*total_batch:i*total_batch+batch_size
           ]
        batch_y = outputData[i*total_batch:i*total_batch+
           batch_size]
        # Run optimization op (backprop) and cost op (to get loss
           value)
```

```
            _, c = sess.run([optimizer, cost], feed_dict={x: batch_x,
                                                          y: batch_y})
            # print c
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", \
                "{:.9f}".format(avg_cost)
print "Optimization Finished!"

# Test model
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
print sess.run(cost, feed_dict={x: [testInput[1]] , y: [testOutput
    [1]]})
# Calculate accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print "Accuracy:", accuracy.eval({x: testInput, y: testOutput})

print "Predicted: " , pred.eval(feed_dict={x: [testInput[1]]})
print "Actual   : " , testOutput[1]
```