Contents

In order to use this file, activate org-babel for ipython and press C-c C-c to execute code blocks.

Case setup

First we import number and set its random seed to a fixed number for reproducibility. We also set the keras backend to theano.

```
import numpy as np
# For reproducibility
np.random.seed(123)

from keras import backend as K
import os

def set_keras_backend(backend):

    if K.backend() != backend:
        os.environ['KERAS_BACKEND'] = backend
        import importlib
        importlib.reload(K)
        assert K.backend() == backend

set_keras_backend("theano")
```

```
net = NeuralNet(input_node_size = 784,
                output_node_size = 10,
                hidden_layers_node_size = [512])
```

```
# net.train(X_train, Y_train, epochs=6)
```

1 NeuralNet

```
class NeuralNet(objefor the ct):

    def __init__(self,
                 input_node_size = None,             # Number of nodes in input layer
                 output_node_size = None,            # Number of nodes in output laye
                 input_shape = None,
                 hidden_layers_node_size = []        # Number of nodes in each hidden
                ):
                    from keras.models import Sequential
                    self.model = Sequential()
                    from keras.layers import Dense, Dropout, Activation, Flatten, LSTM
                    # First layer requires input dimension ie input_shape
                    self.model.add(
                                   LSTM(units=64,
                                         input_dim=input_node_size
                                         )
                                   )
                    self.model.add(Activation('relu'))
                    # Add layers to model for all hidden layers
                    for node_size in hidden_layers_node_size:
                        self.model.add(
                                       Dense(units=node_size)
                                      )
                        self.model.add(Activation('relu'))
                        self.model.add(Dropout(0.3))
                    #          from keras import regularizers
                    #          self.model.add(Dense(64,
                    #                          input_dim=64,
                    #                          kernel_regularizer=regularizers.l2(0.01)
                    #                          activity_regularizer=regularizers.l1(0.0
                    #                 )
                    # Last layer requires activation to be softmax
                    self.model.add(
                                   Dense(units=output_node_size,
                                         activation='softmax'
                                         )
                                  )
                    # Compile model
```

```
                    self.model.compile(loss='categorical_crossentropy',
                                       optimizer='adam',
                                       metrics=['accuracy'])
                    #model.fit(x_train, y_train, epochs=5, batch_size=32)
    def train(self, train_x, train_y, epochs):
        self.model.fit(train_x, train_y, epochs, batch_size = 32)
    def run(self, X, Y, steps):
        metrics = []
        metrics = self.model.evaluate(X, Y, batch_size = 32, steps = steps)
        return metrics
    def label(self, X, steps):
        predictions = self.model.predict(X, batch_size = 32, steps = steps)
        return predictions
```

1.1 init

The Sequential model is a linear stack of layers. We pass in a list of layer instances to it to make a Neural Net.

```
from keras.models import Sequential
self.model = Sequential()
```

Let's import the core layers from Keras which are almost always used.

```
from keras.layers import Dense, Dropout, Activation, Flatten, LSTM
```

The model should know what input shape it should expect. For this reason, we sepcifiy an input size for the first layer.

```
# First layer requires input dimension ie input_shape
self.model.add(
               LSTM(units=64,
                     input_dim=input_node_size
                     )
               )
self.model.add(Activation('relu'))

# Add layers to model for all hidden layers
for node_size in hidden_layers_node_size:
    self.model.add(
                   Dense(units=node_size)
                  )
    self.model.add(Activation('relu'))
    self.model.add(Dropout(0.3))
```

Adding a regularizer does not improve the model

```
#          from keras import regularizers
#          self.model.add(Dense(64,
#                          input_dim=64,
#                          kernel_regularizer=regularizers.l2(0.01),
#                          activity_regularizer=regularizers.l1(0.01))
#                    )

# Last layer requires activation to be softmax
self.model.add(
               Dense(units=output_node_size,
                     activation='softmax'
                     )
              )

# Compile model
self.model.compile(loss='categorical_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
#model.fit(x_train, y_train, epochs=5, batch_size=32)
```

1.2 train

fit the model with training datasets

inputs: train_{x} - training data train_{y} - training labels epochs - number of iterations over the entirity of both the x and y data desired

returns: Nothing

```
def train(self, train_x, train_y, epochs):
    self.model.fit(train_x, train_y, epochs, batch_size = 32)
```

1.3 run

evaluates the model with test data

inputs: X - test data Y - test labels steps - number of iterations over the entire dataset before evaluation is completed

returns: metrics - the test losses as well as the metric defined in init, which in this case is accuracy

```
def run(self, X, Y, steps):
    metrics = []
    metrics = self.model.evaluate(X, Y, batch_size = 32, steps = steps)
    return metrics
```

1.4 label

predicts the labels of the data given

Inputs: X - unlabeled test data steps - number of iterations over the entire dataset before evaluation is completed

returns: predictions - a numpy array of predictions

```
def label(self, X, steps):
    predictions = self.model.predict(X, batch_size = 32, steps = steps)
    return predictions
```

2 PID Controller

2.1 PIDStruct

```
"""
Class that acts as a mutable struct
"""
class PIDStruct(object):
    def __init__(self, input_, Ki, Kp, Kd, oldError, dt, iState):
        self.input_ = input_
        self.Ki = Ki
        self.Kp = Kp
        self.Kd = Kd
        self.oldError = oldError
        self.dt = dt
        self.iState = iState
```

2.2 PID

```
"""
class where the PID is implemented
"""
class PID(object):
    def __init__(self, p_term, i_term, d_term, angle_com):
```

```
        self.p_term = p_term
        self.i_term = i_term
        self.d_term = d_term
        self.controller = PIDStruct(0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00)
        self.min_i_term = -250
        self.max_i_term = 250
        self.angle_com = angle_com
        self.frequency = 100
        self.minAngle = -65
        self.maxAngle = 30
        self.maxFrequency = 1000
        self.buffersize = 2
        self.filteredVal = 0
        self.drive = 0
        self.index = 0
        self.updatedPid = False
        self.filterBuffer = [None] * self.buffersize

    def setup(self):
        # arduino.close()
        # arduino = serial.Serial('/dev/cu.wchusbserial1420', 115200)
        # board.Servos.attach(Esc_pin)
        # board.pinMode(10, "OUTPUT")
        # board.digitalWrite(10, "LOW")
        self.controller.input_ = self.angle_com
        self.controller.Kp = self.p_term
        self.controller.Ki = self.i_term
        self.controller.Kd = self.d_term
        self.controller.dt = 1.0/self.frequency
        # arduino.write_line("press any key to arm or c to calibrate")
        # while arduino.in_waiting && arduino.read():
        # while !arduino.in_waiting
        # if arduino.read().decode('utf-8').lower() == "c":
        #     calibrate(Esc_pin)
        # else:
        #     arm(Esc_pin)

    """
    Resets the PID controller to initialized state
    """
```

```
def resetSystem(self):
    self.drive = 0
    self.updatedPid = False
    for i in range(0,self.buffersize):
        self.angle_com = 0
    self.controller.iState = 0
    self.controller.oldError = self.controller.input_ - self.angle_com

"""
updates PID values as soon as anew pitch request is made

inputs:
com - pitch request

returns:
updatedPid - boolean for if the PID has been updated or not
"""
def updatePID(self, com):

    """
    maps the given float to an integer value between out_min and out_max

    input:
    x - value to map
    in_min - min value that val is within, usually 0
    in_max - max value that val can be
    out_min - min value that val is to be mapped to
    out_max - max value that val is to be mapped to

    returns:
    mapped integer

    """
    def trymap(x, in_min, in_max, out_min, out_max):
        return int((x-in_min) * (out_max-out_min) / (in_max-in_min) + out_min)

    """
    constrains the value given to the range given
```

```
        input:
        val - the value to be constrained
        min_val - min value that val can be
        max_val - max valuse that val can be

        returns:
        value within the range given

        """
        def constrain(val, min_val, max_val):
            return min(max_val, max(min_val, val))

        pTerm, iTerm, dTerm, error = 0,0,0,0
        self.angle_com = com
        error = self.controller.input_ - self.angle_com
        pTerm = self.controller.Kp * error
        self.controller.iState += error * self.controller.dt
        self.controller.iState = constrain(self.controller.iState, self.min_i_term/sel
        iTerm = self.controller.Ki * self.controller.iState
        dTerm = self.controller.Kd * ((error - self.controller.oldError) / self.contro
        self.drive = pTerm + iTerm + dTerm
        # setSpeed(Esc_pin, self.drive)
        self.updatedPid = True
        return self.drive
```

2.2.1 setup

```
def setup(self):
    # arduino.close()
    # arduino = serial.Serial('/dev/cu.wchusbserial1420', 115200)
    # board.Servos.attach(Esc_pin)
    # board.pinMode(10, "OUTPUT")
    # board.digitalWrite(10, "LOW")
    self.controller.input_ = self.angle_com
    self.controller.Kp = self.p_term
    self.controller.Ki = self.i_term
    self.controller.Kd = self.d_term
    self.controller.dt = 1.0/self.frequency
    # arduino.write_line("press any key to arm or c to calibrate")
    # while arduino.in_waiting && arduino.read():
```

```
    # while !arduino.in_waiting
    # if arduino.read().decode('utf-8').lower() == "c":
    #     calibrate(Esc_pin)
    # else:
    #     arm(Esc_pin)
```

2.2.2 resetsystem

```
"""
Resets the PID controller to initialized state
"""

def resetSystem(self):
    self.drive = 0
    self.updatedPid = False
    for i in range(0,self.buffersize):
        self.angle_com = 0
    self.controller.iState = 0
    self.controller.oldError = self.controller.input_ - self.angle_com
```

2.2.3 updatePID

```
"""
updates PID values as soon as anew pitch request is made

inputs:
com - pitch request

returns:
updatedPid - boolean for if the PID has been updated or not
"""
def updatePID(self, com):

    """
    maps the given float to an integer value between out_min and out_max

    input:
    x - value to map
```

```
in_min - min value that val is within, usually 0
in_max - max value that val can be
out_min - min value that val is to be mapped to
out_max - max value that val is to be mapped to

returns:
mapped integer

"""
def trymap(x, in_min, in_max, out_min, out_max):
    return int((x-in_min) * (out_max-out_min) / (in_max-in_min) + out_min)

"""
constrains the value given to the range given

input:
val - the value to be constrained
min_val - min value that val can be
max_val - max valuse that val can be

returns:
value within the range given

"""
def constrain(val, min_val, max_val):
    return min(max_val, max(min_val, val))

pTerm, iTerm, dTerm, error = 0,0,0,0
self.angle_com = com
error = self.controller.input_ - self.angle_com
pTerm = self.controller.Kp * error
self.controller.iState += error * self.controller.dt
self.controller.iState = constrain(self.controller.iState, self.min_i_term/self.co
iTerm = self.controller.Ki * self.controller.iState
dTerm = self.controller.Kd * ((error - self.controller.oldError) / self.controller
self.drive = pTerm + iTerm + dTerm
# setSpeed(Esc_pin, self.drive)
self.updatedPid = True
return self.drive
```

1. trymap

```
"""
maps the given float to an integer value between out_min and out_max

input:
x - value to map
in_min - min value that val is within, usually 0
in_max - max value that val can be
out_min - min value that val is to be mapped to
out_max - max value that val is to be mapped to

returns:
mapped integer

"""
def trymap(x, in_min, in_max, out_min, out_max):
    return int((x-in_min) * (out_max-out_min) / (in_max-in_min) + out_min)
```

2. constrain

```
"""
constrains the value given to the range given

input:
val - the value to be constrained
min_val - min value that val can be
max_val - max valuse that val can be

returns:
value within the range given

"""
def constrain(val, min_val, max_val):
    return min(max_val, max(min_val, val))
```