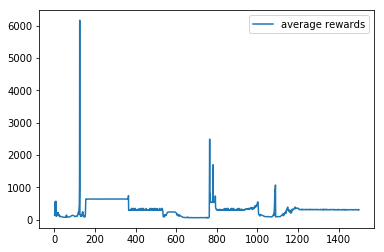
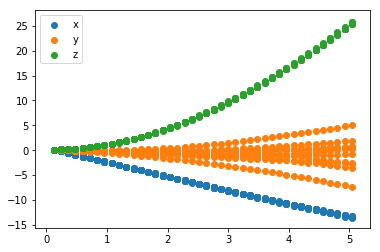
# Multiply by vz for up reward





# Remove multiply by vz up reward

Epi = 1500, score = 0.570 (best = 1.292) in epi 314 simT = 0.120

total num best epi = 8, total num epi > 1s = 98

class Actor:

"""Actor (Policy) Model."""

def \_\_init\_\_(self, state\_size, action\_size, action\_low, action\_high):

"""Initialize parameters and build model.

Params

======

state\_size (int): Dimension of each state

action\_size (int): Dimension of each action

action\_low (array): Min value of each action dimension

action\_high (array): Max value of each action dimension

"""

self.state\_size = state\_size

self.action\_size = action\_size

self.action\_low = action\_low

self.action\_high = action\_high

self.action\_range = self.action\_high - self.action\_low

# Initialize any other variables here

self.build\_model()

def build\_model(self):

"""Build an actor (policy) network that maps states -> actions."""

# Define input layer (states)

states = layers.Input(shape=(self.state\_size,), name='states')

# Add hidden layers

net = layers.Dense(units=32, activation='relu')(states)

net = layers.Dense(units=64, activation='relu')(net)

# net = layers.Dense(units=256, activation='relu')(states)

# net = layers.Dense(units=256, activation='relu')(net)

# net = layers.Dropout(0.3)(net)

# net = layers.Dense(units=256, activation='relu')(net)

# net = layers.Dropout(0.3)(net)

# net = layers.Dense(units=128, activation='relu')(net)

# net = layers.Dropout(0.3)(net)

# net = layers.Dense(units=128, activation='relu')(net)

# net = layers.Dropout(0.3)(net)

# net = layers.Dense(units=128, activation='relu')(net)

# net = layers.Dropout(0.3)(net)

# net = layers.GaussianNoise(1.0)(net)

# net = layers.BatchNormalization()(net) # (SMM) seems to help smooth results

# Try different layer sizes, activations, add batch normalization, regularizers, etc.

# Add final output layer with sigmoid activation

raw\_actions = layers.Dense(units=self.action\_size, activation='sigmoid',

name='raw\_actions')(net)

# Scale [0, 1] output for each action dimension to proper range

actions = layers.Lambda(lambda x: (x \* self.action\_range) + self.action\_low,

name='actions')(raw\_actions)

# Create Keras model

self.model = models.Model(inputs=states, outputs=actions)

# Define loss function using action value (Q value) gradients

action\_gradients = layers.Input(shape=(self.action\_size,))

loss = K.mean(-action\_gradients \* actions)

# Incorporate any additional losses here (e.g. from regularizers)

# Define optimizer and training function

optimizer = optimizers.Adam()

updates\_op = optimizer.get\_updates(params=self.model.trainable\_weights, loss=loss)

self.train\_fn = K.function(

inputs=[self.model.input, action\_gradients, K.learning\_phase()],

outputs=[],

updates=updates\_op)

class Critic:

"""Critic (Value) Model."""

def \_\_init\_\_(self, state\_size, action\_size):

"""Initialize parameters and build model.

Params

======

state\_size (int): Dimension of each state

action\_size (int): Dimension of each action

"""

self.state\_size = state\_size

self.action\_size = action\_size

# Initialize any other variables here

self.build\_model()

def build\_model(self):

"""Build a critic (value) network that maps (state, action) pairs -> Q-values."""

# Define input layers

states = layers.Input(shape=(self.state\_size,), name='states')

actions = layers.Input(shape=(self.action\_size,), name='actions')

# Add hidden layer(s) for state pathway

net\_states = layers.Dense(units=32, activation='relu')(states)

net\_states = layers.Dense(units=64, activation='relu')(net\_states)

# net\_states = layers.Dense(units=256, activation='relu')(states)

# net\_states = layers.Dropout(0.5)(net\_states)

# net\_states = layers.Dense(units=256, activation='relu')(net\_states)

# net\_states = layers.Dropout(0.5)(net\_states)

# net\_states = layers.Dense(units=128, activation='relu')(net\_states)

# net\_states = layers.Dropout(0.5)(net\_states)

# net\_states = layers.Dense(units=128, activation='relu')(net\_states)

# net\_states = layers.Dropout(0.5)(net\_states)

# net\_states = layers.Dense(units=128, activation='relu')(net\_states)

# net\_states = layers.Dropout(0.5)(net\_states)

# net\_states = layers.BatchNormalization()(net\_states) #(SMM)

# Add hidden layer(s) for action pathway

net\_actions = layers.Dense(units=32, activation='relu')(actions)

net\_actions = layers.Dense(units=64, activation='relu')(net\_actions)

# net\_actions = layers.Dense(units=256, activation='relu')(actions)

# net\_states = layers.Dropout(0.5)(net\_states)

# net\_actions = layers.Dense(units=256, activation='relu')(net\_actions)

# net\_actions = layers.Dropout(0.5)(net\_actions)

# net\_actions = layers.Dense(units=128, activation='relu')(net\_actions)

# net\_actions = layers.Dropout(0.5)(net\_actions)

# net\_actions = layers.Dense(units=128, activation='relu')(net\_actions)

# net\_actions = layers.Dropout(0.5)(net\_actions)

# net\_actions = layers.Dense(units=128, activation='relu')(net\_actions)

# net\_actions = layers.Dropout(0.5)(net\_actions)

# net\_actions = layers.BatchNormalization()(net\_actions) #(SMM)

# Try different layer sizes, activations, add batch normalization, regularizers, etc.

# Combine state and action pathways

net = layers.Add()([net\_states, net\_actions])

# net = layers.Activation('softmax')(net)

net = layers.Activation('relu')(net)

# Add more layers to the combined network if needed

# Add final output layer to prduce action values (Q values)

Q\_values = layers.Dense(units=1, name='q\_values')(net)

# Create Keras model

self.model = models.Model(inputs=[states, actions], outputs=Q\_values)

# Define optimizer and compile model for training with built-in loss function

optimizer = optimizers.Adam()

self.model.compile(optimizer=optimizer, loss='mse')

# Compute action gradients (derivative of Q values w.r.t. to actions)

action\_gradients = K.gradients(Q\_values, actions)

# Define an additional function to fetch action gradients (to be used by actor model)

self.get\_action\_gradients = K.function(

inputs=[\*self.model.input, K.learning\_phase()],

outputs=action\_gradients)