```
class Agent():
 def __init__(self, ff_inputsize):
  self.ff_inputsize = ff_inputsize
  self.model = DDDQN_model
  self.memory = Memory(10000, self)  #for definition see code
  self.action_repeat = 4
  self.update_frequency = 4
  self.batch_size = 32
  self.replaystartsize = 1000
  self.epsilon = 0.05
  self.last_action = None
  self.repeated_action_for = self.action_repeat

 def runInference(self, gameState, pastState):
  self.addToMemory(gameState, pastState)
  ff_inputs = self.getAgentState(*gameState)
  self.repeated_action_for += 1
  if self.repeated_action_for < self.action_repeat:
   toUse, toSave = self.last_action
  else:
   self.repeated_action_for = 0
   if self.canLearn() and np.random.random() > self.epsilon:
    action, _ = self.model.inference(ff_inputs)
    toSave = self.dediscretize(action[0])
    toUse = "["+str(throttle)+", "+str(brake)+", "+str(steer)+"]"
   else:
    toUse, toSave = self.randomAction() #for definition see code
   self.last_action = toUse, toSave
  self.containers.outputval.update(toUse, toSave)
  self.numsteps += 1
  if self.numsteps % self.update_frequency == 0 and len(self.memory) > self.replaystartsize):
   self.learnStep()

 def learnStep(self):
  QLearnInputs = self.memory.sample(self.batch_size)
  self.model.q_learn(QLearnInputs)

 def addToMemory(self, gameState, pastState):
   s = self.getAgentState(*pastState)  #for definition see code
   a = self.getAction(*pastState)   #for definition see code
   r = self.calculateReward(*gameState)#for definition see code
   s2= self.getAgentState(*gameState)  #for definition see code
   t = False #will be updated if episode did end
   self.memory.append([s,a,r,s2,t])



class DuelDQN():
 def __init__(self, name, ff_inputsize, num_actions):
  with tf.variable_scope(name, initializer = tf.random_normal_initializer(0, 1e-3)):
   #for the inference
   self.ff_inputs = tf.placeholder(tf.float32, shape=[None, ff_inputsize], name="ff_inputs")
   self.fc1 = tf.layers.dense(self.ff_inputs, 400, activation=tf.nn.relu)
   #modifications from the Dueling DQN architecture
   self.streamA, self.streamV = tf.split(self.fc1,2,1)
   xavier_init = tf.contrib.layers.xavier_initializer()
```

```
   neutral_init = tf.random_normal_initializer(0, 1e-50)
   self.AW = tf.Variable(xavier_init([200,self.num_actions]))
   self.VW = tf.Variable(neutral_init([200,1]))
   self.Advantage = tf.matmul(self.streamA,self.AW)
   self.Value = tf.matmul(self.streamV,self.VW)
   self.Qout = self.Value + tf.subtract(self.Advantage,tf.reduce_mean(self.Advantage,axis=1,keep_dims=True))
   self.Qmax = tf.reduce_max(self.Qout, axis=1)
   self.predict = tf.argmax(self.Qout,1)
   #for the learning
   self.targetQ = tf.placeholder(shape=[None],dtype=tf.float32)
   self.targetA = tf.placeholder(shape=[None],dtype=tf.int32)
   self.targetA_OH = tf.one_hot(self.targetA, self.num_actions, dtype=tf.float32)
   self.compareQ = tf.reduce_sum(tf.multiply(self.Qout, self.targetA_OH), axis=1)
   self.td_error = tf.square(self.targetQ - self.compareQ)
   self.q_loss = tf.reduce_mean(self.td_error)
   q_trainer = tf.train.AdamOptimizer(learning_rate=0.00025)
   q_OP = q_trainer.minimize(self.q_loss)
  self.trainables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=name)


def _netCopyOps(fromNet, toNet, tau = 1):
 op_holder = []
 for idx,var in enumerate(fromNet.trainables[:]):
  op_holder.append(toNet.trainables[idx].assign((var.value()*tau) + ((1-tau)*toNet.trainables[idx].value())))
 return op_holder
```

Initialize replay memory D to capacity N

Initialize action-value function $Q(s, a; \theta)$ with random weights θ

Initialize target action-value function $Q(s, a; \theta^-)$ with weights $\theta^- = \theta$

For episode = 1,M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For 1 = 1,T **do**

    With probability ϵ select random action a_t

    otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Define $a^{max}(\phi_{j+1}; \theta) = argmax_{a'} Q(\phi_{j+1}, a'; \theta)$

    Define $Q^{j+1} = Q(\phi_{j+1}, a^{max}(\phi_{t+1}; \theta); \theta^-)$

    If episode terminates at step $j+1$ **then** set $y_j = r_j$,
      ➥ **Otherwise** set $y_j = r_j + \gamma * Q^{j+1}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to
      ➥ the network parameters θ

    Update target network: $\theta^- \leftarrow \tau * \theta + (1 - \tau)\theta^-$

  End For

End For

```
#see agent
class DDDQN_model():
 def __init__(self, session, ff_inputsize, num_action):
  self.session = session
  self.onlineQN = DuelDQN("onlineNet", ff_inputsize, num_action)
  self.targetQN = DuelDQN("targetNet", ff_inputsize, num_action)
  self.session.run(tf.global_variables_initializer())
  self.session.run(_netCopyOps(self.targetQN, self.onlineQN))

 #see agent
 def inference(self, statesBatch): #called for every step t

  return self.session.run([self.onlineQN.predict, self.onlineQN.Qout], feed_dict={self.onlineQN.
      ➥ ff_inputs: statesBatch})
 #see agent
 #see agent
 #see agent
 def q_learn(self, batch): #also called for every step t
  oldstates, actions, rewards, newstates, terminals = batch
  action = self.session.run(self.onlineQN.predict,feed_dict={self.onlineQN.ff_inputs:newstates})
  folgeQ = self.session.run(self.targetQN.Qout,feed_dict={self.targetQN.ff_inputs:newstates})
  doubleQ = folgeQ[range(len(terminals)),action]
  consider_stateval = -(terminals - 1)
  targetQ = rewards + (0.99 * doubleQ * consider_stateval)
  self.session.run(self.onlineQN.q_OP, feed_dict={self.onlineQN.ff_inputs:oldstates, self.onlineQN.
      ➥ targetQ:targetQ, self.onlineQN.targetA:actions})
  self.session.run(_netCopyOps(self.onlineQN, self.targetQN, 0.001))
  return
```