

Fecha:
Abril 2025

ANALISIS DE IMPLEMENTACION DEL DDQN EN EL GATO

Realizado por:
Garcia chavez Erik 01275863
Inteligencia artificial
Juan Ramon

INTRODUCCIÓN

¿porque el gato?

porque la implementación del algoritmo del DDQN en el juego del gato, este tipo de algoritmo en juegos siempre buscara el ganar o al peor escenario el empatar tanto con otra maquina o con un ser humano, aunque el juego tiene un espacio de estado pequeños, el DDQN demuestra su eficiencia al aprender y evita errores comunes, estos por la mejora que evita la sobreestimación heredada del Q-learning original

la implementación central se encuentra en el archivo dqn.py dentro del repositorio el cual se apoya de memory_buffers.py el cual se usa para almacenar el muestreo de experiencias pasadas del agente durante el entrenamiento

Podemos ver el constructor

```
1
2 class DeepQNetworkModel:
3     def __init__(self,
4                   session,
5                   layers_size,
6                   memory,
7                   default_batch_size=None,
8                   default_learning_rate=None,
9                   default_epsilon=None,
10                  gamma=0.99,
11                  min_samples_for_predictions=0,
12                  double_dqn=False,
13                  learning_procedures_to_q_target_switch=1000,
14                  tau=1,
15                  maximize_entropy=False,
16                  var_scope_name=None):
```

parámetros claves que podemos resaltar del constructor, al código funciona de manera por default como DEEP Q-LEARNING pero cuenta con un parámetro llamado “double_dqn” el cual es de tipo BOOL y active la segunda red neuronal para realizar ese modelo.

“maximize_entropy” si es true maximiza la entropía de los valores Q para fomentar la exploración

creacion de redes neuronales (dqn.py)

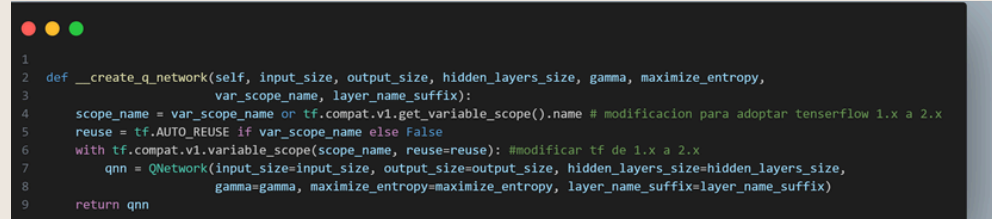
la construcción de las redes neuronales se hacen mediante la funcion `__create_q_network` que define la arquitectura de la red neuronal

tenemos parámetros clave que son “input_size” que es el tamaño de la capa de entrada

“output_size” el tamaño de la capa de salida este va a depender al número de acciones posibles

“hidden_layers_size” lista con el número de nodos en la capa oculta

“var_scope_name” ámbito de variables en tensorflow para compartir pesos entre redes, estas redes se construyen usando capas densas de tensorflow/keras



```
1
2 def __create_q_network(self, input_size, output_size, hidden_layers_size, gamma, maximize_entropy,
3     var_scope_name, layer_name_suffix):
4     scope_name = var_scope_name or tf.compat.v1.get_variable_scope().name # modificacion para adoptar tensorflow 1.x a 2.x
5     reuse = tf.AUTO_REUSE if var_scope_name else False
6     with tf.compat.v1.variable_scope(scope_name, reuse=reuse): #modificar tf de 1.x a 2.x
7         qnn = QNetwork(input_size=input_size, output_size=output_size, hidden_layers_size=hidden_layers_size,
8             gamma=gamma, maximize_entropy=maximize_entropy, layer_name_suffix=layer_name_suffix)
9     return qnn
```

el tamaño de la capa de entrada de una red neuronal se refiere al numero de neuronas de la primera capa y esta determinao directamente por la estrucutra de los datos de entrada.

capas densas: se les conoce porque cada neurona esta conectada a todas las neuronas de la capa anterior.

detalles en class QNetwork (dqn.py)

si la entropía esta “habilitada” entonces incentiva distribuciones de valores Q más diversificadas lo que entraría en el IF

en caso contrario entraría en el else donde self.future_q calcula el valor Q máximo para el próximo estado según la red objetivo

con “self.labels = self.r + (gamma * self.future_q)” que define los valores Q objetivo usando la ecuación de bellman

calcula el costo el cual mide el error entre las predicciones de la red principal (self.predictions) y los Q-tarets (self.labels) calcula el error cuadrático medio (MSE) mediante la función mean_squared_error

```
1
2 if maximize_entropy:
3     self.future_q = tf.log(tf.reduce_sum(tf.exp(self.q_target), axis=1))
4 else:
5     self.future_q = tf.reduce_max(self.q_target, axis=1)
6 self.labels = self.r + (gamma * self.future_q)
7 self.cost = tf.reduce_mean(tf.losses.mean_squared_error(labels=self.labels, predictions=self.predictions))
8 self.optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate).minimize(self.cost)
```

$$Q_{soft}(s, a) = r(s, a) + \gamma \log \left(\sum_{a'} e^{Q(s', a')} \right)$$

self.future_q el valor q maximo del proximo estado

detalles en class QNetwork (dqn.py)

utiliza el optimizador de adam para minimizar el error cuadrático medio (MSE), por lo que se utiliza para actualizar los pesos de la Q-network de manera eficiente. la ventaja que tiene adam es que este suaviza los gradientes usando momentos, lo que estabiliza el entrenamiento. al no maximizar la entropía, el modelo depende aun más de la eficiencia del optimizador para encontrar los valores Q lo que hace a adam un modelo ideal para el caso.

```
1
2 if maximize_entropy:
3     self.future_q = tf.log(tf.reduce_sum(tf.exp(self.q_target), axis=1))
4 else:
5     self.future_q = tf.reduce_max(self.q_target, axis=1)
6 self.labels = self.r + (gamma * self.future_q)
7 self.cost = tf.reduce_mean(tf.losses.mean_squared_error(labels=self.labels, predictions=self.predictions))
8 self.optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate).minimize(self.cost)
```

creacion de la red neuronal con tensorflow/keras (dqn.py)

las primrea líneas lo que crean son placeholders que guardaran valores para nuestra red neuronal:

self.q_tarjet: se utiliza para almacenar los valores Q que se utilizna para la actualizacion de pesos d ela red.

self.r: son las recompensas inmediatas recibidad depsuesde tomar una accion. con "None" puede albregar valores escalares de recompensas para cada muestra en el batch

self.states: alamacena los estados del juego que seran utilizados como entrada para la red.

self.enumerated_actions: contiene las acciones tomadas, donde cada acción esta indexado juego con su posición en el batch, su forma es (None, 2) donde cada fila contiene un índice de batch y un índice de acción

slef.learnig_rate: es un placeholder para la tasa de aprenizaje que se utilizara durante la optimizacion

se inicializa una variable "layer" con self.states que representan la capade entrada. se itera sobre "hiden_layer_size" para construir las caras ocultas

```
1 self.q_target = tf.compat.v1.placeholder(shape=(None, output_size), dtype=tf.float32)
2 self.r = tf.compat.v1.placeholder(shape=None, dtype=tf.float32)
3 self.states = tf.compat.v1.placeholder(shape=(None, input_size), dtype=tf.float32)
4 self.enumerated_actions = tf.compat.v1.placeholder(shape=(None, 2), dtype=tf.int32)
5 self.learning_rate = tf.compat.v1.placeholder(shape=[], dtype=tf.float32)
6 layer = self.states
```

self.q_target: shape)= none, outside, permiteun numero varibale de filas y el outsize es el numero posible de acciones

capas ocultas - capas de salida (dqn.py)

utilizando `tf.keras.layers.Dense` se crea una capa densa, donde se especifica el número de unidades, la función de activación y el inicializador de pesos (`kernel_initializer`), se aplica la capa al tensor

se crea la capa de salida: de igual forma usando la misma función, esta capa tiene un “`output_size`” números posibles de acciones, su nombre y el inicializador de pesos.

se aplica la capa al tensor (`layer`) que es la ultima capa oculta para obtener (`self.output`) que representa los valores de Q posibles para cada acción

```
dense_layer = tf.keras.layers.Dense(
    units=hidden_layers_size[i],
    activation=tf.nn.relu,
    name=f'{layer_name_suffix}_dense_layer_{i}',
    kernel_initializer=tf.keras.initializers.GlorotNormal()
)
layer = dense_layer(layer) # Aplica la capa al tensor anterior

# Para la capa de salida
output_layer = tf.keras.layers.Dense(
    units=output_size,
    name=f'{layer_name_suffix}_dense_layer_{len(hidden_layers_size)}',
    kernel_initializer=tf.keras.initializers.GlorotNormal() # Reemplaza tf.contr
)
self.output = output_layer(layer) # Aplica la capa al tensor anterior
self.predictions = tf.gather_nd(self.output, indices=self.enumerated_actions)
```

se calcula la predicción usando la función `tf.gather_nd` para obtener los valores Q correspondientes a las acciones tomadas especificadas en el parámetro, “`indices=self.enumerated_actions`”

learn (dqn.py)

la función “learn” es clave para el entrenamiento del modelo, veremos los puntos importantes de esta función

el entrenamiento se realiza solo cuando `self.memory.counter` es múltiplo de `current_batch_size` lo que garantiza que se acumulen suficientes experiencias en la memoria antes de entrenar

```
current_batch_size = batch_size if batch_size is not None else self.default_batch_size
if self.memory.counter % current_batch_size != 0 or self.memory.counter == 0:
    logging.debug('Passing on learning procedure')
    pass
```

muestreo de experiencias. la función de este muestreo aleatorio es romper correlaciones temporales, si el modelo se entrena con datos en el orden exacto en que ocurrieron, aprenderá patrones específicos de secuencias temporales lo que genera sobreajuste e inestabilidad. por lo que al mostrar un conjunto de experiencias de la memoria se mezclan las transiciones

learn (dqn.py)

mejora la estabilidad del aprendizaje. las mismas experiencias se usan multiples veces para entrenar a la red. lo que es muy rentable en entornos donde es costoso el recolectar los datos.

```
logging.debug('Starting learning procedure...')
batch = self.memory.sample(current_batch_size)
qt = self.session.run(self.target_q_network.output,
                      feed_dict={self.target_q_network.states: self.__fetch_from_batch(batch, 'next_state')})
terminals = self.__fetch_from_batch(batch, 'is_terminal')
```

entrenamiento de la red principal (q_network)

```
lr = learning_rate if learning_rate is not None else self.default_learning_rate
_, cost = self.session.run([self.q_network.optimizer, self.q_network.cost],
                          feed_dict={self.q_network.states: self.__fetch_from_batch(batch, 'state'),
                                      self.q_network.action: self.__fetch_from_batch(batch, 'reward'),
                                      self.(Variable) q_network: QNetwork (f.__fetch_from_batch(batch, 'action', enum=True),
                                      self.q_network.q_target: qt,
                                      self.q_network.learning_rate: lr)})
logging.debug('Batch number: %s | Q-Network cost: %s | Learning rate: %s',
              self.memory.counter // current_batch_size, cost, lr)
```

lo que se esta viendo es la alimentacion de los datos, con los datos de los estados actuales, las recompensas, las acciones tomadas que seria el enumerated_actions, asi como el uso de valores Q calculador previamente. ejecuta el optimizador par actualizar los pesos de la red Q y calcula el costo, los cuales en el programa se usa ADAM y MSE respectivamente.

learn (dqn.py)

el proceso de actualización periódica funciona para estabilizar el entrenamiento en donde si la red actualiza abruptamente los valores Q objetivo, este va a generar una inestabilidad en el aprendizaje, pero si gradualmente se mezclan los pesos de la red principal con los de la red objetivo usando el parámetro tau, y usando la siguiente formula:

$$Q(s, a; \theta) = \tau \cdot Q(s, a; \theta) + (1 - \tau) \cdot Q(s, a; \theta)$$

también ayuda a la sobreestimación de valores Q que es un problema en DQN que trata de solucionar DDQN

accede a las variables entrenables de tensorflow, divide las variables en dos mitades, las primeras para q_network y las segundas para la red objetivo (target_q_network) aplica la mezcla con tau usando tf.variable.assign()

```
tf_vars = tf.trainable_variables()
num_of_vars = len(tf_vars)
operations = []
for i, v in enumerate(tf_vars[0:num_of_vars // 2]):
    operations.append(tf_vars[i + num_of_vars // 2].assign(
        (v.value() * self.tau) + ((1 - self.tau) * tf_vars[i + num_of_vars // 2].value())))
```

funcion act (dqn.py)

el propósito de la función es seleccionar una acción para un estado dado siguiendo una política ϵ -greedy para equilibrar la exploración y explotación.

la exploración se realiza con probabilidad ϵ , elige una acción aleatoria y la explotación con probabilidad $1-\epsilon$ elige la acción con el valor Q mayor predicho por la red neuronal.

```
eps = epsilon if epsilon is not None else self.default_epsilon
rnd = random.random()
if rnd < eps or self.memory.counter < self.min_samples_for_predictions:
    action = random.randint(0, self.output_size - 1)
    logging.debug("Choosing a random action: %s [Epsilon = %s]", action, eps)
else:
    prediction = self.session.run(self.q_network.output,
                                  feed_dict={self.q_network.states: np.expand_dims(state, axis=0)})
    prediction = np.squeeze(prediction)
    action = np.argmax(prediction)
    logging.debug("Predicted action for state %s is %s (network output: %s) [Epsilon = %s]",
                  state, action, prediction, eps)
return action
```

funcion add_to_memory (dqn.py)

el propósito es almacenar transiciones (state, action, reward, next_state, is_terminal) en la memoria de experiencia

esta nos permite reutilizar experiencias pasadas durante el entrenamiento. si es un estado terminal se ignora el siguiente estado, lo hay acciones futuras

```
def add_to_memory(self, state, action, reward, next_state, is_terminal_state):  
    """  
    Add new state-transition to memory  
    :param state: a Numpy array representing a state  
    :param action: an integer representing the selected action  
    :param reward: a number representing the received reward  
    :param next_state: a Numpy array representing the state reached after performing the action  
    :param is_terminal_state: boolean. mark state as a terminal_state. next_state will have no effect.  
    """  
    self.memory.append({'state': state, 'action': action, 'reward': reward,  
                        'next_state': next_state, 'is_terminal': is_terminal_state})
```

funcion __fetch_from_batch (dqn.py)

el proposito es estrar valores especificos de un lote de datos para alimentar la red neuronal durante el entranamieto.

```
def __fetch_from_batch(self, batch, key, enum=False):
    if enum:
        return np.array(list(enumerate(map(lambda x: x[key], batch))))
    else:
        return np.array(list(map(lambda x: x[key], batch)))
```

repositorio del autor

