DQN

Hong Xingxing

October 18, 2018

Outline

Q Function Update Equations

- MC(Monte Carlo)

$$Q(s,a) \leftarrow Q(s,a) + \alpha(G_t - Q(s,a)) \tag{1}$$

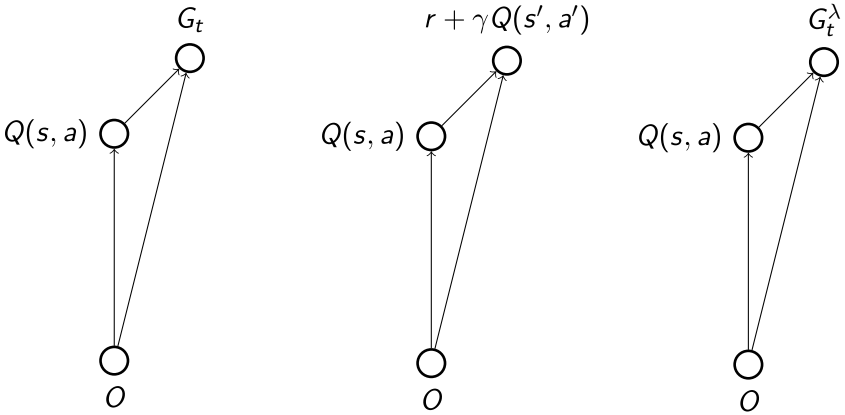
- TD(Temporal Difference)

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma Q(s',a') - Q(s,a)\big] \tag{2}$$

- TD(λ)

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[G_t^{\lambda} - Q(s,a)\big] \tag{3}$$

Q Function Update



$$\arg\min_\theta \left(Q(s,a) - \hat{Q}(s,a,\theta)\right)^2 \tag{4}$$

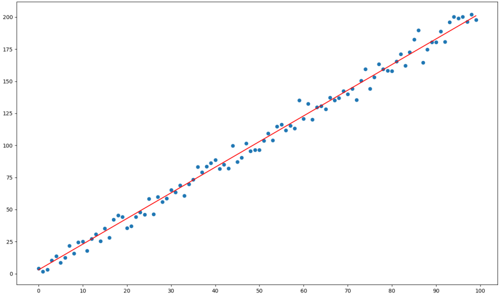
Functions to Approxmiate $\hat{v}(s, \theta)$

- Using functions to approximate $\hat{v}(s, \theta)$ can be regarded as *a Supervised Learning task*
- Then $(Input, Label)$ pairs are (S_t, U_t) in the settings where

$$U_t \simeq \begin{cases} G_t & MC \\ r + \gamma Q(s', a') & TD \\ G_t^{\lambda} & TD(\lambda) \end{cases} \tag{5}$$

- Tabular methods: update each entry (estimated value) of the table separately
- Approximation methods: update parameters θ iteratively then each entry could be updated implicitly

Linear Regression Task and Supervised Learning

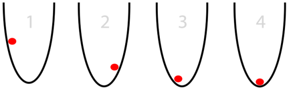



```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0,100)
noise = np.random.normal(0,5,100)
y = 2*x+3+noise
plt.scatter(x, y)
plt.plot(x, 2*x+3, color='red')
F = open("data.csv","w")
for i in range(100):
    line=str(x[i])+"\t"+str(y[i])
    F.write(line+"\n")
plt.xticks(np.arange(0,105,10))
plt.show()
```

$$Y = W * x + b \tag{6}$$

How to estimate W and b?

Linear Regression Task and Supervised Learning



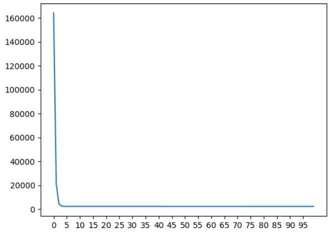
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad (7)$$

$$\frac{\partial}{\partial W} = \frac{2}{n}\sum_{i=1}^{n} -x_i(y_i - (Wx_i + b))$$

$$\frac{\partial}{\partial b} = \frac{2}{n}\sum_{i=1}^{n} -(y_i - (Wx_i + b))$$

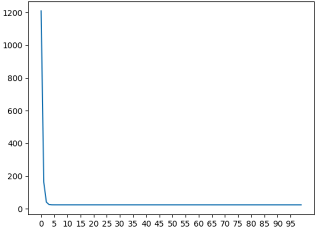
Linear Regression Task and Supervised Learning

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def LR_GradientDescent():
    df=pd.read_csv('data.csv', sep='\t',header=None)
    points = df.values
    N = len(points)
    learning_rate = 0.0001
    num_iterations = 100
    curr_b = 0.0
    curr_m = 0.0
    for epoch in range(num_iterations):
        grad_b = 0
        grad_m = 0
        for i in range(0, N):
            x = float(points[i][0])
            y = float(points[i][1])
            grad_b += -(2.0/N)*(y-((curr_m*x)+curr_b))
            grad_m += -(2.0/N)*x*(y-((curr_m*x)+curr_b))
        curr_b = curr_b-(learning_rate*grad_b)
        curr_m = curr_m-(learning_rate*grad_m)
        totalError = 0
        for i in range(0, N):
            x = float(points[i][0])
            y = float(points[i][1])
            totalError += (y - (curr_m * x + curr_b)) ** 2
        print(epoch, curr_b, curr_m, totalError/N)
```



Linear Regression Task and Supervised Learning

```
import numpy as np
import tensorflow as tf
import pandas as pd

W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)

loss = tf.reduce_mean(tf.square(linear_model - y))
optimizer = tf.train.GradientDescentOptimizer(0.0001)
train = optimizer.minimize(loss)

df=pd.read_csv('data.csv', sep='\t',header=None)
points = df.values
x_train = [float(p[0]) for p in points]
y_train = [float(p[1]) for p in points]

init = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init)
for i in range(1000):
    sess.run(train, {x:x_train, y:y_train})
    curr_W, curr_b, curr_loss = sess.run([W, b, loss],
 ↪    {x:x_train, y:y_train})
    print("W: %s b: %s loss: %s" %(curr_W, curr_b, curr_loss))
```



Gradient Descent and Derivative

Function $J(\theta)$ with n parameters has a first order derivative vector:

$$\frac{\partial J}{\partial \theta} = [\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, ..., \frac{\partial J}{\partial \theta_n}] \tag{8}$$

Second derivative matrix is also called Hessian Matrix:

$$\begin{bmatrix} \frac{\partial^2 J}{\partial^2 \theta_1} & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_2} & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_3} & \cdots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 J}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 J}{\partial^2 \theta_2} & \frac{\partial^2 J}{\partial \theta_2 \partial \theta_3} & \cdots & \frac{\partial^2 J}{\partial \theta_2 \partial \theta_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 J}{\partial \theta_n \partial \theta_2} & \frac{\partial^2 J}{\partial \theta_n \partial \theta_3} & \cdots & \frac{\partial^2 J}{\partial^2 \theta_n} \end{bmatrix}$$

Hessian Matrix

```
import tensorflow as tf
import numpy as np
def cons(x):
    return tf.constant(x, dtype=tf.float32)
def compute_hessian(fn, vars):
    mat = []
    for v1 in vars:
        temp = []
        for v2 in vars:
            # computing derivative twice, first w.r.t v2 and then w.r.t v1
            temp.append(tf.gradients(tf.gradients(f, v2)[0], v1)[0])
        temp = [cons(0) if t == None else t for t in temp]
        # tensorflow returns None when there is no gradient, so we replace None with 0
        temp = tf.stack(temp)
        mat.append(temp)
    mat = tf.stack(mat)
    return mat
x = tf.Variable(np.random.random_sample(), dtype=tf.float32)
y = tf.Variable(np.random.random_sample(), dtype=tf.float32)

f = tf.pow(x, cons(2)) + cons(2) * x * y + cons(3) * tf.pow(y, cons(2)) + cons(4)* x + cons(5) * y +
 ↪    cons(6)
hessian = compute_hessian(f, [x, y])
sess = tf.Session()
sess.run(tf.global_variables_initializer())
print(sess.run(hessian))
```

$$q(x1, x2) = x_1^2 + 2x_1x_2 + 3x_2^2 + 4x_1 + 5x_2 + 6$$

Gradient Mente Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \big[G_t - \hat{v}(S_t,\mathbf{w})\big] \nabla \hat{v}(S_t,\mathbf{w})$

Figure: Incremental Parameters Update via Stochastic Gradient in MC

Semi-Gradient TD(0) for Estimating $\hat{v} \approx v_\pi$

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

Figure: Incremental Parameters Update via Semi-Gradient in TD(0)

Deep Q Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

CartPole-v0 DQN (Tensorflow)[1]

```
import gym
import tensorflow as tf
import numpy as np
import random
from collections import deque

# Hyper Parameters for DQN
GAMMA = 0.9 # discount factor for target Q
INITIAL_EPSILON = 0.5 # starting value of epsilon
FINAL_EPSILON = 0.01 # final value of epsilon
REPLAY_SIZE = 10000 # experience replay buffer size
BATCH_SIZE = 32 # size of minibatch
```

[1] https://zhuanlan.zhihu.com/p/21477488

CartPole-v0 DQN (Tensorflow)

```python
class DQN():
  # DQN Agent
  def __init__(self, env):
    # init experience replay
    self.replay_buffer = deque()
    # init some parameters
    self.time_step = 0
    self.epsilon = INITIAL_EPSILON
    self.state_dim = env.observation_space.shape[0]
    self.action_dim = env.action_space.n
    self.create_Q_network()
    self.create_training_method()

    # Init session
    self.session = tf.InteractiveSession()
    self.session.run(tf.initialize_all_variables())

  def create_Q_network(self):
    # network weights
    W1 = self.weight_variable([self.state_dim,20])
    b1 = self.bias_variable([20])
    W2 = self.weight_variable([20,self.action_dim])
    b2 = self.bias_variable([self.action_dim])
    # input layer
    self.state_input = tf.placeholder("float",[None,self.state_dim])
    # hidden layers
    h_layer = tf.nn.relu(tf.matmul(self.state_input,W1) + b1)
    # Q Value layer
    self.Q_value = tf.matmul(h_layer,W2) + b2
```

CartPole-v0 DQN (Tensorflow)

```
def create_training_method(self):
  self.action_input = tf.placeholder("float",[None,self.action_dim]) # one hot presentation
  self.y_input = tf.placeholder("float",[None])
  Q_action = tf.reduce_sum(tf.multiply(self.Q_value,self.action_input),reduction_indices = 1)
  self.cost = tf.reduce_mean(tf.square(self.y_input - Q_action))
  self.optimizer = tf.train.AdamOptimizer(0.0001).minimize(self.cost)

def perceive(self,state,action,reward,next_state,done):
  one_hot_action = np.zeros(self.action_dim)
  one_hot_action[action] = 1
  self.replay_buffer.append((state,one_hot_action,reward,next_state,done))
  if len(self.replay_buffer) > REPLAY_SIZE:
    self.replay_buffer.popleft()

  if len(self.replay_buffer) > BATCH_SIZE:
    self.train_Q_network()
```

CartPole-v0 DQN (Tensorflow)

```
def train_Q_network(self):
  self.time_step += 1
  # Step 1: obtain random minibatch from replay memory
  minibatch = random.sample(self.replay_buffer,BATCH_SIZE)
  state_batch = [data[0] for data in minibatch]
  action_batch = [data[1] for data in minibatch]
  reward_batch = [data[2] for data in minibatch]
  next_state_batch = [data[3] for data in minibatch]

  # Step 2: calculate y
  y_batch = []
  Q_value_batch = self.Q_value.eval(feed_dict={self.state_input:next_state_batch})
  for i in range(0,BATCH_SIZE):
    done = minibatch[i][4]
    if done:
      y_batch.append(reward_batch[i])
    else :
      y_batch.append(reward_batch[i] + GAMMA * np.max(Q_value_batch[i]))

  self.optimizer.run(feed_dict={
    self.y_input:y_batch,
    self.action_input:action_batch,
    self.state_input:state_batch
    })
```

CartPole-v0 DQN (Tensorflow)

```
def egreedy_action(self,state):
  Q_value = self.Q_value.eval(feed_dict = {
    self.state_input:[state]
    })[0]
  if random.random() <= self.epsilon:
    return random.randint(0,self.action_dim - 1)
  else:
    return np.argmax(Q_value)

  self.epsilon -= (INITIAL_EPSILON - FINAL_EPSILON)/10000

def action(self,state):
  return np.argmax(self.Q_value.eval(feed_dict = {
    self.state_input:[state]
    })[0])

def weight_variable(self,shape):
  initial = tf.truncated_normal(shape)
  return tf.Variable(initial)

def bias_variable(self,shape):
  initial = tf.constant(0.01, shape = shape)
  return tf.Variable(initial)
```

CartPole-v0 DQN (Tensorflow)

```
# Hyper Parameters
ENV_NAME = 'CartPole-v0'
EPISODE = 10000 # Episode limitation
STEP = 300 # Step limitation in an episode
TEST = 10 # The number of experiment test every 100 episode

def main():
  # initialize OpenAI Gym env and dqn agent
  env = gym.make(ENV_NAME)
  agent = DQN(env)

  for episode in range(EPISODE):
    # initialize task
    state = env.reset()
    # Train
    for step in range(STEP):
      action = agent.egreedy_action(state) # e-greedy action for train
      next_state,reward,done,_ = env.step(action)
      # Define reward for agent
      reward_agent = -1 if done else 0.1
      agent.perceive(state,action,reward,next_state,done)
      state = next_state
      if done:
        break
```

CartPole-v0 DQN (Tensorflow)

```
    # Test every 100 episodes
    if episode % 100 == 0:
      total_reward = 0
      for i in range(TEST):
        state = env.reset()
        for j in range(STEP):
          env.render()
          action = agent.action(state) # direct action for test
          state,reward,done,_ = env.step(action)
          total_reward += reward
          if done:
            break
      ave_reward = total_reward/TEST
      print('episode: ',episode,'Evaluation Average Reward:',ave_reward)
      if ave_reward >= 200:
        break

if __name__ == '__main__':
  main()
```

CartPole-v0 DQN (PyTorch)[2]

```
# -*- coding: utf-8 -*-
import gym
import math
import random
import numpy as np
from collections import namedtuple
from itertools import count
from PIL import Image
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms as T

env = gym.make('CartPole-v0').unwrapped
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))
```

[2] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

CartPole-v0 DQN (PyTorch)

```
class ReplayMemory(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

CartPole-v0 DQN (PyTorch)

```
class DQN(nn.Module):
    def __init__(self):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)
        self.head = nn.Linear(448, 2)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

CartPole-v0 DQN (PyTorch)

```
resize = T.Compose([T.ToPILImage(),
                    T.Resize(40, interpolation=Image.CUBIC),
                    T.ToTensor()])
screen_width = 600
def get_cart_location():
    world_width = env.x_threshold * 2
    scale = screen_width / world_width
    return int(env.state[0] * scale + screen_width / 2.0)  # MIDDLE OF CART

def get_screen():
    screen = env.render(mode='rgb_array').transpose((2, 0, 1))
    screen = screen[:, 160:320]
    view_width = 320
    cart_location = get_cart_location()
    if cart_location < view_width // 2:
        slice_range = slice(view_width)
    elif cart_location > (screen_width - view_width // 2):
        slice_range = slice(-view_width, None)
    else:
        slice_range = slice(cart_location - view_width // 2,
                            cart_location + view_width // 2)
    screen = screen[:, :, slice_range]
    screen = np.ascontiguousarray(screen, dtype=np.float32) / 255
    screen = torch.from_numpy(screen)
    return resize(screen).unsqueeze(0).to(device)
```

CartPole-v0 DQN (PyTorch)

```
env.reset()
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10

policy_net = DQN().to(device)
target_net = DQN().to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.RMSprop(policy_net.parameters())
memory = ReplayMemory(10000)

steps_done = 0
```

CartPole-v0 DQN (PyTorch)

```
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(2)]], device=device, dtype=torch.long)

episode_durations = []
```

CartPole-v0 DQN (PyTorch)

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see http://stackoverflow.com/a/19343/3343043 for
    # detailed explanation).
    batch = Transition(*zip(*transitions))
    # Compute a mask of non-final states and concatenate the batch elements
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                        batch.next_state)), device=device, dtype=torch.uint8)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                              if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)
    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken
    state_action_values = policy_net(state_batch).gather(1, action_batch)
    # Compute V(s_{t+1}) for all next states.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch
    # Compute Huber loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))
    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

CartPole-v0 DQN (PyTorch)

```
num_episodes = 50
for i_episode in range(num_episodes):
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None
        memory.push(state, action, next_state, reward)
        state = next_state
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            break
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())

env.render()
env.close()
```

CartPole-v1 DQN (Keras)[3]

```python
# -*- coding: utf-8 -*-
import random
import gym
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

EPISODES = 1000
```

[3] https://github.com/keon/deep-q-learning/blob/master/dqn.py

CartPole-v1 DQN (Keras)

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse',
                      optimizer=Adam(lr=self.learning_rate))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))
```

CartPole-v1 DQN (Keras)

```
def act(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    act_values = self.model.predict(state)
    return np.argmax(act_values[0])  # returns action

def replay(self, batch_size):
    minibatch = random.sample(self.memory, batch_size)
    for state, action, reward, next_state, done in minibatch:
        target = reward
        if not done:
            target = (reward + self.gamma *
                      np.amax(self.model.predict(next_state)[0]))
        target_f = self.model.predict(state)
        target_f[0][action] = target
        self.model.fit(state, target_f, epochs=1, verbose=0)
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

def load(self, name):
    self.model.load_weights(name)

def save(self, name):
    self.model.save_weights(name)
```

CartPole-v1 DQN (Keras)

```
if __name__ == "__main__":
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    agent = DQNAgent(state_size, action_size)
    done = False
    batch_size = 32

    for e in range(EPISODES):
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        for time in range(500):
            # env.render()
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            reward = reward if not done else -10
            next_state = np.reshape(next_state, [1, state_size])
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            if done:
                print("episode: {}/{}, score: {}, e: {:.2}"
                      .format(e, EPISODES, time, agent.epsilon))
                break
            if len(agent.memory) > batch_size:
                agent.replay(batch_size)
```

DQN Over Estimation

- Tabular Q Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big] \tag{9}$$

- Approximation Based DQN

$$\theta_{t+1} = \theta_t + \alpha\big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t) - Q(S_t, A_t; \theta_t)\big)\nabla_{\theta_t} Q(S_t, A_t; \theta_t) \tag{10}$$

Double DQN (DDQN)

- Action Selection: get optimized a^* based on θ_t (Q Network)

$$a^* = \arg\max_a Q(S_{t+1}, a; \theta_t) \tag{11}$$

- TD target: get TD target $Y_t^{DoubleQ}$ based on θ_t' (Target Q Network)

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, a^*; \theta_t') \tag{12}$$

Double DQN (DDQN)

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
    Initialize frame sequence $\mathbf{x} \leftarrow ()$
    for $t \in \{0, 1, \ldots\}$ **do**
        Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
        Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
        if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
        Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
            replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
        Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
        Construct target values, one for each of the N_b tuples:
        Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$

$$y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$$

        Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
        Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
    end
end

Blind Cliff Walking Example

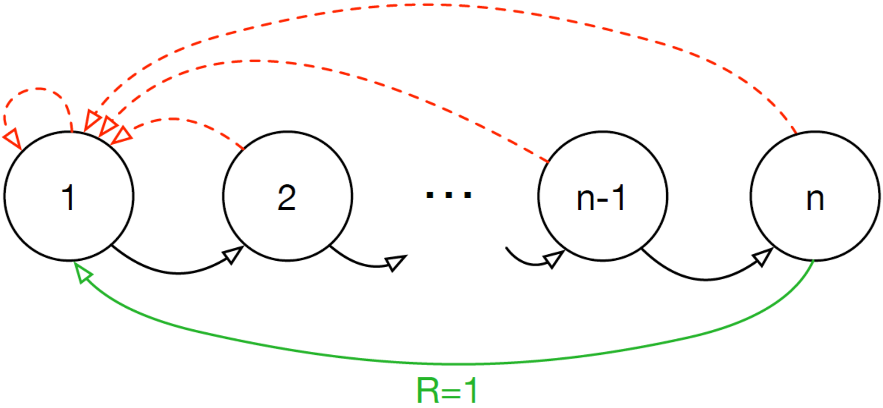


Figure: Delayed and Sparse Reward in Blind Cliff Walk Problem

Prioritizing Experience Replay

- Experience replay in the hippocampus of rodents: the sequences of prior experience are replayed, either during awake resting or sleep
- Uniform randomly replay: most relevant transitions (from rare successes) are hidden in a mass of highly redundant failure cases
- The TD error provides one way to measure these priorities
- Greedy TD-error prioritization
  - A low TD error on first visit may not be replayed for a long time
  - Sensitive to noise spikes, where approximation errors appear as another source of noise

Combine Uniform and Greedy

How to make use of TD-error to prioritize experience replay

$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$

- p_i^α is determined by TD-error δ_i. e.g, $p_i = |\delta_i| + \epsilon$ or $p_i = \frac{1}{rank(i)}$
- When prioritizing experience replay, then we get the bias estimation of the $Q(s, a)$, thus introduce weights $w_i = \left(\frac{1}{N} \times \frac{1}{P(i)}\right)^\beta$

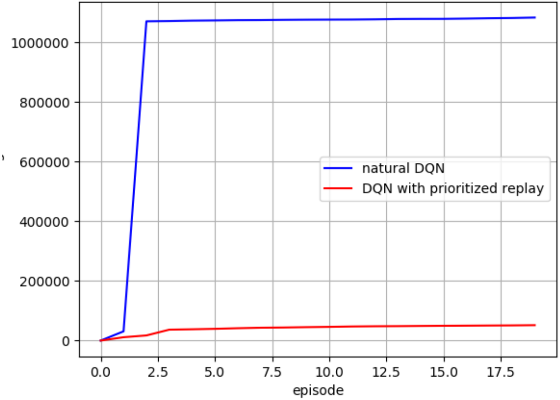
s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
s_4, a_4, r_5, s_5
s_5, a_5, r_6, s_6
$\ldots$
$s_{n-1}, a_{n-1}, r_n, s_n$

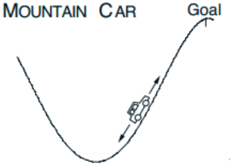
Prioritized Replay

Algorithm 1 Double DQN with proportional prioritization

```
Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory H = ∅, Δ = 0, p_1 = 1
Observe S_0 and choose A_0 ~ π_θ(S_0)
for t = 1 to T do
    Observe S_t, R_t, γ_t
    Store transition (S_{t-1}, A_{t-1}, R_t, γ_t, S_t) in H with maximal priority p_t = max_{i<t} p_i
    if t ≡ 0 mod K then
        for j = 1 to k do
            Sample transition j ~ P(j) = p_j^α / Σ_i p_i^α
            Compute importance-sampling weight w_j = (N · P(j))^{-β} / max_i w_i
            Compute TD-error δ_j = R_j + γ_j Q_target(S_j, arg max_a Q(S_j, a)) − Q(S_{j-1}, A_{j-1})
            Update transition priority p_j ← |δ_j|
            Accumulate weight-change Δ ← Δ + w_j · δ_j · ∇_θ Q(S_{j-1}, A_{j-1})
        end for
        Update weights θ ← θ + η · Δ, reset Δ = 0
        From time to time copy weights into target network θ_target ← θ
    end if
    Choose action A_t ~ π_θ(S_t)
end for
```

Prioritized Replay [4]





[4] https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/tree/master/contents/5.2_Prioritized_Replay_DQN

Dueling DQN

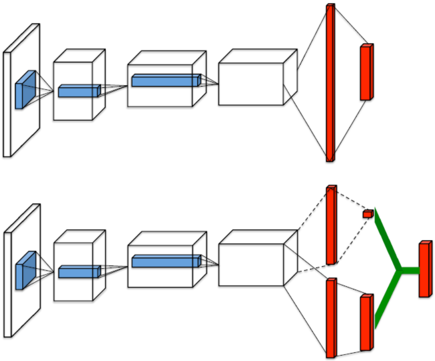


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

Dueling DQN

$$Q^\pi(s,a) = \mathbb{E}\left[G_t | s_t = s, a_t = a, \pi\right] \tag{13}$$

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}\left[Q^\pi(s,a)\right] \tag{14}$$

Thus, $Q^\pi(s,a)$ can be defined as

$$Q^\pi(s,a) = \mathbb{E}_{s'}\left[r + \gamma \mathbb{E}_{a'\sim\pi(s')}[Q^\pi(s',a')] | s, a, \pi\right] \tag{15}$$

We define another important quantity, the advantage function, relating the value and Q functions:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{16}$$

Note that $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. Intuitively, the advantage function subtracts the value of the state from the Q function to obtain a relative measure of the importance of each action.

Dueling DQN

- However, instead of following the convolutional layers with a single sequence of fully connected layers, we instead use two sequences (or streams) of fully connected layers.
- The streams are constructed such that they have they have the capability of providing **separate estimates** of the **value** and **advantage** functions.
- Finally, the two streams are combined to produce **a single output** Q function.

Dueling DQN

- one stream of fully-connected layers output a scalar $V(s;\theta,\beta)$
- the other stream output an $\|A\|$-dimensional vector $A(s,a;\theta,\alpha)$
- θ denotes the parameters of the convolutional layers
- α and β are the parameters of the two streams of fully-connected layers

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \tag{17}$$

$\forall(s,a)$

Dueling DQN

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \max_{a' \in |A|} A(s, a'; \theta, \alpha)\right) \tag{18}$$

An alternative module replaces the max operator with an average:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha)\right) \tag{19}$$

References

- https://zhuanlan.zhihu.com/p/21477488
- https://github.com/keon/deep-q-learning/blob/master/dqn.py
- https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
- Prioritized Experience Replay
- Dueling Network Architectures for Deep Reinforcement Learning
- https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/tree/master/contents/5.2_Prioritized_Replay_DQN

Tensorflow CNN

Hong Xingxing

October 12, 2018

Outline

conv2d API

tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, name=None)

- input tensor of shape: [batch, in_height, in_width, in_channels]
- filter tensor of shape: [filter_height, filter_width, in_channels, out_channels], have the same type as input
- padding: A string from: "SAME", "VALID". The type of padding algorithm to use.
- strides: A list of ints. 1-D tensor of length 4. The stride of the sliding window for each dimension of input.

conv2d example

```
import tensorflow as tf

tf.set_random_seed(9)

input = tf.Variable(tf.random_normal([1,3,3,4]), dtype=tf.float32)
filter = tf.Variable(tf.random_normal([1,1,4,2]))
op = tf.nn.conv2d(input, filter, strides=[1, 1, 1, 1], padding='VALID')
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print(sess.run(input))
    print(sess.run(tf.shape(input)))
    print(sess.run(filter))
    print(sess.run(tf.shape(filter)))
    print(sess.run(op))
    print(sess.run(tf.shape(op))) #[1,3,3,2]
```

Multi-Channel and Multi-Filter

```
import tensorflow as tf
input = tf.constant([
    [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],
    [[0,0,0],[0,0,1],[1,0,0],[2,1,1],[0,0,2],[1,0,2],[0,0,0]],
    [[0,0,0],[2,0,0],[0,1,1],[1,1,2],[0,2,0],[0,1,1],[0,0,0]],
    [[0,0,0],[0,0,0],[0,2,1],[2,1,0],[2,0,2],[1,0,2],[0,0,0]],
    [[0,0,0],[2,2,0],[2,1,1],[0,2,1],[1,1,2],[0,0,0],[0,0,0]],
    [[0,0,0],[1,2,2],[2,1,2],[0,1,2],[0,1,1],[0,2,1],[0,0,0]],
    [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]]
],shape=[1,7,7,3],dtype=tf.float32)
bias = tf.constant([1,0],shape=[2],dtype=tf.float32)
filter = tf.constant([
    [
        [[-1,1],[0,0],[0,1]],
        [[0,1],[-1,-1],[-1,0]],
        [[1,0],[-1,0],[1,0]]
    ],[[[0,-1],[-1,-1],[0,-1]],
        [[1,-1],[1,0],[0,-1]],
        [[0,0],[-1,-1],[-1,0]]
    ],[[[0,1],[0,-1],[0,1]],
        [[-1,1],[-1,-1],[1,-1]],
        [[-1,-1],[1,1],[-1,-1]]
    ]
],  shape=[3,3,3,2],dtype=tf.float32)
op1 = tf.nn.conv2d(input,filter,strides = [1,2,2,1],padding ='VALID')+bias
with tf.Session() as sess:
    result1 = sess.run(op1)
    print(sess.run(input)[0,:,:,0])
    print(sess.run(filter)[:,:,0,0])
    print(result1)
```

```
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 2. 0. 1. 0.]
 [0. 2. 0. 1. 0. 0. 0.]
 [0. 0. 0. 2. 2. 1. 0.]
 [0. 2. 2. 0. 1. 0. 0.]
 [0. 1. 2. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
[[-1.  0.  1.]
 [ 0.  1.  0.]
 [ 0. -1. -1.]]
[[[[-1.  1.]
   [ 4. -4.]
   [ 2. -9.]]

  [[-8. -2.]
   [-8. -7.]
   [ 0. -6.]]

  [[ 1. -4.]
   [-4. -7.]
   [ 1.  0.]]]]
```

Multi-Channel and Multi-Filter

Input Volume (+pad 1) ($7 \times 7 \times 3$)

$x[:,:,0]$

0	0	0	0	0	0	0
0	0	1	2	0	1	0
0	2	0	1	0	0	0
0	0	0	2	2	1	0
0	2	2	0	1	0	0
0	1	2	0	0	0	0
0	0	0	0	0	0	0

$x[:,:,1]$

0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	1	2	1	0
0	0	2	1	0	0	0
0	2	1	2	1	0	0
0	2	1	1	1	2	0
0	0	0	0	0	0	0

$x[:,:,2]$

0	0	0	0	0	0	0
0	1	0	1	2	2	0
0	0	1	2	0	1	0
0	0	1	0	2	2	0
0	0	1	1	2	0	0
0	2	2	2	1	1	0
0	0	0	0	0	0	0

Filter $W_0 (3 \times 3 \times 3)$

$W_0[:,:,0]$

-1	0	1
0	1	0
0	-1	-1

$W_0[:,:,1]$

0	-1	-1
-1	1	-1
0	-1	1

$W_0[:,:,2]$

0	-1	1
0	0	-1
0	1	-1

Bias $b_0 (1 \times 1 \times 1)$

$b_0[:,:,0]$

1

Filter $W_1 (3 \times 3 \times 3)$

$W_1[:,:,0]$

1	1	0
-1	-1	0
1	1	-1

$W_1[:,:,1]$

0	-1	0
-1	0	-1
-1	-1	1

$W_1[:,:,2]$

1	0	0
-1	-1	0
1	-1	-1

Bias $b_1 (1 \times 1 \times 1)$

$b_1[:,:,0]$

0

Output Volume ($3 \times 3 \times 2$)

$o[:,:,0]$

-1	4	2
-8	-8	0
1	-4	1

$o[:,:,1]$

1	-4	-9
-2	-7	-6
-4	-7	0

$$\text{red marked} \; -1 = \left(\sum_{i=0}^{2} \text{sum up the elements of the blue marked submatrices of } x[:,:,i] \cdot W_0[:,:,i] \right) + b_0 \tag{1}$$

Multi-Channel and Multi-Filter

Input Volume (+pad 1) ($7 \times 7 \times 3$)

$x[:, :, 0]$

0	0	0	0	0	0	0
0	0	1	2	0	1	0
0	2	0	1	0	0	0
0	0	0	2	2	1	0
0	2	2	0	1	0	0
0	1	2	0	0	0	0
0	0	0	0	0	0	0

$x[:, :, 1]$

0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	1	2	1	0
0	0	2	1	0	0	0
0	2	1	2	1	0	0
0	2	1	1	1	2	0
0	0	0	0	0	0	0

$x[:, :, 2]$

0	0	0	0	0	0	0
0	1	0	1	2	2	0
0	0	1	2	0	1	0
0	0	1	0	2	2	0
0	0	1	1	2	0	0
0	2	2	2	1	1	0
0	0	0	0	0	0	0

Filter $W_0 (3 \times 3 \times 3)$

$W_0[:,:,0]$

-1	0	1
0	1	0
0	-1	-1

$W_0[:,:,1]$

0	-1	-1
-1	1	-1
0	-1	1

$W_0[:,:,2]$

0	-1	1
0	0	-1
0	1	-1

Bias $b_0 (1 \times 1 \times 1)$

$b_0[:,:,0]$ $\boxed{1}$

Filter $W_1 (3 \times 3 \times 3)$

$W_1[:,:,0]$

1	1	0
-1	-1	0
1	1	-1

$W_1[:,:,1]$

0	-1	0
-1	0	-1
-1	-1	1

$W_1[:,:,2]$

1	0	0
-1	-1	0
1	-1	-1

Bias $b_1 (1 \times 1 \times 1)$

$b_1[:,:,0]$ $\boxed{0}$

Output Volume ($3 \times 3 \times 2$)

$o[:,:,0]$

-1	4	2
-8	-8	0
1	-4	1

$o[:,:,1]$

1	-4	-9
-2	-7	-6
-4	-7	0

$$\text{red marked } 4 = \left(\sum_{i=0}^{2} \text{sum up the elements of the blue marked submatrices of } x[:, :, i] \cdot W_0[:, :, i]\right) + b_0 \tag{2}$$

Multi-Channel and Multi-Filter

Input Volume (+pad 1) ($7 \times 7 \times 3$)

$x[:,:,0]$

0	0	0	0	0	0	0
0	0	1	2	0	1	0
0	2	0	1	0	0	0
0	0	0	2	2	1	0
0	2	2	0	1	0	0
0	1	2	0	0	0	0
0	0	0	0	0	0	0

$x[:,:,1]$

0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	1	2	1	0
0	0	2	1	0	0	0
0	2	1	2	1	0	0
0	2	1	1	1	2	0
0	0	0	0	0	0	0

$x[:,:,2]$

0	0	0	0	0	0	0
0	1	0	1	2	2	0
0	0	1	2	0	1	0
0	0	1	0	2	2	0
0	0	1	1	2	0	0
0	2	2	2	1	1	0
0	0	0	0	0	0	0

Filter $W_0(3 \times 3 \times 3)$

$W_0[:,:,0]$

-1	0	1
0	1	0
0	-1	-1

$W_0[:,:,1]$

0	-1	-1
-1	1	-1
0	-1	1

$W_0[:,:,2]$

0	-1	1
0	0	-1
0	1	-1

Bias $b_0(1 \times 1 \times 1)$

$b_0[:,:,0]$ $\boxed{1}$

Filter $W_1(3 \times 3 \times 3)$

$W_1[:,:,0]$

1	1	0
-1	-1	0
1	1	-1

$W_1[:,:,1]$

0	-1	0
-1	0	-1
-1	-1	1

$W_1[:,:,2]$

1	0	0
-1	-1	0
1	-1	-1

Bias $b_1(1 \times 1 \times 1)$

$b_1[:,:,0]$ $\boxed{0}$

Output Volume ($3 \times 3 \times 2$)

$o[:,:,0]$

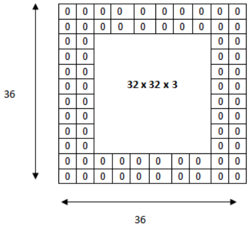
-1	4	2
-8	-8	0
1	-4	1

$o[:,:,1]$

1	-4	-9
-2	-7	-6
-4	-7	0

$$\text{green marked } 1 = \left(\sum_{i=0}^{2} \text{sum up the elements of the blue marked submatrices of } x[:,:,i] \cdot W_1[:,:,i]\right) + b_1 \tag{3}$$

Padding



The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

The formula for calculating the output size for any given conv layer is

$$O = \frac{W - K + 2P}{S} + 1 \tag{4}$$

where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

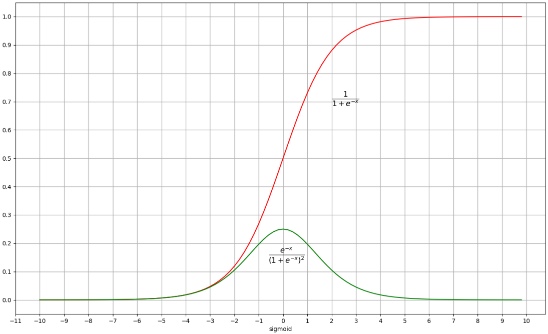
Activation Functions

- sigmod
- TanHyperbolic(tanh)
- Rectified Linear Units(ReLu)
- softplus
- softmax

sigmod

sigmod function looks like $y = \frac{1}{1+e^{-x}}$

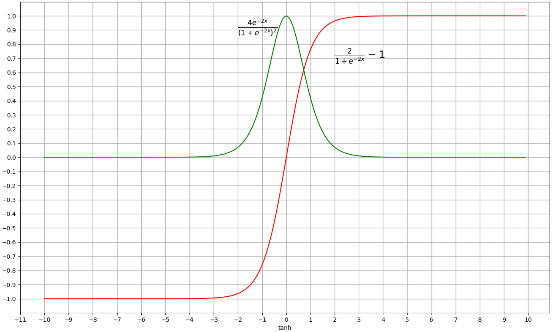
$$\frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{5}$$



tanh

tanh(x) function looks like $\frac{2}{1+e^{-2x}} - 1$

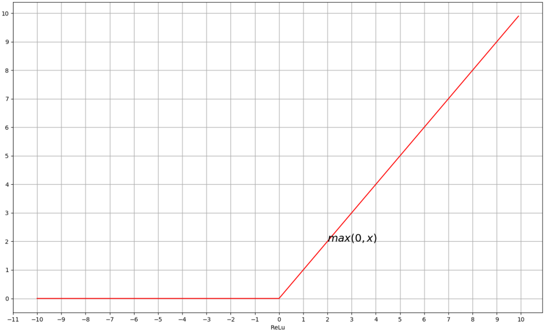
$$\frac{d}{dx}\left(\frac{2}{1+e^{-2x}} - 1\right) = \frac{4e^{-2x}}{\left(e^{-2x}+1\right)^2} \tag{6}$$



ReLu

ReLu function looks like $max(0, x)$ and the derivative of ReLu function is

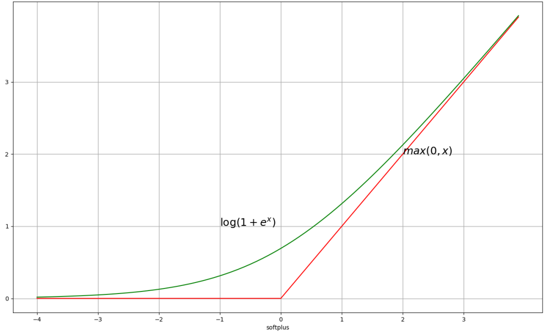
$$\frac{d}{dx}(max(0,x)) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \tag{7}$$



softplus

softplus function looks like $y = \log(1 + e^x)$ and the derivative of softplus function is

$$\frac{d}{dx}\left(\log\left(1 + e^x\right)\right) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}} \tag{8}$$



Softmax

$$pi = \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} \qquad (9)$$

```python
def softmax(X):
    exps = np.exp(X)
    return exps / np.sum(exps)
```

- The numerical range of floating point numbers in numpy is limited. For float64 the upper bound is 10^{308}. For exponential, its not difficult to overshoot that limit, in which case python returns **nan**
- To make softmax function numerically stable, simply normalize the values in the vector, by multiplying the numerator and denominator with a constant C

Stable Softmax

$$
\begin{aligned}
p_i &= \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} \\
&= \frac{Ce^{a_i}}{C\sum_{k=1}^{N} e^{a_k}} \\
&= \frac{e^{a_i+\log(C)}}{\sum_{k=1}^{N} e^{a_k+\log(C)}}
\end{aligned}
$$

```
def stable_softmax(X):
    exps = np.exp(X - np.max(X))
    return exps / np.sum(exps)
```

$\log(C) = -\max(a)$ is chosen.

Derivative of Softmax

$$\frac{\partial p_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} \tag{10}$$

According to the quotient rule $f(x) = \frac{g(x)}{h(x)}$ then $f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$.
In our case we get $g(x) = e^{a_i}$ and $h(x) = \sum_{k=1}^{N} e^{a_k}$.

- In $h(x)$, $\frac{\partial}{\partial a_j}$ will always be e^{a_j}
- In $g(x)$, only $i = j$ can we get e^{a_j}

Derivative of Softmax

if $i = j$ then

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{e^{a_i}\sum_{k=1}^{N} e^{a_k} - e^{a_j}e^{a_i}}{(\sum_{k=1}^{N} e^{a_k})^2}$$
$$= \frac{e^{a_i}(\sum_{k=1}^{N} e^{a_k} - e^{a_j})}{(\sum_{k=1}^{N} e^{a_k})^2}$$
$$= \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{\sum_{k=1}^{N} e^{a_k} - e^{a_j}}{\sum_{k=1}^{N} e^{a_k}}$$
$$= p_i(1 - p_j)$$

if $i \neq j$ then

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{0 - e^{a_j}e^{a_i}}{(\sum_{k=1}^{N} e^{a_k})^2}$$
$$= \frac{-e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}$$
$$= -p_j p_i$$

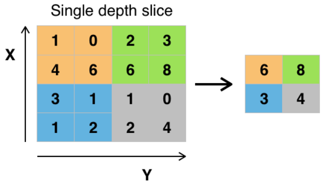
Using the Kronecker delta δij

$$\delta ij = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases} \tag{11}$$

and $\frac{\partial p_i}{\partial a_j} = p_i(\delta ij - p_j)$

pooling

After some ReLU layers, programmers may choose to apply a pooling layer. It is also referred to as a **downsampling layer**.



Example of Maxpool with a 2x2 filter and a stride of 2

This basically takes a filter (normally of size 2×2) and a stride of the same length. It then applies it to the input volume and outputs the **maximum number** in every subregion that the filter convolves around.

pooling

- Other options for pooling layers are
  - average pooling
  - L2-norm pooling
- Pooling layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume
  - The first is that the amount of parameters or weights is reduced by 75%, thus lessening the computation cost.
  - The second is it will control **overfitting**.

pooling API

tf.nn.max_pool(value, ksize, strides, padding, data_format='NHWC',name=None)

- value: A 4-D Tensor of the format specified by data_format. Usually it's like $[1, featuremap\_height, featuremap\_width, 1]$
- ksize: A list or tuple of 4 ints. The size of the window for each dimension of the input tensor. Usually it's like $[1, poolwindow\_height, poolwindow\_width, 1]$
- strides: A list or tuple of 4 ints. The stride of the sliding window for each dimension of the input tensor. Usually it's like $[1, stride, stride, 1]$
- padding: A string, either 'VALID' or 'SAME'

pooling example

```
import tensorflow as tf

a=tf.constant([
        [
            [1,2,3,4],
            [5,6,7,8],
            [8,7,6,5],
            [4,3,2,1]
        ],[
            [4,3,2,1],
            [8,7,6,5],
            [1,2,3,4],
            [5,6,7,8]
        ]])

a=tf.reshape(a,[1,4,4,2])

pooling=tf.nn.max_pool(a,[1,2,2,1],[1,1,1,1],padding='VALID')
with tf.Session() as sess:
    print("image:")
    image=sess.run(a)
    print (image)
    print("reslut:")
    result=sess.run(pooling)
    print (result)
```

```
image:
[[[[1 2]
   [3 4]
   [5 6]
   [7 8]]

  [[8 7]
   [6 5]
   [4 3]
   [2 1]]

  [[4 3]
   [2 1]
   [8 7]
   [6 5]]

  [[1 2]
   [3 4]
   [5 6]
   [7 8]]]]
reslut:
[[[[8 7]
   [6 6]
   [7 8]]

  [[8 7]
   [8 7]
   [8 7]]

  [[4 4]
   [8 7]
   [8 8]]]]
```

1	3	5	7
8	6	4	2
4	2	8	6
1	3	5	7

2	4	6	8
7	5	3	1
3	1	7	5
2	4	6	8

8	6	7
8	8	8
4	8	8

7	6	8
7	7	7
4	7	8

LeNet-5

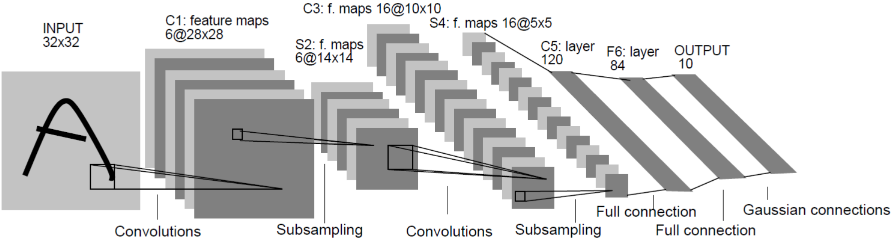


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

MNIST LeNet

```
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('MNIST_data/', one_hot=True)

class LeNet(object):
    def __init__(self, regularizer):
        self.regularizer = regularizer
        self.w = 32
        self.h = 32
        self.c = 1
        self.MAX_EPOCH = 1000
        self.build_net()
        self.create_train_method()
        self.session = tf.InteractiveSession()
        self.session.run(tf.global_variables_initializer())

    def build_net(self):
        self.x = tf.placeholder(dtype=tf.float32, shape=[None, self.w, self.h, self.c] , name='x')
        self.y_ = tf.placeholder(dtype=tf.int32, shape=[None], name='y_')

        with tf.variable_scope('layer1-conv1'):
            conv1_weights = tf.Variable(tf.truncated_normal(shape=[5,5,self.c,6], stddev=0.1),
 ↪     name='weight')
            conv1_biases = tf.Variable(tf.constant(0.0, shape=[6]), name='bias')
            conv1 = tf.nn.conv2d(self.x, conv1_weights, strides=[1,1,1,1], padding='VALID')
            relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_biases))
```

MNIST LeNet

```
with tf.variable_scope('layer2-pool1'):
    pool1 = tf.nn.max_pool(relu1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

with tf.variable_scope('layer3-conv2'):
    conv2_weights = tf.Variable(tf.truncated_normal(shape=[5,5,6,16], stddev=0.1), name='weight')
    conv2_biases = tf.Variable(tf.constant(0.0, shape=[16]), name='bias')
    conv2 = tf.nn.conv2d(pool1, conv2_weights, strides=[1,1,1,1], padding='VALID')
    relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_biases))

with tf.variable_scope('layer4-pool2'):
    pool2 = tf.nn.max_pool(relu2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

pool_shape = pool2.get_shape().as_list()
nodes = pool_shape[1]*pool_shape[2]*pool_shape[3]
reshaped = tf.reshape(pool2, [-1,nodes])
```

MNIST LeNet

```
with tf.variable_scope('layer5_fc1'):
    fc1_weights = tf.Variable(tf.truncated_normal(shape=[nodes,120], stddev=0.1), name='weight')
    if self.regularizer != None:
        tf.add_to_collection('losses', self.regularizer(fc1_weights))
    fc1_biases = tf.Variable(tf.constant(0.1, shape=[120]), name='bias')
    fc1 = tf.nn.relu(tf.matmul(reshaped, fc1_weights)+fc1_biases)

with tf.variable_scope('layer6_fc2'):
    fc2_weights = tf.Variable(tf.truncated_normal(shape=[120,84], stddev=0.1), name='weight')
    if self.regularizer != None:
        tf.add_to_collection('losses', self.regularizer(fc2_weights))
    fc2_biases = tf.Variable(tf.truncated_normal(shape=[84], stddev=0.1), name='bias')
    fc2 = tf.nn.relu(tf.matmul(fc1, fc2_weights)+fc2_biases)

with tf.variable_scope('layer7-fc3'):
    fc3_weights = tf.Variable(tf.truncated_normal(shape=[84,10], stddev=0.1), name='weight')
    if self.regularizer != None:
        tf.add_to_collection('losses', regularizer(fc3_weights))
    fc3_biases = tf.Variable(tf.truncated_normal(shape=[10], stddev=0.1), name='bias')
    self.logit = tf.matmul(fc2, fc3_weights) + fc3_biases
```

MNIST LeNet

```
def create_train_method(self):
    self.cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=self.logit,
↪   labels=self.y_)
    self.loss = tf.reduce_mean(self.cross_entropy) + tf.add_n(tf.get_collection('losses'))
    self.train_op = tf.train.AdamOptimizer(0.001).minimize(self.loss)
    self.correct_prediction = tf.equal(tf.cast(tf.argmax(self.logit,1), tf.int32), self.y_)
    self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, tf.float32))
```

MNIST LeNet

```
    def train_test_lenet(self):
        batch_size = 64
        epoch = 0
        # Train
        for epoch in range(self.MAX_EPOCH):
            batch = mnist.train.next_batch(batch_size)
            imglist = [np.resize(img.reshape(28,28,1), (32,32,1)) for img in batch[0]]
            index_labels = [np.argmax(l) for l in batch[1]]
            print(self.session.run([self.train_op, self.loss, self.accuracy], feed_dict={self.x: imglist,
 ↪    self.y_:index_labels}))
        # Test
        for epoch in range(self.MAX_EPOCH):
            batch = mnist.train.next_batch(batch_size)
            imglist = [np.resize(img.reshape(28,28,1), (32,32,1)) for img in batch[0]]
            index_labels = [np.argmax(l) for l in batch[1]]
            print(self.session.run([self.loss, self.accuracy], feed_dict={self.x: imglist,
 ↪    self.y_:index_labels}))

regularizer = tf.contrib.layers.l2_regularizer(0.001)
ln = LeNet(regularizer)
ln.train_test_lenet()
```

ref

- https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-
- https://deepnotes.io/softmax-crossentropy