Project #2

EE488C Special Topics in EE <Deep Learning & AlphaGo>, 2016 Fall
Information Theory & Machine Learning Lab., School of EE, KAIST

- Contents
  - Fine-tuning AlexNet using Caltech 101 dataset
  - Q-learning for the maze game
  - Q-learning and DQN for a simple version of Breakout
  - Appendix

- Codes for this project
  - Download project2.zip from KLMS, which contains source codes for this project.

- As in project #1, we use TensorFlow. In addition to TensorFlow, you need to have h5py and tflearn installed. If you use one of server computers, they are already installed. If you are using your own machine, then you need to install the packages as follows.

```
$ pip install h5py
$ pip install tflearn
```

- Due: December 5, 1pm

- Image classification using AlexNet
  - AlexNet is a convolutional neural network that consists of 5 convolutional layers and 3 fully-connected layers. This neural network was first proposed by Alex Krizhevsky, *et al*. [1], which won the first place at ILSVRC 2012, achieving a top-5 error rate of 15.3%.
  - Below, we explain how to classify an image using a pre-trained AlexNet with ILSVRC 2012 data set. *ILSVRC* (Imagenet Large Scale Visual Recognition Challenge) is an annual competition for image classification using many (~ million) images in 1,000 categories.
  - In "project2_alexnet_classify.py", we load a pre-trained AlexNet, read an image, and perform image classification that outputs 5 most probable categories and their probabilities. From now on, we will explain the code.
  - In '# Open image' section in the file, we load a test image "cat.jpg" and resize it to 256x256. If the image is not RGB format, then we convert it to an RGB image.
  - In '# Cropping' section, we obtain 10 images of size 227x227 by shifting and cropping the image.
  - We then subtract the mean of the image by using ilsvrc_2012_mean.npy, which can be downloaded from https://github.com/BVLC/caffe/blob/master/python/caffe/imagenet/ilsvrc_2012_mean.npy. It is also in project2.zip.
  - net_data = np.load("bvlc_alexnet.npy").item()
    This function loads the pre-trained AlexNet as a numpy array. "bvlc_alexnet.npy" can be downloaded from http://www.cs.toronto.edu/~guerzhoy/tf_alexnet/. To download it at a Linux prompt, do the following.

```
$ wget http://www.cs.toronto.edu/~guerzhoy/tf_alexnet/bvlc_alexnet.npy
```

  - for x in net_data:
        exec("%s = %s" % (str(x)+"W", "tf.Variable(net_data[x][0])"))
        exec("%s = %s" % (str(x)+"b", "tf.Variable(net_data[x][1])"))
    This defines TensorFlow variables for weights and biases for each layer. 'x' will be one of 'conv1', 'conv2', 'conv3', 'conv4', 'conv5', 'fc6', 'fc7', 'fc8', i.e., names for 5 convolutional layers and 3 fully-connected layers. For the first convolutional layer, the variable 'x' will be 'conv1' and the above two lines containing 'exec' will define "conv1W = tf.Variable(net_data['conv1'][0])" and "conv1b = tf.Variable(net_data['conv1'][1])". The 'for' loop will run 8 times to define TensorFlow variables for weights and biases for all 8 layers.
  - def conv(input, kernel, biases, k_h, k_w, c_o, s_h, s_w, padding="VALID", group=1)
    This function constructs a convolutional layer with 'c_o' convolutional filters with size (k_h)x(k_w). The stride size (s_h)x(s_w). AlexNet was designed to run on two GPU's and it has two different paths as explained in lecture notes. 'group' indicates the number of groups the convolutional filters are split into.
    - This does not mean you need to run it on a computer with two GPU's. It can still run even if you have only one GPU provided that there's enough memory during training.

- Constructing AlexNet
  - conv1 = tf.nn.relu(conv(x, conv1W, conv1b, 11, 11, 96, 4, 4, padding="VALID", group=1))
    This defines the first convolutional layer which consists of 96 11x11 filters. This takes 'x' as an input. Stride sizes are 4x4 and conv1W and conv1b are a weight matrix and a bias vector, respectively, loaded from file "bevlc_alxnet.npy". "tf.nn.relu" is the ReLU function.
  - lrn1 = tf.nn.local_response_normalization(conv1, depth_radius=2, alpha=2e-5, beta=0.75, bias=1.0)
    This normalizes the output of the previous layer. Specifically, this outputs (input/(bias+alpha*sqr_sum)^beta) where 'sqr_sum' means squared sum of inputs within depth_radius. For details, refer to [1].
  - maxpool1 = tf.nn.max_pool(lrn1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1], padding='VALID')
    This performs max pooling, i.e., taking the maximum value in each 3x3 image patch from the input and moving 2 units horizontally and vertically (stride size is 2x2).
  - They remaining convolutional layers are similarly defined.
  - fc6 = tf.nn.relu_layer(tf.reshape(maxpool5, [-1, int(pylab.prod(maxpool5.get_shape()[1:]))]), fc6W, fc6b)
    This is a fully connected layer that takes the output of the previous max-pooling layer, reshapes it, multiplies it by a matrix fc6W, and then adds the bias vector fc6b. Finally, this applies the ReLU activation function to the resulting vector.
  - This is then followed by two more fully connected layers, fc7 and fc8.
  - y_conv = tf.nn.softmax(fc8): This computes the softmax values.
- Printing top-5 categories
  - The output of AlexNet is a 1000x1 vector whose i-th element indicates the estimated likelihood that the input image belongs to the i-th category. We choose 5 most probable categories and print them along with their softmax values. Names of categories are given in 'caffe_classes.py'.
- "project2_alexnet_classify.py" reads "cat.jpg" and performs classification and its outcomes are shown below.
  - Cat.jpg

  - Output

0.60386	tiger cat
0.26037	tabby, tabby cat
0.09442	Egyptian cat
0.03329	lynx
0.00271	Persian cat

- Task #1 – Fine-tuning AlexNet using Caltech 101 dataset
  - For task #1, you will re-train the last fully-connected layer of the pre-trained AlexNet to classify new categories of images from Caltech 101 dataset. Caltech 101 dataset contains a set of images in 101 categories such as airplanes, faces, and motorbikes, collected in 2003 by Fei-Fei Li, et al. There are about 40 to 800 images per category. Since the numbers of images vary a lot, we choose 10 categories with about 100 images in each category. The chosen 10 categories and associated image file names can be found in "101_labels_ten.txt".
  - Among the files in project2.zip, you need the following for task #1. We assume you already downloaded "bvlc_alexnet.npy" as instructed in page 2 of this document.

```
101_labels_ten.txt              # image file names for 10 categories
crop_batch.py                   # cropping and data augmentation
divide_train_val.py             # divide images into train and validation sets
Generate_HDF5.py                # generate HDF5 files
ilsvrc_2012_mean.npy            # mean information for ILSVRC 2012 dataset
project2_alexnet_finetune.py    # code for fine-tuning
project2_HDF5_generation.py     # code for generating *.h5 files
```

  - You need to download Caltech 101 dataset, which can be done as follows at a Linux command prompt.

```
$ wget --no-check-certificate
https://www.vision.caltech.edu/Image_Datasets/Caltech101/101_ObjectCategories.tar.gz
$ tar xvzf 101_ObjectCategories.tar.gz
```

  - Then, run "project2_HDF5_generation.py" to generate *.h5 files in HDF format. This needs to be done only once.
    - Hierarchical data format (HDF) is a data format for organizing and storing a large amount of data.
    - "project2_HDF5_generation.py" reads "101_labels_ten.txt" and calls "divide_train_val()" function in "divide_train_val.py" to randomly split the images specified in the file "101_labels_ten.txt" into a training set and a validation set. The split is 80:20, i.e., 80% for training set and 20% for validation set.
    - After running the code, you will get "Caltech101_ten_train.h5", which contains all the training images and their labels and "Caltech101_ten_val.h5", which contains all the validation images and their labels.
  - Then, run "project2_alexnet_finetune.py", which trains the last layer of AlexNet using the new dataset. You can see the test accuracy is quite high, which is because there are only 10 categories and also because AlexNet was already trained using 1,000 categories and you are only re-training the last layer. In appendix, we explain some key steps in "project2_alexnet_finetune.py".
  - Your job is to modify "project_alexnet_finetune.py" to use three different optimizers and to see how they compare. Test accuracy will not be considered when grading your report, but try to choose three optimizers and their parameters such that their test accuracies are not similar and they show interesting differences. Explain what you observe.

- Q-learning for the maze game
  - In this example, we build a 5 by 5 maze with randomly constructed walls and understand how Q-learning works.
  - Unzip project2.zip and find project2_environment.py and project2_QLearning_maze.py.
  - The maze environment has 25 states, one for each cell, and the agent can perform one of four actions (up, down, left, right) at each state. If a movement is blocked by a wall, then the agent will stay in the current cell. States are numbered from 0 to 24 as shown in Fig. 1. The starting state is state 0, which is the lower left corner and the terminal state is the upper right corner (state 24).
  - The goal of the agent is to reach the terminal state. When the agent reaches the terminal state, the agent gets a reward of 1 and the episode ends.

20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

Figure 1. Maze environment

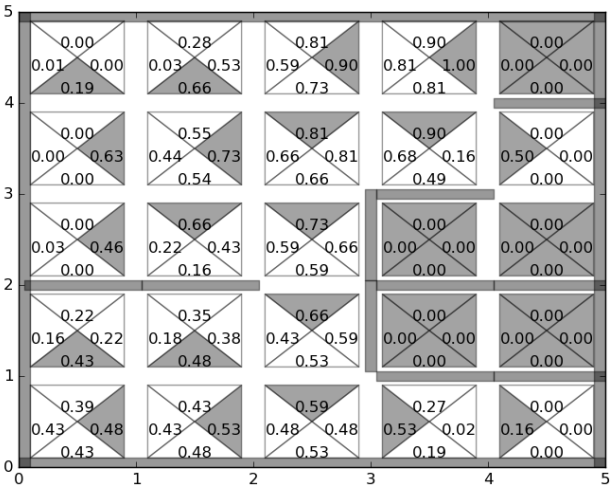

Figure 2. Trained Q values

  - Run project2_QLearning_maze.py, which will perform the following.
    - It will first place walls. Walls are placed with probability 0.2. Mazewall is a vector of length 40, where each of its elements is 1 or 0 indicating whether there exists a wall or not. Since there are 40 possible locations for walls in the 5x5 maze, the length of 'Mazewall' vector is 40.
    - It then performs Q-learning. During training, the function environment_maze(S, A, M) in project2_enrivonment.py is called many times, which takes the current state 'S', current action 'A', and maze information 'M' and returns the reward, the next state, and a flag that indicates whether the next state is a terminal state.
      - Since there are 25 states and 4 actions, the Q matrix is initialized as a 25*4 matrix.
      - ϵ−greedy exploration is used, where we fix the value of ϵ to 0.1.
      - Random tie break is used when taking a greedy action.
    - Since each wall is placed randomly, the maze may not have a solution. In that case, running project2_QLearning_maze.py may take a long time due to unsuccessful tries. You may want to stop the program by pressing Ctrl-C if it takes longer than 10~20 seconds. If the maze is solvable, it shouldn't take more than 10~20 seconds.
    - After learning is completed, it plots the maze with learned Q values as shown in Fig. 2, which shows walls and the learned Q values for each direction (up, down, left, and right). The shaded triangles in each cell indicates the maximum Q value.
    - For more details, read comments in the files.

- Task #2
  - Write codes that can learn to solve the maze shown in Fig. 3 using Q-learning and Sarsa. You should write 3 codes, i.e., “project2_task2_environment.py” that specifies the environment and “project2_task2_QLearning.py” and “project2_task2_SARSA.py” for performing Q-learning and Sarsa, respectively. Place them in your home directory of the server you are using before the deadline. For this task, you don’t need to submit a report.
  - Your code “project2_task2_environment.py” should be based on the function “environment_maze” in “project2_environment.py” in project2.zip. The starting state should be the left bottom corner “S” and the terminal state should be the right bottom corner “T”. See Fig. 3. The 3 red cells in Fig. 3 should give a reward of -100 whenever any of them is visited. When the agent reaches the terminal state, a reward of one should be given.
  - “project2_task2_QLearning.py” and “project2_task2_SARSA.py” should be based on the code “project2_QLearning_maze.py” in project2.zip. You should make the following changes.
    - Instead of generating “Mazewall” vector randomly, manually specify the vector “Mazewall” so that it describes the maze in Fig. 3. Try to analyze the code “project_environment.py” to understand how the length-40 vector “Mazewall” needs to be specified.
    - Modify the line “from project2_environment import environment_maze as environment” to “from project2_task2_environment import environment_maze as environment” so that your codes import the function “environment_maze” from your file “project2_task2_environment.py”.
    - You can change variables such as “alpha”, “gamma”, “epsilon”, and “num_episodes”.
    - For Sarsa, modify the main loop in “project2_Qlearning_maze.py” to perform Sarsa instead of Q-learning.
    - You are not allowed to make any other changes.
  - If successful, Q-learning will choose the shortest but more dangerous path “A” and Sarsa will choose a longer but safer path “B” at the end of training. Paths “A” and “B” are shown below.
  - Grading will be based on whether your codes choose the correct paths and on the value of “num_episodes”. Try to make “num_episodes” as small as possible by adjusting “alpha”, “gamma”, and “epsilon”.

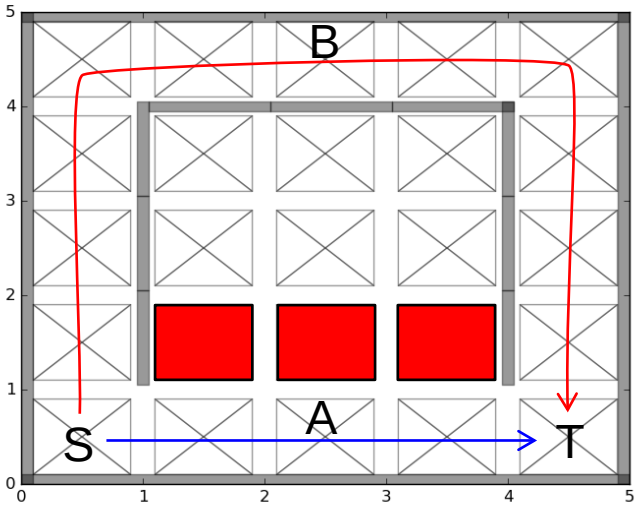

Figure 3. Maze environment for Task #1

- Q-learning and DQN for a simple version of Breakout
  - Unzip project2.zip and find project2_environment.py, project2_state_representation.py, project2_QLearning_breakout1.py, and project2_DQN_breakout*.py.
  - In this example, we train an agent to play a simple version of Breakout. There are 25 bricks in 5 rows and 5 columns. At each time, the agent can take one of 5 actions to remove the bottom brick in one of 5 columns. Unlike the original Breakout, there is no ball and thus you don't need to worry about losing a ball. A reward of 1 is given for removing a brick.
  - There are 3 different environments defined in project2_environment.py.
    - In environment 1, if any one of 5 columns is cleared, then all the remaining bricks are removed and you get a reward equal to the number of removed bricks. Therefore, an optimal policy for this environment is to clear any one of 5 columns. See Fig. 4 below.

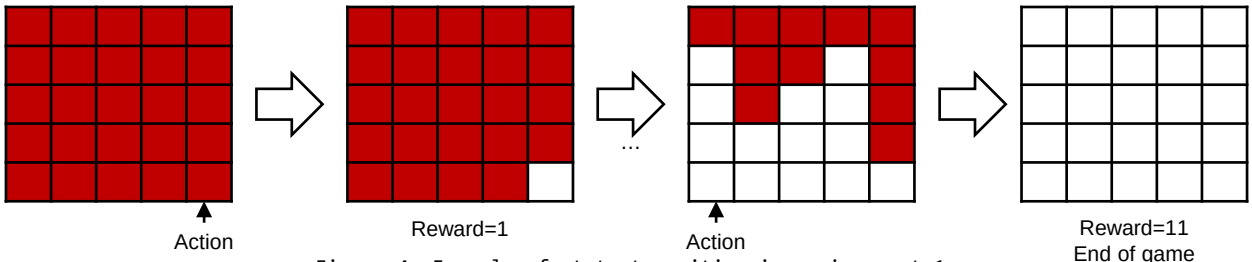

Figure 4. Example of state transition in environment 1

    - In environment 2, if any one of 5 columns is cleared, then all the remaining bricks in the upper 4 rows are removed and you get a reward equal to the number of removed bricks. Therefore, an optimal policy for this environment is to clear any one of 5 columns and then remove the remaining 4 bricks at the bottom row. See Fig. 5 below.

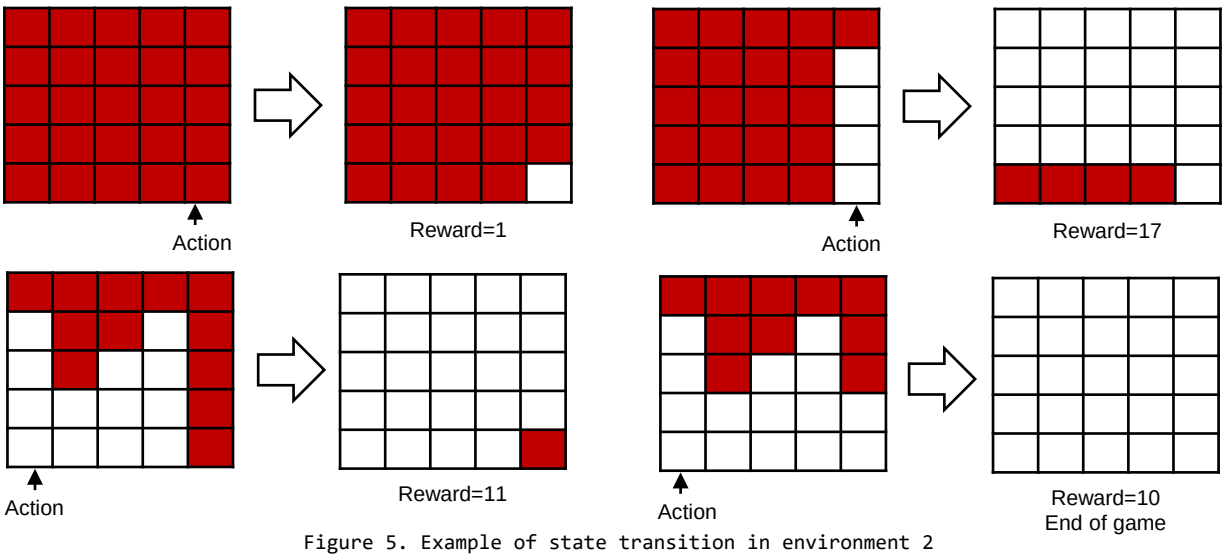

Figure 5. Example of state transition in environment 2

    - In environment 3, if any two consecutive columns are cleared, then all the remaining bricks are removed and you get a reward equal to the number of removed bricks. Therefore, an optimal policy for this environment is to clear any two consecutive columns. See Fig. 6 next page.

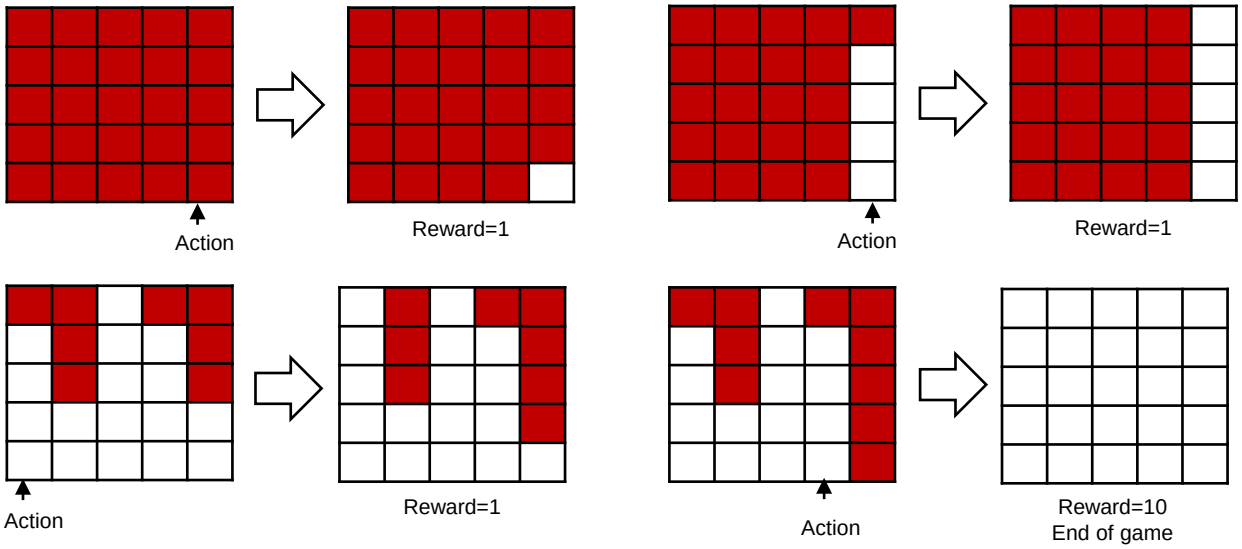

Figure 5. Example of state transition in environment 3

- For all 3 environments, the total accumulate rewards is always 25 at the end of a game. Therefore, the goal is not to get a higher score, but to finish the game as soon as possible. If the discount factor γ is strictly less than 1, then reinforcement learning will try to learn to finish the game as quickly as possible.
- Since there are 6 possible states for each column, i.e., empty, only the top brick is present, only the top two bricks are present, …, and all 5 bricks are present, the total number of states is 6^5.
- There are 4 codes for training, i.e., project2_QLearning_breakout1.py, project2_DQN_breakout1.py, project2_DQN_breakout2.py, and project2_DQN_breakout3.py, where the first two use environment #1, the third uses environment #2, and the last one uses environment #3. The first one use Q-learning and the other three use DQN.
- A state is represented 3 different ways.
  - For storing a state in replay memory in DQN, we use a vector representation, i.e., $S = [s_1, s_2, s_3, s_4, s_5]$, where s_i is 0~5 indicating the number of remaining bricks in the i-th column.
  - For feeding a state to the neural network in DQN, we use a matrix representation that is 5x5 matrix of 0's and 1's indicating the presence of each brick. A vector representation can be converted to an equivalent matrix representation by using the function "matrix_state" in "project2_state_representation.py".
  - For Q learning, we use a scalar representation, i.e., $S = 0, \ldots, 6^5 - 1$, where $S = 0$ means no bricks and $S = 6^5 - 1$ means all bricks are present. A vector representation can be converted to an equivalent scalar representation by using the function "scalar_state" in "project2_state_representation.py".

- Run “project2_QLearning_breakout1.py” and observe the output.
  - It performs Q-learning to train an agent to play Breakout environment #1.
  - ϵ−greedy exploration is used, where the value of ϵ is set to 1 for the first episode and then decreased linearly to 0 for the last episode.
  - The number of episodes is set to 10,000. For the i-th episode, ‘num_trials[i]’ is set to the number of actions taken during the episode.
  - At the end of training, it will show the number of actions taken for the last episode, i.e., ‘num_trials[n_episodes-1]’. Since ϵ=0 for the last episode, it measures the performance of the learned agent without any random exploration. If training is done well, the number of actions taken for the last episode should be 5. Due to randomness, i.e., random explorations during training, this may be higher than 5 sometimes. In that case, try to run the code again until you get 5.
  - It also shows a plot of how the number of actions changes over training episodes. Each data point in the plot shows an average of ‘num_trials’ over (num_episodes / 100) episodes. In Fig. 6 below, we show an example of the plot.

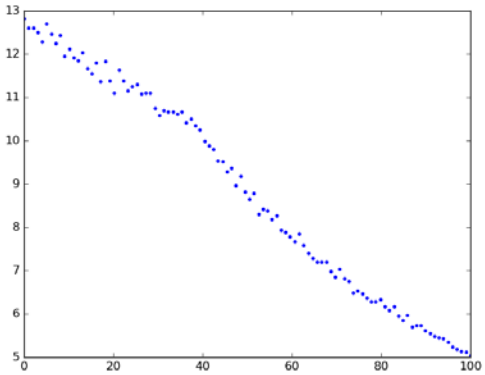

Figure 6. Performance of Q-learning for environment 1

- Run “project2_DQN_breakout1.py” and observe the output.
  - In this code, DQN is used instead of Q-learning. The environment is still environment #1.
  - The neural network observes the state as a 5x5 matrix as explained before.
  - The neural network has only one layer, but it can learn to play optimally with only 10 episodes. As explained in page 12 of lecture notes #18, it is possible for such a simple neural network to produce an optimal policy.
- Run “project2_DQN_breakout2.py” and observe the output.
  - In this code, DQN is used for environment #2.
  - The neural network still has only one layer.
  - It takes about 5,000 ~ 10,000 episodes to train the network. More episodes are needed than environment #1 since this game is more complicated.
- Run “project2_DQN_breakout3.py” and observe the output.
  - In this code, DQN is used for environment #3.
  - The neural network still has only one layer
  - It now takes about 20,000 episodes to train the network.

- Deep Q network design
  - We now explain how DQN works for “project2_DQN_breakout*.py”.
  - Basic algorithm is the same as in [3]. We use an online network and a target network. Initially, they are both identical copies of each other. Then, by using the experiences of the agent, the online network is updated. Then, after some iterations, the target network is updated by copying the parameters of the online network to the target network.
  - Hyperparameter design
    - Discount factor and learning rate are set to 0.9 and 0.00025.
    - In the DQN, we can use experience replay as done in [3]. We store the state, action, new state and reward in the memory and sample them in batch when we need parameter update. Replay memory size is set to be big enough to store the whole experience during training.
    - size_minibatch = 32 # Size of the sample batch
    - replay = 1 # 0 means we sample from the most recent transition. 1 means sample from replay memory.
    - num_episodes = 20000 # Number of training episodes
    - target_net_update_period = 20 # Period for updating the target network
  - Q network architecture (function DeepQNetwork)
    - Q network consists of single fully connected layer, where we get the image of the state as a matrix and output the Q values for all possible actions
    - weights1 = tf.get_variable("weights1", [5*5, 5], initializer = init)

      biases1 = tf.get_variable("biases1", [5], initializer = init)

      These are the weights and biases for the network
    - in1 = tf.reshape(state, [-1, 5*5])

      This reshapes the state to a length-25 vector. -1 means undecided, which will be automatically set later to be equal to the batch size when you feed data to the neural network.
    - return tf.matmul(in1, weights1) + biases1
      This defines the output Q values without activation function.
    - QN, state_dim = DeepQNetwork, [None, 5, 5]

      state_representation = matrix_state

      We set the Q network and the input state dimension as 5*5. ‘state_representation’ function converts the state vector to a matrix form as explained before.
  - Placeholder and optimizer design
    - We use five placeholders S,A,R,Sn, and T for current state, action, reward, next state, and indicator for episode termination, respectively. The shape of each placeholder is set properly. [None] means unspecified, which will be set later to match the batch size.
    - online_Q = QN(S)

      This specifies ‘online_Q’ is the output of the online network with input ‘S’.
    - target_Q = QN(Sn)

      This specifies ‘target_Q’ is the output of the target network with input ‘Sn’.
    - online_net_variables = tf.get_collection(tf.GraphKeys.VARIABLES, scope = "online_net")

      target_net_variables = tf.get_collection(tf.GraphKeys.VARIABLES, scope = "target_net")

      We gather the variables such as weights and biases that contribute to the update of online_Q and target_Q.

- Y_targets = R + gamma * tf.mul(T, tf.reduce_max(target_Q, 1))
  This corresponds to the R+max(Q(sn,:)) in the Q-learning. 'T' is multiplied since we only need to have 'R' if 'Sn' is a terminal state.
- Y_onlines = tf.reduce_sum(tf.mul(online_Q, A), 1)
  This extracts the Q value for the current state and action.
- loss = tf.reduce_mean(tf.square(tf.sub(Y_targets, Y_onlines)))
  This is the loss function defined as $E\left[\left(Q(s,a)-(R+\gamma*\max(Q(s',:))\right)^2\right]$.
- optimizer = tf.train.RMSPropOptimizer(alpha).minimize(loss, var_list = online_net_variables).
  Finally, we train the weights and biases of the neural network to minimize the loss. RMSPropOptimizer is a well-known optimize, which is also used in DQN in [3].
- This optimization is performed by 'online_net_update' function.
- target_net_update = [target_net_variables[i].assign(online_net_variables[i])
  for i in range(len(online_net_variables))]
  This updates the target network by copying the variables of the online network.

- Main training algorithm
  - Training is done similarly as in [3]. Here, we explain things that are specific to our implementation.
  - 'replay_memory' and 'minibatch' are initialized to be lists with 5 empty lists.
  - online_net_variables and target_net_variables are initialized.
  - epsilon = 1 - float(i_episode) / (num_episodes – 1)
    We linearly decrease epsilon from 1 in the first episode to 0 in the final episode.
  - For each episode, we initialize the state 'S_', indicator of the terminal state 'T_', and time stamp 'time_episode_start'.
  - Q_ = sess.run(online_Q, feed_dict = {S: state_representation([S_])})
    Obtain Q values for the current state 'S_' using the online network.
  - After we take action and observe the new state, reward, and indicator for a terminal state, we store the observed values into the replay memory. Then, we 'num_sampling' random samples from the replay_memory and train the Q network. If 'replay' variable is set to 0, only the most recent observed sample is used for training.
  - if target_net_update_counter < target_net_update_period … sess.run(target_net_update)
    This is for updating the target network.

- Task #3
  - Modify "project2_DQN_breakout3.py" so that the agent can learn using fewer episodes for environment #3.
  - You are allowed to change only the following.
    - alpha, gamma, size_minibatch, replay, num_episodes, and target_net_update_period
    - You can change the neural network architecture specified in the function DeepQNetwork. You can have up to 3 layers. If you use a convolutional layer, the size of the convolutional filter should not exceed 3x3 and the number of convolutional filters should not exceed 10. If you use a fully connected layer, the number of neurons should not exceed 100.
    - Loss function
    - Optimizer and its parameters
    - You are not allowed to change any other parts of the code.
  - Grading will be based on whether the agent learns to finish the game in 10 time steps and on the value of 'num_episodes'. Try to make 'num_episodes' as small as possible.
  - Name your code as "project2_task3.py" and save it in your home directory before the deadline. For this task, you don't need to submit a report.

- References

[1] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton, (2012) Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25 (NIPS'2012)

[2] Xavier Glorot and Yoshua Bengio, (2010) Understanding the difficulty of training deep feedforward neural networks. In Journal of Machine Learning Research, W&CP 9:249-256.

[3] V. Mnih, et al., (2015) Human-level control through deep reinforcement learning, Nature

Appendix

- Explanations on some steps in “project2_alexnet_finetune.py”.
  - arrayoutput_weight_filename='./Caltech101_finetune_weight.npy'
    After fine-tuning AlexNet, we save the parameters of the network in a numpy file. This line specifies the file name.
  - ‘fan1 = math.sqrt(6.0/(4096.0+10.0))’ and ‘fc8W = tf.Variable(tf.random_uniform([4096,10], minval=-fan1, maxval=fan1))’
    We use Xavier initialization.
  - train_step = opt.minimize(cross_entropy, var_list=[fc8W,fc8b])
    We train the last fully-connected layer while fixing other layers. The variables that need to be optimized, i.e., the weight and bias of the last fully connected layer, are specified as ‘var_list’.
  - npy_save = {}

    npy_save[0] = fc8W.eval()

    npy_save[1] = fc8b.eval()

    np.save(output_weight_filename, npy_save)

    We save the parameters of the last fully-connected layer as a numpy file.