CSCE 790-002: Deep Reinforcement Learning and Search
Homework 3
Due: 10/28/2020

Installation

We will be using an updated `conda` environment.

To update, follow the same instructions from Homework 1 using the updated `spec-file.txt` file.
You can either erase the current virtual environment that you have or create a new one.
The entire GitHub repository should be downloaded again as changes were made to other files. You can download it with the green “Code” button and click “Download ZIP”.

1 Backpropagation

For this exercise, you will implement forward propagation and backward propagation for a linear layer and a pointwise non-linear layer for a neural network using numpy. These two functions that you will implement will build your understanding of how neural networks work. The two layers you implement in this exercise can be composed in a variety of ways to build powerful function approximators.

There are N inputs of dimension D. The input matrix is $\mathbf{X} \in \mathbb{R}^{N \times D}$.
Where $\mathbf{x}_n$ is the n^{th} input and x_{nd} is the n^{th} input at index d.

For each layer, the gradient of some error function E will be calculated with respect to the parameters of the layer as well as with respect to the inputs to the layer. The simplicity of the backpropagation algorithm is in the fact that you do not need to know what E is. You only have to execute the chain rule when given the backpropagated gradients δ.

To check your implementations run:

```
python run_check_gradients.py --layer relu --num_inputs 5 --input_dim 10 --hidden_dim 100
python run_check_gradients.py --layer linear --num_inputs 5 --input_dim 10 --hidden_dim 100
```

The goal is to have the squared error between your implementation of the output computation step and the gradient computation be zero.

Vary `--num_inputs`, `--input_dim`, and `--hidden_dim` to verify your implementation.

See the slides on neural networks and deep learning for more information on backpropagation.

1.1 Gradient of a Rectified Linear Unit (ReLU) Layer (10 pts)

Implement `relu_forward` and `relu_backward` in `assignments_code/assigment3.py`.

Given an input $\mathbf{X} \in \mathbb{R}^{N \times D}$, `relu_forward` returns the output $\mathbf{O} \in \mathbb{R}^{N \times D}$.
where $o_{nd} = \max(x_{nd}, 0)$.

Given a backpropagated gradient, $\delta \in \mathbb{R}^{N \times D}$, `relu_backward` backpropagates the gradient to the:

- inputs: $\frac{\partial E}{\partial x_{nd}} = \delta_{nd}(x_{nd} > 0)$.

1.2 Gradient of Linear Layer (40 pts)

Implement `linear_forward` and `linear_backward` in `assignments_code/assigment3.py`.
This linear layer takes as input a D dimensional vector and outputs a K dimensional vector.

There is a weight matrix $\mathbf{W} \in \mathbb{R}^{K \times D}$ and a bias vector $\mathbf{b} \in \mathbb{R}^{K}$.
`linear_forward` computes $\mathbf{O} \in \mathbb{R}^{N \times K} = \mathbf{X}\mathbf{W}^T + \mathbf{b}$
$o_{nk} = \sum_{i=0}^{D} x_{ni} w_{ki} + b_k$

Given a backpropagated gradient, $\delta \in \mathbb{R}^{N \times K}$, `linear_backward` backpropagates the gradient to the:

- weights: $\nabla_{\mathbf{W}} E = \delta^T \mathbf{X}$, $\frac{\partial E}{\partial w_{kd}} = \sum_{n=0}^{N} \delta_{nk} x_{nd}$
- biases: $\frac{\partial E}{\partial b_k} = \sum_{n=0}^{N} \delta_{nk}$
- inputs: $\nabla_{\mathbf{X}} E = \delta \mathbf{W}$, $\frac{\partial E}{\partial x_{nd}} = \sum_{k=0}^{K} \delta_{nk} w_{kd}$

2 Deep Q-Network (DQN)

2.1 Implement a DQN (10 pts)

Implement `get_dqn` in `assignments_code/assignment3.py`.

The implementation should be in PyTorch. The DQN should take an input for dimension 100, followed by a hidden layer of size 75, followed by a hidden layer of size 50, and, finally, followed by an output layer of size 4. The hidden layers should have a ReLU activation function while the output layer should have a linear activation function. The output layer represents $\hat{q}(s, a, \mathbf{w})$ for all four actions.

See the slides on PyTorch and the PyTorch tutorial `https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`.

2.2 Deep Q-learning (40 pts)

Implement `deep_q_learning_step` (shown in Algorithm 2) in `assignments_code/assignment3.py`.

Like in the previous homework about Q-learning, we will be implementing the Q-learning algorithm with a deep neural network. To see the answers to Homework 2, see Course Content - Homework - Homework2.

These commands have already been run:

```
torch.set_num_threads(1)
device: torch.device = torch.device("cpu")
dqn: nn.Module = get_dqn()
optimizer: Optimizer = optim.Adam(dqn.parameters(), lr=0.001)
```

Running the code:

```
python run_deep_q_learning.py --map maps/map1.txt --wait_greedy 0.1
```

Compare your results to the solution videos provided in Blackboard under Course Content - Homework - Homework 3. Keep in mind that this algorithm is stochastic, so results will not be exactly the same.

Algorithm 1 Deep Q-learning

```
procedure Deep Q-learning($\gamma, \epsilon, \alpha, B$)
    Initialize $\hat{q}, \hat{q}_\tau$                                  ▷ DQN and target DQN
    $D = []$                                                ▷ initialize replay buffer as empty list
    for $episode \in 1...N$ do
        Initialize $s$
        for $t \in 1...T$ do
            $s, \hat{q}, D$ = Deep_Q_learning_Step($s, \hat{q}, \hat{q}_\tau, \gamma, \epsilon, \alpha, B, D$)
        end for
        if Update then
            $\hat{q}_\tau = \hat{q}$
        end if
    end for
    return $\hat{q}$                                           ▷ Approximation of $q_*$
end procedure
```

Algorithm 2 Deep Q-learning Step

```
procedure Deep_Q_learning_Step($s, \hat{q}, \hat{q}_\tau, \gamma, \epsilon, \alpha, B, D$)
    Sample an action $a$ from $\epsilon$-greedy policy derived from $Q$
    $s', r$ = env.sample_transition(s,a)
    Store transition $(s, a, r, s')$ in $D$
    Sample a batch of data $(\mathbf{X}, \mathbf{y})$ of size $B$ from $D$
      $\mathbf{X} \in \mathbb{R}^{B \times d}$ where $d$ is the dimension of the input to $\hat{q}$
      $\mathbf{y} \in \mathbb{R}^B$
      $\mathbf{x}_i$ = env.state_to_nnet_input($s_i$)
      $\mathbf{x}'_i$ = env.state_to_nnet_input($s'_i$)
      $y_i = \begin{cases} r_i & \text{if } s'_i \text{ is terminal} \\ r_i + \gamma \max_{a'} \hat{q}_\tau(x'_i, a', \mathbf{w}^-) & \text{otherwise} \end{cases}$
    Calculate $\nabla_{\mathbf{w}} E(\mathbf{w})$ with error $E(\mathbf{w}) = \frac{1}{B} \sum_{i=1}^{B} (y_i - \hat{q}(x_i, a_i, \mathbf{w}))^2$
    Update $\mathbf{w}$ with learning rate $\alpha$
    return $s', \hat{q}, D$
end procedure
```

What to Turn In

Turn in your implementation of `assignments_code/assignment3.py`.