CEDL HW2 Report

Roy, Colin, Tommy, Jeff

1. Problem 5

We observe the variance of advantages and training performance before and after adding baseline. Specifically, we observe the change of the advantages without normalized. Since normalized advantages have zero mean and unit variance no matter we enable the baseline or not.

1.1. Let's define some terms :

1. *Iteration to convergence*: the number of iteration our training algorithm need to reach convergence.
2. *Average advantage variance*: the average of advantage variance from different sample path when our training algorithm reach convergence.
3. *Performance*: the final optimization result of our policy.

1.2. How we compute the variance :

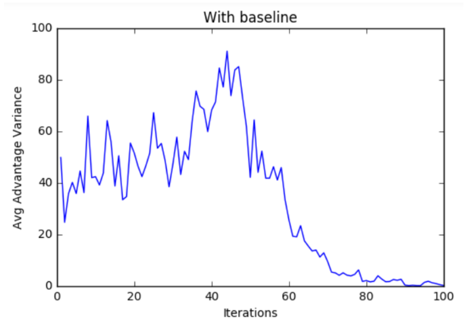
1. In the method **`process_paths`** of class `PolicyOptimizer`, we first compute the variance of advantage for each time stamp in a sample path. Then return the results of different paths as an numpy array.
2. In the method **`train`** of class `PolicyOptimizer`, we compute the mean of variance from different paths, stored in `data["adv_var"]` as the average advantage variance of k-th iteration.
3. Namely, we compute the variance of advantage of different states in one path, and the take the mean of that from every paths in a iteration.

1.3. Experiments and results:

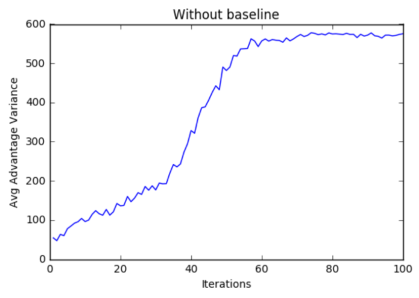
1. Perform training with baseline for 5 times, record the iteration to convergence and average advantage variance.
2. Perform training without baseline for 5 times, record the iteration to converge and average advantage variance.

time	Iteration to convergence Baseline	No Baseline	Avg. advantage variance Baseline	No Baseline
1	69	87	7.38	574.44
2	58	88	6.97	576.53
3	49	67	5.39	571.55
4	73	90	5.97	573.71
5	70	60	5.49	572.03
Avg	63.8	78.4	6.24	572.18

Table 1. Compare converge speed and variance with and without baseline subtraction. **Baseline** means subtracting baseline from the value function

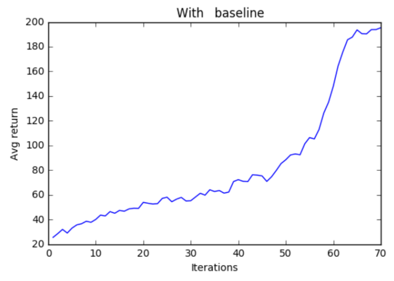


(a)

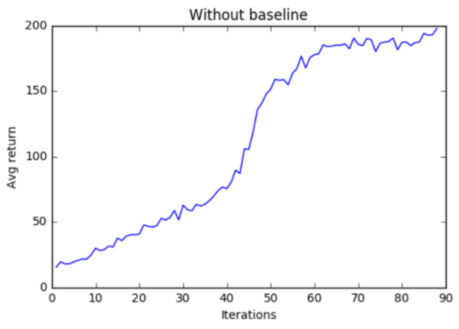


(b)

Figure 1. Compare the change of average advantage variance in training process



(a)



(b)

Figure 2. Compare the training performance with and without baseline

1.4. Discussions

1. We find that average advantage variance will converge to a small value if we subtract baseline while training. Otherwise, the variance will converge to an interval of [570, 575].
2. Also during the training process, baseline subtraction always gives lower variances comparing to no baseline subtraction. The difference becomes more significant at larger iterations.
3. It's reasonable referring to the math. When training is getting closer to the optimal policy, the return of each state, namely the estimated value function, should also get closer to the optimal value function, which is the baseline here. So the advantage of each state will approach to zero, and thus the variance will be smaller.
4. If we don't subtract baseline, the return of each state in a path will suffer from the bias of different reward accumulating length. Thus give rise to the big variance in the gradient of objective function. That cause more noisy update to the weights of policy function.
5. When it comes to training convergence speed, subtracting baseline leads to faster convergence around 10 iteration. That's because lower variances result in much more stable gradients.
6. As for performance, there seems no big difference in this task whether subtract baseline or not.

2. Problem 6

2.1. Convergence efficiency

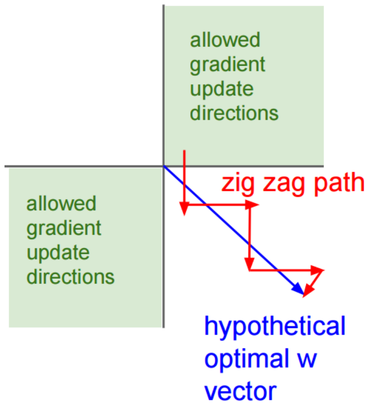
For example,

$$a_j = \sum_i f(w_i x_i + b_i) \tag{1}$$

$$\nabla_{w_i} a_j = \nabla_{w_i} f * x \tag{2}$$

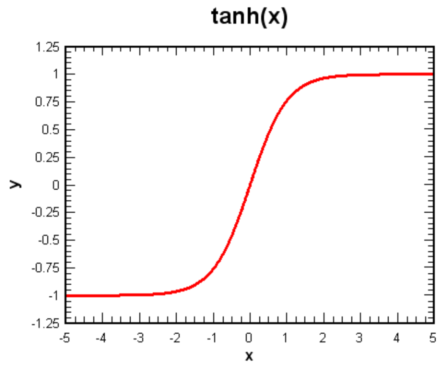
x is input data, w is weight, b is bias and f is activation function

If all $x_i's$ are positive, then the gradients of equation (1) for each weight parameter w_i will be all positives or all negatives. This restricts the update direction of each weight to have the same tendency.


allowed gradient update directions
zig zag path
allowed gradient update directions
hypothetical optimal w vector

2.2. Weight initialization

We use `tahn` as activation function for each neuron in this homework.



As you can see from the figure above, `tanh` will saturate at large values, causing the problem of **gradient vanishing** . So it is crucial to initialize the weights in a way such that the output of each neuron will fall in the range of [-1,1] mostly.

The output of neuron depends on the input, and the very first input of the neural network is the input data. If we normalize the input data to have zero mean and unit standard deviation, we can then initialize our weights based on the normalized data distribution.

We use Xavier initialization method in this homework:

Python code for understanding

```
W = np.random.randn(in_dim, out_dim) / np.sqrt(in_dim)
```

We use Gaussian distribution instead of uniform distribution of the original paper.

$$w_i \sim N(0, \frac{1}{\sqrt{in\_dim}}) \tag{3}$$

With weight distribution above, we can expect that the output values will have *mean* of 0 and *variance* of 1 because of the following properties:

$$\begin{aligned} Var(X+Y) &= Var(X) + Var(Y) \\ Var(XY) &= Var(X) \times Var(Y) \end{aligned}$$

Data normalization and proper weight initialization can stable the gradient propagation, avoiding gradient vanishing and exploding. These are important for neural network to actually learn something.

If we change our weight initialization method to `Glorot and Bengio (2010)`'s method:

$$stddev = \sqrt{\frac{6}{in\_dim \times out\_dim}} \tag{4}$$

We can further **reduce the convergence iteration to about 40 iterations**. This states that weight initialization is crucial for convergence efficiency.