Justin Teufel

CSPB 3202 Project

https://github.com/jteufel/CSPB_3202_FinalProject

Description

The following project is a reinforcement learning agent tested on the Open AI Gym **Pendulum-v0** environment. The base architecture of my agent is a **Dense Neural Network** with Hyperparam Tunning.

Approach/Architecture

My agent builds on an existing agent designed for the **Pendulum-v0** environment, attempts to improve training time while maintaining the performance of the agent as much as possible through an iterative process:

1. Introduce structural changes to the code to improve efficiency - reduce the number of model training iterations and modify hard-coded training parameters.
2. Make adjustments to the layering in the DNN.
3. Introduce Hyperparam Tunning.

The core architecture of chose to use a Dense Neural Network (DNN). A DNN consists of a series of *dense layers* - which are 'fully connected' to the input. These are paired with an activation function/layer which is responsible for updating the weights of the neural network. Below is a description of the initial layers in the DNN.

1. **Dense Layer with ReLU Activation:**
    - Consists of a dense layer and an activation layer. The dense layer consists of 128 'nodes', with connections to each of the inputs.
2. **Dense Layer with ReLU Activation:**
    - Consists of a dense layer and an activation layer. The dense layer consists of 64 'nodes', with connections to each of the 128 inputs.
3. **Dense Layer with Linear Activation:**
    - The final dense layer consists of 10 nodes from the 64 input nodes.

Results - Step 1

My first step was to introduce a series of structural changes to the training code to reduce the number of training iterations /model fits. The model is trained by generating a series of random samples in the format: **(existing state, action, reward, new state)**. The existing training loop was refitting the model on each new sample collected - so a training iteration of 500 samples would call a model fit 500 times. I improved this by reformatting the code such that for each training iteration, the model would only be fit once on multidimensional data. With this change, I also increased the number of passes through the training data (epochs) - one training iteration required 20 epochs.

The following changes resulted in a drastic decrease in training time - down to roughly 200-300 seconds from roughly 1 hour. However, there was a noticeable drop in the effectiveness of the agent. In ten trails, the agent was only successful 4/10 times (in the best case), as compared to 10/10 for the original. In order to improve this, I increased the sample size up to 2000 from 500 - this increased the training time to roughly 18 minutes, but the agent was successful 9/10 times (See **demo1.mp4** and **agent.py**)

Results - Step 2

My next step was to make changes to increase the complexity of the DNN, while also implementing measures to limit overfitting. My approach for this model was to implement a series of new dense layers, with increased connectivity, each followed by a dropout layer. The purpose of the dropout layer is to limit overfitting given the increased node count of the dense layers - it allows for a certain percentage of connections to be 'ignored' as defined by the dropout rate. Below is the updated ordering of the DNN:

1. **Dense Layer with ReLU Activation:**
    - 512 nodes total
2. **Dropout Layer:**
    - Dropout rate of .8
3. **Dense Layer with ReLU Activation:**
    - 256 nodes total

4. **Dropout Layer:**
    - Dropout rate of .8
5. **Dense Layer with ReLU Activation:**
    - 128 nodes total
6. **Dropout Layer:**
    - Dropout rate of .8
7. **Dense Layer with ReLU Activation:**
    - 64 nodes total

This model was first trained with a sample size of 500 - this resulted in an agent that was quite unsuccessful in the pendulum environment (0/10 successes on each attempt). In response, I removed layers 1-4 and increased the sample size - while the agent appeared to perform slightly better, it still had a 0/10 success rate.

(See **demo2.mp4** and **agentDNN.py**)

Results - Step 3

My next step was to attempt to introduce Hyperparam tunning in order to optimize some of the parameters in the neural net, to allow for more efficient training. The layers of the neural network were modified only slightly from step 1 for this iteration - 1 flattening layer, followed by 2 dense layers. The following hyperparameters were introduced:

1. **Dense Layer 1 - Units:**
    - Optimize the number of nodes in the first dense layer - in 32 node increments ranging from 64 to 256.
2. **Learning Rate:**
    - Change in the model in response to the estimated error on each weight update.

Unfortunately due to some encountered bugs - I was not able to successfully complete agent training. All code for Hyperparam tunning is included in agentHyperParam.py.

Conclusion

Below are a few key observations/points from my implementations I took note of:

1. **Attempts to reduce training time all resulted in a drop in agent performance/accuracy.**
2. **Attempts to combat overfitting did more harm than good.**
3. **Increase sample sizes for each training iteration was by far the most effective technique in improving agent performance/accuracy.**

Overall, it was evident that there is a tradeoff between agent performance and training effectiveness for the architectures/techniques I used for this environment. It also seemed that the environment has a high overfitting threshold since the original model was the best performing and the overfitting reduction techniques I used in Step 2 drastically hindered the agent performance. The sample size observation is evident based on the results of step 1.

A possible improvement could could come from successful implementation of hyperparam tunning.

References.

1. **https://github.com/gregd190/OpenAI-Gym-PendulumV0**
2. **https://www.tensorflow.org/tutorials/keras/keras_tuner**