Ahmed El Sheikh – 1873337
Ismagil Uzdenov – 1873718
Michal Ostyk-Narbut – 1854051

July-2019

La Sapienza University of Rome

Population Based Training of Neural Networks

Presented to Prof. Aureilo Uncini & Prof. Simone Scardapane

Contents

1. Introduction

Neural Networks showed great success in most of the domains they are used in starting from audio/images classification, playing video games, as well as, machine translation, and other tasks Natural Language Processing (NLP) related.

Building and training a neural network for a specified task is easy, however, optimization of networks is still a hard task to do, especially when dealing with General Adversarial Networks (GANs) or Temporal Difference/Actor Critic based Deep Reinforcement Learning techniques. As there are a lot of hyper parameters to tune, this tuning can be done using different optimization techniques.

Hyperparameters are the set of parameters that define how the model will be structured. It can be thought of it as searching parameters space to find the best parameters. i.e. our aim is to find the optimal set of hyperparameters as per task. Generally, this process can be broken down into the following:

1. Defining and building the model
   - Task specific
2. Define a range of possible values
     - Recurrent Neural Network LSTM based
     - Dropout
     - Recurrent Dropout
     - Batch Size
     - Number of Epochs
   - Decision Trees classifier
     - Number of trees
     - Maximum depth
3. Define an evaluation criterion

2. Optimization Techniques

Optimization techniques can be categorized into 2 classes:

1. Parallel Search: many parallel optimization processes, train multiple networks with different set of hyperparameters, for example Grid Search & Random Search.
2. Sequential Optimization: it follows the same paradigm as parallel search for few iterations of optimization and get the output of these iterations utilizing them to improve NNs performance gradually, for example Manual tuning & Bayesian Optimization.

- Sequential optimization will be better compared to parallel, however, it is not feasible for long optimization processes.

1. Grid Search: Most basic tuning method, which is based on building and training models as per each set of hyper parameters, which is very time and resources consuming, e.g. if we have a model and 100 combinations of hyperparameters, this means we have to wait for training of 100 models and then pick the best. This illustrates the suffering of Grid Search if the number of parameters grow.
2. Random Search: several neural networks are built and trained asynchronously, and the best one in terms of performance is selected. But, all NNs are trained at the same time regardless of how promising the network will be at the end. It is easy to notice that some NNs are not as good and will consume computational resources that can be saved.
3. Manual Tuning: Deep Learning practitioner has to guess a set of hyper parameters and train the NN. This is a basic iterative optimization method which takes a lot of time, which can be invested in something else, which increase the chances of finding best fit params comparatively higher; as the random search ends up optimizing parameters without any aliasing.
4. Bayesian Optimization: Unlike what was mentioned earlier, it keeps track of past evaluation results, which are used to form a probabilistic model mapping.

3. Population Based Training of Neural Networks

It basically is an optimization technique that aims to get the best of both worlds parallel and sequential optimization. So, it trains multiple networks at the same time, and also able to use fewer computational resources in comparison with random/grid search. It leverages information sharing across a population of concurrently running optimization process and allows for online transfer of hyperparameters between members based on performance.

It starts like parallel search by randomly sampling hyperparameters and weights initialization of the model. Nevertheless, each training runs asynchronously evaluate its performance periodically. If the model is under-performing, it will exploit the rest of the population by replacing itself with a better performing model, and it will explore new hyperparameters by modifying the better model's hyperparameters, before training is continued. The result is a hyperparameter tuning method that while very simple, results in faster learning, lower computational resources, and often better solutions.

So, simply, it is an asynchronous optimization algorithm which effectively uses a fixed computational budget to jointly optimize a set of neural networks (will be referred to as "population" later) and their hyperparameters to maximize the performance.

PBT discovers a schedule of hyperparameters settings rather than following the generally sub-optimal strategy of trying to find a single fixed set to use through the whole course of training.

3.1. Problem Formulation

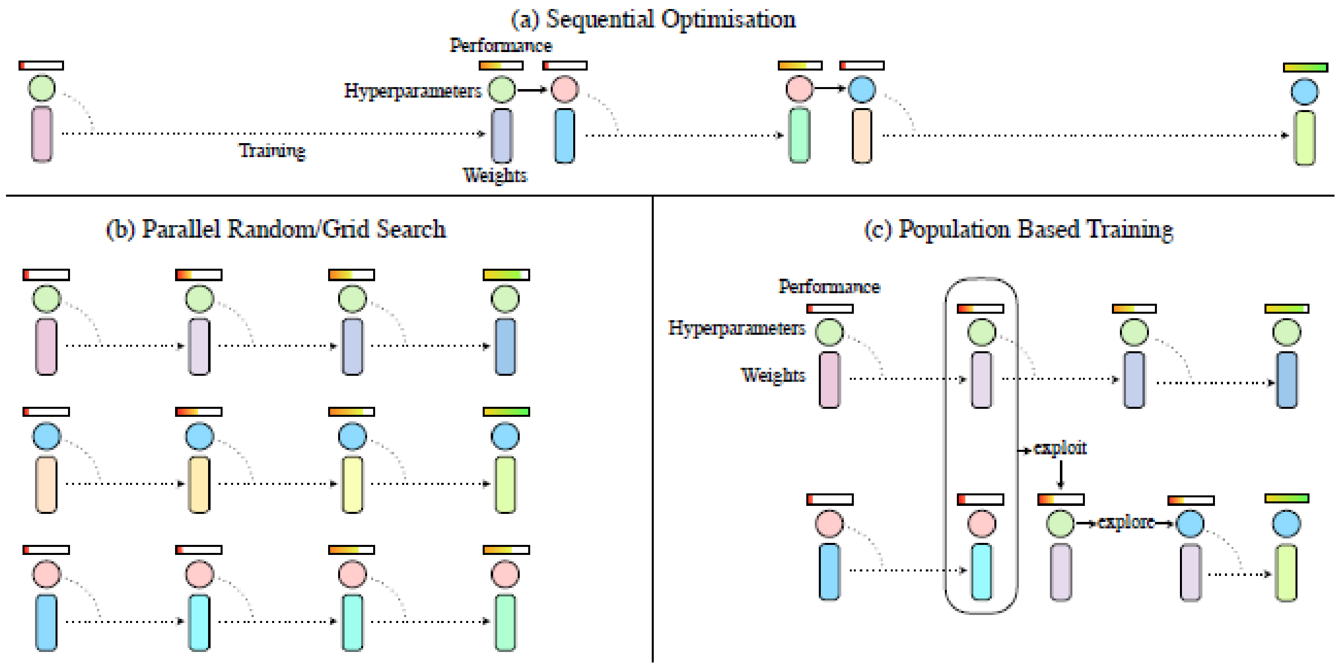
In Neural Networks our aim is to optimize the parameters of the network θ of a model f to maximize a given objective function $Q^\wedge$. Updating the parameters is done iteratively using stochastic gradient descent (SGD), updates are done based on performance metric Qwhich is different from $Q^\wedge$, which can be accuracy on validation set, inception score for measuring performance of GANs, or BLEU for neural machine translation. PBT aims to improve both weights $\theta$and hyperparameters h jointly.

To find optimal model weights

$$\theta \quad = argmax\big(eval(\theta)\big); \theta \in \Theta$$

This iterative approach of optimizing the weights is computationally expensive, due to the number of required steps, as well as, the computational cost of every step. This process might even take days to do. Not to mention that, the hyperparameters have a huge impact on how the model will perform, so if they were chosen haphazardly, the model might not converge.

Refer to figure 1 for visual explaining PBT in comparison with parallel and sequential optimization.



3.2. Methodology

PBT can be broken down into 2 methods which can be invoked independently on each member of the population

1. Exploit: which selects to abandon the current member and focus on more promising members, given the performance of the whole population
2. Explore: which given the current solution and hyperparameters proposes new ones to possibly improve the solution space.

As mentioned earlier, each member is trained individually synchronously with other members, with iterative calls of “step” to update the member weights and evaluate its current performance. But, if a member reaches the threshold of performance or is deemed ready, its weights and hyperparameters are updated using “exploit” and “explore”. Exploit to replace the current weights with the weights of the best model and Explore to randomly perturb the hyperparameters with noise. And then it continues training iteratively until convergence (repeating the previously mentioned approach). Eval was mentioned multiple times, with no clarification, so eval is the mean episodic return or validation set performance of the metric used to optimize the networks. Exploit selects another member of the population to copy weights and hyperparameters from. Explore, creates new hyperparameters for the next timesteps of gradient based learning either by perturbing the current parameters (by adding noise to it) or resampling the hyperparameters from originally defined prior distribution.

Combining all of that, PBT benefits from local optimization by GD, and periodic model selection, as well as, hyperparameters refinement from process similar to genetic algorithms. Refer to figure 2 for pseudo code.

Algorithm 1 Population Based Training (PBT)

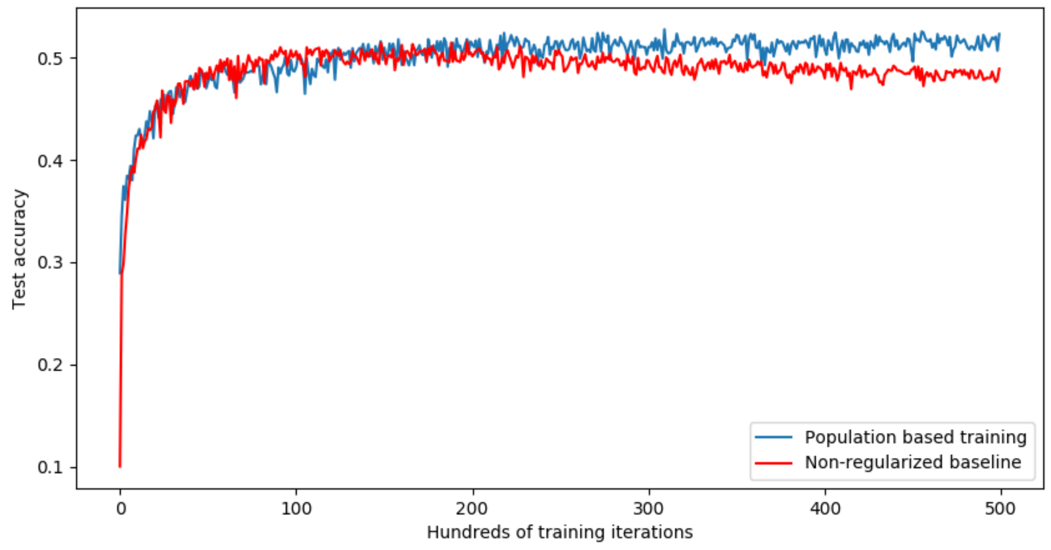
```
1:  procedure TRAIN($\mathcal{P}$)                                    ▷ initial population $\mathcal{P}$
2:      for $(\theta, h, p, t) \in \mathcal{P}$ (asynchronously in parallel) do
3:          while not end of training do
4:              $\theta \leftarrow$ step$(\theta|h)$                  ▷ one step of optimisation using hyperparameters $h$
5:              $p \leftarrow$ eval$(\theta)$                         ▷ current model evaluation
6:              if ready$(p, t, \mathcal{P})$ then
7:                  $h', \theta' \leftarrow$ exploit$(h, \theta, p, \mathcal{P})$   ▷ use the rest of population to find better solution
8:                  if $\theta \neq \theta'$ then
9:                      $h, \theta \leftarrow$ explore$(h', \theta', \mathcal{P})$   ▷ produce new hyperparameters $h$
10:                     $p \leftarrow$ eval$(\theta)$                 ▷ new model evaluation
11:                 end if
12:             end if
13:             update $\mathcal{P}$ with new $(\theta, h, p, t + 1)$   ▷ update population
14:         end while
15:     end for
16:     return $\theta$ with the highest $p$ in $\mathcal{P}$
17: end procedure
```

4. Experiments

Experiments were done in 4 different domains, first of them, Images Classification using CIFAR-10 dataset, secondly, GANs using CIFAR-10 dataset, third, Deep Double Sarsa on OpenAI Gym's LunarLander-v2 environment, as well, lastly, Neural Machine Translation using Europarl v7 dataset. All of the experiments were implemented using TensorFlow or Pytorch which are deep learning libraries built on top of Python.

4.1. Images Classification

The network used was with standard fully connected layers and applying l1 regularizers. Regularizers are one of the approaches to combat overfitting, regularization boils down to adding a penalty term to our objective function, which allows us to control our model complexity. Back to the experiment, in this experiment the layers' capacity, in addition to, l1 regularization value will be tuned by PBT, training a model regularly for 50K iterations, and then training a population of 10 neural networks for same number of iterations. As it was mentioned earlier, PBT follows 2 main steps exploit, explore, and of course we evaluate at frequency of 100 iterations.
Exploit the best 3 models, by replacing the rest with the best hyperparameters and weights.
Explore by perturbing (adding noise to our regularizer value) to the worst 3 models. Noise is of type normal distribution with a mean of 0.0, and standard deviation of 0.5
Evaluation was done on the based on the accuracy of prediction.



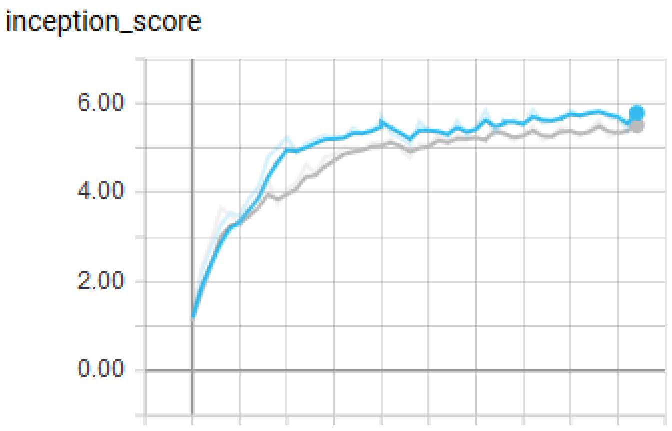
As we can see PBT started to show differences after the 20k iteration and increase on test accuracy, however the test accuracy was low based on the fact that the simplest model was used to train on CIFAR-10.

4.2. GANs

Generative Adversarial Networks are networks that have the ability to generate new content from the input's probability distribution or in other words GANs can be taught to mimic the input/generating outputs similar to inputs. GANs are made up of 2 ANNs one known as Generative Network which models the distribution between individual classes, and Discriminative models which learns how to classify them. Optimizing GANs hyperparameters is one of the most fragile optimization problems, unlucky random initialization might cause the Network to diverge.
As per this experiment we are trying to optimize the learning rates of both generative and discriminative models, by creating a population of 20 workers and evaluate model using inception score which measures the quality of samples produced, and their diversity.

2D Conv >> LeakyReLu >> 2D Conv >> BatchNorm >> 2D Conv >> Batch Norm followed by output layer

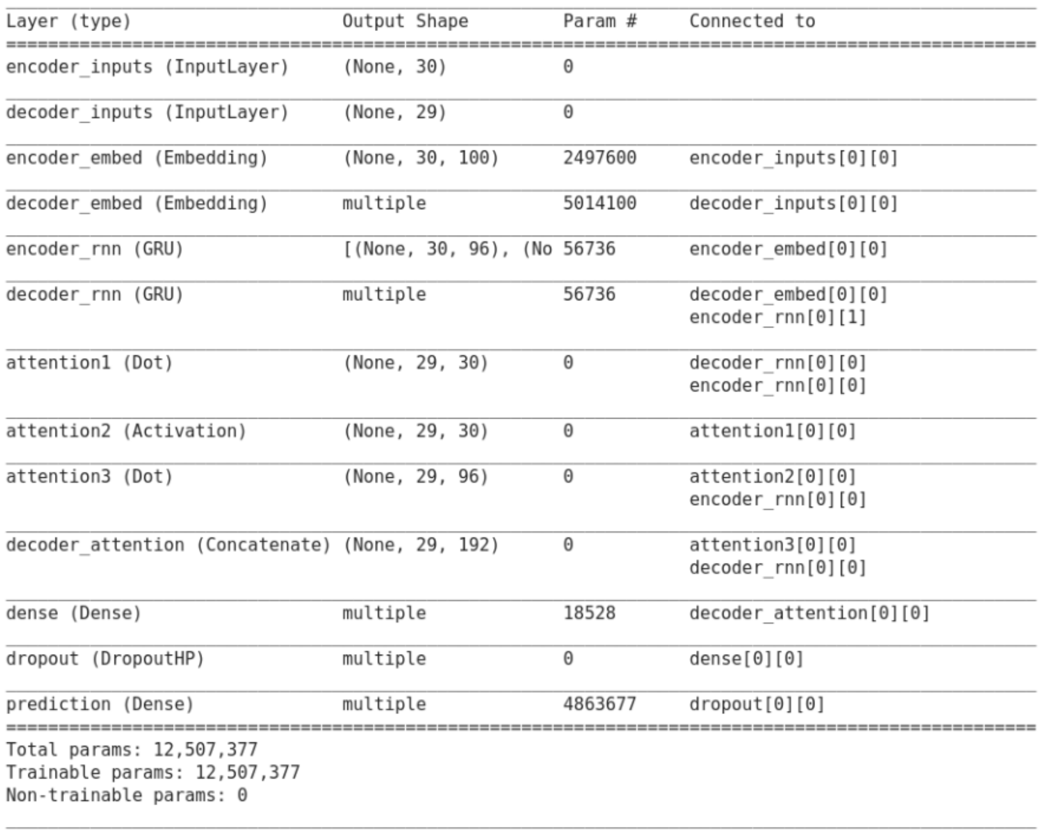
The model was trained on CIFAR-10 dataset, 64 batch size, evaluation interval is based on number of epochs multiplied by the number of batches. As mentioned earlier, exploration and exploitation are depending on whether this member belongs to the top 20% or the worst 20% of the population.



Inception score of the network (Blue is PBT score, Gray is baseline model)

4.3. NMT

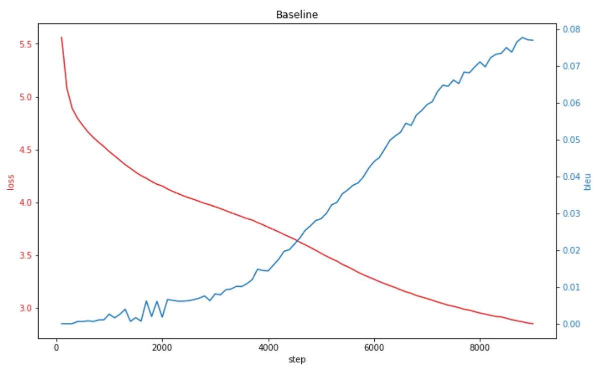
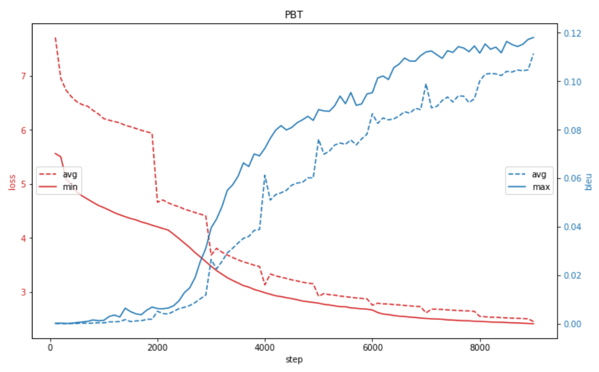
Neural Machine translation (NMT) is basically machine translation but aided with Artificial Neural Networks, this experiment was approached by several steps starting with downloading and understanding the English-to-German europarl v7 dataset, needless to say, the process of training requires pairs of English-to-German sentences, the dataset contains 1.9 million sentences, training on all of that will require too much time, so only 40% of the dataset was used, which corresponds to 576k sentences.
Before training the model, data was parsed, tokenized, trimmed and padded to length of 40 tokens as per sentence, as well as, replacing the words that occurred less than 5 times with <UNK> token, so to reduce our vocabulary size. So, our input shape was [768083, 40].
Given that the problem is a translation problem, so a sequence of words is our input, and expected to have a sequence of words as our output, hence a sequence to sequence model was implemented. This model is made up of 2 RNN (recurrent neural networks), one accepts a sequence and produce a corresponding context, which is known as context vector known as Encoder RNN, the other RNN uses the context vector, as well as, the target sequence to predict the output word by word. During training phase both networks are jointly trained.



```
________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================================
encoder_inputs (InputLayer)     (None, 30)           0
________________________________________________________________________________________________
decoder_inputs (InputLayer)     (None, 29)           0
________________________________________________________________________________________________
encoder_embed (Embedding)       (None, 30, 100)      2497600     encoder_inputs[0][0]
________________________________________________________________________________________________
decoder_embed (Embedding)       multiple             5014100     decoder_inputs[0][0]
________________________________________________________________________________________________
encoder_rnn (GRU)               [(None, 30, 96), (No 56736       encoder_embed[0][0]
________________________________________________________________________________________________
decoder_rnn (GRU)               multiple             56736       decoder_embed[0][0]
                                                                 encoder_rnn[0][1]
________________________________________________________________________________________________
attention1 (Dot)                (None, 29, 30)       0           decoder_rnn[0][0]
                                                                 encoder_rnn[0][0]
________________________________________________________________________________________________
attention2 (Activation)         (None, 29, 30)       0           attention1[0][0]
________________________________________________________________________________________________
attention3 (Dot)                (None, 29, 96)       0           attention2[0][0]
                                                                 encoder_rnn[0][0]
________________________________________________________________________________________________
decoder_attention (Concatenate) (None, 29, 192)      0           attention3[0][0]
                                                                 decoder_rnn[0][0]
________________________________________________________________________________________________
dense (Dense)                   multiple             18528       decoder_attention[0][0]
________________________________________________________________________________________________
dropout (DropoutHP)             multiple             0           dense[0][0]
________________________________________________________________________________________________
prediction (Dense)              multiple             4863677     dropout[0][0]
================================================================================================
Total params: 12,507,377
Trainable params: 12,507,377
Non-trainable params: 0
________________________________________________________________________________________________
```

This was the model used for this task, the model was trained only for a single iteration which was taking 23 hours to train, Adam optimizer was used with a learning rate of 0.001 and dropout was fixed to 0.2
To evaluate NMT tasks, there a score metric used known as BLEU which is valid till Quadro-grams -BLEU stands for bilingual evaluation understanding-, perfect score of BLEU is 1.0, worst will be 0.0, it works by

counting the number of matching n-grams in the candidate translation to n-grams, and in the modified version of BLEU, it normalizes the n-grams by their frequency of occurrence. The model was evaluated every 100 steps against a validation set of 6k sentences
So, the baseline model BLEU score was 0.07951, and then a population of 8 models were trained and evaluated synchronously only to tune to hyperparameters learning rate ranges from 1e-4 to 1, and dropout rate of 0.1 until 0.5

 Baseline showing BLEU score of 0.07951 after 8000 steps	 As we can see the effect of PBT reaching better results in 8000 steps

4.4. Deep Double Sarsa

Sarsa is a TD learning algorithm used for control problems. Instead of value function, it estimates state-action values. To do that it takes into account the state, the action, the next state, the next action and the reward, namely the tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Thus, the name, born from the acronym of the tuple, SARSA. The update of the Sarsa is shown below:

$$Q(S_t, A_t) \Leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

The goal of TD algorithms, as is the goal of other reinforcement learning algorithms, is to find an optimal policy that maximizes the expected reward collected by the agent. TD algorithms find the value of a state or the state-action pair value to achieve this goal and the policy is determined according to the learned values.

Double Sarsa is the double learning variation of the Sarsa algorithm. The main idea is to overcome maximization bias problem using two different networks, where one of the networks serves as the reference value for the update of the other. Deep Double Sarsa is the version of Double Sarsa that uses neural networks as it's estimators, instead of tableaus. To train a model like this, a loss function is constructed as shown below:

$$Y^A = r + \gamma Q(s', a'; \theta^B) - Q(s, a; \vartheta^A)$$

By using this loss in backpropagation, the networks learn to estimate the values of the states. Deep Double Sarsa updates the networks uniformly randomly one at an episodic step. The pseudo algorithm for Deep Double Sarsa can be seen below.

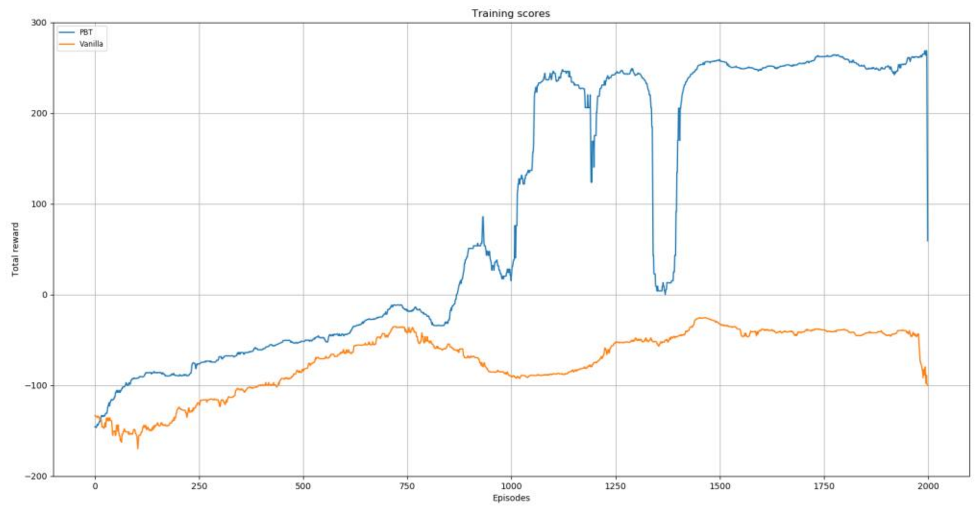
Deep Double Sarsa

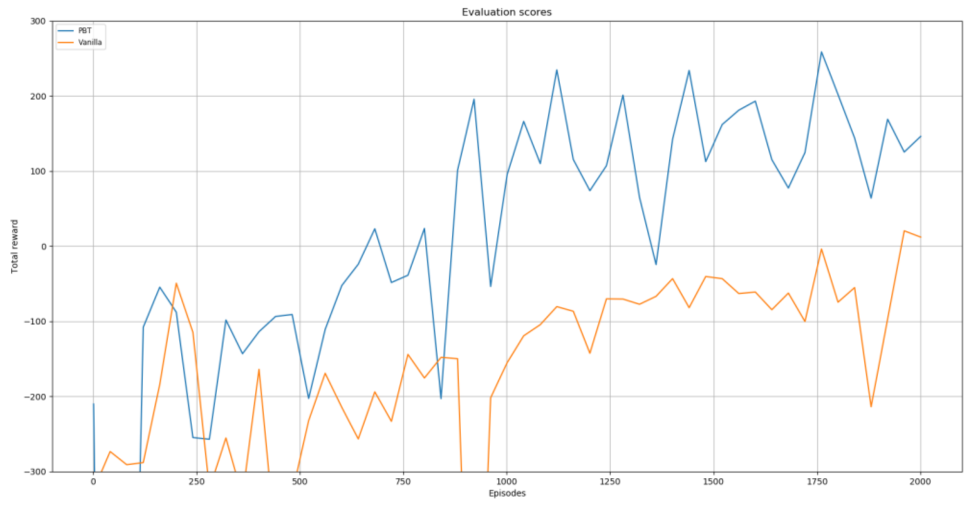
```
1.  Initialize $\theta^A$ and $\theta^B$ arbitrarily
2.  loop{over episodes}
3.      Initialize $s$
4.      Choose $a$ from $s$ using arbitrary policy
5.      repeat{for each step of the episode}
6.          Take action $a$, observe $r, s'$
7.          Choose $a'$ from $s'$ using policy $\pi$ derived from average of $Q(s,a;\theta^A)$ and $Q(s,a;\theta^B)$
8.          $Y^A = r + \gamma Q(s,a;\theta^B) - Q(s,a;\theta^A)$
9.          Train neural network using $\theta^A$ , $\phi(s)$ , and $Y^A$
10.         $s \leftarrow s'; a \leftarrow a'$
11.         with probabilty 0.5:
12.             swap$(\theta^A, \theta^B)$
```

Our implementation of Deep Double Sarsa uses a technique described in DeepMind's Double Q-Learning paper. Instead updating both networks randomly, one of the networks is assigned as the behavior network and the other as the target. The target network serves as the reference value for the behavior network and the behavior network is updated at every step. The target network is updated periodically and during the update it only takes the parameters of the behavior network.

We do the experiment on OpenAI Gym's LunarLander-v2 environment. We train Deep Double Sarsa algorithm by itself and its PBT version. Both models are trained for 2000 episodes with an epsilon value starting from 1.0 and decreasing to the value of 0.1 over time. This is done for exploration of new states during training. Every 40 episodes evaluation of the agents is performed where no update is done, and the epsilon value is 0. The PBT version uses 20 agents, 4 worst of which are changed to 4 random best in the population during the exploration and exploitation sequence.

The results of the experiment show that PBT version clearly outperforms the standard version. This is due to the instability of the Deep Double Sarsa algorithm, i.e. it is prone local minima and one bad update could drop policy’s performance extremely. The PBT version is able to overcome that due to the number of agents and them sharing their parameters. This can be inferred from the training sequence of both models. The maximum total reward during training that standard model achieves is 232.84, which is a high score. But this score is lost during the training due to the aforementioned reasons and is never recovered again. While the PBT version doesn’t lose the performance at that stage and evolves to get better performance, where it’s highest total training reward is 317.407. The training and evaluation scores of both models can be seen in the plots given below.





Improvements and Future Work

Instead of using basic model to train on CIFAR dataset for the images classification, it can be implemented using CNN, which will show way better results as per this Venice boats classification repository classifying 18 different instances of Venice boats.

NMT task can be further improved by using the entire of the dataset instead of a subset, or by using more than one dataset like it was done in this word sense embeddings repository, as well as, padding sentences as per the length of the longest sentence per batch, as It was done herein this Chinese word segmentation repository. Both repositories belong to the author Ahmed El Sheikh.

The training was performed on AWS Tesla K80 GPU, using only a single GPU. The training takes for all of these population members takes a lot of time, i.e. the image classification took approx. 5 hours, NMT consumed 23 hours' worth of GPU computational time, and finally, GANs took the largest share of the time with 48 hours to finish. So, in order to reduce this computational time, either to improve the complexity of the codes, or to use multiple machines.

Conclusion

We tried to augment PBT with different domains of Deep Learning. We have shown how PBT affects the models' performance in the same number of steps showing better results, and how this algorithm has shown consistent improvements in optimizing model weights and hyperparameters. PBT discovers that fixed set of hyperparameters is not as good as adaptive schedule of hyperparameters tuning.

References

1. https://arxiv.org/abs/1711.09846
2. https://deepmind.com/blog/population-based-training-neural-networks/
3. https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f
4. https://medium.com/@senapati.dipak97/grid-search-vs-random-search-d34c92946318
5. https://skymind.ai/wiki/generative-adversarial-network-gan
6. “Reinforcement Learning: An Introduction”, Richard S. Sutton and Andrew G. Barto ,MIT Press, Cambridge, MA, 2018
7. “Double Sarsa and Double Expected Sarsa with Shallow and Deep Learning”, Ganger M., Duryea E., Hu W., Journal of Data Analysis and Information Processing, 2016, 4, p. 159-176
8. “Deep Reinforcement Learning with Double Q-Learning”, van Hasselt H. Guez A. and Silver D., 2015
9. https://gym.openai.com/envs/LunarLander-v2/