Hyperparameters Optimization with Deep Reinforcement Learning

Faisal Mohamed, Amna Elmustafa, Maab Elrashid, Amina Rufai, Ahmed A. A. Elhag
African Institute for Mathematical Sciences
{fmohamed, aelmustafa, mnimir, arufai, aelhag}@aimsammi.org

Abstract

Hyperparameters tuning is important to obtain a state of the art performance in modern deep learning models. There exist several hyperparameters optimization approaches such as grid search, random search and Bayesian optimization. In this work, hyperparameters optimization is viewed as a sequential decision making problem, which is solved by using Deep Reinforcement Learning. Our experimental results in regression and classification tasks show that this method is competitive with other baselines such as grid search.[1]

1 Introduction

The core of most Machine learning problems is "Optimization". We could be dealing with convex or non-convex optimization problems, but in the end, the aim is to find the best possible/optimal solution, to a given problem. Choosing the right hyperparameters is synonymous to finding the best control variables that defines the best model to a given learning problem. However, this process can require the use of a lot of resources in terms of computation and this is not very time efficient. Automating this process increases efficiency and saves computational costs for larger/more complex networks.

Hyperparameters refer to the parameters that are independent of the data but can be adjusted/tuned to improve model performance. They can be categorized into two groups: those used for training and those used for model design.

From previous studies on deep neural architectures, the two most important hyperparameters considered during training are batch-size and learning rate. For model design,the most common are; the number of hidden layers and size of layers. In this study, We explored the use of reinforcement learning for automating hyperparameter search/optimization. The aim of the reinforcement learning approach is to maximize the expected accuracy(reward) of the architecture while finding the best hyperparameter that produces the best possible model performance for the given problem. We implemented this using a simple Multilayer Perceptron(MLP) network, and optimizing the learning rate, weight decay parameter, batch size, and number of hidden units using REINFORCE algorithm for two tasks; regression and classification. the rest of this report is as follows: section discussed some related work, section 3 explain the methodology and the experimental setup, section 4 explains the results and some discussion.

[1]This work is part of Deep RL course in the African Master of Machine Intelligence (AMMI)
The code for this work is in the provided link `https://github.com/AMNAALMGLY/HypOptRL`

2 Related Work

2.1 Previous Search Algorithms

2.1.1 Grid Search

Grid Search is a common Hyperparameters optimization technique. It performs an exhaustive search on a set of specific hyperparameters specified by the user. It is relatively straightforward and easy to implement, as long there are sufficient resources. However, it is not time efficient and might require a lot of computational resources for deeper and more complex networks.

2.1.2 Random Search

This is a slight improvement on the Grid search approach. It indicates a randomized search over hyperparameters from certain distributions over possible parameter values. The searching process continues till the predetermined budget is exhausted, or until the desired accuracy is reached. It performs better than Grid search when some hyper-parameters are not uniformly distributed. However, despite this benefit, it is also a computationally intensive method and requires more computational resources than a Grid search.

2.2 Bayesian Optimization Methods

It is a sequential model-based method aimed at finding the global optimum with the minimum number of trials. It is described as a Global optimization technique, applying the principle of Bayes theorem. It is a robust technique that can deal with noisy black-box functions, otherwise known as empirical risk minimization problems. Bayesian optimization is efficient in tuning few hyperparameters but its efficiency degrades as the search space increases. It was proven to have lower performance than random search with larger search space/search parameters. Also, Bayesian optimization can only work on continuous hyperparameters.

2.3 Reinforcement Learning for Hyperparameters Optimization

2.3.1 Neural Architecture Search with Reinforcement Learning

This method was proposed by Barret Zoph and Quoc V. Le in 2016 [3]. They used a recurrent network to generate convolutional architectures. So, the controller was implemented as a recurrent neural network. It predicts filter height, filter width, stride height, stride width, and the number of filters. The RNN was trained with a policy gradient method, to maximize the expected accuracy of the sampled CNN architectures.

2.3.2 Hyp-RL : Hyperparameter Optimization by Reinforcement Learning

This work is done by Hadi S. Jomaa, Josif Grabocka, and Lars Schmidt-Thieme in 2019 [1]. They proposed a novel policy based on Q-learning, Hyp-RL, that can navigate high-dimensional hyper-parameter spaces. The formulation of hyperparameter response surface of a neural network as an environment with a defined reward function and action space. The agent is an LSTM that learns to explore the hyperparameter space of fixed network topology.

3 Methodology/Frameworks

3.1 Dataset

For the regression task, the tabular dataset used is the winequality-red dataset; it has 12 features and 1599 instances, and the target is to predict the residual sugar of the wine.
For the classification task, we used the letter recognition dataset. The goal is to predict an English letter, and the feature vector size is 16.
Each dataset has been split into training, validation, and test sets, with ratios of 0.7, 0.1, 0.2, respectively.

3.2 Environment

The environment's starting state consists of randomly selected hyperparameters, and after each step in the environment, the new state is the action from the previous time step.
The action space is discrete, each entry in the action vector is the index of the selected hyperparameter from a predefined list of values, an action vector is defined as $\vec{a}$ =[learning rate, weight decay, batch size, no. of hidden units].
The predefined set of values for each hyperparameter are shown below

Learning rate $\in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 3\times10^{-5}, 3\times10^{-4}, 3\times10^{-3}, 3\times10^{-2}, 3\times10^{-1}\}$
Weight decay $\in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 3\times10^{-5}, 3\times10^{-4}, 3\times10^{-3}, 3\times10^{-2}, 3\times10^{-1}\}$
Batch size $\in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096\}$
No. of hidden units $\in \{2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$

Each step in the environment is a training of the machine learning task. Since the goal is to optimize for the best hyperparameter, the natural choice for the reward is the negative validation loss.

$$R(a) = -Validation\ loss \tag{1}$$

The model choice for the machine learning task is a single layer (MLP) with the size of the hidden units selected by the policy.

3.3 Policy Training

The agent was trained with the policy gradient algorithm (REINFORCE) [2], shown in algorithm 1, in which the goal is to find a policy that maximizes the reinforcement learning objective as shown in equation 3; this objective can be approximated by Monte Carlo sampling as shown in equation 2, where γ is a discount factor.

$$J^{\pi_\theta} = E_{\pi_\theta}\Big[\sum_{t=0}^{T}\gamma^t r(s_t, a_t)\Big] \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\gamma^t r(s_t, a_t) \tag{2}$$

$$\theta^* = \arg\max_{\theta} J^{\pi_\theta} \tag{3}$$

Similarly, the gradient is estimated by Monte Carlo sampling and applying the log-trick (also called the score function)

$$\nabla_\theta J^{\pi_\theta} = E_{\pi_\theta}\Big[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\Sigma_{t'=t}^{T}\gamma^{t'-t} r(s_t^{'}, a_t^{'})\Big]$$

$$\approx \tfrac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\Sigma_{t'=t}^{T}\gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \tag{4}$$

Algorithm 1 REINFORCE

1: Initialize random policy parameters θ.
2: **for** k=1,2,... **do**
3: Collect trajectories in the set $D_k = \{\tau_i\}$ by running the policy π_{θ_k} in the environment.
4: Compute the sum of rewards for each timestep as shown in equation (3).
5: Compute the policy gradient $\nabla_\theta J^{\pi_\theta}$ as shwon in equation 3.3.
6: update the policy parameters $\theta_{k+1} \leftarrow \theta_k + \alpha\nabla_\theta J^{\pi_\theta}$.
7: **end for**

3.4 Experimental Setup

The experiments have been conducted for two learning tasks: regression and classification. The training model is a single-layer MLP with hidden units size selected by the policy. The number of training epochs is 15. The agent has been trained with two models: a single layer MLP with multiple heads, each head correspond to a hyperparameter as shown in figure 1, the first and second layers are considered the backbone with 64 and 16 hidden units respectively, all heads are of the same output

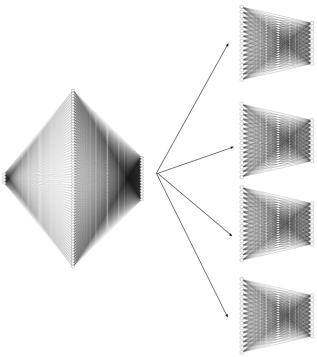


Figure 1: MLP policy

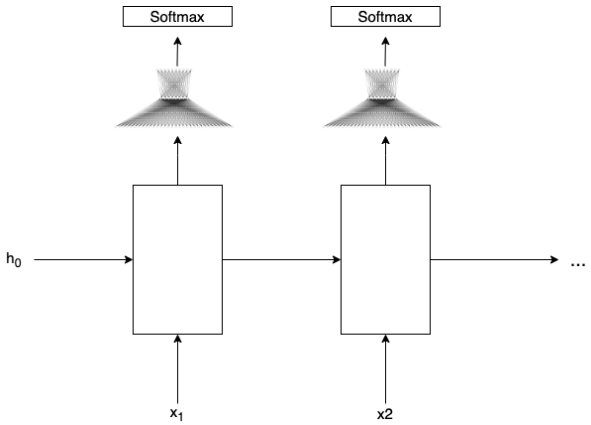


Figure 2: RNN policy

size (10 units) that are passed to a softmax function to sample the value of each hyperparameter, the input size is 4 (number of hyperparameters that are optimized), each entry corresponding to the index of the selected hyperparameter, the learning rate is 10^{-3}, Adam is used for optimization with 300 epochs, the batch size is 16, and the discount factor was set to 0.99. For more details about the experiment a python code is provided in the appendix.

The other policy model experimented also is the sequence model with RNN shown in figure 2, the idea behind using autoregressive model is that when choosing between different hyperparameters, the decison on the previous time steps (previous chosen hyperparameter) affects the prediction of the next hyperparameter, and here comes the RNN with its sequential nature. The input to the RNN is the one-hot encoded states which is then concatenated with the hidden vector to be fed to 2 linear layers with Relu non-linearity, then passed to a softmax to give the probability distribution of the action. RNN and linear layers hidden sizes are chosen to be 16 and 8 units respectively.

4 Results and Discussion

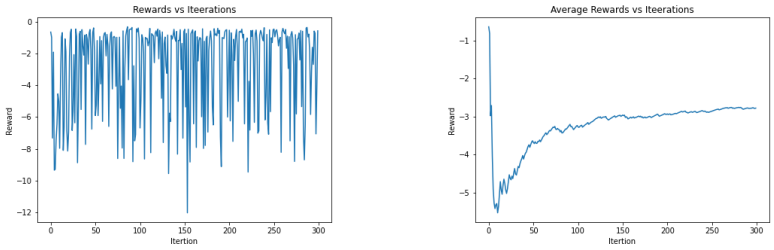
To compare our results, we used the Ray Tune library, for hyperparameter optimization. Because the environment space is too large (we want 10000 trials to do all the hyperparameter combinations), we were not able to complete all the trials due to limitations in the computing resources, for the regression the completed trails are 1610 and for the classification are 489, the results for the regression and classification are shown in table 1 and 2, respectively.

Figures 3 and 4 show the learning curves for the regression task, optimized by the MLP and RNN policies, respectively.
The reward curve in part (a) of both figures is noisy and does not show any trend, in order to extract more useful information from it, the moving average of the rewards has been calculated and plotted as shown in part (b) of both figures.
The reward in the MLP policy starts with a large value, then drops rapidly (this is probably an effect of the random seed), then it grows at a fast rate and starts to plateau after 100 iterations.
In contrast, the RNN policy starts with a low reward then it largely oscillates until the 50th iteration, after that it stabilizes until the end of the training.
The final test loss for the hyperparameters chosen by both policies is shown in table 1 with a baseline for comparison, the best results were from the RNN policy, which introduced a performance gain of about 33% compared with the baseline.

Similarly, Figures 4 and 5 show the results of optimizing the classification task, for the MLP and RNN policies, both of the reward curves in part (a) were smoothed by calculating the moving average.

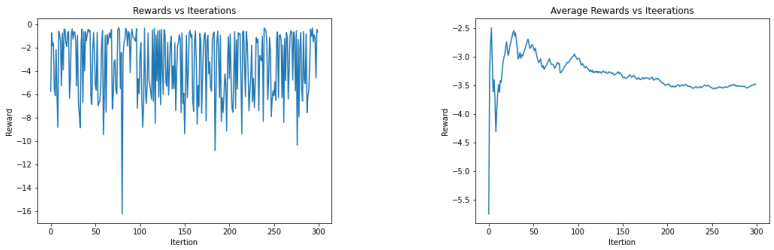
The resulting plot of the MLP policy shows that it starts at a low reward value, then it oscillates rapidly, and after the 50th iteration, it grows almost linearly until the end of the training.
For the RNN policy, the curve starts at a high reward (as stated before, this is probably an effect of the random seed), after that it drops and increases rapidly after the first few iterations, and until the end of the training, the reward increases gradually with minor drops.
The final test loss for the hyperparameters chosen by both policies is shown in table 2 with a baseline for comparison, the best results were from the MLP policy, which introduced a performance gain of about 13% compared with the baseline.

It is important to emphasize that due to computing limitations, all these experiments were done for 300 epochs, REINFORCE is high variance and needs more epochs to reach the optimal performance as indicated in the learning curves. Also, the experiments should be conducted in multiple random seeds, and the resulting figure should include the mean result and the standard deviation among all runs to average out the effect of choosing a single random seed.



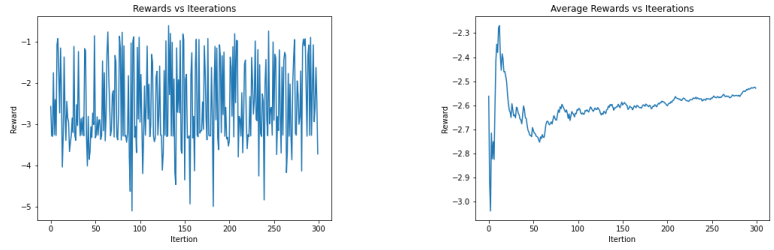
(a) Reward curve (b) Average reward curve

Figure 3: Regression with MLP policy



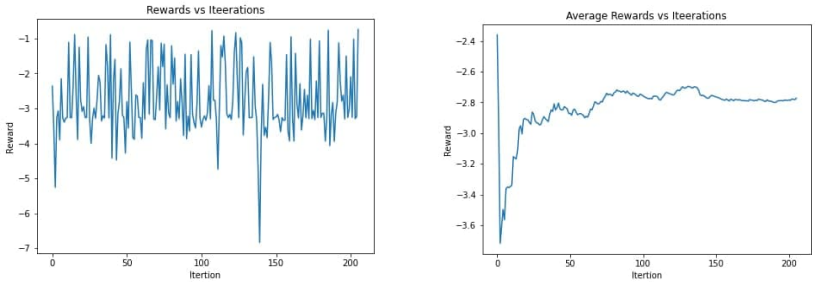
(a) Reward curve (b) Average reward curve

Figure 4: Regression with RNN policy



(a) Reward curve (b) Average reward curve

Figure 5: Classification with MLP policy



(a) Reward curve (b) Average reward curve

Figure 6: Classification with RNN policy

Table 1: Final results of the regression task

Method	Learning rate	Weight decay	Batch size	Hidden units	Test error
Baseline	0.01	10^{-4}	128	18	0.5948
MLP policy	3×10^{-2}	3×10^{-4}	16	18	0.4232
RNN policy	10^{-1}	3×10^{-4}	128	8	**0.3957**

Table 2: Final results of the classification task

Method	Learning rate	Weight decay	Batch size	Hidden units	Test error
Baseline	0.01	10^{-5}	64	20	0.7674
MLP policy	3×10^{-3}	10^{-5}	8	20	**0.6678**
RNN policy	10^{-3}	10^{-3}	8	20	0.7772

5 Conclusion and Future Work

Hyperparameters tuning is important to obtain a state of the art performance in modern deep learning models. There exist several hyperparameter optimization approaches such as grid search, random search and Bayesian optimization. In this work, hyperparameter optimization is viewed as a sequential decision making problem, which is solved by using Deep Reinforcement Learning. Our experimental results in regression and classification tasks show that this method is competitive with other baselines such as grid search.
For future work, more complex tasks and architecture such as CNN for image classification can be tested with more sophisticated Deep RL algorithms, also more baselines are needed such as Bayesian optimization methods to asses the quality of the results.

6 References

References

[1] H. S. Jomaa, J. Grabocka, and L. Schmidt-Thieme. Hyp-rl : Hyperparameter optimization by reinforcement learning, 2019.

[2] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[3] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendix

A Experiment's code

```
def experiment(task="regression", model_type="MLP", dataset=
                                    dataset_regression, n_inputs=11,
                                    n_outputs=1, policy_type="MLP"):
  # define the task
  task = task
  # define the model type
  model_type = model_type
  # dataset
  dataset = dataset
  # create the envirnment model
  env = Env(n_inputs=n_inputs, n_outputs=n_outputs, dataset=dataset,
                                      task=task, model_type=model_type)
  state = env.reset()

  # size of the state
  Ns = env.Ns
  # size of the action
  Na = env.Na
  # number of hyperparamers
  n_hyperparams = env.N_hyperparams

  # create the policy model and the optimizer
  if policy_type == "MLP":
    model = MLP_Policy(Ns, Na, n_hyperparams).to(device)
  elif policy_type == "LSTM":
    model = LSTM_Policy(Ns, Na).to(device)

  optimizer = opt.Adam(model.parameters(), lr=policy_lr)

  # track the learning dynamics
  rewards_per_iteration = []
  probs_per_step = []
  best_reward = -10000

  # maximum length of the trajectory
  Tmax = 1

  for step in range(num_steps):
    # batch_storage
    batch_losses = torch.zeros(batch_size)
    batch_returns = np.zeros(batch_size)

    # batch loop
    for batch in tqdm(range(batch_size)):
      rewards = []
      log_proba = []

      # reset the environment
      state = env.reset()

      for t in range(Tmax):

        # pick an action
        action = model.sample_action(state)
        next_state, reward, done, _ = env.step(action)

        # calculate the log probabilities of the selected actions
        log_probs = log_probabilites(model, state, action)

        rewards.append(reward)
```

```
        log_proba.append(log_probs)

        # iterate
        state = next_state

        if done:
          break

      # compte the policy loss and cum reward for a trajectory
      policy_loss, cum_rewards = compute_policy_loss(gamma, rewards,
                                        log_proba)

      # Store batch data
      batch_losses[batch] = policy_loss
      batch_returns[batch] = cum_rewards[0]

    loss = batch_losses.mean()
    # update the policy
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    rewards_per_iteration.append(batch_returns[-1])

    # save a ckech point in case of a better performance and save the
                                       value of the hyperparmters and
                                       the loss in a csv file
    if batch_returns[-1] > best_reward:
      test_loss = env.evaluate()
      best_reward = batch_returns[-1]
      print(f"New Best reward of {batch_returns[-1]}, test_loss:{
                                        test_loss},learning_rate={env
                                        .learning_rates[action[0]]},
                                        weight decay={env.
                                        weight_decays[action[1]]},
                                        batch size={env.batch_sizes[
                                        action[2]]}, number of hidden
                                         units={env.n_hiddens[action[
                                        3]]}")
      torch.save(model.state_dict(), f"best_model_{task}_{policy_type}
                                        _policy.ckpt")
      best_hyperparams = [env.learning_rates[action[0]], env.
                                        weight_decays[action[1]], env
                                        .batch_sizes[action[2]], env.
                                        n_hiddens[action[3]]]
      save_data(f"best_hyperparams_{task}_{policy_type}_policy",
                                        best_hyperparams, -
                                        batch_returns[-1], test_loss)


    print('Step {}/{} \t reward: {:.2f} +/- {}'.format(
          step, num_steps, batch_returns[-1], np.std(batch_returns)))


  return rewards_per_iteration
```