REPNET: A Reproductive Training Architecture for Reinforcement Learning Neural Networks

Tiernan Lindauer

August 27, 2023

Contents

Abstract

Since 2001, actor-critic algorithms have combined "the strong points of actor-only and critic-only methods" (Konda & Tsitsiklis, 1999) to reduce the time needed to train a reinforcement learning neural network. The REProductive NETwork architecture, REPNET[1], uses a custom data structure built on top of this idea to train networks in significantly less time. During training, the models in REPNET (called branches) can reproduce, rewarding models that achieve a high running reward change. Branches can also be pruned due to a lack of reproduction, penalizing stagnant branches. These branches are contained in the custom recursive data structure of REPNET. Through the use of automatic add-ons such as Adaptive Pruning Time Adjustment (APTA) and Auto-Generating Nonlinear Reproduction Threshold Functions, these hyperparameters can be tuned without human input. After training the neural network on OpenAI's "CartPole" environment, REPNET reduced the mean of the training episodes by 29.3% and the standard deviation of episodes by 72.62%. Overall, REPNET made reinforcement learning training significantly faster and less variable in comparison to its actor-critic counterpart. Decreasing the training times of reinforcement learning neural networks can accelerate progress in AI research, leading to faster development of technologies that can improve various aspects of the world, from healthcare to environmental sustainability.

1 Introduction

A. Experimental Problem - How can the number of episodes required to train a reinforcement learning model be decreased to improve training efficiency?

B. Hypothesis/Engineering Goal - Implementing REPNET through a recursive tree data structure and series of actor-critic models will decrease the number of episodes required to reach a running reward threshold in comparison to a singular actor-critic model.

C. Rationale - With an increase in the complexity of tasks being solved by machine learning, the training times and computing resources needed to train these models are increasing as well. However, these models can take a significant amount of time to train despite running on state of the art machines, due to the sheer number of parameters in the model to tweak. REPNET is a way to substantially reduce reinforcement learning training times and decrease the amount of training episodes needed to achieve a minimum running reward (i.e. "solving" the task). REPNET performs this through a custom tree data structure containing branches, each with its own model. REPNET applies a dual-sided evolutionary pressure of reproduction and pruning on each branch to select the best possible model from the data structure's active models. Beyond the base implementation, add-ons such as the Adaptive Pruning Time Adjustment (APTA) system and Auto-Generating Nonlinear reproduction threshold functions reduce sources of human error, making them more predictable. These improvements in Reinforcement Learning will allow models to be trained faster, resulting in more research and development in the field of machine learning.

[1]This is different from Google's RepNet, which is based on ResNet and used for analyzing repetition in videos.

2 Literature Review

2.1 Overview

To create this project, the author used the software library TensorFlow (Abadi et al., 2016) to construct the reinforcement learning neural networks. Other libraries such as PyTorch offer similar functionality, but the author is most familiar with TensorFlow due to the wide influence it has.

OpenAI's Gym library (Brockman et al., 2016) was used to create an environment to test REPNET in. CartPole was the specific environment used to test REPNET, due to its simple nature. CartPole requires the agent to balance a pole to within a certain angle range by moving a cart on the floor. The cart is additionally required to stay within a distance from the origin.

Research done on the soft actor critic approach to reinforcement learning (Christodoulou, 2019) was used to demonstrate additional optimizations that could be made to the actor-critic reinforcement learning model.

Long Short Term Memory (Hochreiter, n.d.) is an optimization that can be applied to recurrent neural networks that has a similar concept to REPNET. LSTM creates both long-term and short-term memory through a series of activation functions for the task of sequence prediction. Similarly, REPNET uses the Adam optimizer to make short-term improvements in each branch, while the process of pruning and branch reproduction emphasizes effective long-term trends.

Among many optimizers, Adam (Kingma & Ba, 2014) was used to perform the gradient descent adjustments in the neural network is critical to decreasing training times in comparison to other optimizers like Momentum, Gradient Descent, or Adadelta. Adam overall has faster computation time and requires fewer parameters for tuning, making it ideal.

The actor-critic network structure (Konda & Tsitsiklis, 1999) is what was used as the baseline for testing REPNET and how each branch node trains. Deep Q Networks are the other main type of reinforcement learning neural networks, but were not used due to the nature of their construction.

Deep reinforcement learning has been tested on many different games or simplified environments, such as playing Atari games (Mnih et al., 2013), and provides a representation of how the network will perform on a larger scale or different reinforcement learning application.

Distributed reinforcement learning is available (Ong, Chavez, & Hong, 2015), however it uses deep Q-networks that determine future rewards from many (or all) possible actions the agent can take. This can oftentimes be a large group of tasks, and so is easily distributed across multiple processors/threads. However, actor-critic algorithms are not easily distributed as they attempt to find the most optimal policy directly.

Reinforcement learning is used in many different applications such as vision-based robotics (Wang, Vasan, & Mahmood, 2022) and adaptive traffic signal control (Abdulhai, Pringle, & Karakoulas, 2003), and it is important to note the importance it plays in the machine learning scene.

2.2 An Applied Introduction to Machine Learning

Neural networks, initially invented in 1958 (Rosenblatt, 1958), have taken the world by storm through their wide range of applications, incredible versatility, and surprising accuracy. They have enabled everything from smart thermostats to vision-based robotics (Wang et al., 2022), and become an integral part of our society.

> Neural networks are often used for statistical analysis and data modelling, in which their role is perceived as an alternative to standard nonlinear regression or cluster analysis techniques. (Gurney, 2018)

A simple fully-connected neuron is seen below:

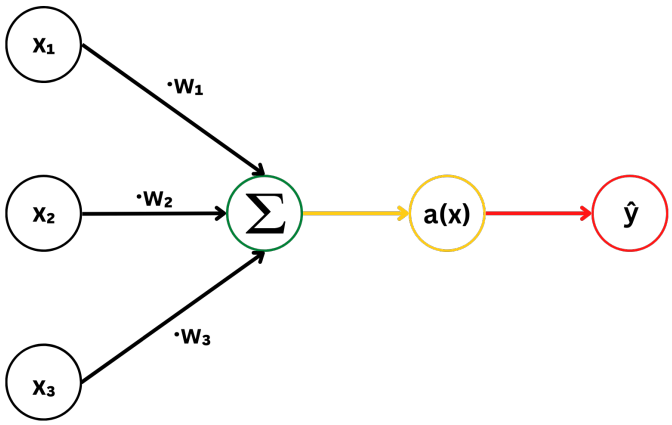
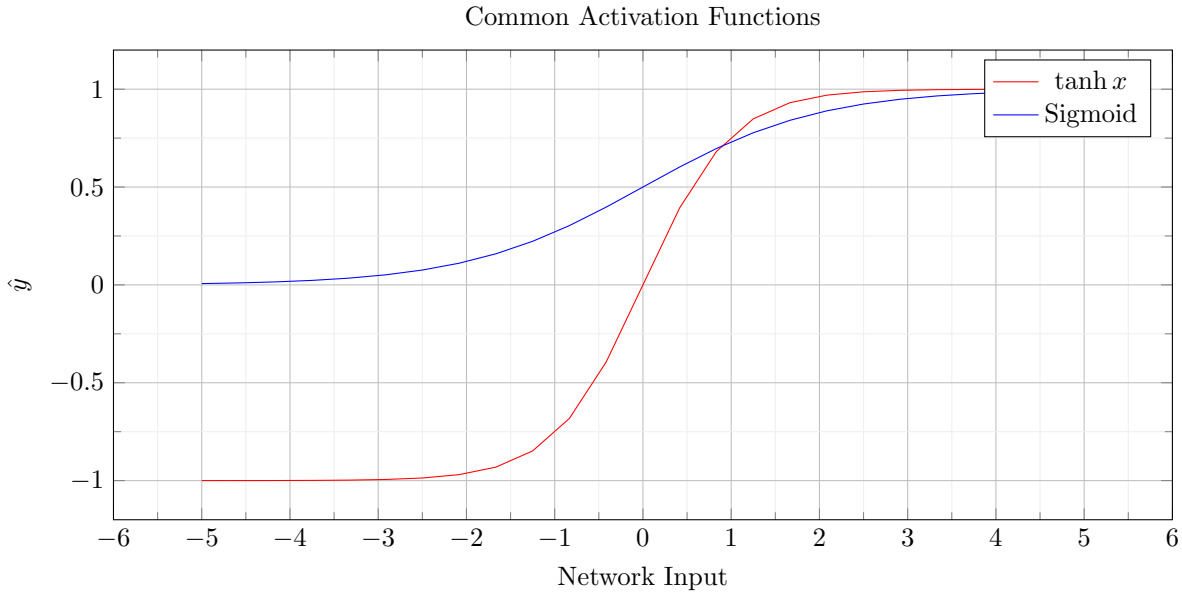


Figure 1: A basic neuron, comprising of a series of inputs, weights, an activation function $a(x)$, and the end result $\hat{y}$

It takes in a set of inputs, either as network inputs or the outputs of a previous layer of neurons, multiplies each input by a weight adjusted using a feed-forward algorithm. The weighted inputs are then summed and then applied through an activation function. The activation function creates an output ($\hat{y}$) based on the specified input; two common activation functions are the sigmoid and tanh functions:



After running the model on a set of input data, the loss function then returns a numerical value describing how got a set of input weights $\mathbf{W}$ are, given loss function $\mathbf{J}$. The loss function used for this project was the Mean Squared Error, as seen below:

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}(y^i - f(x^i; \mathbf{W}))^2$$

The first term used in the loss function inside of the summation is the matrix of labels y^i, or actual values the network should have returned. The second term is the matrix of resultant values after training the neural network based on the matrix of inputs x^i and weights $\mathbf{W}$.

The idea is to then minimize $\mathbf{J}(\mathbf{W})$ in the following way:

$$\mathbf{W}^* = \text{argmin}_{\mathbf{W}}(\mathbf{J}(\mathbf{W}))$$

Based on this loss, the network can then perform gradient descent to update each of the individual weights on the following algorithm:

Algorithm 1 Basic Gradient Descent Algorithm

$\mathbf{W} \leftarrow N(O, \sigma^2)$ ▷ Weights are randomized initially
while $\mathbf{J}(\mathbf{W}) \leq \tau$ **do** ▷ Repeat until the loss decreases past a threshold τ
  $\mathrm{G} = \frac{\partial \mathbf{J}(\mathbf{W})}{\partial \mathbf{W}}$ ▷ Compute the partial derivative of each weight, how each weight affects the loss
  $\mathbf{W} \leftarrow \mathbf{W} - \eta G$ ▷ Update the weights by subtracting the partial derivatives times a learning rate
end while

In this way, a neural network can approximate an output function for each given set of inputs, the complexity of which depends on how many neurons and what kind of activation functions they have. These networks are commonly used for classification, but this paper focuses on one type of reinforcement learning, where the input to the neural network is a set of observations in an environment and the output is the action the network takes.

3 Procedure

3.1 Hypothesis/Engineering Goal

Implementing REPNET through a recursive tree data structure and series of actor-critic models will decrease the number of episodes required to reach a running reward threshold in comparison to a singular actor-critic model.

3.2 Location of Experiment

3903 Simon Ridge Court, Cedar Park TX 78613

3.3 Materials

1. Computer

3.4 Procedures

1. Construct the recursive tree data structure
2. Test the data structure to ensure proper operation
3. Construct the basic actor-critic model
4. Test the actor-critic model and determine the baseline number of episodes required to achieve the minimum running reward threshold across 100 trials
5. Integrate the basic actor-critic model and the data structure to create the linear reproduction threshold REPNET
6. Test the linear reproduction threshold REPNET model and determine the number of episodes required to achieve the minimum running reward threshold across 100 trials
7. Compare the mean and standard deviation of the linear reproduction threshold REPNET with the mean and standard deviation of the basic actor-critic model.
8. Construct various improvements to REPNET by editing the basic REPNET architecture.
9. Test the improvements and determine the mean number of episodes required to achieve the minimum running reward threshold across 100 trials for each.
10. Compare the mean and standard deviation of the REPNET variants with the mean and standard deviation of the basic REPNET and actor-critic model.
11. Compile findings and determine if REPNET and its variants yielded an improvement.

4 Analysis

4.1 CartPole Data Analysis

As a practical demonstration of the improvement REPNET offers upon the actor-critic architecture, it performed against a separate actor-critic architecture. Both were designed in TensorFlow (Abadi et al., 2016), which has been used to find the optimal policy for many games (Mnih et al., 2013). This architecture uses one model and one Adam optimizer (Kingma & Ba, 2014) with a learning rate of 0.01. The environment of choice was OpenAI's CartPole environment (Brockman et al., 2016), an important baseline used in the field of Reinforcement Learning. In this environment, the actor is required to keep a pole balanced on top of a cart which can move left and right through the actor's actions, while staying in a certain range of the origin. The pole is subject to gravity, and if the pole falls below a certain angle, then the actor loses. REPNET used a pruning hyperparameter of 100 subsequent weight updates and a constant reproduction threshold function of 75.

$$T(P) = 75 \tag{1}$$

For this model, if the improvement in running reward is greater than 75 between any two instances, a new branch will be created. Like the normal actor-critic architecture, REPNET used Adam optimizers with a learning rate η of 0.01. Both architectures used a γ (discount factor) of 0.99. The reward threshold τ was 195, such that the model is considered trained after the running reward is greater than or equal to 195. After 100 pairs of networks were run, the results were as below:

Table 1: Normal Actor-Critic Network versus REPNET

Network	**Mean Number of Episodes**	**Standard Deviation of Episodes**
Actor-Critic	255.49	163.67
REPNET	200.57	44.27

Compared to the normal actor-critic method, the results show a 21.5% improvement in the mean and a 72.95% improvement in the standard deviation. These results therefore indicate that REPNET provides a model that is quicker to train and more consistent when training. A representative training session for REPNET and the normal actor-critic architecture is provided below for clarity:

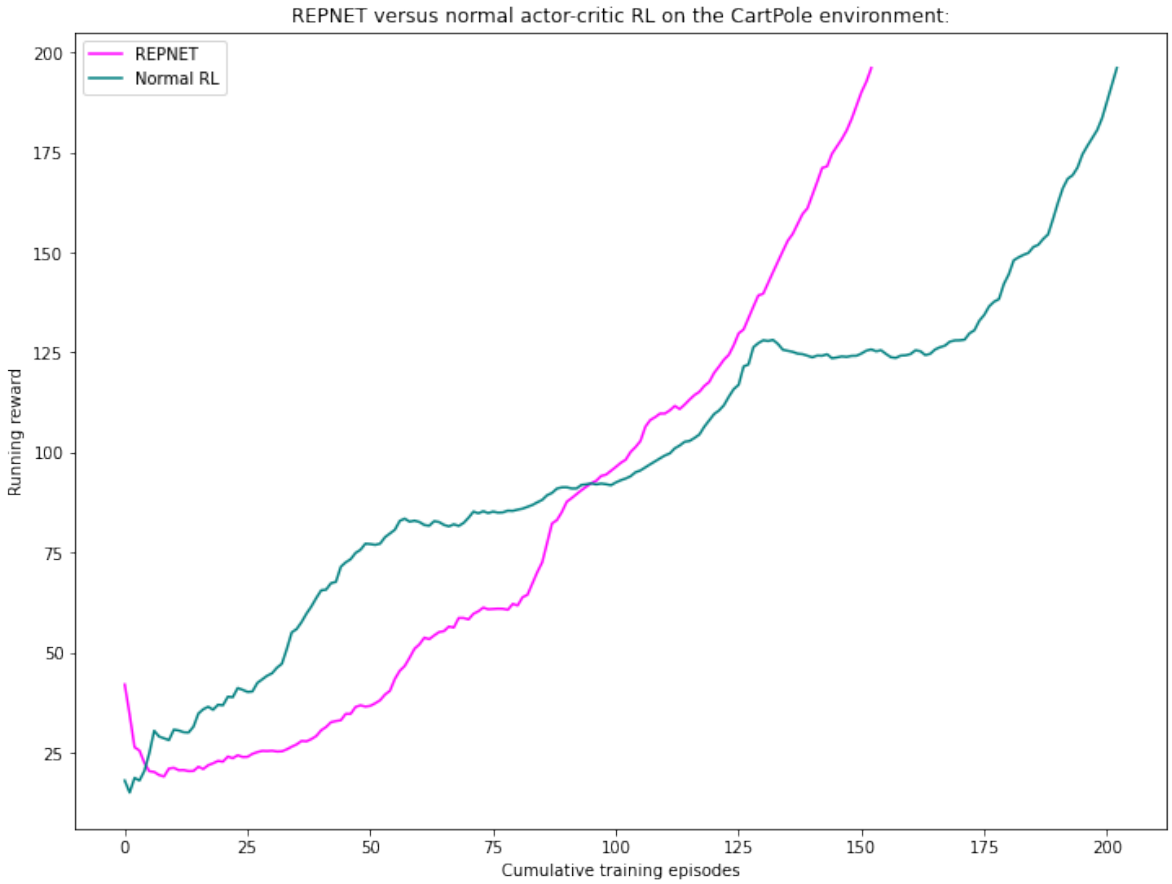


Figure 2: A representative training session of REPNET versus the normal actor-critic RL architecture. Note that the same seed (100) was used for both instances.

REPNET was trained on CartPole with a variety of hyperparameter combinations, from 5 to 100 in steps of 5 for both the pruning time and constant reproduction threshold function. The corresponding color map, with the brightest colors training in the fewest number of episodes, is seen below.

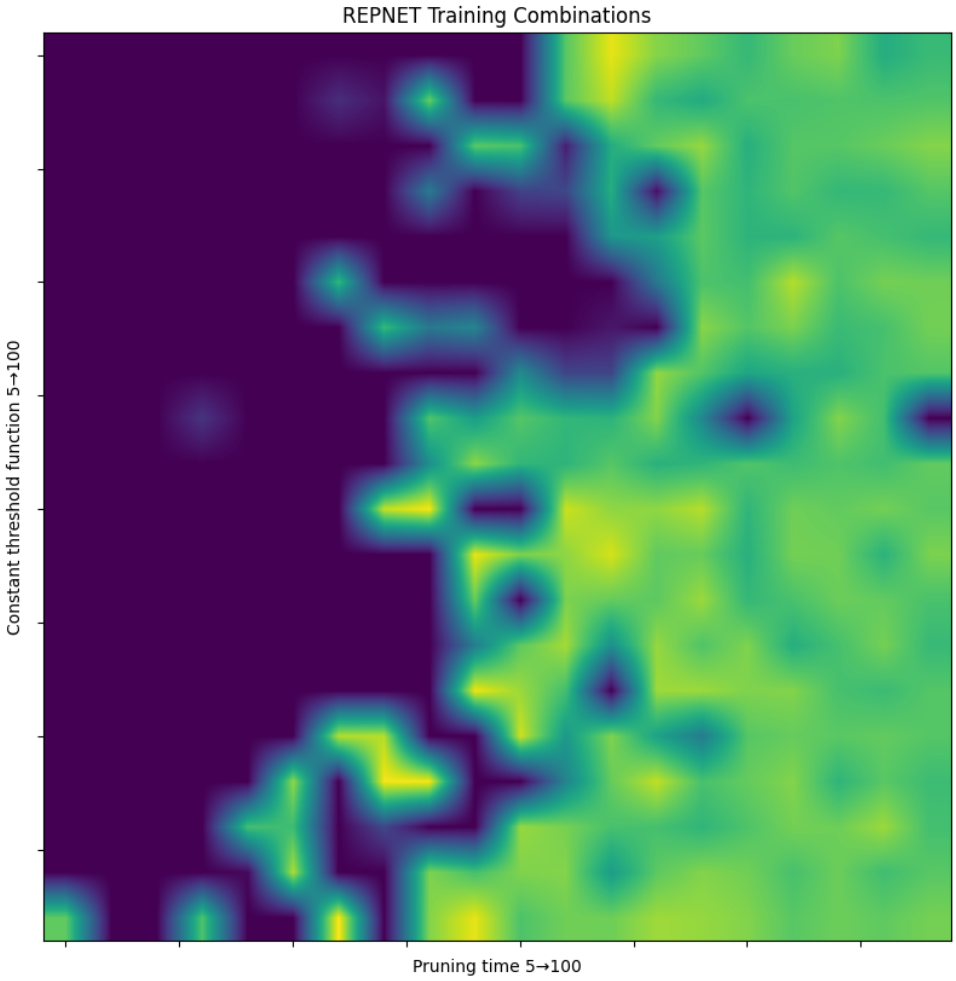


Figure 3: A color map of common training combinations for REPNET.

4.2 Nonlinear Reproduction Threshold Functions with CartPole

The previous reproduction threshold functions used to control the branch creation of REPNET have been horizontal constants. However, the change in running reward over time increases substantially as the running reward increases, until training halts. Further and further along into training, there's fewer reasons to create more branches due to the increasing change in running reward. This increase in the change in running reward, also creates more and more branches, creating dead weight in the network. This problem can be solved by implementing reproduction threshold functions like the one below. These functions take the current running reward and determine the minimum change in running reward necessary to create a new branch.

$$T(P) = \frac{P^2}{100} + 20 \tag{2}$$

However, the manual creation of a reproduction threshold function can lead to an increase in the loss of the network if improperly tuned. To avoid this, an auto-generating reproduction threshold function was constructed as seen below. τ_0 is the starting running reward, or the approximate performance of the model with randomized weights, and τ_1 is the running reward threshold to consider the environment solved.

$$T(P) = \frac{-\tau_1 \tau_0}{P - \tau_1} \tag{3}$$

Creating a function like this does two things:

1. Making the change in running reward approach an asymptote prevents the creation of new branches once the network has "almost figured it out". Due to the nature of policy recognition, this pattern always occurs, assuming the optimal policy is achievable. Normally, the high change in running reward during this time with a constant reproduction threshold causes many new branches to be created, which increases the overall number of episodes that must be trained. So, removing this dead weight can improve training time significantly. An example auto-generated reproduction threshold function graph is provided below. It's important to note that as the input to the reproduction threshold function is the current running reward, this approach works regardless of the length of training time in episodes.

2. Making more branches towards the start increases the variety of solutions in future episodes, as those initial children of the main branch have time to settle into different minima. This increases the likelihood of a quicker training time and increases the redundancy this architecture affords.

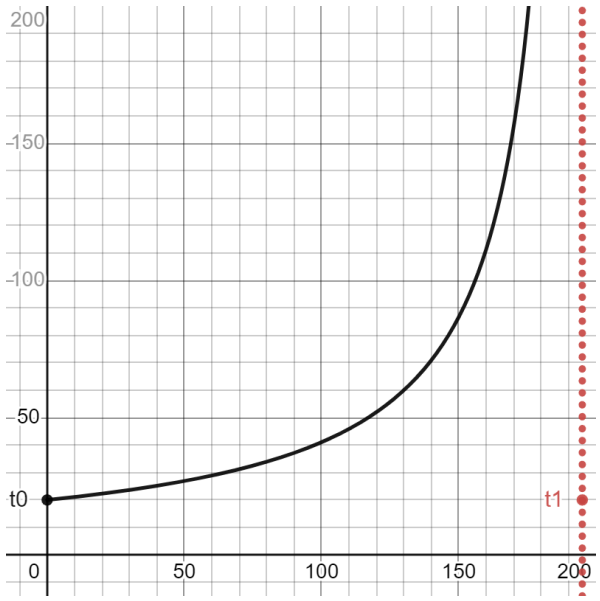


Figure 4: An auto-generated reproduction threshold function based on environment parameters specified.

This auto-generated nonlinear reproduction threshold function was trained 100 times on the Cart-Pole environment, with the statistical results below:

Table 2: The normal Actor-Critic network, REPNET, and Auto-Generating Nonlinear Reproduction Threshold Function, across 100 trials each

Network	**Mean Number of Episodes**	**Standard Deviation of Episodes**
Actor-Critic	255.49	163.67
REPNET	200.57	44.27
Nonlinear Threshold REPNET	180.62	44.81

The histogram of these trials can be seen below.

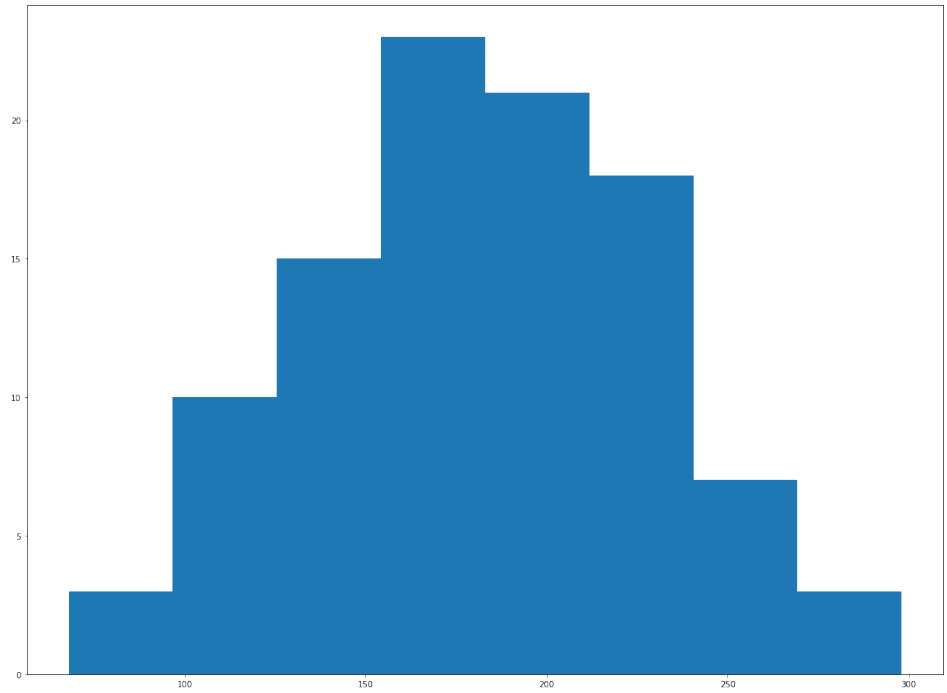


Figure 5: A histogram based on 100 training sessions of the nonlinear reproduction threshold function REPNET.

After testing the auto-generated reproduction threshold function across 100 training sessions, it decreased the training time by 10.24% when compared to the default REPNET (using the same pruning time hyperparameter) and 29.3% compared to the normal actor-critic approach.

4.3 Adaptive Pruning Time Adjustment (APTA) with CartPole

APTA is a powerful add-on tool that removes human error with regard to the pruning time parameter. When paired with the auto-generating nonlinear reproduction threshold function, this version of REPNET only one additional hyperparameter compared to the normal actor-critic models.

APTA uses a pre-tuned PID controller to attempt to stabilize the number of branches towards a target amount of branches per branch using the following control function.

$$u(t) = K_P e(t) + K_I \sum_{i=1}^{t} e(i) + K_D(e(t) - e(t-1)) \tag{4}$$

where,

$u(t)$	PID control variable, the change to the pruning time parameter
K_P	Proportional gain
$e(t)$	Measured error at episode t between the desired and actual number or growth rate of the number of branches
K_I	Integral gain
K_D	Derivative gain
t	A discrete variable representing episode number ($t \in \mathbf{N}_0$)

This hyperparameter is much easier to determine, as it is directly proportional to the number of branches in the tree. A figure showing the benefit APTA adds is provided below.

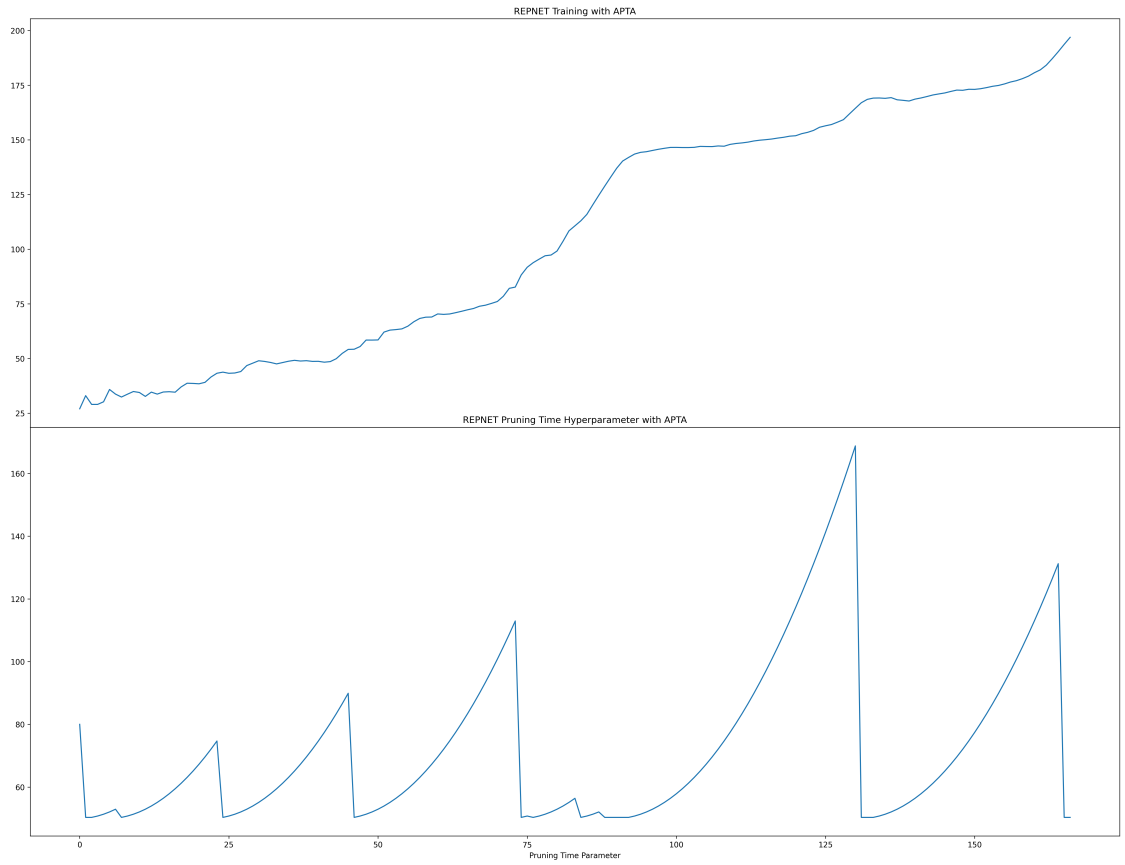


Figure 6: Training APTA on CartPole, viewing the running reward over episodes (top plot) and pruning time hyperparameter (bottom plot)

This network was initialized with a pruning time hyperparameter of 80, which is significantly suboptimal for the CartPole environment. A REPNET using a constant pruning time hyperparameter

would create far more branches than need be, slowing the training of the network. The APTA PID catches this sub-optimal ballooning, and drops the pruning time to approximately 55. In this state, the child branches over time are creating fewer and fewer branches, and so to prevent the network from pruning all of its branches, the pruning time hyperparameter is gradually increased back to 80. This process repeats several times to maintain the specified number of branches, and the network trains in approximately 170 episodes.

Below is the updated table with APTA's performance compared to the other network variants, based on 100 trials on the CartPole environment:

Table 3: The normal Actor-Critic network, REPNET, Auto-Generating Nonlinear Reproduction Threshold Function, and APTA networks, across 100 trials each

Network	**Mean Number of Episodes**	**Standard Deviation of Episodes**
Actor-Critic	255.49	163.67
REPNET	200.57	44.27
Nonlinear Threshold REPNET	180.62	44.81
APTA	195.81	45.78

A histogram is also provided for reference to the distribution of training times:

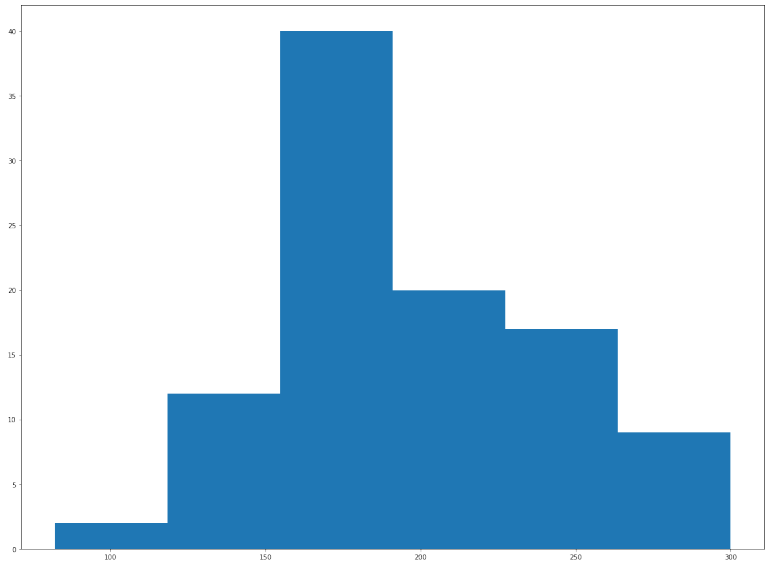


Figure 7: A histogram of training REPNET on CartPole across 100 trials.

Across the 100 trials, APTA had a 2.37% decrease in training times compared to the default REPNET, however it performed 8.41% worse than the manually controlled nonlinear reproduction threshold REPNET. This is likely due to the local minimas present in the optimization function, resulting in longer training times.

To determine whether or not APTA performed worse than the strict Auto-Generating Nonlinear Reproduction Threshold REPNET, a 2-Sample T-Test was performed:

i. 2-Sample T-Test
$\mu_1 =$ true mean number of training episodes for the nonlinear reproduction threshold REPNET without APTA
$\mu_2 =$ true mean number of training episodes for the nonlinear reproduction threshold REPNET with APTA
$\bar{x}_1 = 180.62$, $S_{x1} = 44.81$, $\bar{x}_2 = 195.81$, $S_{x2} = 45.78$, $\mathbf{df} = 197.91$, $n = 100$
$H_0 : \mu_1 = \mu_2$, $H_a : \mu_1 \leq \mu_2$

ii. Randomization is present due to the pseudo random number generator used to seed the network. As the size of both random samples was 100 and therefore ≥ 30, the sample sizes are sufficiently large enough to guarantee normality through the Central Limit Theorem. Due to the near infinite variation that training a network provides, it is safe to assume there are more than $100 \cdot 10 = 1000$ possible training instances in the population of training sessions for both variations of REPNET on the CartPole Environment.

iii.
$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_{x_1}^2}{n} + \frac{s_{x_2}^2}{n}}} = \frac{180.62 - 195.81}{\sqrt{\frac{44.81^2}{100} + \frac{45.78^2}{100}}} = -2.3712$$
$$\mathbf{df} = 197.91, \text{ p-value} = 0.00934, \alpha = 0.05$$

iv. The author rejects H_0 because the p-value $\leq \alpha$. There is enough statistically significant evidence to conclude the true mean number of training episodes for the nonlinear reproduction threshold REPNET without APTA is less than with APTA.

Although the mean training times decrease in comparison to the arbitrary pruning time in the example above, it still is a significant improvement compared to the default REPNET and, given the additional security given to training resources, it is a valuable tool and should not be overlooked when training REPNETs.

4.4 Statistical T-Interval Analysis

Raw statistical sample data can often misrepresent the true parameters of a population, in this case the true mean number of episodes required to train REPNET and its variants. In order to avoid statistical confusion, confidence intervals for the two main REPNET variants were established. A confidence level of 90% was used for each interval, indicating that 90% of intervals constructed from a sample of the same size n would contain the true mean population parameter.

4.4.1 Nonlinear Reproduction Threshold REPNET

To analyze the result of the auto-generated reproduction threshold function with REPNET, a t-interval test was performed using a confidence interval of 90%.

i. μ = the true mean of the number of episodes required to train the nonlinear reproduction threshold REPNET on the CartPole environment.
$\bar{x} = 180.62$, $t^* = 1.660$, $S_x = 44.81$, $n = 100$

ii. Randomization is present due to the pseudo random number generator used to seed the network. As the size of the random sample was 100 and therefore ≥ 30, the sample size is sufficiently large enough to guarantee normality through the Central Limit Theorem. Due to the near infinite variation that training a network provides, it is safe to assume there are more than $100 \cdot 10 = 1000$ possible training instances in the population of training sessions for the nonlinear reproduction threshold REPNET on the CartPole environment.

iii.
$$\begin{aligned} I &= \bar{x} \pm t^* \cdot \frac{S_x}{\sqrt{n}} \\ &= 180.62 \pm 1.660 \cdot \frac{44.81}{\sqrt{100}} \\ &= (173.18, 188.06) \end{aligned} \tag{5}$$

iv. The author is 90% confident that the mean number of episodes required to train the nonlinear reproduction threshold REPNET on the CartPole environment is between 173.18 and 188.06. All conditions for accuracy were met.

4.4.2 Nonlinear Reproduction Threshold REPNET with APTA

To analyze the result of the auto-generated reproduction threshold function with REPNET with APTA, a t-interval test was performed using a confidence interval of 90%.

i. μ = the true mean of the number of episodes required to train the nonlinear reproduction threshold REPNET with APTA on the CartPole environment.
$\bar{x} = 195.81$, $t^* = 1.660$, $S_x = 45.78$, $n = 100$

ii. Randomization is present due to the pseudo random number generator used to seed the network. As the size of the random sample was 100 and therefore ≥ 30, the sample size is sufficiently large enough to guarantee normality through the Central Limit Theorem. Due to the near infinite variation that training a network provides, it is safe to assume there are more than $100 \cdot 10 = 1000$ possible training instances in the population of training sessions for the nonlinear reproduction threshold REPNET with APTA on the CartPole environment.

iii.
$$\begin{aligned} I &= \bar{x} \pm t^* \cdot \frac{S_x}{\sqrt{n}} \\ &= 195.81 \pm 1.660 \cdot \frac{45.78}{\sqrt{100}} \\ &= (188.21, 203.41) \end{aligned} \tag{6}$$

iv. The author is 90% confident that the mean number of episodes required to train the nonlinear reproduction threshold REPNET with APTA on the CartPole environment is between 188.21 and 203.41. All conditions for accuracy were met.

5 Discussion

Through its reproductive recursive branch approach, REPNET decreased training times by 21.5% compared to its actor-critic counterpart, and reduced the standard deviation of training episodes by 72.95%. The default version of REPNET, as well as the nonlinear reproduction threshold function and APTA variant, were all trained one hundred times to reduce the margin of error when determining the true population parameters. Additionally, an approximate nonlinear reproduction threshold function can be employed to speed up training by an additional 10.24% past the default REPNET, and 29.3% in total compared to the actor-critic method. An Adaptive Pruning Time Adjustment protocol can additionally be put in place that decreases training times by 2.37% compared to the default version of REPNET. That being said, it increases training times by approximately 8.41% compared to the pure nonlinear reproduction threshold function REPNET variant with a manually set pruning time hyperparameter. However, the reliability and reduction of potential human errors are significant, and it should be considered as an important tool when training.

5.1 Underlying Data Structure

The data structure used in REPNET is a treelike recursive organizer. Each branch contains a model as well as pointers to the child branches that spawn from it, with a main branch starting off the process. Each branch is created with five instance variables - the running reward, a boolean state of whether the branch is killed or not, a counter of the number of episodes since reproduction, a reference to the model's weights, and a list containing the child branches. In an update method, a new running reward value and reference to weights is provided. The new running reward value is the running reward reached by the model during the episode.

If the change in running reward between the previous episode and the new episode is greater than the output of the reproduction threshold function based on the current running reward, then a new child branch is created. This occurs by adding a new branch object into a list in the parent branch, These child branches can be easily accessed through a whole tree recursive query that starts in the main branch. From the main branch, it adds the pointer to each child branch into a main list which can then be indexed through and updated accordingly. Each time a branch is updated, the branch's counter variable increases by one if no new child branch is created. If a new branch is created, the counter is reset to zero. A check also occurs to determine if the counter has passed the kill time hyperparameter, and if so, the branch is set to killed. The branch is not deleted if it has active child branches when killed, otherwise the child branches would be isolated from the main branch. If the branch is killed, it won't appear in the whole tree recursive query used to update each branch and therefore will not be updated further. After training, each set of weights from the whole tree recursive query can be analyzed and filtered. After this process, the best model can then be selected. This approach also allows for a large variety of approaches to the same problem, lowering the probability of REPNET settling into local minima.

Algorithm 2 REPNET Branch Algorithm

$X = [\,]$ ▷ List of child branches
$K =$False ▷ Branch pruned or not
$C_{reproduce} = 0$ ▷ Reproduction counter
$R_{running} = 0$ ▷ Current branch running reward

$T_{reward} \leftarrow t$ ▷ Threshold for the running reward
$K_{thresh} \leftarrow k$ ▷ Number of iterations without reproduction until branch is pruned
$T(P) \leftarrow t$ ▷ Reproduction threshold function for creating new branches
$W \sim N(0, \sigma^2)$ ▷ Initialize branch weights
▷ Reference to this model's weights ▷ Reproduction counter
while $R_{running} < T_{reward}$ **do** ▷ Check kill condition
  if $C_{reproduce} > K_{thresh}$ **then**
    $K =$True
  end if
  ▷ Update this branch
  if not K **then**
    $R_{running}, W \leftarrow$ self.runEpisode()
    $C_{reproduce}$++
  end if

  if $\Delta R_{running} \geq T(R_{running})$ **then** ▷ If change in $r_{running} \geq$ threshold function, create child
    X.addChild($R_{running}$,W_{new})
    reset this branch one episode ▷ Resets W and $R_{running}$ (and optimizer)
    $C_{reproduce} = 0$
  end if
  ▷ Update all child branches
  child.update() for child in X
end while
save W ▷ Communicate results to main program

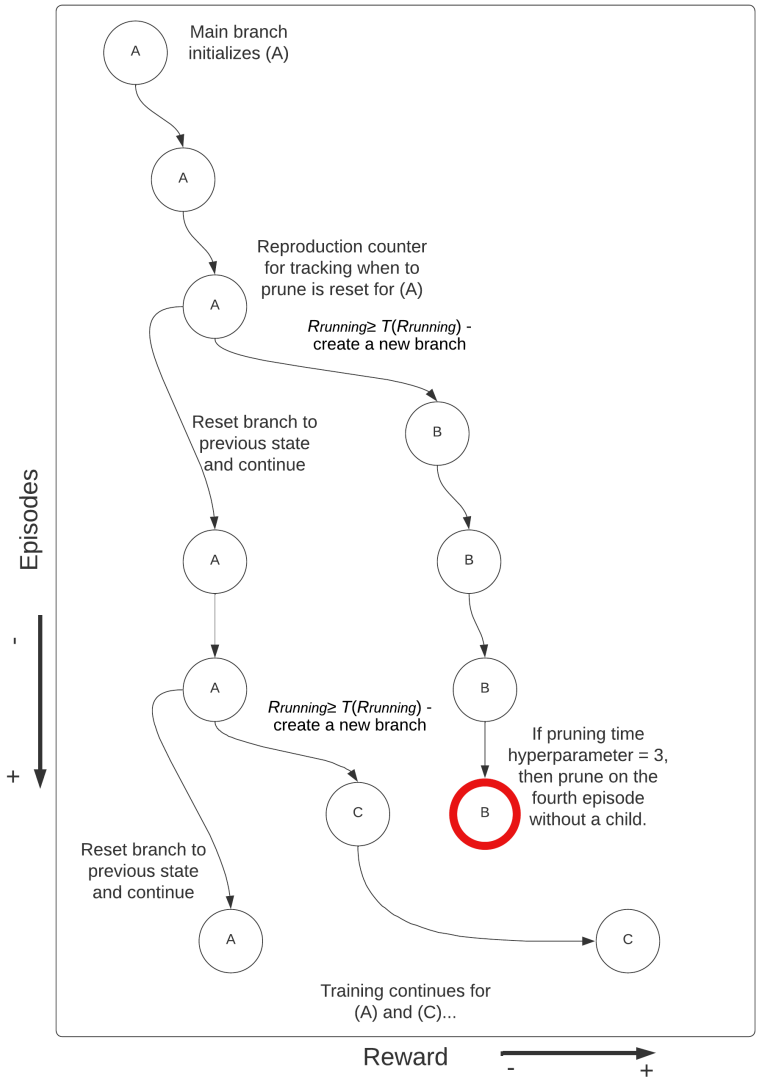


Figure 8: An example of the REPNET architecture. The X axis is the performance (running reward) and episodes increase vertically down the Y axis.

5.2 Weight Updates

REPNET initializes the neural network with weights in the main branch and an optimizer. To train the network on a reinforcement learning based task, the network accesses an action space and observation space. The actions taken in the action space must affect the observations seen in the observation space for reinforcement learning to work. Initially, the model in the main branch is fed the observation space as an input and makes a decision in the action space based on each branch's actor-critic model policy. Over the training episodes, the optimizer of choice increases the running reward gained.

Each episode, the whole tree recursive query list is looped through and updated. For each branch in the whole tree recursive query, the saved weight reference is assigned to the active model. The model is then optimized and subsequently trained on one episode, and the new weights and running reward are stored back to each respective branch. Training halts when a branch achieves the desired running reward threshold τ, and the whole tree recursive query can return the weights and running reward from each branch for further analysis.

5.3 Pruning

When a new branch is created, it also can be pruned due to a lack of producing child branches. After the counter passes a hyperparameter set as the kill value, then the branch is effectively dead (not updated anymore). This pruning process reduces the total number of branches currently being updated, lowering computing power better spent on higher performing branches. Without it, a problem of suitably large complexity or with a low reproduction threshold function could cause the number of branches to increase uncontrollably.

5.4 Reasoning

REPNET works due to the evolutionary pressure it places on branches. By pruning poor performing branches and having strict requirements for creating new branches, the best models can converge to the most optimal solution much quicker than other approaches. The running rewards of the best performing models then grow at a faster rate in comparison to the average performing model, causing REPNET to reach a the running reward threshold τ in significantly fewer episodes given properly tuned hyperparameters.

REPNET also significantly reduces the standard deviation of the number of episodes required to train the network due to the multiple branches that increase redundancy and prevent unusually long training sessions. A hard stop after a number of episodes of training used with the normal actor-critic architecture is a common approach to eliminate these outliers. However, the model has to be retrained if the hard stop is hit, causing the total number of episodes to increase significantly, minimally decreasing the mean training times of the normal actor-critic architecture in aggregate.

5.5 Python Implementation

5.5.1 Basic REPNET

To implement REPNET in Python, all a user is required to do is import the package, and declare a new `Repnet` object.

```python
import tensorflow as tf

from repnet.core import repnet, Tree
from repnet.basic_examples.cartpole import CartPoleEnv
import matplotlib.pyplot as plt

THRESH = 195

# Environment object for Cartpole (has an action and observation space)
cartpole = CartPoleEnv()

# Tree object that contains branches used in REPNET with a threshold function
tree_structure = Tree(80, lambda P: 45)

# Establish the optimizer with a learning rate hyperparameter of 0.01
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

# Create the REPNET object to train the model
data, weights = repnet(cartpole, tree=tree_structure, optimizer=optimizer,
reward_threshold=THRESH)

# Plot the resultant data
plt.plot(data)
plt.xlabel("Episodes")
plt.ylabel("Running Reward")
plt.title("Training with REPNET")
plt.show()
```

Figure 9: An example network based on the CartPole environment that plots the running reward over time for a basic constant reproduction threshold REPNET.

The `repnet()` function must have an optimizer, environment, and `Tree` object passed into it in order for the proper functionality to be present. The tree object requires the pruning time hyperparameter and the reproduction threshold function. The `repnet()` function can be run to train the network on the specified `env` environment object, storing the resultant running reward over time and finalized weights into two variables for further analysis. The environment object should be formatted according to the OpenAI Gym (Brockman et al., 2016) format.

5.5.2 Auto-Generating Nonlinear Reproduction Threshold Function & APTA

Both the auto-generating nonlinear reproduction threshold function and APTA are powerful tools that can significantly reduce training times and remove the subjective human aspect of setting hyperparameters.

```python
import tensorflow as tf

from repnet.core import repnet, Tree, Thresholds
from repnet.basic_examples.cartpole import CartPoleEnv
from repnet.core import APTAControl
import matplotlib.pyplot as plt

THRESH = 195

# Environment object for Cartpole (has an action and observation space)
cartpole = CartPoleEnv()

# Tree object that contains branches used in REPNET with a threshold function
# The threshold function is automatically generated, removing potential user error.
apta = APTAControl(50, 30)
tree_structure = Tree(80, Thresholds.autogen(20, 195), apta=apta)

# Establish the optimizer with a learning rate hyperparameter of 0.01
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

# Train the model
data, weights = repnet(cartpole, tree=tree_structure, optimizer=optimizer, reward_threshold=THRESH)

plt.plot(data)
plt.xlabel("Episodes")
plt.ylabel("Running Reward")
plt.title("Training with REPNET")
plt.show()
```

Figure 10: Usage of both the APTA object and auto-generating reproduction threshold function.

To use APTA, import `APTAControl` from `repnet.core`. Pass in the starting pruning time as the first argument and the optimal number of child branches per branch as the second argument. `p`, `i`, and `d` are optional keyword arguments that control the PID adjustment. These values should generally not be tweaked.

5.5.3 Into the API

While the entirety of the REPNET API constructed is too long to fit into a paper, below is an example of the Tree object used in the REPNET API:

```python
class Tree:
    """
    Tree object containing a main branch "trunk" and weights / other info
    """

    def __init__(self, max_kill_iters, threshold_func, max_steps_per_episode=1000, apta=None):
        self.max_kill_iters = max_kill_iters
        self.threshold_func = threshold_func

        # This is the branch all the rest of the branches come from.
        self.main = Branch(max_kill_iters, threshold_func, apta=apta)

        # Stores data to eventually return
        self.best_weights = None
        self.best_performance = 0

        self.max_steps_per_episode = max_steps_per_episode

        self.apta = apta

    def get_num_branch_ends(self) -> Callable[[], int]:
        """
        Get the number of branch ends in this tree
        :return: # of branch ends
        """
        return self.main.get_branch_num_ends

    def get_branch_ends(self) -> np.array:
        """
        Get the branch end references in the tree (global recursive tree query)
        :return: The list of references to branch ends
        """
        return self.main.get_branch_ends()

    def update_end(self, end, performance, weights) -> bool:
        """
        Update an individual branch in the tree provided its reference
        :param end: branch reference
        :param performance: NEW performance of the branch
        :param weights: NEW weights of the branch
        :return:
        """
        if performance > self.best_performance:
            self.best_weights = weights
            self.best_performance = performance
        return end.update(performance, weights)

    def reset(self) -> None:
        # Reset the entire tree back to default
        self.__init__(self.max_kill_iters, self.threshold_func,
max_steps_per_episode=self.max_steps_per_episode,
                      apta=self.apta)

    def __str__(self) -> str:
        return self.main.__str__()

```

Figure 11: The custom tree object used in the REPNET API. It stores the instance data pertaining to the structure of the network, as well as a function to return the references to each branch end in the tree, a function to update a specified branch based on a training session, and a function to reset the tree, useful in case of running multiple subsequent trials. It also contains the APTA object to be applied to each branch.

The `Branch` class is instantiated in the `Tree` class, and collectively contains the training data for REPNET. The `Tree` class then adds the recursive structure used based on the input hyperparameters, and the `Repnet` class performs the recursive updates. Additional functions have been put into place to perform statistical analysis, such as the one below:

```python
def compile_data(network_data) -> dict:
    """
    Get statistics data from the network
    :param network_data: episodes until running reward threshold reached
    :return: dictionary with important stats
    """
    compiled_length_data = [len(x) for x in network_data]
    poly_funcs = [np.poly1d(x) for x in compiled_length_data]
    derivatives = [func.deriv() for func in poly_funcs]
    return {
        "mean": np.mean(compiled_length_data),
        "standard deviation": np.std(compiled_length_data),
        "variance": np.var(compiled_length_data),
        "max": np.max(compiled_length_data),
        "min": np.min(compiled_length_data),
        "max rate of change": np.max(derivatives),
        "min rate of change": np.min(derivatives),
        "median": np.median(compiled_length_data),
        "first quartile": np.quantile(compiled_length_data, 0.25),
        "second quartile": np.quantile(compiled_length_data, 0.50),
        "third quartile": np.quantile(compiled_length_data, 0.75),
        "fourth quartile": np.quantile(compiled_length_data, 1),
    }
```

Figure 12: A helper function to perform statistical analysis.

For those getting started with REPNET, the `repnet.basic_examples` sub-package has been provided to give the developer some examples demonstrating REPNET's capability, such as the CartPole environment. Currently, the more in-depth documentation of the API is available at txkl.gitbook.io/repnet.

6 Conclusion

REPNET decreased the training episodes by a maximum of 29.3% compared to the standard actor-critic approach through the recursive branch structure present in the network architecture. REPNET can be combined with more complex reinforcement learning algorithms to yield lower training times and help develop models quicker. This is important, as larger reinforcement learning models present in robotics systems (Wang et al., 2022), self-driving cars, adaptive traffic signal controls (Abdulhai et al., 2003), and much more can take weeks or months to train, and so a time reduction of any amount is significant. Past this exploration, more research should be conducted into optimizing the APTA protocol and combining it with different approaches such as soft-actor critic algorithms (Christodoulou, 2019).

7 Acknowledgements

I would like to thank my parents for instilling in me a love of computer science, and my school for providing me with great resources such as the FTC Robotics team where I can grow my skills.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... others (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Abdulhai, B., Pringle, R., & Karakoulas, G. J. (2003). Reinforcement learning for true adaptive traffic signal control. *Journal of Transportation Engineering*, *129*(3).

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Christodoulou, P. (2019). Soft actor-critic for discrete action settings. *arXiv preprint arXiv:1910.07207*.

Greenwade, G. D. (1993). The Comprehensive Tex Archive Network (CTAN). *TUGBoat*, *14*(3), 342–351.

Gurney, K. (2018). *An introduction to neural networks*. CRC press.

Hochreiter, S. (n.d.). Ja1 4 rgen schmidhuber (1997).“long short-term memory”. *Neural Computation*, *9*(8).

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Konda, V., & Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems*, *12*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Ong, H. Y., Chavez, K., & Hong, A. (2015). Distributed deep q-learning. *arXiv preprint arXiv:1508.04186*.

Rosenblatt, F. (1958). *Two theorems of statistical separability in the perceptron*. United States Department of Commerce Washington, DC, USA.

Wang, Y., Vasan, G., & Mahmood, A. R. (2022). Real-time reinforcement learning for vision-based robotics utilizing local and remote computers. *arXiv preprint arXiv:2210.02317*.