ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

DESIGN AND EVALUATION OF NEURAL ARCHITECTURE, USING REINFORCEMENT LEARNING AND DISTRIBUTED COMPUTING.

Διπλωματική Εργασία

του

Κυριακίδη Γεώργιου

Θεσσαλονίκη,

DESIGN AND EVALUATION OF NEURAL ARCHITECTURE, USING REINFORCEMENT LEARNING AND DISTRIBUTED COMPUTING

Κυριακίδης Γεώργιος

Πτυχίο Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, 2015

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Μαργαρίτης Κωσταντίνος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την

Μαργαρίτης Κωσταντίνος	Σαμαράς Νικόλαος	Ρεφανίδης Ιωάννης
.....	.....	.....

Κυριακίδης Γεώργιος

.....

Περίληψη

Στηριζόμενο στα νευρωνικά δίκτυα, το Deep Learning έχει καταφέρει να λύσει διάφορα δύσκολα προβλήματα μηχανικής μάθησης. Αν και η βελτιστοποίηση των βαρών του δικτύου είναι ιδιαίτερα σημαντική για την απόδοσή του, η αρχιτεκτονική του δικτύου φαίνεται να έχει την ίδια αν όχι μεγαλύτερη σημασία. Ενώ η βελτιστοποίηση των βαρών είναι μία αριθμητική διαδικασία, η τελευταία παραμένει μία διαδικασία που εξαρτάται σε μεγάλο βαθμό από την ανθρώπινη εμπειρία και πειραματισμό. Σε αυτή τη μελέτη, προσπαθούμε να αναπαράγουμε την έρευνα που διεξάγεται στην αναζήτηση αρχιτεκτονικής νευρωνικών δικτύων, χρησιμοποιώντας την ενισχυτική μάθηση και δοκιμάζοντας διαφορετικές προσεγγίσεις. Εφαρμόζουμε ένα μικρό πείραμα, χρησιμοποιώντας Double Deep Q-learning Networks, πάνω στο γνωστό σύνολο δεδομένων MNIST χειρόγραφων ψηφίων. Στη συνέχεια, εφαρμόζουμε τον Synchronous Advantage Actor-Critic τόσο για διακριτούς όσο και για συνεχείς χώρους δράσεων. Τέλος, πειραματιζόμαστε με μερική εκπαίδευση των νευρωνικών δικτύων, προκειμένου να μειωθούν οι υπολογιστικοί πόροι που απαιτούνται για την αξιολόγηση των αρχιτεκτονικών. Στόχος της μελέτης είναι η επαλήθευση της βιωσιμότητας της χρήσης κατανεμημένου Reinforcement Learning στην αναζήτηση αρχιτεκτονικών νευρωνικών δικτύων, καθώς και η εφικτότητα της χρήσης μερικής εκπαίδευσης για την αξιολόγηση της αρχιτεκτονικής ενός νευρικού δικτύου. Διαπιστώνουμε ότι το κατανεμημένο Reinforcement Learning μπορεί πράγματι να χρησιμοποιηθεί για την εύρεση βέλτιστων αρχιτεκτονικών καθώς και τη χρήση μερικής εκπαίδευσης για την αξιολόγηση μιας αρχιτεκτονικής.

Λέξεις Κλειδιά:

Deep Learning, Ενισχυτική Μάθηση, Αναζήτηση Αρχιτεκτονικών Νευρωνικών Δικτύων, Μερική Εκπαίδευση, Ανεκπαίδευτα Δίκτυα, Advantage Actor-Critic

Abstract

Relying on neural networks, Deep Learning has solved many difficult machine learning problems. Although the optimization of the network's weights is of paramount importance for its performance, the network's architecture has been shown to contribute significantly as well. While the former is a relatively straightforward, numerical method, the latter remains a procedure heavily relying on human expertise and experimentation. In this study, we try to reproduce research conducted on Neural Architecture Search by utilizing Reinforcement Learning techniques as well as try different approaches. We implement a small-scale framework, using Double Deep Q-learning Networks while applying it to the well-known MNIST dataset of hand-written digits. We then apply Synchronous Advantage Actor-Critic for both discrete as well as continuous action spaces. Finally, we experiment with partial training of the neural networks, in order to reduce the computational resources required to evaluate the architectures. The aim of the study is to verify the viability of using distributed Reinforcement Learning in Neural Architecture Search, as well as the feasibility of using partial training in order to evaluate a neural network's architecture. We find that distributed Reinforcement Learning can indeed be used to find optimal architectures as well as the use of partial training in order to evaluate an architecture.

Keywords:

Deep Learning, Reinforcement Learning, Neural Architecture Search, Partial Training, Untrained Networks, Advantage Actor-Critic

ACKNOWLEDGEMENTS

I would like to thank my parents, for their constant encouragement and financial support that I have gotten over the years. Furthermore, I would like to thank my thesis supervisor, Professor Kostantinos Margaritis for his support, encouragement and open-minded approach to the thesis' topic, as well as the interesting conversations we frequently had.

Contents

Table of Figures

Table of Tables

1 Introduction

1.1 Deep neural network architectures

In recent years, with the explosion of Deep Learning, artificial neural networks, empowered by the massive parallelism offered by modern GPUs have enabled the execution of many complex tasks, previously regarded as difficult. The most popular are perception based applications, such as image and speech recognition [1], [2], [3], but other even unsupervised learning application are equally impressive, such as neural style transfer [4].

The only shortcoming of such feats is the uncertainty on why neural networks perform these tasks so well. In their quest for an explanation, researchers have discovered that the architecture (topology) of a network is as important as its training. Even untrained networks can perform exceptionally well in image representation [5]. It is thus imperative to develop methodologies that automatically generate good neural network architectures, as they contribute greatly to the overall system's performance.

1.2 Research Aims

In this research, we aim to investigate the feasibility of applying distributed Advantage Actor-Critic Reinforcement Learning methods, in order to search for neural network architectures. Furthermore, we investigate the possibility of using quick evaluation techniques, such as partial training, in order to quickly evaluate a topology's quality. The main research questions that we address are:

- Can we use distributed actor-critic methods in order to train an agent to design good neural network architectures in continuous spaces?
- Can we speed up the evaluation of a network architecture's quality, by using partial training (training only the output layer)?

1.3 Approach

The first part of this thesis presents the theoretical background that is necessary in order to understand the basic components utilized in the application of Deep Reinforcement Learning methods for Neural Architecture Search. The second part presents the experimental methodology and results of Deep Reinforcement Learning methods in discrete, as well as continuous spaces.

We first apply a relatively simple algorithm, Double Deep Q-Learning [6] in discrete spaces. We compare the performance of a simple Deep Reinforcement Learning algorithm in a small search space (fully connected and dropout layers on the MNIST dataset) [7] with random search, in order to gauge its ability to find better architectures than random search, as well as to evaluate our ability to correctly apply and code the methodology. We then apply a distributed Advantage Actor-Critic algorithm to the same problem space, in order to evaluate its performance on discrete spaces. Finally, we extend the methodology in continuous spaces, with fully convolutional layers on the CIFAR10 dataset [8].

1.4 Contribution

The main contributions of this thesis are:

- The application of distributed Advantage Actor-Critic algorithms in Neural Architecture Search.
- The demonstration of using partial training in order to evaluate a deep neural architecture.

1.5 Outline

Chapter 2 presents the mathematical and conceptual preliminaries of Reinforcement Learning. Furthermore, the three most important categories of Reinforcement Learning are presented, namely On-Policy methods, Off-Policy methods as well as Actor-Critic methods.

Chapter 3 explains the basics of neural networks and introduces Deep Neural Networks. The importance of architecture on the neural network's performance is discussed through relevant studies, as well as the contribution of neural networks in Deep Reinforcement Learning.

Chapter 4 presents High-Performance Computing and its contribution in the recent development of Deep Neural Networks. The basic categories of data and instruction parallelism are presented, as well as MPI, the standard for inter-process communication. Finally, we present Nvidia's CUDA, the main driving force of massive parallelism behind Deep Neural Networks.

Chapter 5 presents the first and most simple method used in our experiments, Double Deep Q-Learning

Chapter 6 presents the second Deep Reinforcement Learning method used in our experimentation, Advantage Actor-Critic.

Chapter 7 evaluates the performance of Double Deep Q-Learning in discrete action spaces, with neural networks consisting of simple layers, evaluated on the MNIST dataset.

Chapter 8 introduces the Advantage Actor-Critic algorithm to the problem of Chapter 7.

Chapter 9 extends the methodology to continuous action spaces, with fully convolutional neural networks and the CIFAR10 dataset.

Chapter 11 presents a summary and conclusions of the findings, as well as provide directions of future work.

2 Reinforcement Learning

2.1 Introduction

Reinforcement Learning is a sub-field of machine learning. Instead of mapping inputs to a target (Supervised Learning) or trying to find structure in the data (Unsupervised Learning) Reinforcement Learning tries to map inputs to actions. Lately, Reinforcement Learning has been used for natural language processing [9], to train robots how to navigate and manipulate objects [10], as well as play board games [11]. The following sub-chapters have been largely based on [12].

2.2 Markov Decision Process

A Markov Decision Process (MDP) is a way to model decision-making problems, when there is a stochastic element in the problem's outcome, regardless of the deterministic behavior of the decision maker. MDPs can be solved using a range of optimization methods, including Reinforcement Learning (RL). A MPD can be defined as a tuple of 5 elements, (S, A, P, R, γ):

- S is a set of states.
- A is a set of available actions.
- $P_a(s, s')$ denotes the transition probability; Given state s, action a will result in state s' with probability $P_a(s, s')$.
- $R: S \times S \times A \to \mathbb{R}$ is a function mapping the action, current and next state to the set of real numbers. (s, s'). It is the immediate reward, resulting from action a, given state s and transitioning to state s'.

- $\gamma \in [0,1]$ is a discount factor, denoting the difference in importance between immediate and future rewards.

Generally, in a Reinforcement Learning setting, only the available states, actions and discount factor are known. Thus, in order to decide optimally, the decision maker (or autonomous agent) must first explore the reward (and transition probability) space and then try to formulate an optimal solution (policy). This, in optimization theory is known as the exploration-exploitation tradeoff [13].

An agent acting in a discrete-time Markov Decision Process interacts with the environment at discrete time steps (Figure 1). At each time step t, the agent has some information regarding the environment's current state S_t, selects an action A_t from the set of possible actions A and receives a reward R_{t+1} as well as information about the environment's next state S_{t+1} .

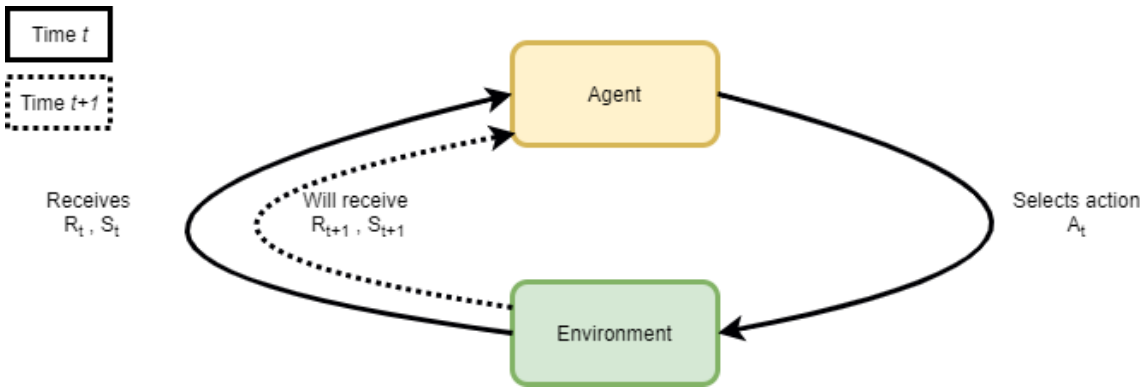

Figure 1 Agent interacts with environment in an MDP

In Reinforcement Learning, the agent's goal at any given time t is to maximize the cumulative rewards, from the current time t until a terminal state has been reached (or another termination condition has been met) at time T. Starting from state S_0 at time t_0, the sequence of interactions between the agent and the environment until reaching a terminal state is called an episode.

2.3 Returns and discounted rewards

Assuming that an agent receives a (possibly zero) reward for every action taken, the sum of expected rewards from time t until the episode's end is called return and defined as:

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T = \sum_{k=0}^{T} R_{t+1+k} \tag{2-1}$$

It is easy to see that $\lim_{T \to +\infty} G_t = \pm\infty$, depending on the distribution of R and the policy π. For this reason, we apply a discount rate $0 \leq \gamma \leq 1$ to future rewards, such that equation (2-1) becomes:

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T = \sum_{k=0}^{T} \gamma^k R_{t+1+k} \tag{2-2}$$

Thus, the agent's objective is to maximize the expected discounted return, which depending on the value of γ could imply a completely different acting policy.

2.4 Policy

The policy π is a mapping from the state space to the action space. In other words, the policy dictates how an agent should act in all possible states. It could be deterministic, thus mapping each state to a single action, or stochastic, mapping each state to a distribution of possible actions. An optimal policy π^* maximizes the expected discounted return.

2.5 Value Functions

The value of a state s under a policy π is the expected return, starting from state s and following π. It can be formally written as:

$$V_\pi(s) = E(G_t | s, \pi), \forall\, t \tag{2-3}$$

V_π is called the state-value function for policy π. Similarly, we can define the value of deciding to take the action a in state s under a policy π:

$$Q_\pi(s, a) = E(G_t | s, \pi, a), \forall\, t \tag{2-4}$$

q_π is called the action-value function for policy π.

In order to maximize the state-value function, we need to find an optimal policy. In other words, we need to compute

$$V_{\pi^*}(s) = \max_{\pi} V_{\pi}(s) = \max_{\alpha} Q_{\pi^*}(s,a)$$
$$= \max_{\alpha} E\left(\sum_{k=0}^{T} \gamma^{\kappa} R_{t+1+k} \,|s,\pi,a\right)$$
$$= \max_{\alpha}\left(\sum_{s\prime} p(s'|s,a)\,(R(s,a,s') + \gamma V_{\pi^*}(s'))\right) \quad (2\text{-}5)$$

Equation (2-5) is the Bellman's equation for V_{π}. It expresses the need to choose an action that maximizes the sum of expected discounted reward at any given state, in order to maximize the state-value function.

2.6 Dynamic Programming

Dynamic Programming (DP) is a family of algorithms designed to derive optimal policies, assuming the environment can be perfectly modeled as a MDP. Although DP is computationally expensive and needs a perfect model of the environment (which in many cases is impossible to obtain), it is a useful pedagogic tool, in order to introduce and understand RL concepts. Furthermore, many classical problems can be solved using DP. For example, Dijkstra's shortest path algorithm can be seen as a DP problem [14].

The main idea of DP is to break the problem into smaller sub-problems and iteratively solve these sub-problems. For example, suppose that an agent has to navigate through 8 rooms. The agent's goal is to reach room 8. Thus, room 8 yields a reward of 1 and is a terminal state, while all the other rooms yield a reward of 0 (Figure 2 The 8 rooms and their respective rewards.).

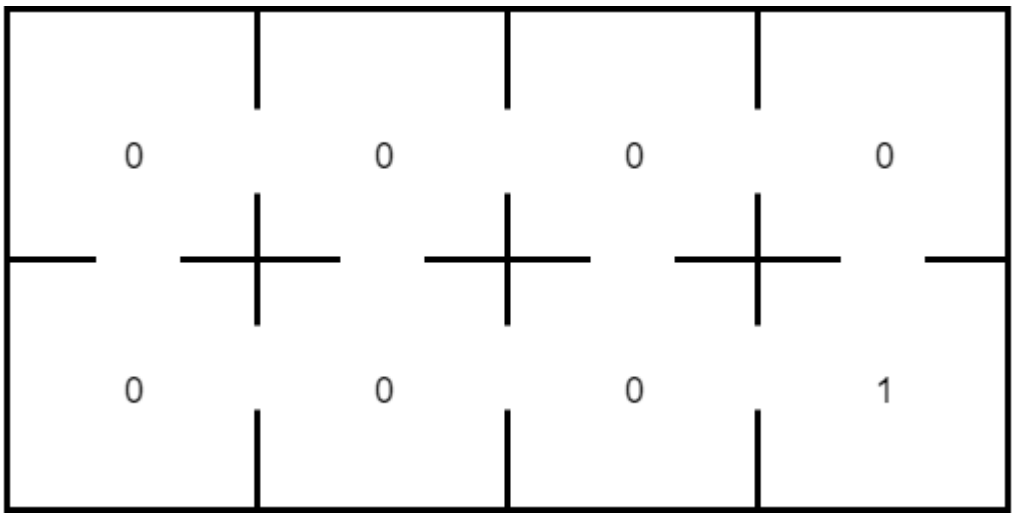

Figure 2 The 8 rooms and their respective rewards.

Solving this problem with DP would require that we compute the state-value function $V_\pi(s)$ for a given policy π and for every state s. By breaking the problem into smaller sub-problems (in this case, each room is a sub-problem), we can iteratively compute the state-value function. Using a uniform distribution to select from 4 possible actions (up, down, left, right), i.e. a random policy π_r, we compute $V_{\pi_r}(s)\ \forall\ s$, until it converges, for a given tolerance. In every iteration, for every state s, we compute

$$V'(s) = \sum_{a \in A} \pi(a|s) \sum_{s'} p(s'|s,a)[R_a(s,s') + \gamma V(s')] \tag{2-6}$$

Given a tolerance τ, we stop when the maximum difference is less than τ. In our example, for $\tau = 0.01$ the computed state-value function at each iteration i is shown below

	0.00	0.00	0.00	0.50
i=1	0.00	0.00	0.33	1.00
	0.00	0.00	0.27	0.50
i=2	0.00	0.11	0.49	1.00
	0.00	0.13	0.33	0.63
i=3	0.05	0.16	0.58	1.00
	0.09	0.16	0.44	0.66
i=4	0.08	0.25	0.65	1.00

Figure 3 Iterative policy evaluation

Using this information, we can improve our policy: for every state, we deterministically choose the action with the best action-value function. Thus, our new policy would be

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, Q_\pi(s,a) \tag{2-7}$$

Dynamic programming, as mentioned earlier, requires a perfect model of the environment, which means that probabilities must either be known a priori or must be easy to be estimated online. As this is not always the case in real-world problems, model-free methods have been developed to solve environments where a perfect model is difficult or impossible to be constructed.

2.7 On-policy methods

On-policy methods use the current policy to act and generate experiences from these acts. By learning from experiences, they are able to improve their policy and use the new policy in order to generate new experiences.

One such method is SARSA (State-Action-Reward-State-Action) [15]. SARSA learns the action-value function, or Q-value, by remembering (as the name suggests) the current State and Action, as well as the next Reward, State and Action. For each state/action pair, at each time step, the new Q-value is calculated as

$$Q(s_t, a_t) = Q(s_t, a_t) + a[R_a(s_t, s_{t+1}) + +\gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (2\text{-}8)$$

where a is the learning rate parameter.

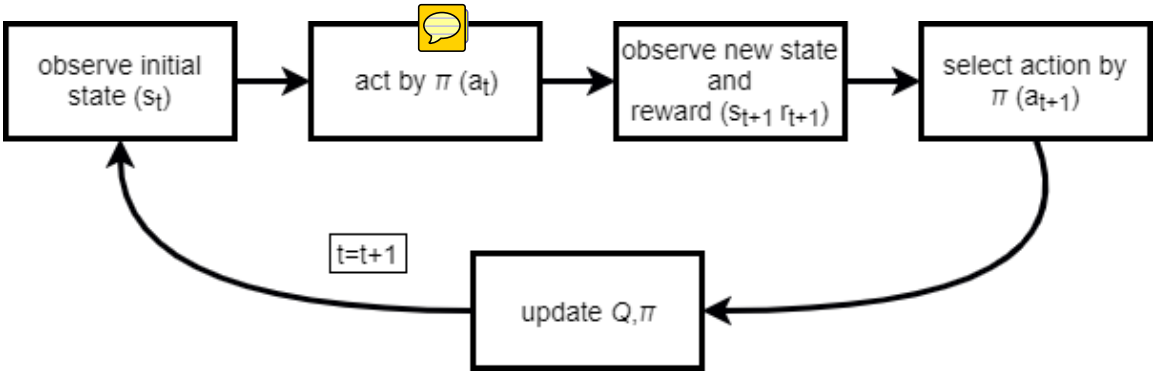

Figure 4 SARSA

2.8 Off-policy methods

Off-policy methods use any viable policy (exploration method) to generate experiences and then use these experiences to learn and update their current policy.

A well-known off-policy method is Q-Learning. Unlike SARSA, Q-Learning does not remember the next action taken. Instead, from each generated experience, only the current State and Action as well as the next State and Reward are memorized. At each step, the Q-value is calculated as

$$Q(s_t, a_t) = Q(s_t, a_t) + a[R_a(s_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (2\text{-}9)$$

It becomes clear why Q-Learning is an off-policy method: instead of acting based on the current policy π and remembering the action, the agent acts greedily, based on the calculated Q-Value

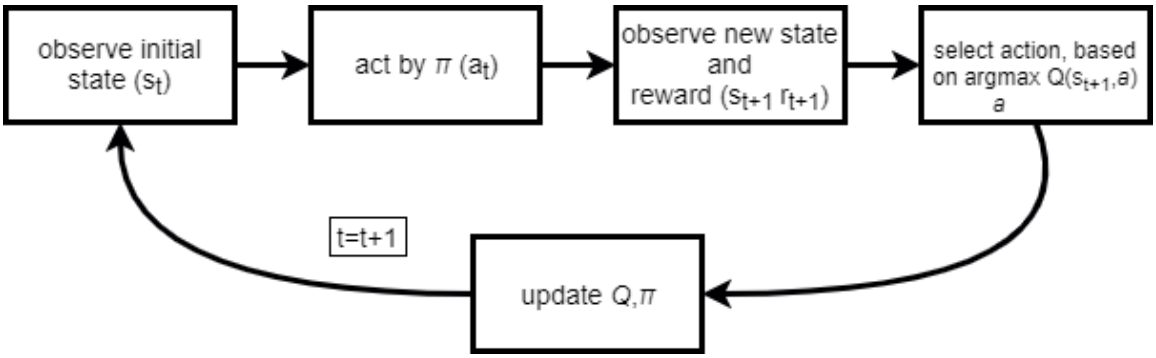

Figure 5 Q-Learning

2.9 Actor-Critic Methods

In Actor-Critic methods, the agent employs two different parameterized functions: A function that estimates the policy (Actor) and a function that estimates the value function (Critic).

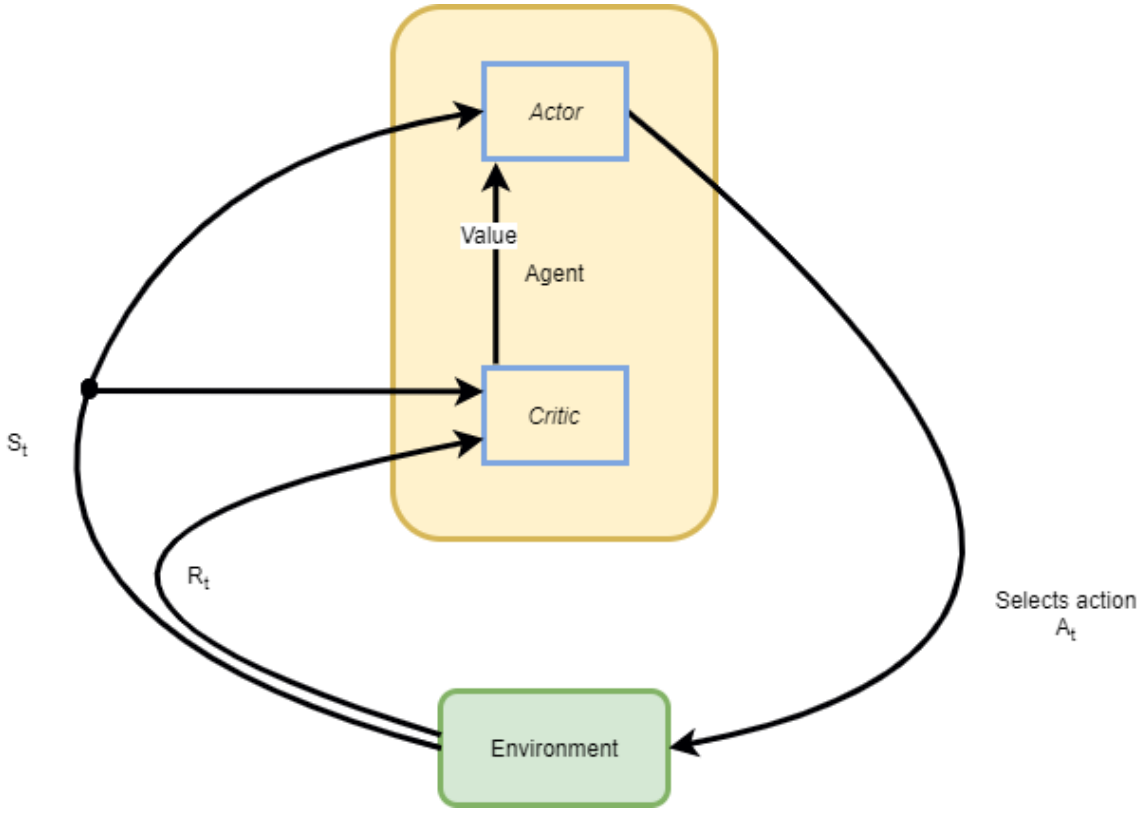

Figure 6 Actor-Critic Agent

One of the simplest Actor-Critic methods is One-step Actor-Critic. Assuming parameters θ, w of the Actor $A(s,w)$ and Critic $C(\theta,s)$ functions respectively, at each step, being at state s_t, the Actor function selects action a_t and the Agent observes state s_{t+1} and reward $R_a(s_t, s_{t+1})$. The Agent updates the parameters as follows:

$$\delta = R_a(s_t, s_{t+1}) + \gamma A(s_{t+1}, w) - A(s_t, w)$$

$$w = w + a^w I \delta \nabla_w A(s_t, w)$$
$$\theta = \theta + a^\theta I \delta \nabla_\theta lnC(s_t, w)$$
$$I = \gamma I$$

Where $\nabla_k f(x, k)$ is the partial derivative of f with respect to its parameters, k and a^w, a^θ are the learning parameters for the Actor and Critic functions. A more advanced Actor-Critic method is presented in Chapter **Error! Reference source not found.**.

3 Deep Neural Networks

3.1 Introduction

Artificial Neural Networks are computational systems inspired by biological brains. The main analogy is that they both use small, simple functional units (neurons) in order to execute complex, non-linear computations. Lately, (Deep) Neural Networks have been the driving force behind significant achievements in Machine Learning, such as Image Recognition [16], Style Transfer from one image's style to another's content [4] and Speech Recognition [3].

In this chapter, we discuss the most common neuron (layer) types, the use of Deep Learning (Deep Neural Networks) in Reinforcement Learning, the effects of a network's architecture to its performance as well as the various architecture-building methods that have been investigated in the literature.

3.2 Neurons

The most fundamental building block of Artificial Neural Networks is the Artificial Neuron. Each neuron is a combination of two functions, $\varphi \circ f(x)$, where x are the neuron's inputs, stored in a vector of size m, $f(x) = \sum_{j=0}^{m-1} w_j x_j$ is a parameterized summation function and $\varphi(x)$ is the activation function. Common activation functions are the linear function, the sigmoid, the hyperbolic tangent (tanh) the Rectifier Linear Unit [17], the softmax and softplus (Figure 7).

- Linear $f(x) = x$
- Sigmoid $f(x) = \frac{1}{1 + e^{-x}}$
- Hyperbolic Tangent $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Rectifier Linear Unit $f(x) = \max(0, x)$

- Softmax $$f(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, x \in \mathbb{R}^k$$
- Softplus $$f(x) = \log(e^x + 1)$$

Figure 7 Common activation functions.

In order to train the neuron, a cost function is defined and the neuron is optimized, in order to minimize the cost function. Usually, the cost is a function of the current output and the desired (or target) output. The neuron is trained through a method called backpropagation, which updates the neuron's weights, via stochastic gradient descent. Assuming a cost function C and a learning rate a, the weights at each step t are updated using the following equation:

$$w_i(t+1) = w_i(t) + a\frac{\partial C}{\partial w_i} + \xi(t) \quad (3\text{-}1)$$

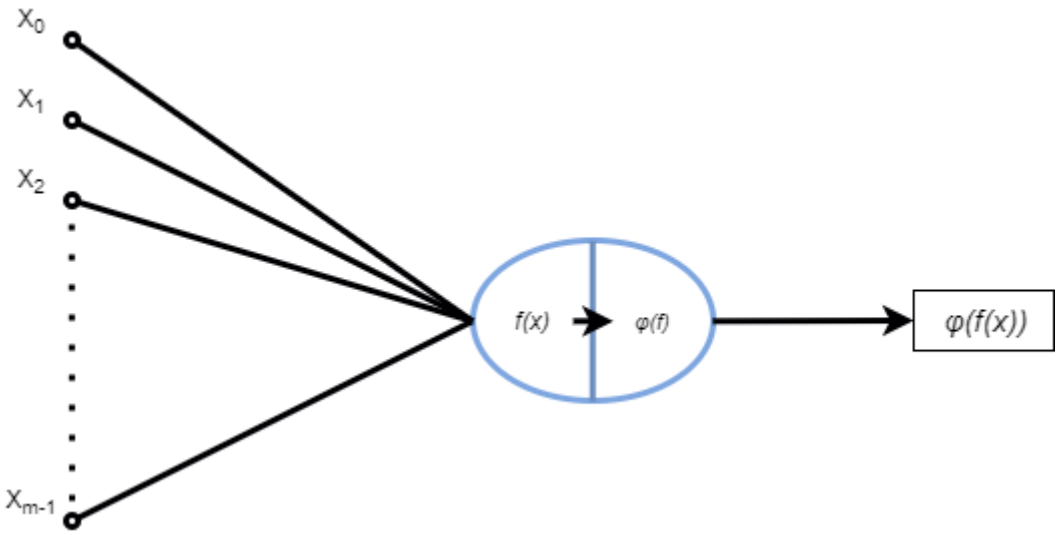

Figure 8 Artificial Neuron

3.3 Networks

An Artificial Neural Network consists of many neurons, organized in layers. Each layer's neuron is connected to all previous and next layers' neurons, but to none of the neurons that belong to the same layer. There are three distinct layer positions: The input layer, where the network is fed with the input values. The output layer, that outputs the network's computed values. Finally, all the intermediate layers are called "hidden" layers and are responsible for learning how to correctly transform the inputs to outputs.

In order to train the network, the outputs are calculated in a feed-forward manner: Starting from the input layer, each neuron computes its output and passes it on to the next

layer, until the output layer's neurons produce a value. Then, the weights' changes are calculated, in a backward-pass manner, starting from the output layer (where the targets are known, thus it is possible to calculate the cost) and propagating the errors to the hidden layers. Finally, the weights are updated, using the calculated changes.

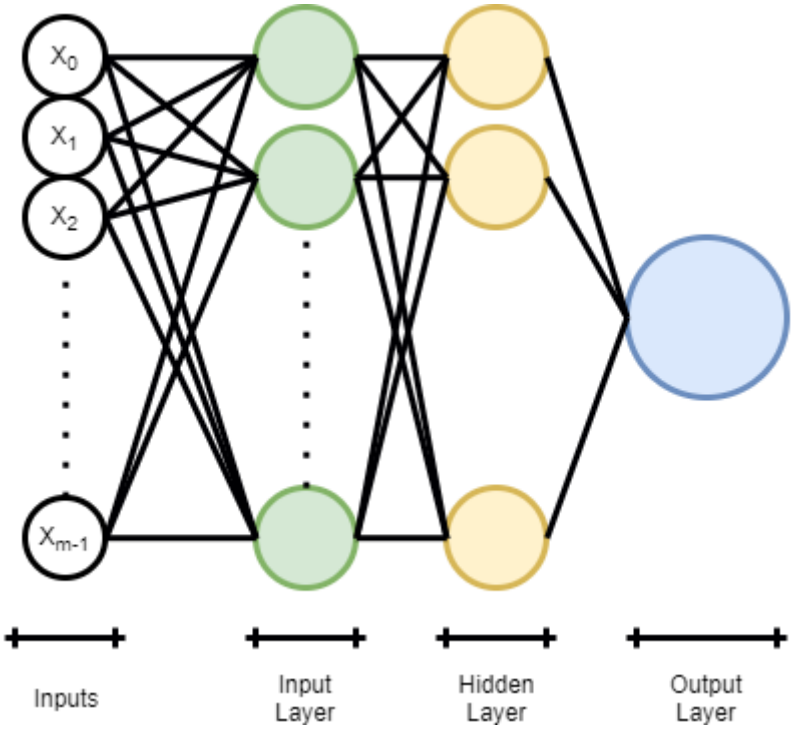

Figure 9 Neural network with one output neuron (regression).

3.4 Layer Types

Through the years, many different layer (neuron) types were created, in order to enhance the capabilities of Artificial Neural Networks.

Dropout layers were created in order to reduce overfitting in neural networks. Overfitting is a serious problem, which causes the neural network to optimize its weights so much, that while it performs very well on the training dataset, its performance greatly degrades on out of sample data (loss of generalization). In order to counter this problem, during training, dropout neurons with a probability of p (hyper-parameter, not trainable) deactivate, outputting zero and with a probability of $1 - p$ activate, outputting their input intact [18]. This technique essentially creates an ensemble of smaller networks, by deactivating parts of a big network. It could be seen as a form of ensemble learning [19].

Convolutional layers [20] extract spatial features from the inputs (usually images). In their simplest form, they are $n \times n$ matrices, applied to a two-dimensional (grayscale) image. For example, a 3×3 identity matrix can be used in order to determine if an image

block of 9 pixels contains a diagonal line. Starting from the top left of the input image, the filter's matrix is multiplied element-wise with the inputs and the elements of the resulting matrix are summed. Then, the filter is moved to the right, by m pixels (strides), or if the filter has reached the end of the input, it is moved down m pixels and to the left of the input, such that it is at the beginning of the next block-row. The output is a three-dimensional matrix, with the last dimension being dependent solely on the layer's number of filters (neurons). Thus, a special case of convolution layer is one with dimensions 1×1 and a stride of 1. Assuming a $i \times j \times k$ input, the output space is $i \times j \times K$, where K is the layer's number of neurons. Thus, if $K < k$, the layer performs a dimensionality reduction, to its inputs.

Another popular layer type is pooling. A pooling layer with $m \times n$ dimensions, reduces its inputs dimensions by m, n respectively. It reduces the dimensions, by aggregating $m \times n$ items, outputting the maximum (max pooling) [2] or the average (average pooling) item of the group.

3.5 Use in Reinforcement Learning

In Deep Reinforcement Learning, Artificial Neural Networks are used as function approximators. In very large state spaces, it is almost impossible to store the state/action value for all states. Thus, Artificial Neural Networks are used in order to learn the state/action value. For example, in Deep Q-Networks [21], a neural network is used to output the expected Q-value of each possible action, with the current state as input (which is computationally more efficient than training the network to output the expected Q-value of a state/action pair). The states are color images, represented as $84 \times 84 \times 4$ tensors (matrices). The neural network consists of three convolutional layers, followed by two fully connected layers. The activation functions used for all layers, except the output layer was a Rectified Linear Unit (Chapter 3.4).

Using this setup, Google DeepMind managed to achieve human-like or better performance in a number of classic arcade games. Other approaches utilize two different networks [6], in order to estimate the Q-value with one network and choose the optimal action with the other. This technique eliminates the bias that arises from trying to maximize the expected value, while trying to estimate the maximum expected action value from the maximum action value [22].

3.6 Effect of Architecture and Random Weights

Recent research has questioned how Deep Neural Networks achieve their performance. One study tried to replicate style transfer, texture synthesis and representation inversion, using untrained deep architectures, initialized with random weights [5]. Figure 10 is extracted from the paper and on the first row shows the original style image, on the second row the style transfer using an untrained deep architecture, while on the third row is the style transfer achieved with a fully trained network. While not exactly the same, the second row achieves an impressive performance, far from random.

Figure 10 Style transfer with untrained architectures [5]

Another study, funded by the DARPA Deep Learning program, compared random-weighted architectures with various fill trained architectures on classification tasks [23]. The results are quite surprising, as on the NORB dataset the random weight architecture achieved an accuracy of 94%, while the trained version achieved a 96.1% accuracy.

Encouraged by these findings, we can speculate that the performance of deep neural networks can be attributed to the architecture itself, as well as the optimization methods used to train them.

3.7 Neural Architecture Search

The process of designing and testing the architecture of a neural network has traditionally been a manual task, requiring knowledge about the dataset's nature as well as the behavior of the network. As such, it is a task performed by a few knowledgeable individuals.

In the past, there have been efforts to automate the task of discovering good architectures. One notable effort used genetic algorithms in order to gradually evolve various architectures [24]. Genetic algorithms evolve a solution to an optimization problem iteratively. A population of candidate solutions is evaluated and at each iteration(generation) new solutions are generated by a certain rule (for example randomly combine elements of two good solutions) and low-quality solutions are discarded. Each solution's parameters are stored in a vector (called gene) and each iteration is called a generation. In [24], the gene explicitly represents every connection and neuron, instead of aggregating them into common layer types. A more recent approach, uses hierarchical representations, in order to more efficiently represent the network's architecture [25]. Still, genetic algorithms, while having the ability to produce diverse and powerful solutions, are computationally expensive, as each iteration needs to validate the whole population, in order to decide how to reproduce (generate the next population's solutions). This is a problem that most population-based meta-heuristics face.

Another approach is to use Reinforcement Learning. In fact, using deep Reinforcement Learning, predictive ability of neural networks is leveraged, in order to design other neural networks. This is the approach that a Google team has followed, in order to find high-performing architectures for several datasets [26]. Using REINFORCE [27], they trained an agent to create neural architectures. The results were quite interesting, as many of the architectures learned, closely resembled the state of the art hand-crafted architectures for the given datasets. Furthermore, they were able to outperform state of the art architectures for certain datasets. Another independent study from MIT Media Laboratory also explored the viability of using Reinforcement Learning to train an agent at the architecture design task [28]. This particular study used Q-Learning, demonstrating that using Reinforcement Learning is a viable solution to the architecture search problem.

4 High Performance Computing

4.1 Introduction

Generally, High Performance Computing refers to computing methods and systems, used for tasks with higher computational needs than those performed by the vast majority of computer users. Example uses include weather forecasting, aerodynamic research and proteomics [29] [30].

Most supercomputers are actually a group of many regular computers, connected through high-speed networks. From the world's top 500 supercomputers, as of November 2017, 87.4% are cluster computers, while 12.6% are massively parallel processing systems [31].

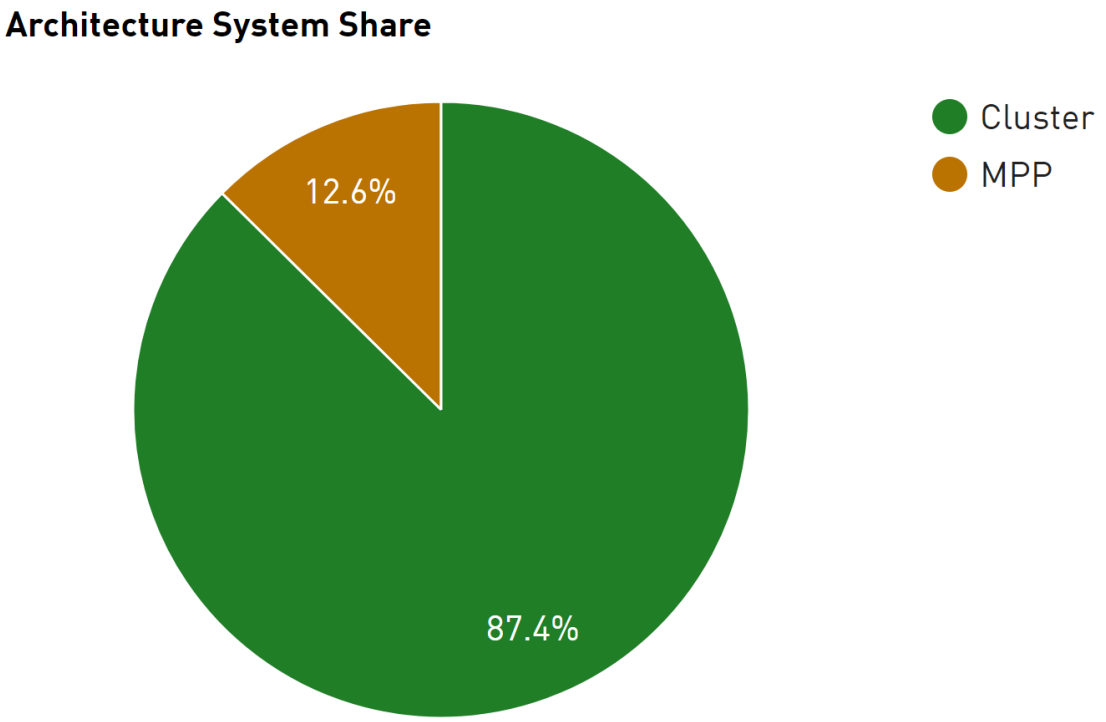

Figure 11 Architectures of the top 500 supercomputers

4.2 Parallel Architectures

One of the most well-known classifications of parallel architecture schemes is Flynn's Taxonomy, which was proposed by Michael J. Flynn in 1966 [32]. Flynn categorizes computer systems, based on their parallelization ability, both on the instructions, as well as the data.

There are four different categories: Single Instruction, Single Data (SISD), Single Instruction, Multiple Data (SIMD), Multiple Instruction, Single Data (MISD) and Multiple Instruction, Multiple Data (MIMD) (Table 1).

4.2.1 SISD

SISD architectures exhibit no parallelism at all. The most common example is the uniprocessor, a computer where all the tasks share the same processor.

4.2.2 SIMD

SIMD architectures employ a single instruction stream, but multiple data streams. The most common example are Graphics Processing Units, where one instruction (for example to increase brightness) is applied to all the image's pixels.

4.2.3 MISD

MISD is the least common of the four architectures. An MISD system exhibits instruction parallelism but not data parallelism. One example is the systolic array, where the results of one function are forwarded to the next.

4.2.4 MIMD

MIMD systems exhibit both data as well as instruction parallelism. The most common type of architecture exhibiting MIMD properties is the multiprocessor, where many different tasks can be simultaneously performed. For example, one core may be decoding a video stream, while another compiles a program.

Table 1 Flynn's Taxonomy

	Single Instruction	*Multiple Instruction*
Single Data	**SISD**	**SIMD**
Multiple Data	**MISD**	**MIMD**

4.3 Message Passing Interface

Another distinction of parallel systems can be made based on their memory. Shared Memory systems execute processes that have access to the same memory addresses, Distributed Memory systems run processes with separate address spaces. Message Passing Interface (MPI) is a message passing standard, designed to simplify the inter-process

communication of processes running a parallel program across distributed memory. MPI allows point-to-point (one process to another) as well as collective (one-to-many, many-to-one, many-to-many) communication. The main concepts of MPI are communicators, datatypes, point-to-point communication and collective communications [33].

4.3.1 Communicators

Communicators define group of processes and arranges the group in an ordered topology.

4.3.2 Data types

MPI defines its own common data types (int-MPI_INT, char-MPI_CHAR, double-MPI_DOUBLE etc.) in order to support heterogenous environments where data types are represented in different ways (ex. big-endian vs small-endian).

4.3.3 Point-to-point

The most common MPI functions are MPI_SEND and MPI_RECV, in order to send and receive data respectively. Their definition, as of MPI specification, ver. 3.0 are:

- MPI_SEND (buf, count, datatype, dest, tag, comm)
    - buf: initial address of a buffer containing the message
    - count: number of elements in the buffer
    - datatype: datatype of the buffer's elements
    - dest: integer, denoting the destination process' rank
    - tag: message tag
    - comm: communicator
- MPI_RECV (buf, count, datatype, source, tag, comm, status)
    - buf: initial address of a buffer to store the message
    - count: number of elements in the buffer
    - datatype: datatype of the buffer's elements
    - source: integer, denoting the source process' rank
    - tag: message tag
    - comm: communicator
    - status: status object, containing information about the operation

These are blocking operations. When an MPI_RECV is called, the execution is halted, until a message is received. Non-blocking versions of these methods exist, named MPI_ISEND and MPI_IRECV.

4.3.4 Collective communication

Collective communication functions enable many processes to communicate and synchronize. The simplest form of synchronization is achieved through MPI_BARRIER, where all processes must call the function in order any of them to continue their execution. Collective communication is achieved through:

- MPI_BCAST, where a buffer is broadcasted from a root source to many destination
- MPI_GATHER, where all processes send a buffer to the root process
- MPI_SCATTER, where a buffer sent from a root source is divided among the receiving processes
- MPI_ALLGATHER, where all the processes receive a buffer from all the other processes.
- MPI_REDUCE, where all the processes send a buffer to the root process, and they are aggregated through a reduction operation (max, min, average etc.).

4.4 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) was developed by Nvidia Corporation, in order to enable general-purpose computing on graphics processing units (GPGPU) on their graphics cards [34]. Originally designed to extend the C/C++ programming language, it enabled programmers to exploit the power of massively parallel, simple processors (CUDA cores) on Nvidia Graphics Processing Units (GPUs). CUDA was one of the driving forces behind the recent advances in Deep Learning, as it enabled the massive parallelization of Deep Neural Networks' training, thus making experimentation and evaluation of different architectures feasible [1].

CUDA cores execute threads. Each thread has private, local memory and belongs to a thread block, which in turn belongs to a grid. Threads within a block can share memory (have access to memory of other inter-block threads) but not within the grid. Grids read data from and write data to the GPU's global memory.

In order to execute a CUDA kernel (CUDA program) on the GPU, a conventional programmer must learn to think at a thread-level. As operations are performed by threads,

the kernel must contain instructions about the thread's actions. The most important piece of information that is needed in order to determine a thread's action is the thread's position (index) in the grid and the block. This information is stored in the built-in variables threadIdx and blockIdx, where the thread's position in the block and the block's position in the grid are specified.

The block is a three-dimensional structure, thus threadIdx.x, threadIdx.y and threadIdx.z determine the (x, y, z) coordinates of the thread in the block. Variables blockIdx.x, blockIdx.y, blockIdx.z determine the (x, y, z) coordinates of the block in the grid. Furthermore, blockDim and gridDim store the dimensions of the block and the grid.

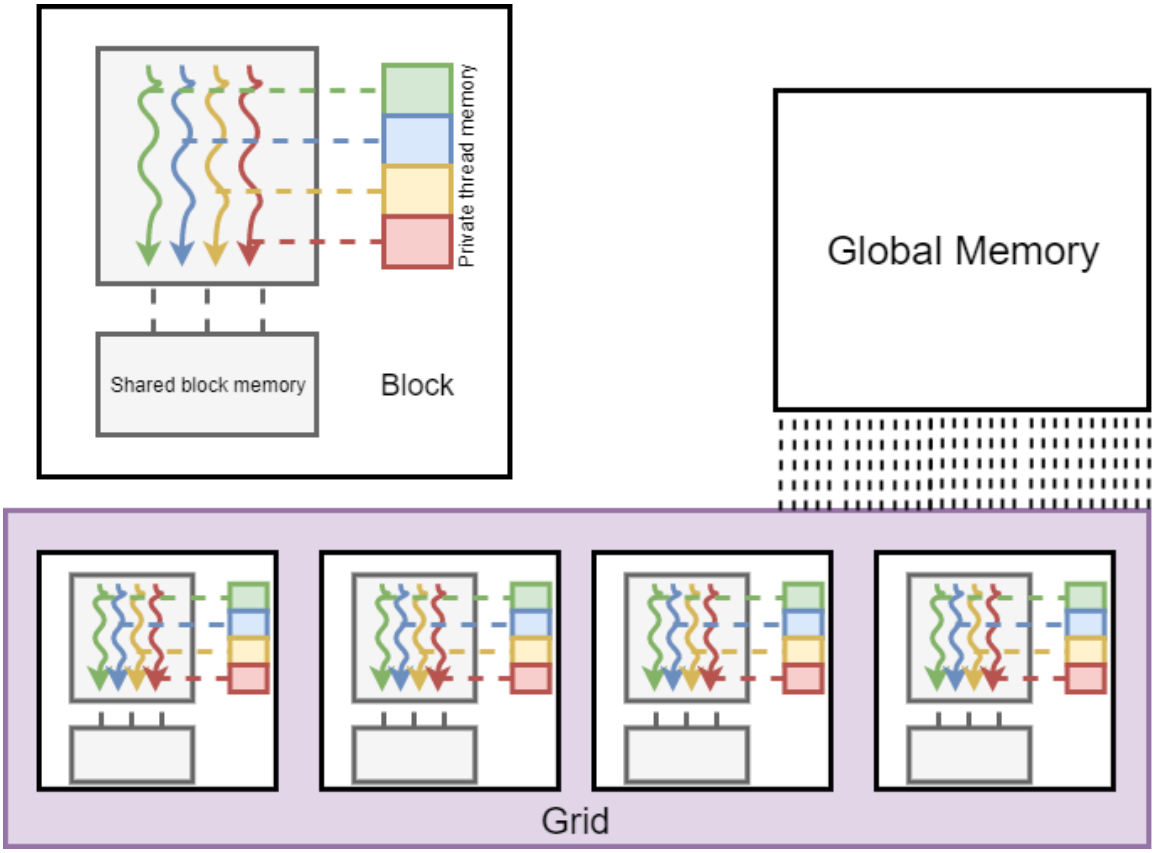

Figure 12 CUDA GPU architecture

The simplest example of a CUDA kernel is the addition of two vectors. Assuming vectors $A, B \in \mathbb{R}^n$, the following kernel computes $C = A + B$.

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < n)
    {
        C[i] = A[i] + B[i];
    }
}
```

Figure 13 Vector addition CUDA kernel

4.5 HPC in Neural Architecture Search

As the evaluation of a Neural Network's Architecture is a computationally intensive task, HPC techniques have been used in order to speed it up. In [26], the authors used a set of parameter servers and a set of controller replicas. Each replica and server were run on a separate machine. Furthermore, each controller replica sampled many architectures, and evaluated each one in parallel, on separate machines. A heavy use of distributed, MIMD is observed.

Furthermore, [24] could very well distribute the evaluation of each generation's population to separate machines. Assuming that each population is comprised of n individuals, n separate machines could be used, in order to evaluate the population in parallel. Due to the time required for each architecture to be trained and evaluated and the independency of the evaluation procedure from other individuals, the communication overhead would be small, compared to the potential speedup that could be achieved.

5 Double Deep Q-Learning Networks

5.1 Introduction

As mentioned in Chapter 3.5 , there is overestimation (positive bias) of potential actions' Q-values when Deep Q-Learning Networks are used. This led the Google DeepMind team to experiment with other methodologies in order to eliminate this bias. A solution that eliminated this bias was the usage of two Deep Q-Learning Networks. One of them was used in order to select the optimal action at each step, while the second was used in order to actually estimate the Q-Value [6].

5.2 The algorithm

The algorithm is the same as the Deep-Learning version (function approximation) of Q-Learning, with the following modification. Assuming two neural networks, A, B the original Q-target function is

$$Q = R_a(s_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) \quad (5\text{-}1)$$

In Double Deep Q-Learning, equation (5-1) is modified into

$$Q = R_a(s_t, s_{t+1}) + \gamma Q_A\left(s_{t+1}, \max_a Q_B(s_{t+1}, a)\right) \quad (5\text{-}2)$$

Here, network A is used to estimate the actual Q-value, while network B is used in order to select the optimal action. Network A is the target network, while network B is the primary network.(6-1)

6 Advantage Actor-Critic

6.1 Introduction

In this chapter, we discuss the estimation of the advantage function, as well as the asynchronous and synchronous versions of the Advantage Actor-Critic algorithm. These algorithms were used in our research, in order to train an agent to build optimal neural network architectures.

6.2 Advantage Function

For a given policy π, the advantage of a state/action pair, at time t is defined as the difference of the state-value function from the action-value function [35]:

$$A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t) \quad (6\text{-}1)$$

The advantage function provides a baseline (the state-value function) in order to evaluate the quality of a state/action pair (how advantageous the action is, given the baseline). In actor-critic frameworks, it provides stability, by reducing variance of the policy gradient, given that the critic approximates the advantage function and corrects the actor based on it.

6.3 Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic (A3C) was proposed by the Google DeepMind team, in a paper proposing many asynchronous methods for deep Reinforcement Learning [36]. The algorithm uses an actor-critic neural network, were

some of the parameters are shared (i.e. there is only one input layer, some hidden layers and two output layers, connected to the last shared hidden layer). The critic approximates the value function while the actor maintains a policy. After a certain number of steps, or when a terminal state is reached, both the actor as well as the critic are updated. Furthermore, the authors found that adding the policy's entropy to the actor's loss function increased exploration, thus preventing the agent from converging early to sub-optimal solutions. Assuming actor, critic parameters θ_a, θ_c, t steps for an episode that has ended and a vector $r \in R^t$ where the immediate reward of each step has been recorded, the parameters are updated as follows:

$$R = \begin{cases} 0 & \text{if } s_t \text{ is terminal} \\ V(s_t) & \text{for any other } s_t \end{cases}$$

for $i \in \{t-1, \dots, 0\}$

$$d\theta_a = d\theta_a + \nabla_{\theta_\alpha} \log \pi\,(a_i|s_i)\big(R - V(s_i)\big) + \beta \nabla_{\theta_\alpha} H\big(\pi(s_i)\big)$$

$$d\theta_v = d\theta_v + \frac{\partial \big(R - V(s_i)\big)^2}{\partial \theta_v}$$

Figure 14 The update rule for A3C

The algorithm is designed in order to be executed asynchronously. A master thread retains the actor-critic network's parameters (a parameter server) and many worker threads copy the parameters to their local memory, execute an episode, calculate the network's gradients and immediately send them to the parameter server. Thus, for any given time, it is possible that some threads have outdated local parameters, compared to the parameter server. Figure 15 depicts a possible execution scheme, where two threads reach terminal states, while a third thread never reaches a terminal state and is instead ran until the maximum number of steps is reached.

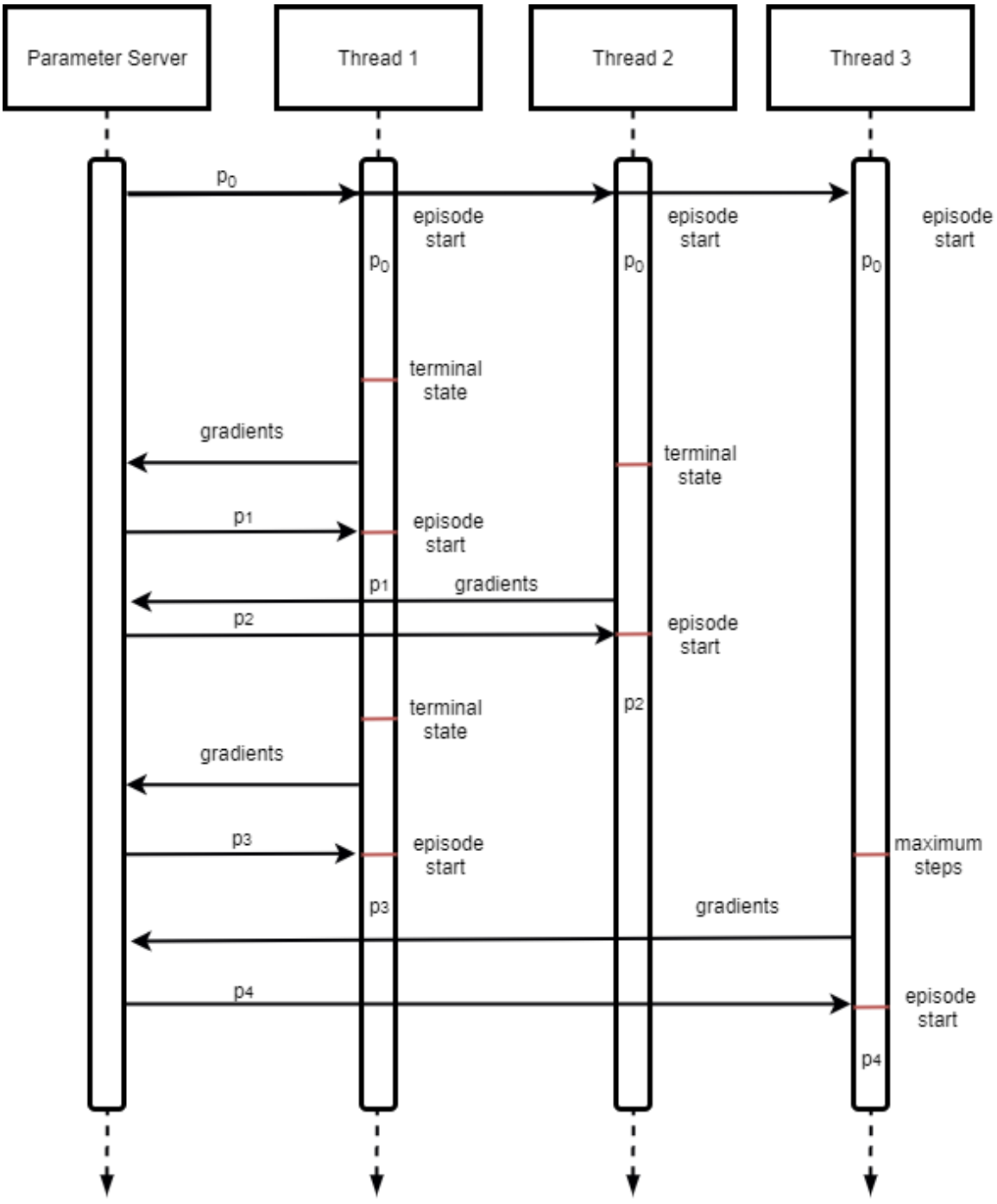

Figure 15 Asynchronous execution of A3C

6.4 Synchronous Advantage Actor-Critic

OpenAI, a non-profit AI research company, found that the synchronous updates lead to better performance. Furthermore, the company claims that it can more effectively use GPUs, as they are able to operate on larger batch sizes. Moreover, they claim that there was no notable benefit of introducing noise due to asynchrony (delayed propagation of updated parameters) [37]. They introduced a synchronous version of A3C, called A2C

where the parameter server waits until all workers have finished before updating the network parameters and broadcasting them to the workers. The parameters are updated by averaging the gradients of all workers.

7 DDQN Application in discrete spaces

7.1 Introduction

Our first Neural Architecture Search with Reinforcement Learning utilizes a relatively simple method, Double Deep Q-Learning [6]. We apply Double Deep Q-Learning to a discrete action and state space, training an agent to learn the best possible neural architecture.

7.2 Methodology

7.2.1 Problem Formulation

In order to formulate the problem as a Markov Decision Process, we need to define the possible states *S*, the available actions *A* and the rewards *R* for each state-action pair. To keep the implementation as simple as possible, the available actions are the expansion of the current architecture, with fully-connected layers, with sizes 128,256,384,512,640,768,896,1024 or dropout layers with a possible dropout rate of 0.1,0.2,0.3,0.4,0.5,0.6,0.7. Furthermore, we define an additional action $\emptyset$, which keeps the network in its current state (i.e. no additional layer is added). Given that we impose a limit of a maximum of 4 layers, the states are defined as an $(\#A - 1) \cdot 4 = 60$ bit vector (number of possible actions that alter the network's state times the maximum depth).

Assuming a one-hot encoding of the actions except for $\emptyset$, the state is the concatenation of the actions that led to the current network's configuration. In other words, if a network resulted from actions $a_1, \emptyset, a_3, a_2$ and there are only 3 available actions plus $\emptyset$, the representation would be the one-hot vectors representing 1,3,2 i.e. 100,001,010,000. Notice that the representation length is 12 ($3 \cdot 4$) and by using action $\emptyset$ we leave the network intact, thus having a final depth of 3.

Finally, as a reward we consider the improvement in accuracy from the previous state. For example, if a network in state s has 95% Out of Sample (OOS) accuracy, performs action a which induces a transition to state s' with a 95.5% OOS accuracy, the reward r would be 0.5% or 0.05. This formulation ensures that the total reward for any

state is less than or equal to 1. Furthermore, deterioration of OOS accuracy is represented by negative values.

In order to avoid high deviation in rewards due to the first layer (as a zero-layer network has 0% OOS accuracy) we assume a baseline of 90% OOS accuracy, as most configurations of one-layer networks achieve OOS accuracy greater than 85% and define the reward of the single-layer network as its accuracy minus the baseline.

7.2.2 Use of Reinforcement Learning

In order to solve the formulated MDP, we use a Double Deep Q-learning Network [6]. As exploration policy, we use an ε-greedy approach with initial $\varepsilon = 1.0$ and decay rate $d = 0.995$ in order to retain a high level of exploration. The DDQN is trained for 200 episodes (which is not very much, but restrictions in computational resources prohibit us from running lengthy experiments). Each generated network is trained for 3 epochs (again, due to computational resources restrictions, as the networks are evaluated on CPU) and then evaluated on the validation set. In order to improve the estimation of single-layer networks' accuracy (as it in turn provides better estimation for multi-layer configurations' rewards) and to reduce the total time required, we start with a maximum depth of 1 and increase it every 10 episodes, up to 4. As mentioned in 7.2.1 , the total reward of each possible state is bounded, so we use a discount rate of $\gamma = 1$. Finally, a memory of length 200 was used. State-action combinations previously evaluated were not re-evaluated. Instead, they were recalled from memory.

7.2.3 Implementation

We implemented the method using Keras, a Deep Learning wrapper library for the Python programming language [38]. The Q-Networks are comprised of a single dense layer with 128 neurons, followed by a Rectified Linear Unit [17] and a dropout layer with 0.1 dropout rate. We used RMSProp in order to train the networks. The generated networks were evaluated on the MNIST dataset of hand-written digits. Figure 16 shows the layer type, as well as the input and output shape of each Q-Network's layer. "None" means that the batch size is variable, while the second dimension is fixed.

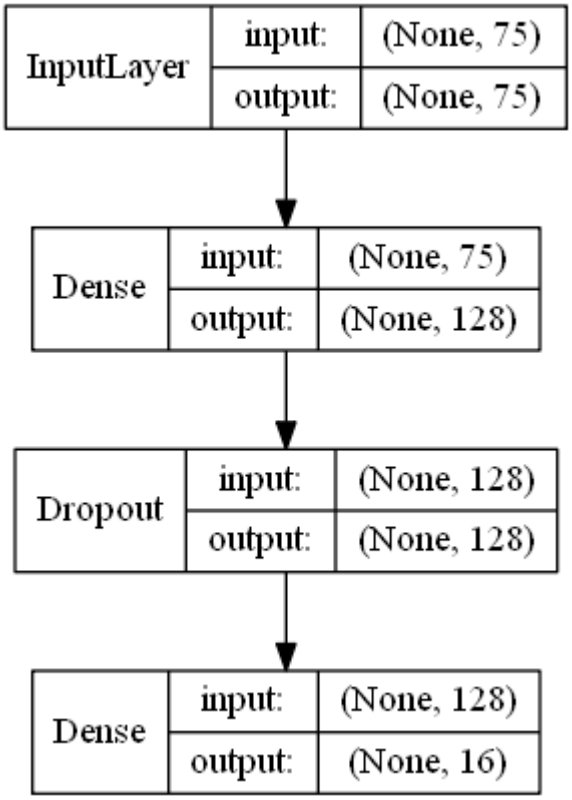

Figure 16 Q-Network Architecture

7.3 Experimental Results

Figure 17 depicts the current architecture's final (after all layers have been added) accuracy for each episode. It is clear that it converges to high quality solutions, although due to the ε-greedy exploration policy, there are considerable deviations. In order to evaluate the method's quality, we compare it to random search as a benchmark, which can be quite difficult to surpass [39]. Instead of training an agent to design the architecture, we sample random architectures, following the same rules that the agent is following. It can be seen as following an ε-greedy policy, with $\varepsilon = 1$ for the full duration of the search.

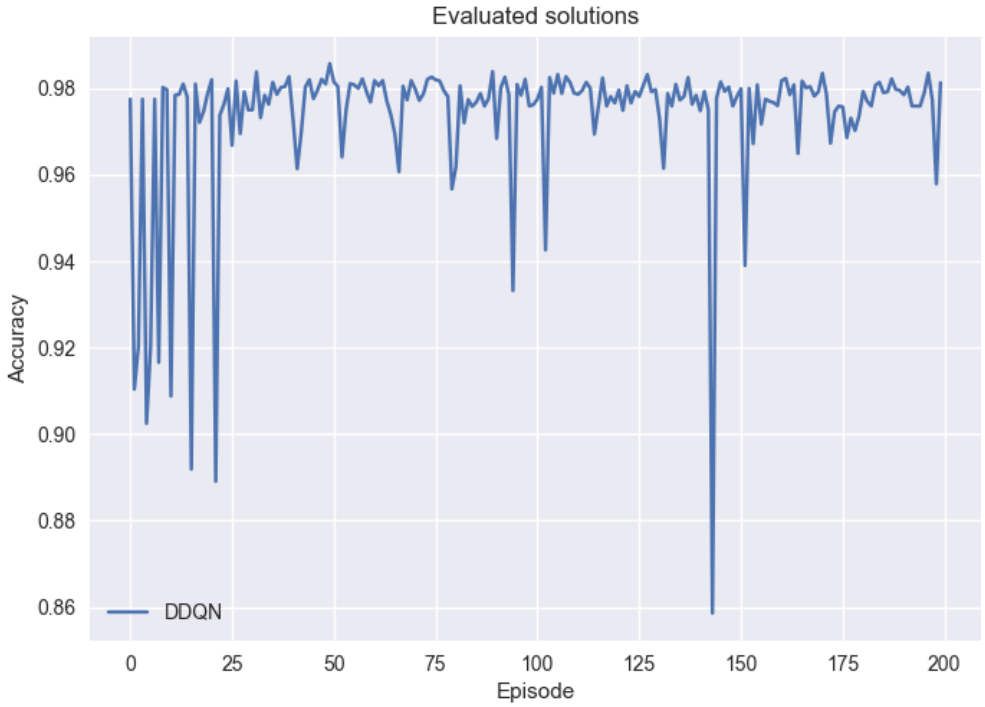

Figure 17 DDQN current solutions' accuracies.

In Figure 18 and Figure 19 the empirical cumulative distribution functions of the Double Deep Q-Learning Network and Random Search are plotted, for the current best, as well as for the current solution. We can clearly see, that in both figures, the DDQN approach exerts first order stochastic dominance over the random search, i.e. its ECDF is to the right of the random search. Furthermore, by applying the Kolmogorov-Smirnov two-sample statistical test, we find that the two distributions differ ($p < 1e - 6$), for both the current as well as the current best solution. In Table 2, the mean of each distribution is depicted, as well as the total time required for each method. DDQN has a higher mean for both the current best, as well as the current solution, while it finished 4031 seconds earlier (about 1h10m faster, a speedup of 1.16).

Concluding, the DDQN approach, although quite simple, managed to produce better results (higher probability to find a good architecture) in a shorter time than Random Search.

Table 2 Statistics of DDQN, Random Search.

Method	*Best Mean*	*Current Mean*	*Total Time (s)*
DDQN	98.5%	97.4%	25598
Random Search	98.2%	96.4%	29629

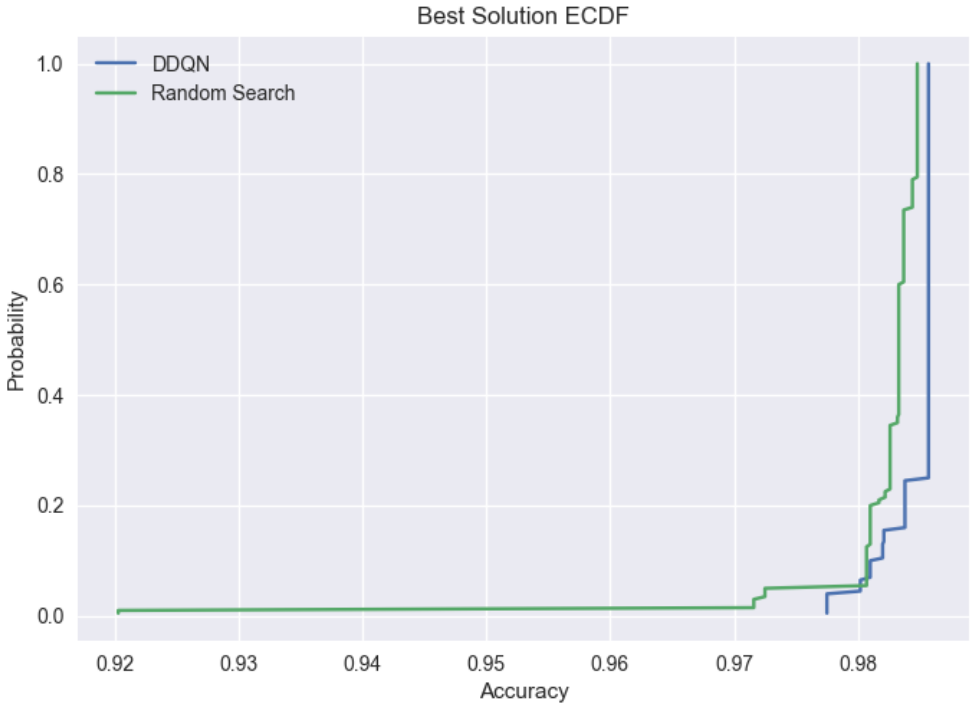

Figure 18 Empirical Cumulative Distribution Functions of DDQN and Random Search for the best architectures found.

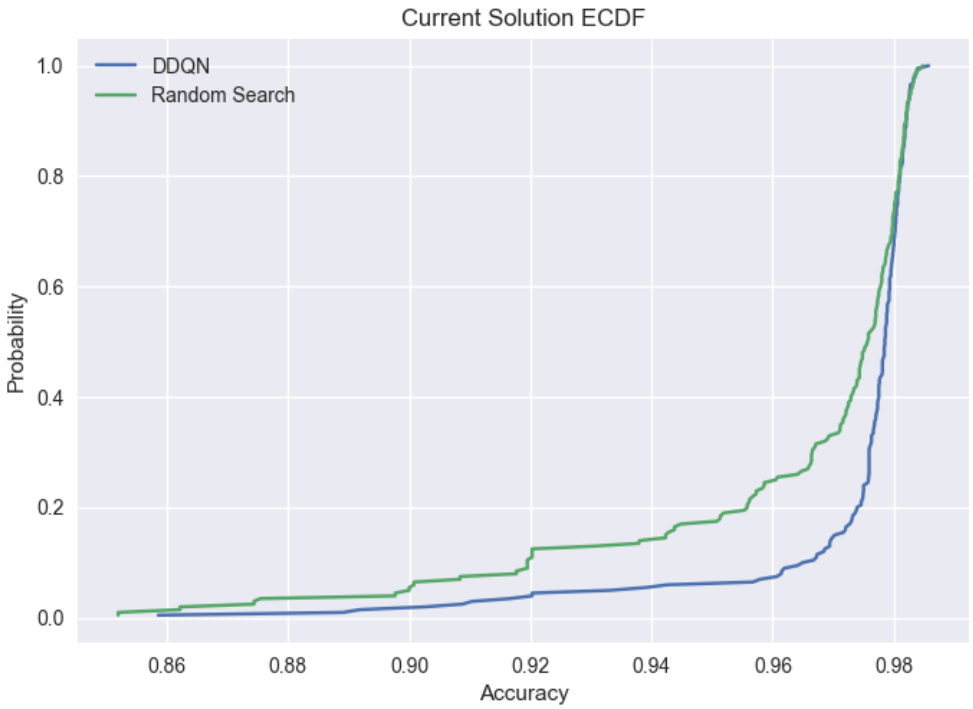

Figure 19 Empirical Cumulative Distribution Functions of DDQN and Random Search for all the architectures.

7.4 Advantages

The application of Double Deep Q-Learning Networks in Neural Architecture Search proved e have an advantage over the Random Search. Furthermore, it provided us with the following feedback:

- It is feasible to use Reinforcement Learning in discrete spaces for the Neural Architecture Search problem.
- It is more probable to find a good architecture with Reinforcement Learning than with Random Search.
- It is quicker than Random Search, due to the fact that as the agent learns, it resamples the same architectures, which have already been evaluated and thus their accuracy is cached.

7.5 Limitations

Even though this approach seems promising, there are some limitations:

- The execution time, although lower than Random Search, is still quite high. This is partly due to the fact of using a wrapper library (Keras), which is not optimized to be used in such applications (massive interactive architectures definitions).
- It is not designed to leverage HPC methods, such as distributed computing. Thus, it is not scalable without modifications.
- It is not easily extensible to continuous spaces. In order to design modern architectures, we need continuous actions.

7.6 Remarks

We found that using the improvement of accuracy is a better reward than accuracy itself. First, it bounds the returns and the rewards to [-1,1]. Second, it avoids rewarding stagnating networks, thus encouraging the method to search for even small improvements in accuracy.

Hyperbolic tangent works better to approximate the expected rewards than linear activation, as the rewards are bounded to [-1,1].

8 A2C Application in discrete spaces

8.1 Introduction

In order to leverage the capabilities of distributed computing, we decided to implement a distributed Reinforcement Learning algorithm to train our agent. Synchronous Advantage Actor-Critic (A2C) was chosen, as it offers the following advantages:

- It can be extended to continuous action spaces with small modifications.
- It can be used to train Recursive Neural Networks.
- It is a distributed algorithm, with a master parameter server and exploration workers.
- It is more robust than A3C, as all the workers share the same parameters at all times [37].

8.2 Methodology

8.2.1 Problem Formulation

The problem formulation remains the same as in 7.2.1 with the following changes:

- Action space consists of
    - Fully connected layers with 64, 128, 256, 512, 1024 neurons.
    - Dropout layers with 0.1, 0.3, 0.5 dropout rate.

8.2.2 Implementation

In order to utilize more efficiently the available computational resources, we implemented the architecture search framework in Tensorflow [40]. Both the meta-network (the network trained by A2C) as well as the candidate architectures are trained and evaluated using Tensorflow. Moreover, the custom loss functions of the method are implemented more naturally than they would be on Keras (where we would have to use lambda layers with additional inputs). The experiment was executed with a total of 4 workers.

We use an entropy beta factor of 0.01 for the loss function and the Adam optimization algorithm with a learning rate of $1e-5$ [41]. The gradients are clipped by the network's weights global norm. We calculate the discounted rewards for the whole episode, as episodes are relatively short. Furthermore, we do not use an ε-greedy exploration policy. Instead, at each step, we sample each action with the probability

outputted by the meta-network's Policy output layer. Figure 20 shows the meta-network's architecture. Using the current state as input, the network outputs:

- The estimated probability that each action is optimal in the current state (Policy layer)
- The state-value function of the current state.

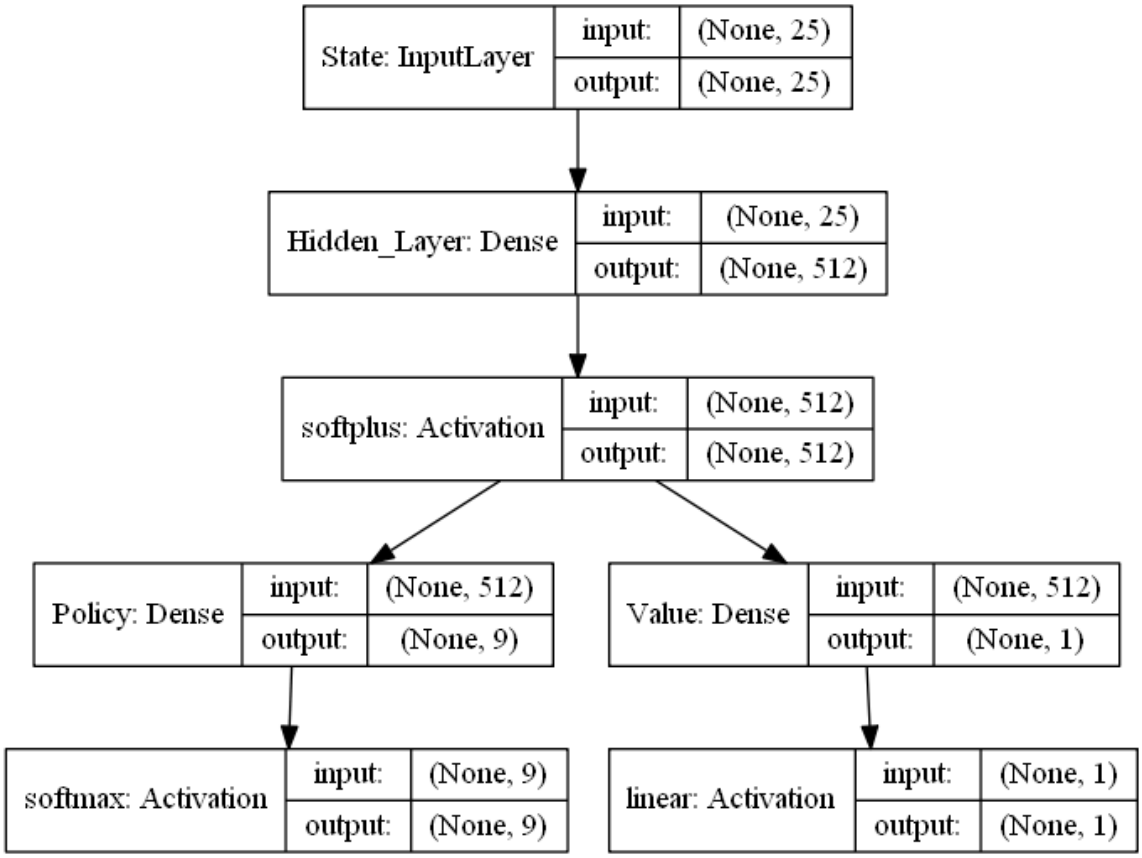

Figure 20 Discrete A2C meta-network architecture.

8.3 Experimental Results

The method seems to quickly converge to optimal solutions. As seen on Figure 21, at episode 50, it has already converged. Once again, we compare the method with a control experiment, using random search in order to generate a distribution of architectures. Once again, the method displays stochastic dominance over the random search, for both the current solution as well as the current best solution. This can be visually verified in Figure 22, Figure 23. In Figure 23 we see that the Random Search eventually finds the same best solution as A2C. Applying the two-sample Kolmogorov-Smirnov test, we conclude that the Random Search and A2C distribution are statistically significant different ($p < 1e-6$). Table 3 depicts basic statistics about both methods.

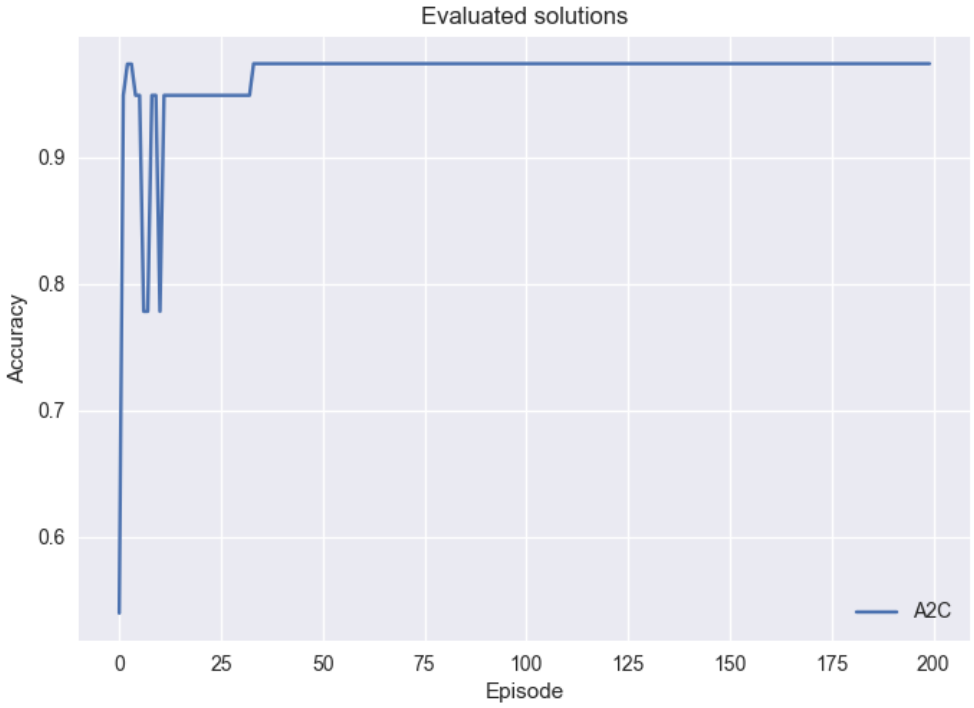

Figure 21 Discrete action space A2C Current solution's accuracy.

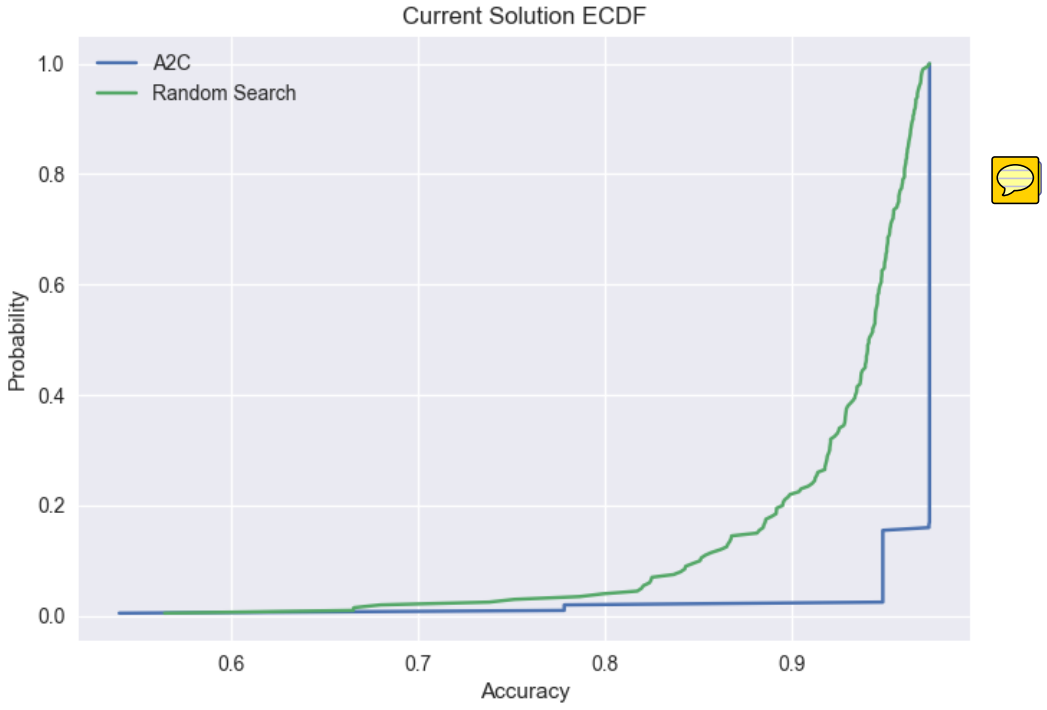

Figure 22 Empirical Cumulative Distribution Functions of discrete action space A2C and Random Search for all the architectures

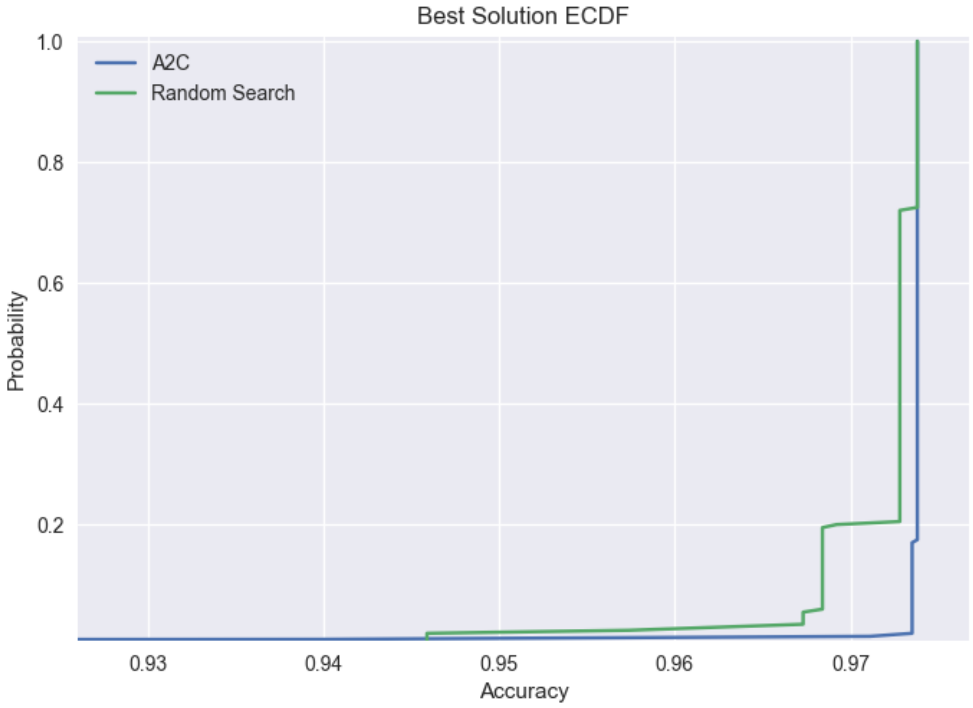

Figure 23 Empirical Cumulative Distribution Functions of discrete action space A2C and Random Search for the best architectures found.

Table 3 Statistics of discrete action space A2C, Random Search

Method	*Best Mean*	*Current Mean*	*Total Time (s)*
A2C	97.4%	96.5%	1212
Random Search	97.2%	92.2%	295

Due to the inter-process communication, as well as the need to evaluate all intermediate architectures, we found A2C to be significantly slower than Random Search. The average round-trip time (construction of the network, evaluation and accuracy report to the master) for Random Search was 1.5 seconds, while A2C needed 5.9 seconds. This implies a speedup of 4 for the Random Search. Still, A2C manages to converge before episode 50. By taking 100 samples of 50 architectures, we can estimate the distribution of the best architecture found for the Random Search. The probability of finding the best architecture (with accuracy 0.738) is 0.2 (Figure 24).

Assuming that the same computing resources are available for both methods, Random Search can be run 4 times (as it is 4 times faster than A2C) and finish at the same

time as A2C, but the probability that none of the 4 runs will find the best architecture in the first 50 episodes is $(1 - 0.2)^4 = 0.4096$.

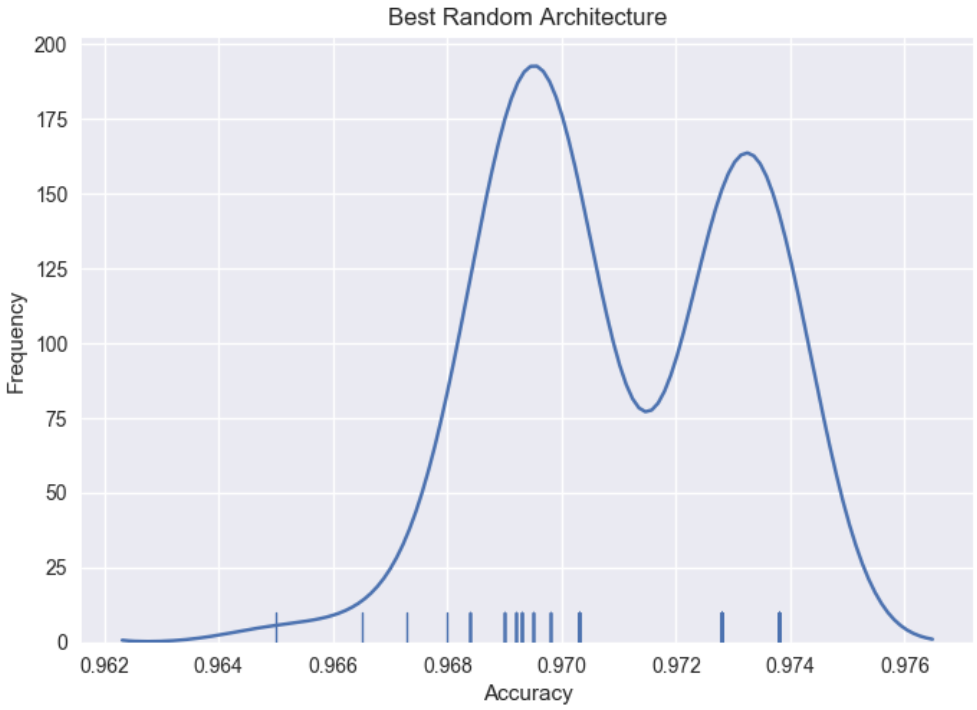

Figure 24 Kernel density estimation and rug plot for the estimated distribution of Random Search best architecture.

8.4 Advantages

The application of Synchronous Advantage Actor-Critic in the Neural Architecture Search problem offers the following advantages over Double Deep Q-Learning Networks:

- It is easily extensible to continuous action spaces.
- It converges quickly to good architectures.
- It is natively a distributed algorithm
- There is no need to explicitly define an exploration policy, as the network outputs stochastic policies (action probabilities distribution) which can be followed.

8.5 Limitations

As it was experimentally shown, Random Search has a significant probability (20%) to find the same best architecture as A2C in small discrete action spaces. This is logical, as our action space was comprised of 8 actions. A 4-layer neural network

developed using these actions can have $8^4 = 4096$ different architectures, which is a small enough space to be explored using brute force. Furthermore, the need to transmit gradients and wait for all the workers to complete in order to broadcast the new network parameters makes Random Search an enticing alternative. Still, Random Search has a lower probability to produce high quality architectures, even if both methods are run for equal lengths of time.

9 A2C Application in continuous spaces

9.1 Introduction

Fully connected and dropout layers, although essential in many applications, are not state-of-the-art. In order to design more modern architectures, such as convolutional networks, we need to expand our framework in continuous spaces. In this chapter we present the theoretical modelling of continuous action spaces, as well as an A2C implementation for the problem of Neural Architecture Search on the CIFAR10 dataset.

9.2 Methodology

9.2.1 Problem Formulation

We consider fully convolutional layers with two main hyperparameters. First, there is the filter's dimensions, namely height and width. For the sake of simplicity, we assume a square filter, with equal height and width. The second hyperparameter concerns the number of filters that exist in each layer. Thus, we have two axes (filter size, filter number), bounded below by 1 (in order to produce valid layers) and bounded above by (u_s, u_n). Thus, the agent's actions are $A \in \mathbb{Z}^{u_s \times u_n}$. Note that although the actual action space is discrete, it can quickly explode in size, as for $(u_s, u_n) = (5,128) \rightarrow \#\mathbb{Z}^{u_s \times u_n} = 640$. Furthermore, the state space becomes even bigger, as for a two-layer deep network, the state space has a size of 1280.

The output layer remains a fully connected layer with number of neurons equal to the number of the dataset's classes. Thus, the state S can be represented by a vector of tuples, each denoting the layer's (filter size, filter number). As a reward R, we continue to use the improvement over the previous state's accuracy.

9.2.2 Implementation

As the agent acts in a continuous action space, the network cannot output a distribution of probabilities for a set of actions. Instead, the output must be a set of distributions, one for each of the possible axes of action. In our application, we output two probability distributions, one for the filter size and one for the filter number. The network's actual output is the mean and the standard deviation of a normal distribution. At each step, we take a random sample from the outputted distribution for each axis.

Moreover, at each step the agent performs the same decision (what dimensions should the network's next layer be?), which depends on its previous decisions (which define its state). Thus, a Recurrent Neural Network was chosen as the meta-network. The network sequentially reads the tuples defining its layers and decides what the next layer's parameters should be. The layer we chose as a recurrent layer for our network was a Gated Recurrent Unit (GRU), as they have fewer parameters than Long Short Term Memory layers, while achieving the same performance [42].

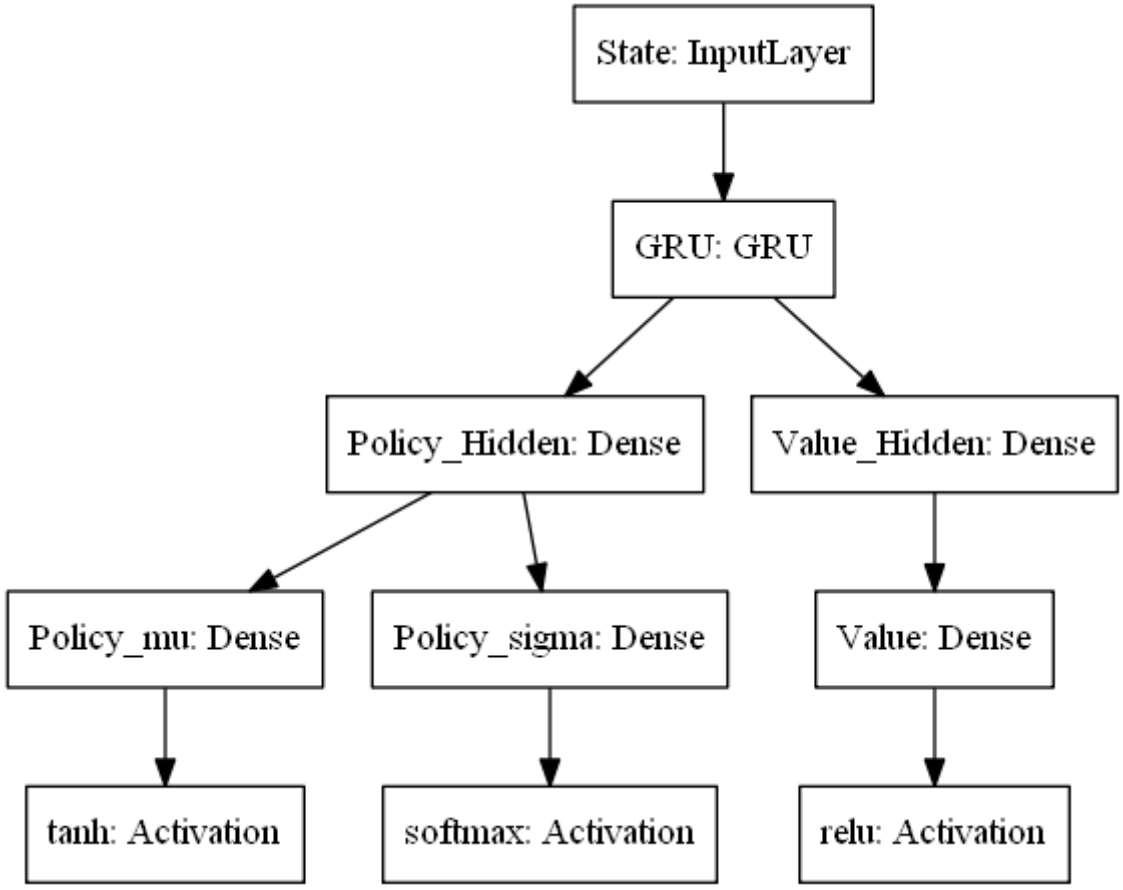

Figure 25 Continuous action space A2C network.

The Value output layer backpropagates the errors and gradients to the recurrent layer, while the two policy output layers backpropagate their errors and gradients only to

the hidden policy layer. This stabilizes the network's training. Furthermore, this enables the use of different learning rates for the two sub-networks. The network that is trained by the value output layer has a learning rate of $1e-2$ while the policy part of the network has a learning rate of $1e-5$, both using the RMSProp optimizer. This ensures that the policy does not oscillate or change dramatically, while the value of the current policy is quickly learned.

In order to evaluate our implementation, we trained an agent to learn to design topologies for the CIFAR10 image classification dataset. The agent designed two-layer networks, and evaluated each architecture by training it for 3 epochs and then evaluating the trained network on a validation set of 10.000 samples. The agent was trained for 4.000 episodes. Architectures that were not valid or did not fit in the GPU's memory were discarded. The experiment was executed on a cluster with GTX730 gpus, on 8 worker stations.

9.3 Experimental Results

In Figure 26 the accuracy of each episode is depicted. Although the agent learns to avoid low-quality architectures, it seems that it cannot converge to one particular solution. This is due to the fact that the network's evaluation procedure entails a 3-epoch training, due to restricted computational resources. This creates great variation in the accuracy produced for a given architecture to different worker processes (as each process has a local cache of evaluated architectures). Once again, in order to assess the overall quality of the solutions, we compare the agent's results with a random search.

In Figure 27 the ECDFs of the best solutions found are plotted, while Figure 28 depicts the ECDFs of all the architectures evaluated, for both methods. It is interesting, that while there is no clear stochastic dominance, A2C is more probable to find high-quality solutions (above 50% accuracy) while Random Search is less probable to find low-quality solutions. Furthermore, A2C manages to find better architectures, although Random Search's worst best architecture is better than A2C's 20% of best architectures. The statistics of each method are available at Table 4 .

Table 4 Statistics of continuous action space A2C, Random Search for CIFAR10.

Method	*Best Mean*	*Current Mean*	*Total Time (s)*
A2C	62.4%	53.9%	88679
Random Search	61.8%	53.4%	106909

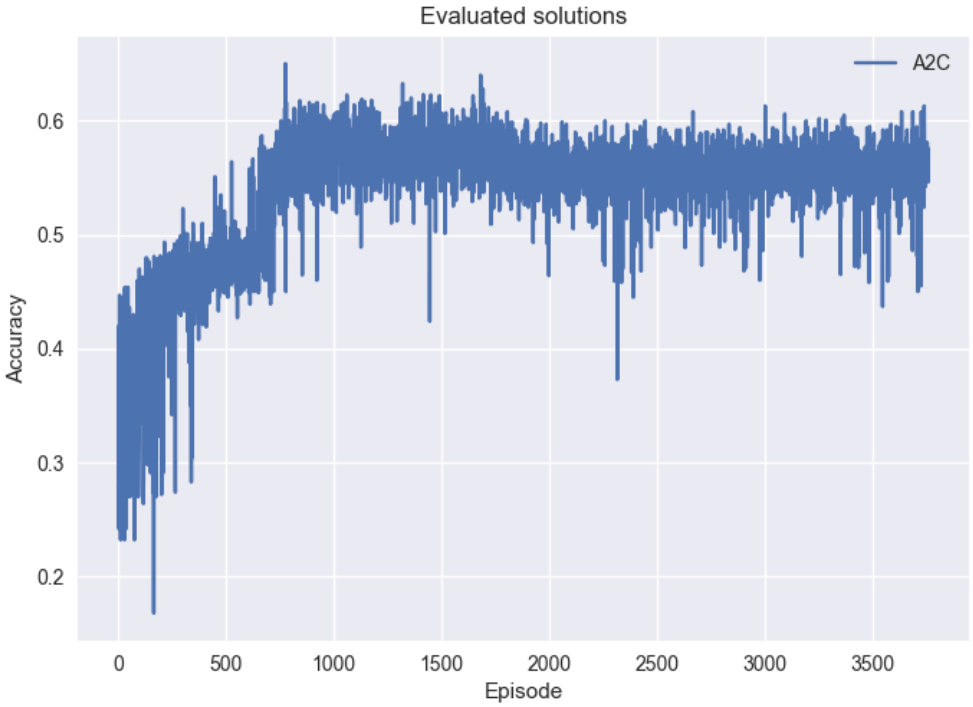

Figure 26 Continuous action space A2C Current solution's accuracy.

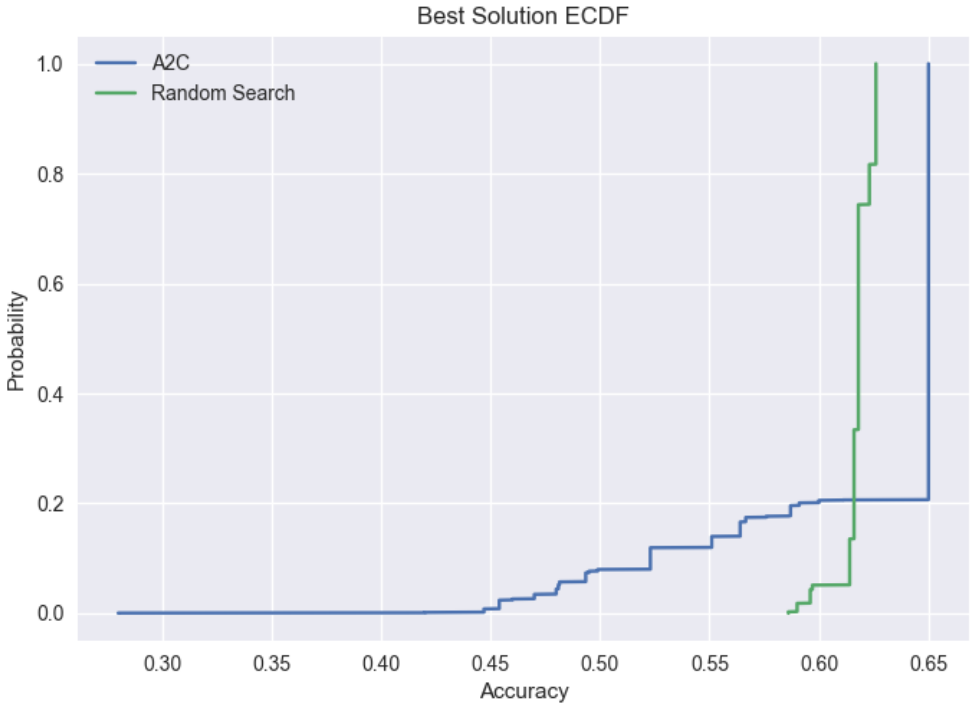

Figure 27 Empirical Cumulative Distribution Functions of continuous space A2C and Random Search for the best architectures found.

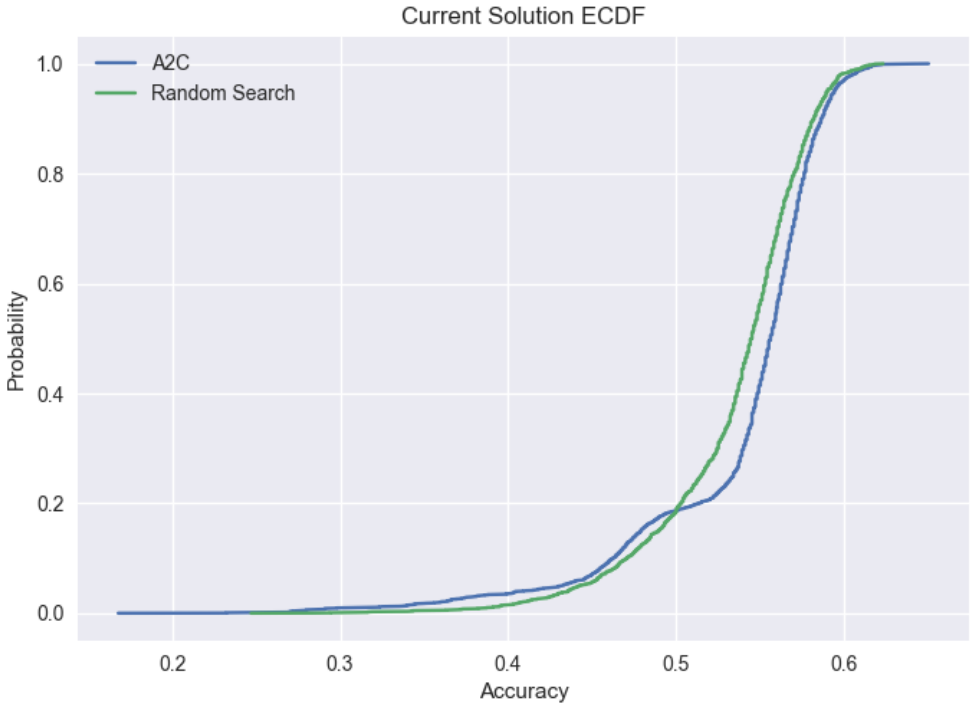

Figure 28 Empirical Cumulative Distribution Functions of continuous space A2C and Random Search for the all the architectures evaluated.

This behavior could be attributed to the fact that some architectures did not fit in the memory and were thus discarded, as well as to the fact that the training time of 3 epochs is quite small. Furthermore, the network instead of being assigned the task to learn the relative ranking of a small number of actions, has to optimize one distribution for each axis. Combined with the fact that the 8 workers may have very different perceptions of reality (the quality of a specific architecture) the outcome is not very surprising.

9.4 Advantages

The implementation of continuous action space, recurrent A2C has the following advantages over discrete action space A2C:

- It provides a way to act in continuous action spaces.
- It provides a compact representation of the action as well as the state space.
- It is able to produce better solutions than random search.

9.5 Limitations

The main limitations of the implementation are the following:

- It does not clearly dominate the Random Search approach.
- It needs more episodes in order to learn to avoid low-quality solutions.
- The expanded state space increases the state-reward variance between workers.

10 Evaluation of network architectures through partial training

10.1 Introduction

In order to reduce the computational resources required to evaluate a network architecture, as well as to reduce the trainable parameters (and thus the variance between workers) we experiment with partial training of the evaluated networks. In this chapter, we repeat the experiment of chapter 9 , but train only the last layer (the classification layer) of the candidate architecture.

10.2 Motivation and methodology

As mentioned in Chapter 3.6 , some studies have shown that even untrained networks with essentially random weights, can perform considerably well on some tasks. By taking advantage of these findings, we reduce the trainable layers of the candidate architecture to only one, the last (classification) layer. Our motivation is that by reducing the number of parameters, we also reduce the variance of validation accuracy between the workers, the memory needed on the GPU in order to store inter-layer matrix operations as well as reduce the time needed for training.

Furthermore, it is interesting to see if different architectures retain their relative ranking, when they are fully or partially trained. If a network's performance can be attributed to the architecture as well as its training, a good architecture should outperform most inferior architectures, given that they are trained the same way.

10.3 Experimental Results

As it is evident by Figure 29, the variation between workers was not improved by reducing the number of trainable layers. However, as the models were not rejected due to memory restrictions, the method now shows clear dominance over the Random Search in Figure 30 and Figure 31. Furthermore, by fully training the best architecture found for 3 epochs (layer-1 parameters: (2,75), layer-2 parameters: (4,114)) and evaluating it on the

validation set, the accuracy achieved is 59.6%, which is at the top 5% of the fully-trained version of the experiment.

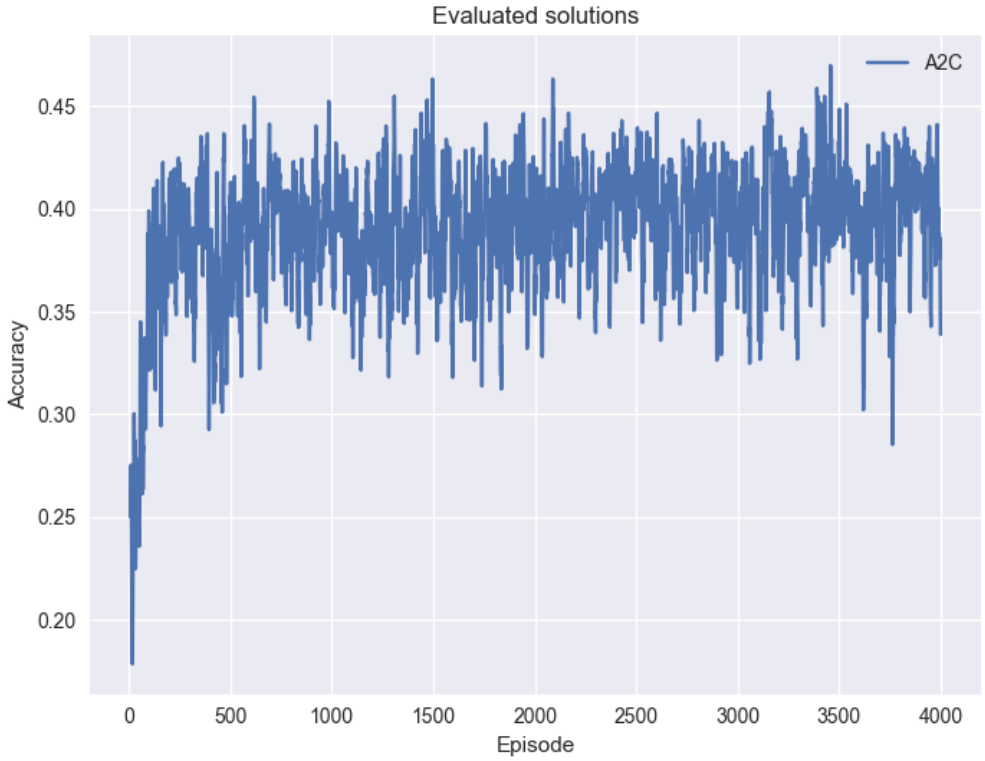

Figure 29 Reduced training current solution accuracy.

As seen on Table 5, A2C has higher means for both the current best as well as the current solution. Furthermore, it finished 4000 episodes 4 times faster than the fully trained version.

Table 5 Statistics of continuous action space A2C, Random Search for CIFAR10, reduced training.

Method	*Best Mean*	*Current Mean*	*Total Time (s)*
A2C	52.8%	39.0%	21934
Random Search	50.2%	36.2%	56463

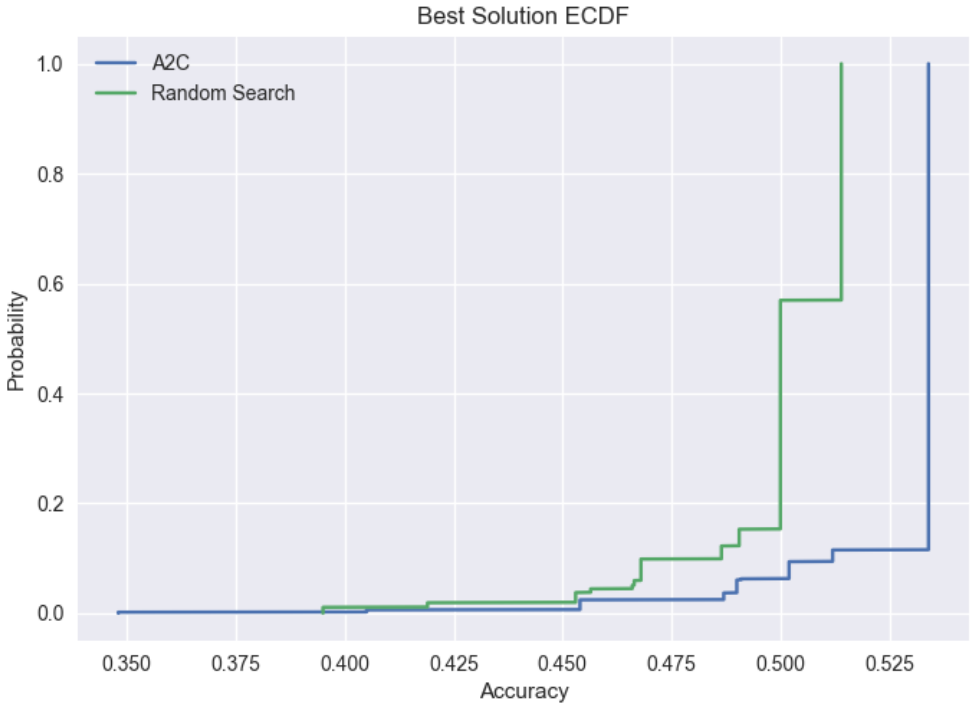

Figure 30 Empirical Cumulative Distribution Functions of continuous space A2C and Random Search for the best architectures found, reduced training.

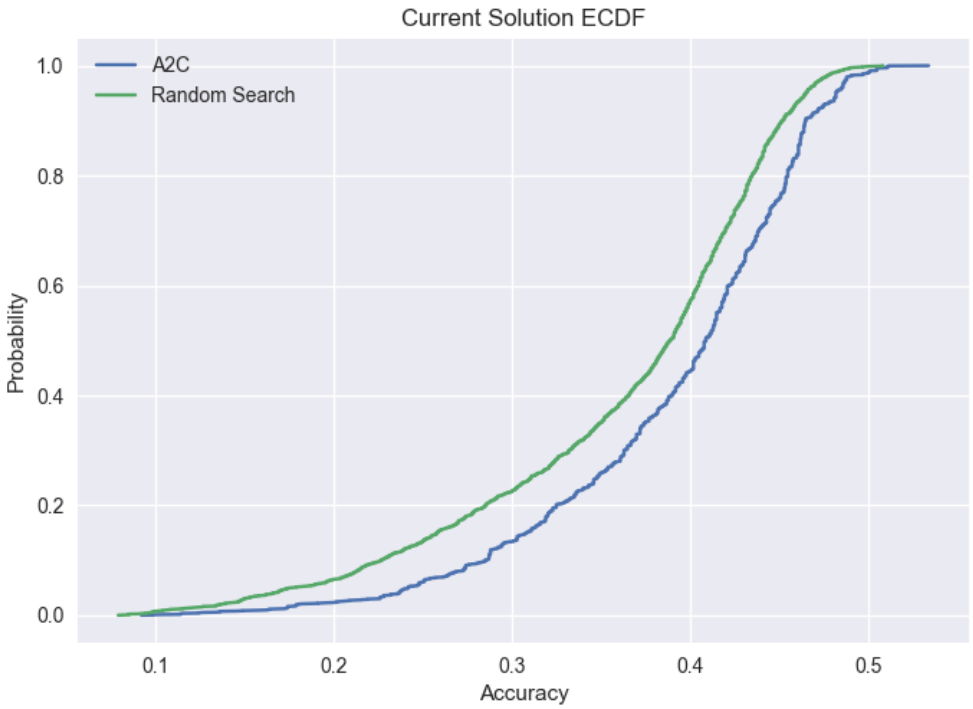

Figure 31 Empirical Cumulative Distribution Functions of continuous space A2C and Random Search for the all the architectures evaluated, reduced training.

10.4 Relative Rankings

In order to examine if the relative ranking of the architectures remains the same regardless of the training scheme, we compiled two lists of architectures and accuracies, one for each version of the experiment. In case an architecture was evaluated by many workers, the highest accuracy was used. After ranking the architectures based on their highest accuracy, we compared the two lists using Kendall's tau coefficient [43]. Kendall's tau is a non-parametric statistical test, utilizing the number of concordant and discordant pairs in order to assess the ranking correlation of two ordered sets. Using the architectures evaluated in our experiments, we found a correlation of 0.752 in the relative rankings ($p < 1e-5$). Thus, around 75% of the relative rankings seem to be preserved when using partial training.

11 Conclusion

In this study we investigated the feasibility of using Reinforcement Learning in the problem of Neural Architecture Search. Motivated by recent findings about the importance of a network's architecture to its performance, as well as the fact that the design process is a tedious, trial-and-error manual task, performed by experts, we tried to implement a simple method of Neural Architecture Search. By defining a state space, an action space and a reward function, necessary in order to formulate a Markov Decision Process, we solve it using Reinforcement Learning.

At first, we present the theoretical background behind Markov Decision Processes and Reinforcement Learning, in order to provide a basic understanding of the concepts discussed later. We then provide the basics of neural networks, their use in Reinforcement Learning, as well as recent research about the importance of architecture in the network's performance. A small introduction to High Performance Computing is presented, as it is an essential part of Deep Learning, as well as distributed Reinforcement Learning.

While our main aim was to apply a distributed, continuous action space algorithm to the problem, we approached it by solving the simplest problem first, the design of a fully-connected neural network for the image classification task, using the MNIST dataset of hand-written digits. At first, we implemented a simple Reinforcement Learning algorithm, Double Deep Q-Learning, using discrete actions in order to build a fully connected neural network. The methodology proved enough in order to outperform the

Random Search approach, indicating that Reinforcement Learning can indeed be utilized in order to build simple neural networks (Chapter 7).

In order to slowly progress towards a distributed, continuous action solution, we implement a distributed algorithm, trying to solve the same problem. Using Synchronous Advantage Actor-Critic, we design a small fully-connected network for the MNIST dataset. Once again, the algorithm outperforms Random Search, while it converges to optimal solutions faster than Double Deep Q-Learning (Chapter 8).

Following the successful application of A2C to our problem, we extend it, by modifying the network, in order to produce not a distribution of probabilities for a set of actions, but a set of probabilities for a set of axes of actions. Furthermore, we design fully convolutional neural networks for the CIFAR dataset. Although the method does not clearly outperform Random Search, it manages to find better solutions. Another problem that arose during this experiment was the variance between workers, due to the small number of epochs used to evaluate candidate solutions, as well as the rejection of architectures due to memory constraints (Chapter 9).

The problems encountered in Chapter 9 led us to investigate ways to reduce the computational resources needed in order to evaluate a candidate architecture. Inspired by recent studies, we tried to evaluate the architecture itself, by training only the output layer, while leaving the rest of the network random. This approach yielded better results than the fully trained approach. Architectures with big number and size of filters were not rejected due to memory constraints. The method outperformed Random Search, having a higher probability to find good architectures, as well as finding better architectures overall. Furthermore, the best architecture found was at the top 5% of the architectures found by the fully trained version of the experiment, while the speedup due to partial training was a factor of 4. Finally, we found that architectures retain their relative rankings, even when partially trained.

In summary, distributed Reinforcement Learning can be used in order to design neural architectures in continuous action spaces. Partially training a neural network architecture is a good way to reduce the computational resources needed to complete the evaluation, as it does not severely alter the relative rankings of the architectures.

11.1 Future Work

The study provided interesting results concerning the design of neural architectures using distributed Reinforcement Learning. The main limitation of the approach is the

enormous computing power required in order to scale and granulate the approach. We can identify the following directions that can be explored in order to further reduce the computational requirements of the approach:

- Further reduce the resources needed to evaluate an architecture. By reducing the complexity of the dataset, while retaining its statistical properties, the computational resources needed to evaluate the architecture can be reduced.
- Improve the Reinforcement Learning algorithms used in the search. Algorithms that converge faster, or make better use of the available information can solve the problem more quickly, thus reducing the episodes needed.
- Utilize past information more efficiently. Being on-policy, A2C cannot learn by using information gathered with another policy. By using other, more sample efficient algorithms, less information is needed to solve the problem.
- Increase the available computational power. Being natively distributed, by adding more, or upgrading existing machines, the actual time needed to design an optimal network can be reduced, thus enabling researchers to increase the action space complexity, while solving the problem in reasonable time.
- Transfer learning. By training the agent on datasets of incremental complexity, but of the same nature (image classification, natural language processing etc.) it may be possible to train a domain-specific agent instead of a dataset-specific agent, thus reducing the episodes needed for each dataset.

References

[1] A. Krizhevsky, I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu and G. Zweig, "Achieving human parity in conversational speech recognition," *arXiv preprint arXiv:1610.05256*, 2016.

[4] L. A. Gatys, A. S. Ecker and M. Bethge, "Image Style Transfer Using Convolutional Neural Networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[5] K. He, Y. Wang and J. Hopcroft, "A powerful generative model using random weights for the deep image representation," in *Advances in Neural Information Processing Systems*, 2016.

[6] H. Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *CoRR*, vol. abs/1509.06461, 2015.

[7] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[8] A. Krizhevsky, V. Nair and G. Hinton, "CIFAR-10 (Canadian Institute for Advanced Research)".

[9] T. Young, D. Hazarika, S. Poria and E. Cambria, "Recent trends in deep learning based natural language processing," *arXiv preprint arXiv:1708.02709*, 2017.

[10] S. Amarjyoti, "Deep reinforcement learning for robotic manipulation-the state of the art," *arXiv preprint arXiv:1701.08878*, 2017.

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton and others, "Mastering the game of go without human knowledge," *Nature*, vol. 550, p. 354, 2017.

[12] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction, vol. 1, MIT press Cambridge, 1998.

[13] W. G. Macready and D. H. Wolpert, "Bandit problems and the exploration/exploitation tradeoff," *IEEE Transactions on evolutionary computation,* vol. 2, pp. 2-22, 1998.

[14] M. Sniedovich, "Dijkstra's algorithm revisited: the dynamic programming connexion," *Control and cybernetics,* vol. 35, pp. 599-620, 2006.

[15] G. A. Rummery and M. Niranjan, On-line Q-learning using connectionist systems, vol. 37, University of Cambridge, Department of Engineering, 1994.

[16] K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition,* 2016.

[17] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10),* 2010.

[18] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research,* vol. 15, pp. 1929-1958, 2014.

[19] T. G. Dietterich and others, "Ensemble methods in machine learning," *Multiple classifier systems,* vol. 1857, pp. 1-15, 2000.

[20] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation,* vol. 1, pp. 541-551, 1989.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski and others, "Human-level control through deep reinforcement learning," *Nature,* vol. 518, pp. 529-533, 2015.

[22] H. V. Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel and A. Culotta, Eds., Curran Associates, Inc., 2010, pp. 2613-2621.

[23] A. M. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh and A. Y. Ng, "On Random Weights and Unsupervised Feature Learning.," in *ICML*, 2011.

[24] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation,* vol. 10, pp. 99-127, 6 2002.

[25] H. Liu, K. Simonyan, O. Vinyals, C. Fernando and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *arXiv preprint arXiv:1711.00436,* 2017.

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

[27] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning,* vol. 8, pp. 229-256, 5 1992.

[28] B. Baker, O. Gupta, N. Naik and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167,* 2016.

[29] F. Allen, G. Almasi, W. Andreoni, D. Beece, B. J. Berne, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, P. Coteus, P. Crumley, A. Curioni, M. Denneau, W. Donath, M. Eleftheriou, B. Flitch, B. Fleischer, C. J. Georgiou, R. Germain, M. Giampapa, D. Gresh, M. Gupta, R. Haring, H. Ho, P. Hochschild, S. Hummel, T. Jonas, D. Lieber, G. Martyna, K. Maturu, J. Moreira, D. Newns, M. Newton, R. Philhower, T. Picunko, J. Pitera, M. Pitman, R. Rand, A. Royyuru, V. Salapura, A. Sanomiya, R. Shah, Y. Sham, S. Singh, M. Snir, F. Suits, R. Swetz, W. C. Swope, N. Vishnumurthy, T. J. C. Ward, H. Warren and R. Zhou, "Blue Gene: A vision for protein science using a petaflop supercomputer," *{IBM} Systems Journal,* vol. 40, pp. 310-327, 2001.

[30] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM,* vol. 21, pp. 63-72, 1978.

[31] "Top500.org," 1 11 2017. [Online]. Available: https://www.top500.org/. [Accessed 5 1 2018].

[32] M. J. Quinn, "Parallel Programming," *TMH CSE,* vol. 526, 2003.

[33] J. Dongarra and others, "MPI: A Message-Passing Interface Standard, Version 3.0," *High Performance Computing Center Stuttgart (HLRS),* 2013.

[34] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang and V. Volkov, "Parallel Computing Experiences with CUDA," *{IEEE} Micro,* vol. 28, pp. 13-27, 7 2008.

[35] J. Schulman, P. Moritz, S. Levine, M. Jordan and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438,* 2015.

[36] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *ICML 2016,* 4 2 2016.

[37] OpenAI, *OpenAI Baselines: ACKTR & A2C,* OpenAI Blog, 2017.

[38] F. Chollet and others, *Keras,* GitHub, 2015.

[39] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281-305, 2012.

[40] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu and X. Zheng, "TensorFlow: A System for Large-scale Machine Learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, 2016.

[41] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 22 12 2014.

[42] J. Chung, C. Gulcehre, K. Cho and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," 11 12 2014.

[43] H. Abdi, "The Kendall rank correlation coefficient," *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pp. 508-510, 2007.