[entry]nyt/global/

UKRAINIAN CATHOLIC UNIVERSITY

MASTER THESIS

Neural architecture search: a probabilistic approach

Author:
Volodymyr LUT

Supervisor:
Yuriy KHOMA
Vasilii GANISHEV

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2020

Declaration of Authorship

"It's inspiring to see how AI is starting to bear fruit that people can actually taste. There is still a long way to go before we are truly an AI-first world, but the more we can work to democratize access to the technology—both in terms of the tools people can use and the way we apply it—the sooner everyone will benefit."

Sundar Pichai, CEO Alphabet Inc., May 17, 2017

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Master of Science

Neural architecture search: a probabilistic approach

by Volodymyr LUT

Abstract

In this project we introduce Gaussian Process (GP) Bayesian Optimization (BO) implementation of NAS algorithm that is exploiting patterns found in most optimal unique architectures sampled from the most popular NAS dataset and benchmarking tool **NASbench-101** (**nas102**). Proposed solution leverages novel approach to path-encoding and is designed to perform reproducible search even on relatively small initial batch obtained from random search. This implementation does not require any special hardware, it is publicly available. In AppendixA we share all details needed to fully reproduce our results.

Acknowledgements

First of all, I would like to thank my supervisor Mr. Yuriy Khoma and Mr. Vasilii Ganishev for all their support, patience and knowledge shared with me during this project.

I would also like to thank all my colleagues in UCU - those who already graduated, who were working on their masters during this year, and those who left UCU - for being a bright lighthouse, which navigated me towards excellence, for showing a good example and for being a strong and supportive community.

Many thanks to the UCU team - to all those people, who build and design the educational system I am proud being a part of, to all those who inspired me, shared their vision and made my time being a masters program student an unforgettable experience. I thank everyone who works every day in the UCU - no matter whether they are responsible for brilliant coffee or brilliant courses and workshops. Special cheer outs to Mr. Artem Chernodub for interesting homework and deep learning course at all. It helped me a lot during writing about ConvNets in this master's thesis.

Special thanks go to my friend and business partner Serhii Chepkyi for providing big support during my studies and also for providing a big help with visual materials for this project.

Last but definitely not least - I would like to thank my closest people and my family - without your eternal love, this would not be possible. This was the hardest years in my life and I'm glad and thankful that every second I've felt that I'm not alone.

Contents

List of Figures

List of Tables

List of Abbreviations

ML	**M**achine **L**earning
AutoML	**Auto**mated **M**achine **L**earning
NAS	**N**eural **A**rchitecture **S**earch
RL	**R**einforcement **L**earning
MDP	**M**arkov **D**ecission **P**rocess
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
UCB	**U**pper **C**onfidence **B**ound
MLE	**M**aximum **L**ikelyhood **E**stimate

For all the brave people who make it possible for millions of young Ukrainians to hold books in their hands instead of rifles and grenades.

Chapter 1

Introduction

As machine learning provides a huge variety of automation possibilities for different industries the problem of automation of ML industry itself seems natural. For decades ML engineers were pioneers in the new era of computer science research. As a result, the new industry was shaped and this industry requires automation.

AutoML is a general name of automation in routine work of ML engineers including but not limited to data preparation, feature engineering, feature extraction, neural architecture search, hyperparameters selection, etc.

ML is reshaping businesses and other aspects of everyday life worldwide. We believe that everyone would benefit from the democratization of these new tools. Having the ability to run models on portable devices, IoT chips, and other mass-market hardware we treat AutoML as a big move towards in terms of a variety of different applications created.

Existing AutoML techniques require lots of computational resources and most of the research in the field is covered by tech giants nowadays.

However, since pioneering work by of A new area of research, neural architecture search (NAS), seeks to automate this process.

We are focusing on neural architecture search problems, especially on hyperparameter optimization tasks because historically this problem is solved mainly using exhaustive search techniques, such as grid search. Engineers often follow their empirical knowledge and try to guess optimal parameters to tune models.

We are using the reinforcement learning paradigm since it is performing well in solving NAS problems. RL agents can design better architectures than related hand-designed models in terms of error-rate and efficiency - **ZophL16**.

Moreover, we believe that RL could benefit from probabilistic approaches. We are deeply inspired by DeepAR **2017arXiv170404110S** used by Amazon to build forecasting models. We show that Gaussian probability distribution could be used to effectively balance the exploration and exploitation of RL agents solving NAS tasks.

Chapter 2

Background overview

2.1 History

The idea of using RL agents to build neural networks is not new, however, there are not so many research projects nowadays. Mostly the reason for this is that most of the research is held by the business, and business usually is not optimistic about RL in production.

However, some good progress was made in recent years. In 2015, ResNet becomes a winner of ILSVRC 2015 in image classification, detection, and localization and winner of MS COCO 2015 detection and segmentation. This enormous network contained 152 layers optimized by a lot of professional engineers manually. This process is expensive in terms of time and resources. Image classification contests are constantly showing a growing amount of layers for best-performing networks (AlexNet, 2012 - 8 layers, GoogleNet, 2014 - 22 layers). Resnet has 1.7 million parameters. Each competition is turning researchers more and more towards automation of this work - and this is a place where NAS becomes a new trend.

Barret Zoph and Quoc Le. in **ZophL16** used a recurrent network to generate the model descriptions of neural networks and train this RNN via RL agent to maximize the expected accuracy of the generated architectures on a validation set. This paper is one the most cited in this field and our research is heavily based on it.

In 2019, Google researchers developed a family of models, called EfficientNets, which surpass state-of-the-art accuracy with up to 10x better efficiency (smaller and faster) using AutoML - see **2019arXiv190511946T**.

Amazon has two AutoML products to offer - Amazon SageMaker Autopilot for the creation of the classification and regression machine learning models and Amazon DeepAR for forecasting scalar (one-dimensional) time series using RNN. This paper is also heavily based on the probabilistic approach used in DeepAR because of it's spectacular results (see 2.2).

This section would not be full without the paper which shares a lot in common with this project. RL agent (Q-Learning, epsilon-greedy exploration rate control,

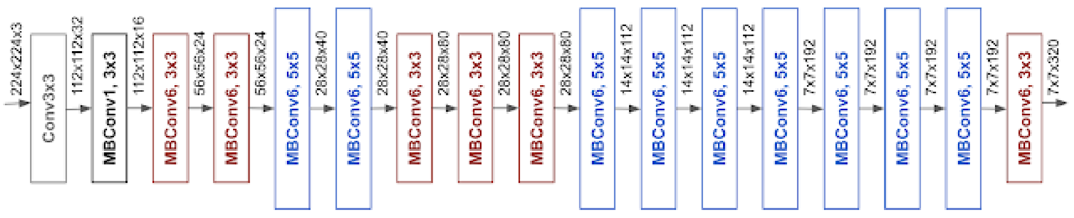

FIGURE 2.1: The architecture for EfficientNet's baseline network EfficientNet-B0 from **2019arXiv190511946T**

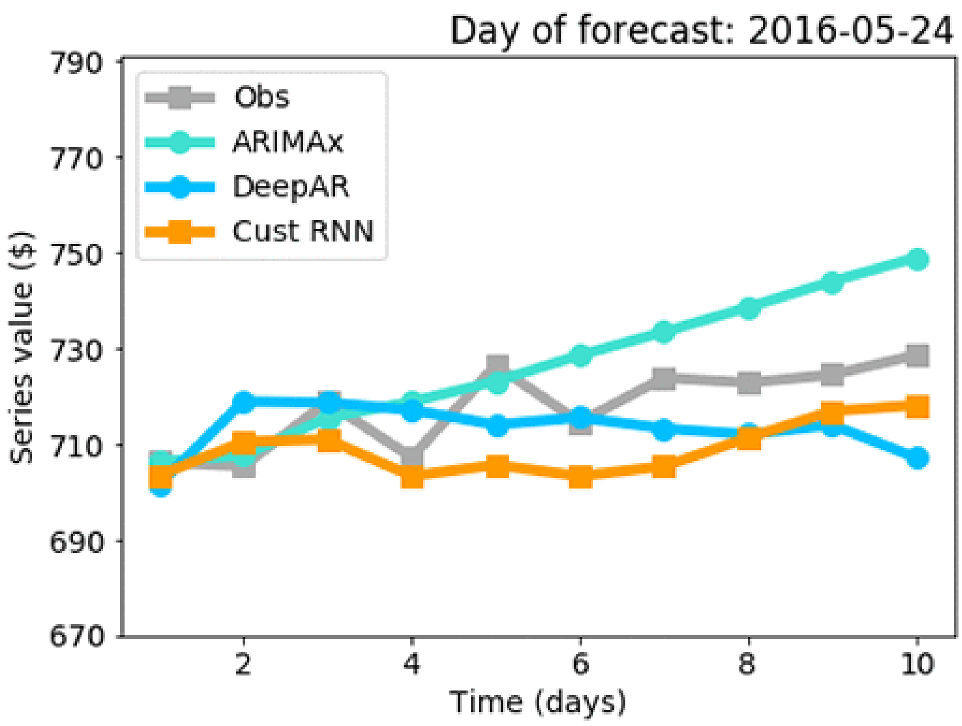

FIGURE 2.2: Benchmark analysis: Visualization of forecasts produced by the RNN combo, ARIMAx and DeepAR together with what was actually observed for the series A values

finite state space) described in the paper outperformed meta-modeling approaches for network design on image classification tasks **Baker2016DesigningNN**.

The interest to the topic becomes even hotter when the NAS benchmark and dataset was introduced by Google Research team **pmlr-v97-ying19a**

A variety of methods have been proposed to perform NAS, including reinforcement learning, Bayesian optimization with a Gaussian process model, sequential model-based optimization (SMAC), evolutionary search, and gradient descent over the past few years. We see a lot of research potential in this field and we share a big passion for RL paradigm - and that's why this project exists.

2.2 Reinforcement Learning

RL is a machine learning paradigm often used when the exact mathematical model is unknown and data is unlabeled. In this project, we would mainly concentrate on the Q Learning approach. We require having an observable environment where software agent would be able to take actions which would lead to reward. An agent is designed in the way it should maximize reward by utilizing existing knowledge (exploitation) or exploring random actions (exploration). The algorithm needs to learn a policy that determines which action should be taken next given current state (or set of recent states). The environment agent is operating with is typically a Markov Decision Process.

2.2.1 Markov Decision Process

MDP is a generalization of the mathematical framework which allows performing a sequence of actions in an environment where future states would depend on current states and yield partly random outcomes.

It is described by:

- State (or states) $\mathcal{S}$ that are fully observable by agent
- Set of actions $\mathcal{A}$ which agent could take in a given state
- Transition model $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longmapsto [0, 1]$ - an effect in the state accused by action taken by the agent
- A reward function R which defines a reward received by an agent for taking this action
- A discount factor $\gamma \in [0, 1)$ which is used to value a reward that could be received in future
- A policy $\pi : \mathcal{S} \times \mathcal{A} \longmapsto [0, 1]$ agent should learn

A policy that allows the agent to maximize its reward is called a solution of given MDP. Notation of MDP used above is taken from **Thomas15a**.

MDP that we will refer to later in this thesis, would have a Markov property because the reward received from action taken in the current state is not depended on previous states nor previous actions. In other words, we are talking about the Markov chain in this thesis.

2.2.2 Exploration and exploitation dilemma

To demonstrate exploration over exploitation problem we usually often refer to Multiarmed Bandit Problem.

The multiarmed bandit problem is MDP with stochastic reward. Imagine having N machines with reward probabilities $R_1..R_n$. Playing k-th bandit could result in the reward of 1 with a probability of R_k or in reward 0 with a probability of $1 - R_k$. Real reward probabilities are unknown to the agent. Also, resources are limited, so, exploring them using the law of large numbers is inefficient and impossible in terms of this problem.

The goal of the agent playing multiarmed bandits is the maximization of cumulative reward.

Simply said, while playing a known action (performing exploitation) agent may miss a better option. This becomes a loss function of such algorithms - a regret player might have due to not selecting the optimal action. If the agent always plays random actions (exploration) it will be completely useless and would yield no good results with limited resources given. In other words, to balance exploration and exploitation problem we would need to ensure that the agent is not overfitting to known actions but still uses gained knowledge. It often happens, that RL agents observes some **good enough** action and starts to use it constantly. This is not the way we generally want an RL algorithm to behave.

2.2.3 Epsilon-greedy approach

Epsilon-greedy approach to solving exploration over exploitation dilemma is based on the idea, where an agent would generally take best actions, but at some time iterations t, it would explore random actions. It often assumes that at the beginning of the training number of random actions would be bigger (to ensure that agent would be able to explore good actions at the very beginning) and later it would decay (to ensure that agent is actually playing best actions). An expected reward is often represented as a mean of previously received rewards when playing this action.

2.2.4 Upper-confidence bound

UCB is another instrument to solve exploration over exploitation dilemma. The general approach is similar to the epsilon-greedy algorithm, however, UCB allows us to use unexplored actions in favor of actions which algorithm is very certain about being bad ones.

UCB measures the potential of action using the upper confidence bound of the reward value in such a way that a larger number of trials of certain action should give smaller bound. This also prevents the algorithm from overfitting.

In this work we would refer to Theorem 1 from **Auer2002** using a simple UCB approach which allows receiving logarithmic regret uniformly and without any preliminary knowledge about the reward distributions.

$$U(a) = Q(a) + \sqrt{\frac{2logn_a}{N}}$$

Where Q(a) is expected reward from action a (generally a mean of rewards received by playing that action), n_a is the number of times this action was played and N is a total number of plays.

2.2.5 Deep Q Learning

While Q learning is an algorithm that allows the agent to learn an effective policy for maximizing cumulative reward in general MDP, deep Q learning is the same, model-free approach that uses Deep Neural Network to approximate the values of expected reward.

In recent years a lot of research is done in this field, even though Deep Q learning is treated as an unstable solution because a slight change in input can change outputs significantly.

However, this approach has a proven success story. This work is inspired by **MnihKSGAWR13** which was one of the first successful approaches to use a deep neural network as a backend for reinforcement learning agent.

In general Deep Q learning algorithms stores experienced rewards in replay memory, from where they are later batched and used as input to the underlying neural network.

This work uses a convolutional neural network as a backend for RL algorithm.

2.3 Image classification problem

The image classification problem is a machine learning problem aiming to recognize a visual concept of a given image and assign some class (label) to it as an output.

A data-driven approach for this problem required to train the algorithm on loads of images in order to understand which features images from one class have in common to other images of this class and different from images of other classes.

Naturally, in the beginning, KNN (K Nearest Neighbors) and other simple clusterization algorithms were used.

However, the real move forward started when backpropagation techniques were applied to neural network architectures, which opened a way for deep learning techniques. As Yann LeCun released LeNet-5 (see **lecun-89c**) modern convolutional architectures started to develop rapidly.

There are several popular datasets such as MNIST [**mnist**], Flowers [**flowers**], ImageNet [**imagenet**] etc. which are constantly used in image classification contests. Modern deep neural networks outperform human-level accuracy on those datasets.

2.4 Convolutional Neural Networks

CNN or ConvNet are neural networks sharing architecture of multi-layer perceptrons. Since multi-layer perceptrons have fully-connected layers, they are expensive in terms of computation resources. CNN shares a different approach - every convolutional neuron processes data only related to its receptive field of small size. Most ConvNets use fully connected layers at the end anyway.

Though CNN is capable of efficiently handling a huge variety of machine learning tasks, the simplest for understanding would be an image recognition example - and this is also related to the CNN architectures generated by the algorithm introduced by this project. One of the most popular ConvNet was LeNet-5. Figure 2.3 shows architecture of LeNet-5.

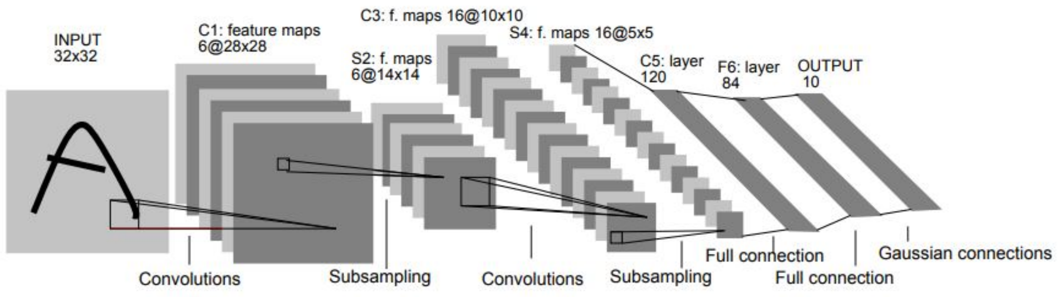

FIGURE 2.3: Original image from **lecun-89c** displaying LeNet-5 architecture

Each neuron in a CNN computes an output applying a function over input values received from the receptive field (a small subset of input) in the previous layer. This function is usually a dot product over a matrix of weights. Bias also could be applied (matrix addition operation).

The learning process is performed by updating weight and bias matrices. Huge progress was gained after the backpropagation technique was introduced for modern deep neural networks [**lecun-98b** - this is not an invention of the technique; it was introduced by lots of people before Yann LeCun, but this paper is a very good explanation of basic principles]. Backpropagation is a way to efficiently compute the gradient of the loss function with respect to the weights of the network.

2.5 Gaussian Distribution and MLE

Gaussian distribution or normal distribution is a continuous probability distribution of a random variable. This distribution is described by its mean and standard deviation.

The density curve is symmetrical and centered above its mean. It's spread is by the standard deviation.

Density curve is defined by following function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}(\frac{x-\mu}{\sigma})^2}$$

We would use the maximum likelihood estimate of the variance of the distribution σ^2 as a loss function of master CNN.

We want to estimate the parameters of distribution by maximizing a likelihood function so that observable data is most probable.

It is convenient to work with a log-likelihood interpretation

Let l be a likelihood function (probability of observing the given data as a function of σ) and f be a density function noted above.

$$l(\sigma, \mu) = \sum_{i=1}^{n} log(f(x_i|\sigma, \mu))$$

Calculation of partial derivatives and setting them to zero gives MLE of the target variable (in our case - σ).

Chapter 3

Proposed approach

Our primary goal is exploring the effectiveness of probability approaches applied in DeepAR [**2017arXiv170404110S**] for a deep RL agent applied for solving NAS problems.

Leaving alone RL, probabilistic approaches seem to be quite effective for solving hyperparameter optimization tasks. Modern Bayesian optimization approaches showed good results in hyperparameter optimization for image classification problems[**thornton2013a**
SnoekLA12].

Rewards received from a different model of the same architecture built with different hyperparameters form a highly nonlinear response surface [**ZeilerF13**]. How could probabilistic approaches affect the performance of RL agent solving hyperparameter optimization tasks? To answer this question, we would build two RL agents which generate slave CNNs, one with a classical master CNN and another with modified CNN with a Gaussian Layer as it's output and compare the way they learn and transfer knowledge.

To calculate the loss function for the algorithm, we will use MLE of variance for Gaussian modification of CNN and Huber loss for the classical one.

3.1 RL agent

To implement RL agent we heavily rely on the pipeline described in **ZophL16**.

Since there is no physical space $\mathcal{S}$ for this MDP, we will need to create an abstraction Manager which is responsible for:

- Generating slave CNN in accordance with actions taken by the agent
- Evaluate slave CNN for a certain number of epochs
- Return reward in terms of CNN accuracy

Since there is no initial state, random action is generated in order to provide proper state transition $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longmapsto [0, 1]$.

RNN was used as a backend for RL algorithm in the original paper [**ZophL16**] but we use simple CNN to make integration of Gaussian Layer easier.

Since RL needs a finite set of actions *aset* we are limiting hyperparameters options to a set of predefined values.

Each epoch RL agent can either explore random action or predict one using backend CNN. After the reward would be returned, state transition should be stored in the replay memory. Random batches of those transitions should be taken and used as input to update master (backend) CNN (see 3.1).

Since there is no actual next state for the controller, transition objects would contain the previous state, action taken and reward received.

Using the accuracy of CNN directly as a reward would lead to suboptimal policies learned by RL agent. In order to avoid it, we would use a trick called Exponentially Weighted Moving Average. Fun fact: this kind of problem (choosing a correct representation of reward for RL agent) is also a hyperparameter optimization problem and other RL agents are used to optimize it - see **rlhyperparamrl**.

Since master CNN actually performs a regression task, we also use a converter that converts its outputs into actions that should be taken next. Obviously, the output layer could be modified to make this CNN solve a classification problem, but this trick was used in order to appropriately evaluate outputs of Gaussian Layer.

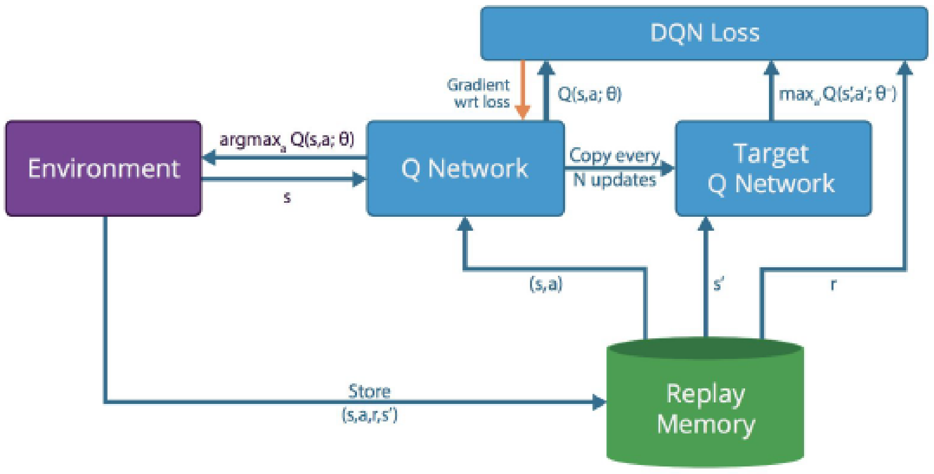

FIGURE 3.1: General overview of DQN RL architecture [original from **Nair**]

3.1.1 Exploration and exploitation

Since the action space is limited, during experiments we've observed a tendency of the algorithm to overfit. In order to resolve this, we use two different ways to handle exploration over exploitation dilemma:

- Decaying random effect occurrence in epsilon-greedy approach
- Applying UCB to received reward

Both methods were evaluated during experiments.

3.2 Master CNN

Master CNN is used as a backend for the RL agent. Since neural networks are universal function approximators, we can use one to make a prediction of the reward received for taking an action.

This CNN takes in a batch of transition objects. Its outputs are of the size of action space. In effect, the network is trying to predict the expected return of taking each action given the batch of transitions in the classic approach.

In the updated approach this network outputs a μ and σ of Gaussian distribution of expected return of taking each action.

3.2.1 Gaussian layer

When it comes to RL algorithms, scoring becomes very important and may have a big impact on action taken by the controller - **tilmann**

If master CNN in RL agent is solving a regression problem (which is exactly our case) this CNN would use backpropagation to update its weights (as noted above) in a way that error metrics of a test set would be minimized. Originally outputs of the last layer would be real values - in our case, values that determine the actions that would be taken by the controller.

Those values obviously depend on input and weights. In order to receive a Gaussian distribution, we would need to modify the last layer so that it would return the mean and variance of the output variable, which is enough to describe a Gaussian distribution of this variable. This allows us to bring a prediction uncertainty into the outputs of CNN. As described in **2017arXiv170404110S** this requires also another approach to the computation of loss function.

We are using Gaussian distribution instead of single values because we need to have a measure of uncertainty of prediction of output. The underlying idea is that RL agent should benefit from playing actions with lower uncertainty.

For these needs, any other distribution could be obviously used. DeepAR [**2017arXiv170404110S**] allows users to choose Gaussian, Student T and beta distributions. We've chosen Gaussian distribution for the needs of this project, but any other distribution of real-valued variables could be used with an appropriate likelihood loss function.

3.3 Slave CNN

Slave CNN is a CNN generated by the controller. It could be of any architecture, however, in experiments, the number of hidden layers was restricted to 4 hidden convolutional layers and one GlobalAveragePooling layer.

Here is code listing of slave CNN Model implemented in TensorFlow:

```
ip = Input(shape=(28, 28))
x = Conv2D(filters_1, (kernel_1, kernel_1), activation='relu')(ip)
x = Conv2D(filters_2, (kernel_2, kernel_2), activation='relu')(x)
x = Conv2D(filters_3, (kernel_3, kernel_3), activation='relu')(x)
x = Conv2D(filters_4, (kernel_4, kernel_4), activation='relu')(x)
x = Dense(100, activation='softmax')(x)
model = Model(ip, x)
```

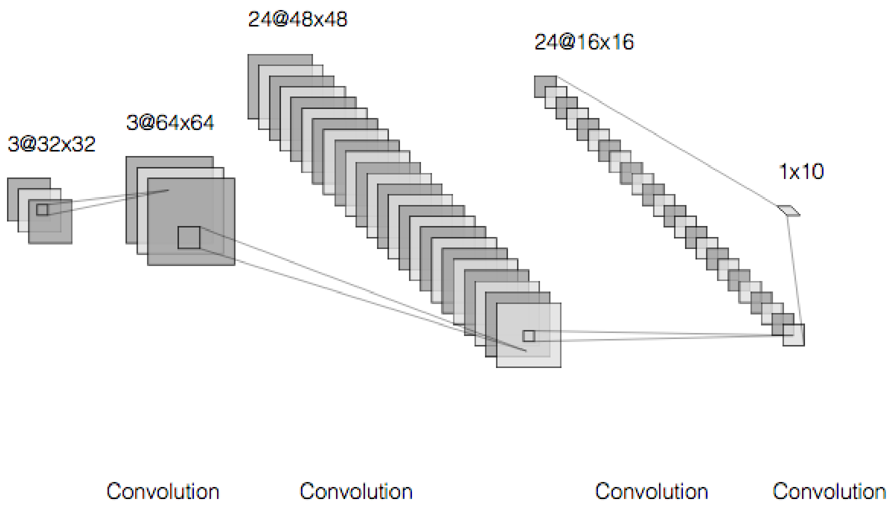

FIGURE 3.2: Example of slave CNN architecture that could be generated by RL agent for CIFAR10 dataset

Chapter 4

Experiments

4.1 Datasets

Our RL agent is generating CNN architectures that are performing an image classification task on the CIFAR-10 and CIFAR-100 datasets.

CIFAR datasets are described in very deep detail in Chapter 3 of **CIFAR**, especially details about its collection.

I would not go into details, just will note that CIFAR is a set of 32×32 color images depicting real-world objects.

After training the RL algorithm on CIFAR10 it is heavily switched to use its knowledge on CIFAR100 to provide us with an idea about how this algorithm would handle knowledge transferring during the live experiment.

Key differences between CIFAR10 and CIFAR100 is denoted in table 4.1.

Dataset	Size	Number of classes	Images in class
CIFAR10	60000	10	6000
CIFAR100	60000	100 grouped into 20 superclasses	600

TABLE 4.1: Comparison of CIFAR10 and CIFAR100 datasets

Both in CIFAR10 classes are exclusive and do not assume instances overlapping - see 4.1.

We use CIFAR datasets as is, meaning that we also share the same train/test dataset split for evaluating generated CNN architectures. There are 50000 training images and 10000 test images in the dataset.

4.2 Metrics

Because of the nature of the problem that we are solving one metric would be in common for any approach described: it's a generated CNN accuracy. However, to see more details of the RL agent learning, we will use also additional metrics since we are handling two parallel experiments and since the Gaussian layer of a master CNN provides a distribution which gives us some additional knowledge.

4.3 Environment and training

Our models are implemented using Pytorch [**NEURIPS2019_9015**] (master CNN) and Tensorflow [**tensorflow2015-whitepaper**](slave CNN). There was no particular need to develop them using different frameworks, it's a historical issue.

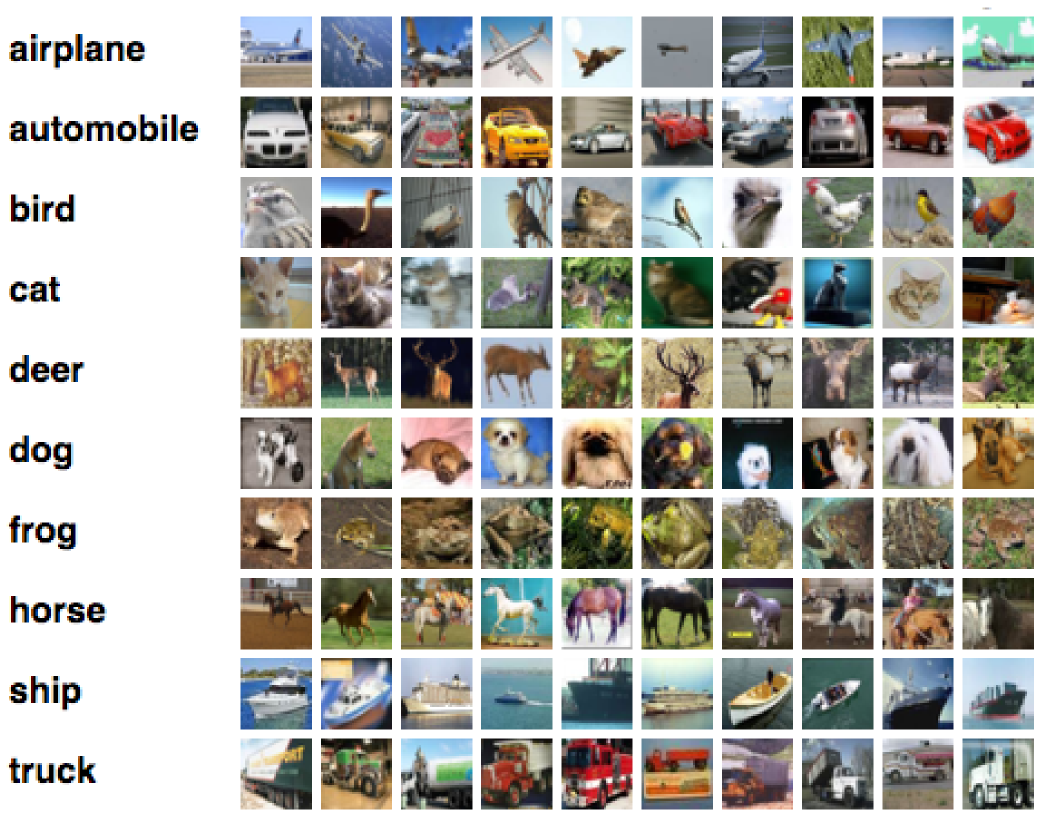

FIGURE 4.1: Classes of CIFAR10 including 10 random images from each **CIFAR**

Algorithm	Loss Function	Additional metrics
Classic CNN	Huber loss	6000
CNN with Gaussian Layer	MLE of variance	μ and σ

TABLE 4.2: Additional metrics of RL algorithm

RL agent was trained on the GPU resources provided by Google Colaboratory [**colab**] service.

Each RL agent was trained for 80 epochs on the first dataset (CIFAR10) and then hard-switched to CIFAR100 (leaving weights of the backend CNN, but clearing other metadata).

First 10 epoch there were 9 random actions explored. This number decayed by 1 every 10 epochs until int finally will become 1 random action per 10 epochs.

RL agent was choosing 4 kernel size and filter sizes tuples from available sizes. Types of slave CNN hidden layers were predefined.

Action type	Values available in experiment
Filter size	1, 3, 6, 9, 12, 24
Filter size	2, 4, 8, 16, 32, 64

TABLE 4.3: Action space description

Each slave CNN was trained for 8 epochs.

4.4 Results

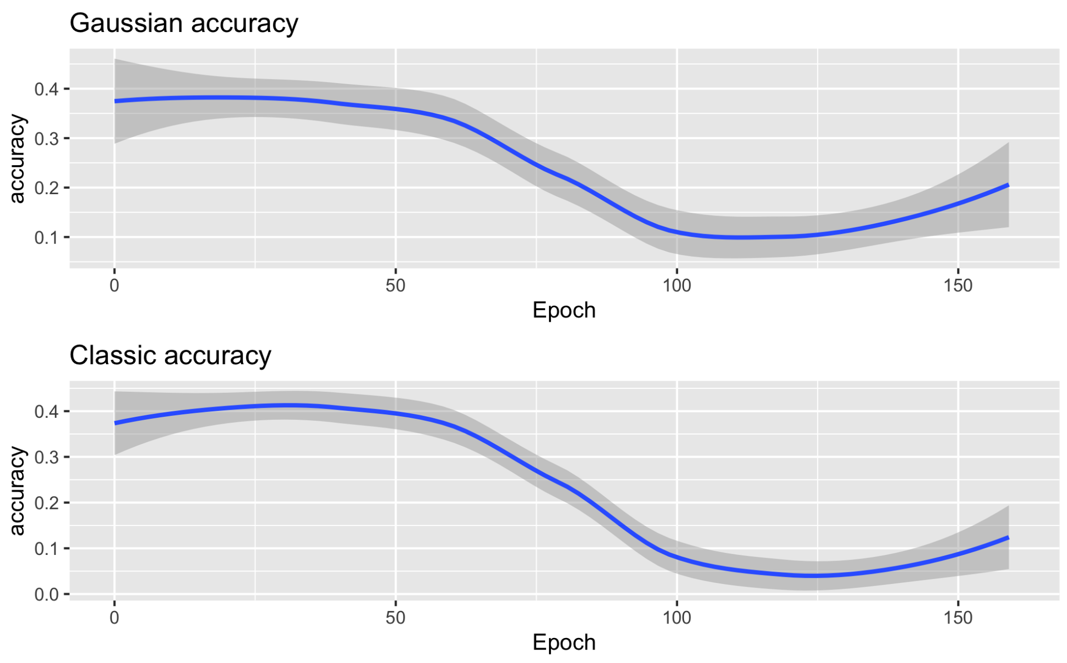

FIGURE 4.2: Accuracy of slave CNN generated via epsilon-greedy approach

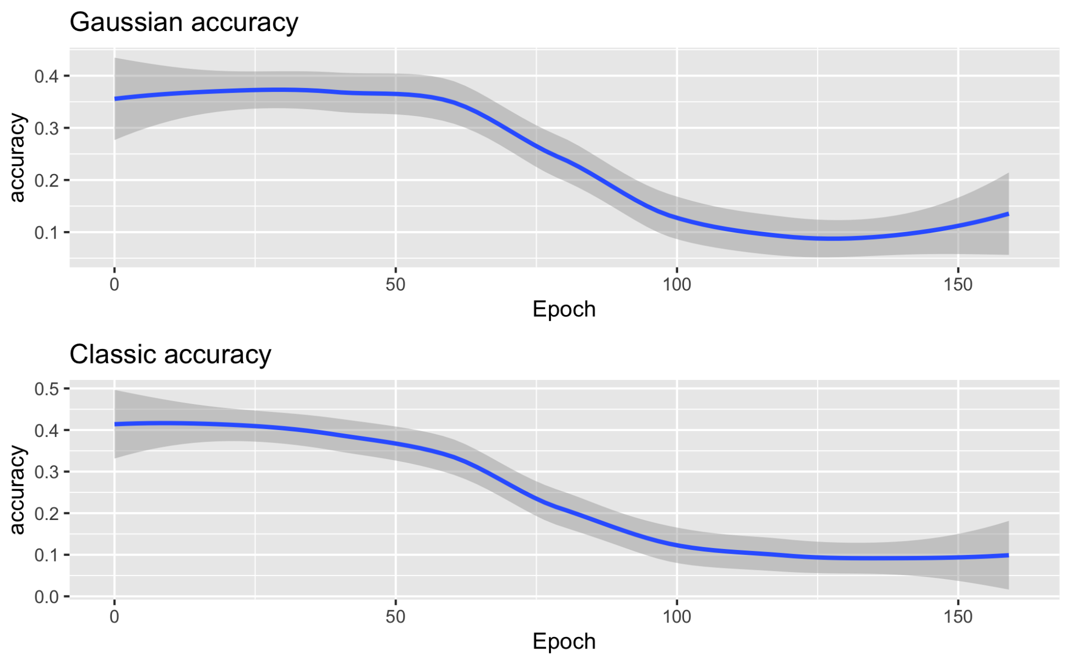

FIGURE 4.3: Accuracy of slave CNN generated via upper confidence bound approach

Even though that chart looks similar to each other there are few things to note here:

- UCB does not provide any benefits to the algorithm in this configuration and environment setup
- During numerous sets of experiments we've seen that Gaussian modification yields better CNN architectures while handling knowledge transfer problems.

Even though yielding 0.2 accuracies after 8 epochs of training on CIFAR100 is a good result, it is interesting to see how these agents was actually training. Since the goal of RL agent is the maximization of cumulative reward it would be beneficial to see how this reward was changing over time (see 4.4).

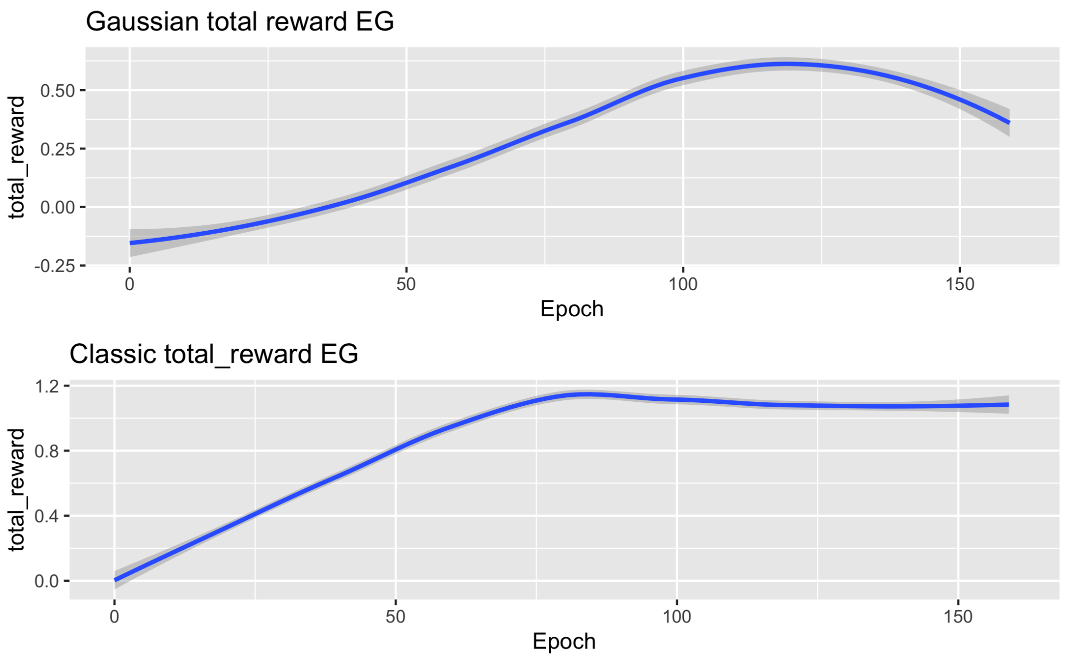

FIGURE 4.4: Total reward of both RL controllers compared

Since RL agent receives moving accuracy from the controller both of those lines will stop increasing after the dataset would changes. Obviously, results received on CIFAR100 would be much lower than on CIFAR10. That's why we should see those lines moving down after the dataset was changed if the RL algorithm is still training. This is the reason why the classic algorithm performed twice as low as Gaussian one after the dataset was hard-changed.

To estimate how the agent learns, we must see how it exploits it's knowledge (see 4.5). Blue lines indicate an action that was already generated or explored by the controller at some point in time before. It's interesting to see how both algorithms react to the change in their MDP state because of the database change.

Finally let's see how μ and σ distribution changed during the period of training (see 4.6).

Notice how the standard deviation of the distribution becomes bigger as the underlying CNN train and then reduces again when the dataset was updated. This would be more interesting to explore on a larger time scale, however, even on the 80-epoch experiment Gaussian approach shows it's strong ability to generate better architectures exploiting existing knowledge. It also seems to avoid a lot of overfitting problems.

However, during this experiments some issues were found:

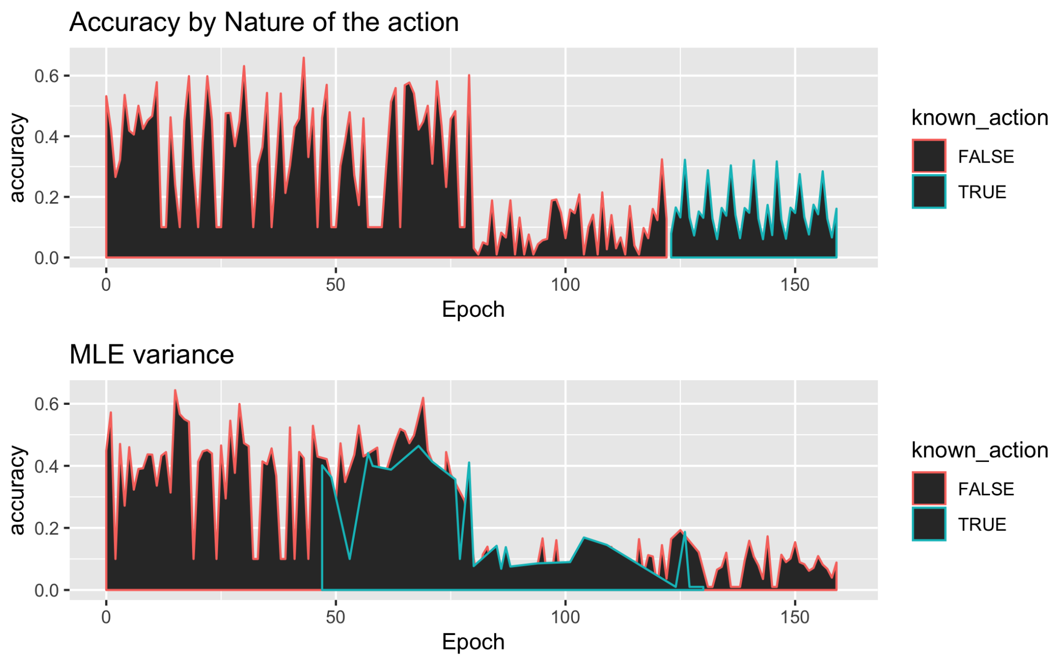

FIGURE 4.5: Distribution of knowledge exploitation during the training

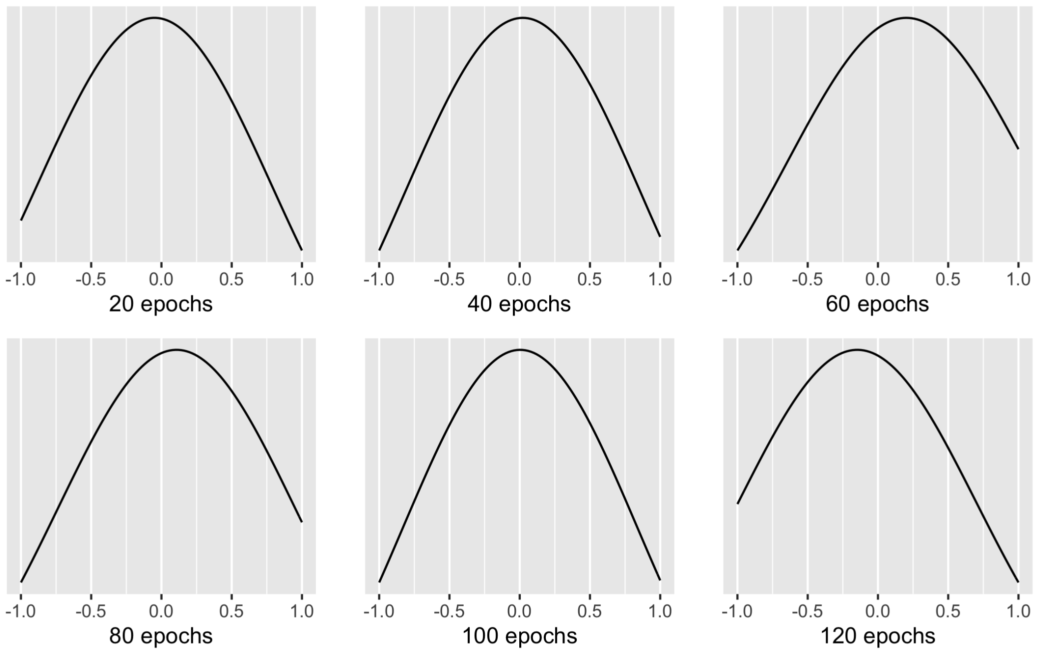

FIGURE 4.6: Gaussian distributions received on the last layer of CNN

- RL algorithms are not very stable and occasionally they can yield bad results as seen at 4.5
- Training of those algorithms in a way proposed is resource-ineffective and in any other research should be done using NASbench **pmlr-v97-ying19a**
- Even though gaussian modification yields better results it hasn't prevented algorithm from overfitting at small action space.

Algorithm	CIFAR10 avg last 10 epoch	CIFAR100 avg last 10 epoch
Gaussian epsilon-greedy	0.3807	0.1588
Classic epsilon-greedy	0.3308	0.0846
Gaussian UCB	0.3875	0.1078
Classic UCB	0.3737	0.0799

TABLE 4.4: Accuracy achieved by slave CNN

Some of them could be resolved by tuning the RL agent or master CNN, but some require additional research.

Chapter 5

Conclusion

In this work, we were researching possibilities of effective application of probabilistic approaches described at DeepAR [**2017arXiv170404110S**] in a field of AutoML using DQN. As a result of the research, we've come with a solution that requires the implementation of a Gaussian Layer to master CNN and usage of MLE of variance as a loss function. A proposed solution was validated on a set of experiments showing results compared to the original algorithm and outperformed it during transfer learning validation.

The solution is available on the GitHub [**github**]. Note that this code is heavily inspired and build on top of this two projects [**nasgithub**; **rltutorial**].

We hope that this example will show that AutoML research using RL is already a thing and could be held by other students and researches even on an occasion of low resources available. We hope to see more and more research in the filed in the upcoming years.

A few days before I was writing this conclusion NAS-Bench-102 was published. It introduced different search space, results on multiple datasets, and more diagnostic information [**nas102**]. As the community is building up this ecosystem we have no doubt NAS would become a problem solved during challenges, and, what's more important, more reproducible. This, however, opens new challenges for us, and one of the biggest ones would be to move existing codebase of the project to NASbench.

Appendix A

A.1 Reproducibility and random seed

Something on the topic