Neural Architecture Meta-learning via Reinforcement

Undergraduate Thesis - End-Semester Report

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Sagnik Majumder
ID No. 2014A8TS0464P

Under the supervision of:

Prof. Dr. Visvanathan Ramesh
&
Prof. Dr. Surekha Bhanot

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
October 2018

Declaration of Authorship

I, Sagnik Majumder, declare that this Undergraduate Thesis - End-Semester Report titled, 'Neural Architecture Meta-learning via Reinforcement' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ***Sagnik Majumder***

Date: ***May 3, 2018***

Certificate

This is to certify that the thesis entitled, "*Neural Architecture Meta-learning via Reinforcement*" and submitted by Sagnik Majumder ID No. 2014A8TS0464P in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor	*Co-Supervisor*
Prof. Dr. Visvanathan Ramesh	Prof. Dr. Surekha Bhanot
W3 Professor,	Professor,
Goethe University, Frankfurt	BITS-Pilani Pilani Campus
Date:	Date:

"Success is going from failure to failure without losing your enthusiasm"

Winston Churchill

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Bachelor of Engineering (Hons.) Electronics and Instrumentation

Neural Architecture Meta-learning via Reinforcement

by Sagnik Majumder

Neural architecture search is a very rapidly growing domain with the objectives being the mitigation of the problems that come along with hand-engineering neural network architectures and the production of very good performance in different machine learning applications. Use of reinforcement learning, in particular, for architecture search has gained widespread popularity in recent times owing to the ability to generate highly efficient models for certain types of machine learning tasks. First, we study in detail the reinforcement learning algorithms for architecture search, their applicability, their merits and demerits on supervised image classification and unsupervised image reconstruction tasks. It is shown that the reward for architecture search on these two tasks can be defined as the inverse of the loss metric during validation and an architecture search with this reward produces top architectures which are also the best architectures the search searches over. The architecture search using reinforcement learning works on supervised image classification and unsupervised image reconstruction tasks because the search reward in the form of the inverse of the loss metric, acts as a heuristic that reflects the performance of the candidate architectures with respect to the particular task. But, the most important aspect of this work is to analyze the applicability and the performance of reinforcement learning based architecture search techniques for the task of completely unsupervised image generation using variational autoencoders [29]. A successful linking of architecture search with variational autoencoders necessitates defining a joint formulation of loss functions that train the variational autoencoder on the task of image generation, and the formulation of a reward that reflects the image generation potential of the architecture, from the loss metric for the search. In this case, a new loss formulation is proposed that can be optimized well for the individual variational autoencoder models (candidate models in the search). But, the methods used in this work to formulate the reward from the loss metric do not work i.e. the top architectures from the search are not the best architectures the search searches over, from the point of view of the visual fidelity. This shows that not only the loss function to be optimized, is task dependent but also, the search reward depends on the task to a certain extent, and the formulation of both is very critical for the reinforcement learning based architecture search to yield results.

Acknowledgements

I would like to extend my gratitude to Prof. Visvanthan Ramesh for giving me this opportunity to work under his supervision. I would like to thank him and Mr. Martin Mundt, a PhD scholar at the Center for Cognition and Computation, for their time and devotion to support me through this thesis. It would be very difficult to reach this stage of the thesis without their advice and motivation. I will always remain indebted to them for this invaluable exposure.

I would also like to thank my co-supervisor, Prof. Surekha Bhanot, for her support and mentorship without which it would have been difficult to accomplish this stage of the project. Besides, I am grateful to my labmates Andres, Martin, Sreenivas, Sumit, Tobias, Julia, Diana, Christian, Timm and Dennis for their friendship and support.

I thank the Electrical and Electronics Engineering Department and the Academic Research Division at BITS Pilani for allowing me to write my off-campus thesis at Goethe University in Frankfurt, Germany. I express my heartfelt gratitude to Goethe University and its administration for having me over in Frankfurt for the thesis.

Last but not the least, I would like to thank my family, especially my parents for their love and support. This work would not have been possible without their help and belief in me.

Contents

List of Figures

List of Tables

Abbreviations

RL	**R**einforcement **L**earning
ML	**M**achine **L**earning
SMBO	**S**equential **M**odel **B**ased **O**ptimization
GP	**G**aussian **P**ocess
TPE	**T**ree of **P**arzen **E**stimators
PPO	**P**roximal **P**olicy **O**ptimization
MDP	**M**arkov **D**ecision **P**rocess
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort **T**erm **M**emory
GPU	**G**raphics **P**rocessing **U**nit
NAS	**N**eural **A**chitecture **S**earch
ENAS	**E**fficient **N**eural **A**chitecture **S**earch
DAG	**D**irected **A**cyclic **G**raph
AI	**A**rtificial **I**ntelligence
NLL	**N**egative **L**og **L**ikelihood
FC	**F**ully **C**onnected
SPP	**S**patial **P**yramidal **P**ooling
SM	**S**oft **M**ax
CAE	**C**onvolutional **A**uto **E**ncoder
BCE	**B**inary **C**ross **E**ntropy
MSE	**M**ean **S**quared **E**rror
t-SNE	**t**-distributed **S**tochastic **N**eighbor **E**mbedding
VAE	**V**ariational **A**uto **E**ncoder
GAN	**G**enerative **A**dversarial **N**etwork

KL	**K**ullback **L**eibler
MC	**M**arkov **C**hain
i.i.d	**I**ndependent and **I**dentically **D**istributed
EM	**E**xpectation **M**aximization
ELBO	**E**vidence **L**ower **B**ound

Dedicated to Maa, Baba and Indumati.

Chapter 1

Introduction

Deep neural networks have shown a lot of promise in different types of tasks which are both very challenging and very useful for humanity [2, 31, 59]. They have shown their strong capability in automatically learning good, representative and class discriminating feature maps directly from raw input. But designing such deep neural architectures has often involved a lot of experts' knowledge and hand-engineering [20, 60, 68, 69, 70, 77] right from the day of inception of deep neural networks. Such hand-engineering is extremely task-specific and time-consuming in addition to demanding a lot of domain knowledge and experience. In this report, the structure/configuration of the network which includes the types of layers, the number and the size of the filters, the different non-linearities in the network, among others is being referred to as the architecture of the network. The terms architecture and network will be used interchangeably in the same sense through the rest of the text.

A way to overcome the difficulties in hand-engineering network architectures is to automatically design architectures using an architecture search algorithm. Such architecture-designing algorithms essentially come under the category of meta-learning a.k.a learning-to-learning algorithms [72, 1].

1.1 Motivation

Over the years, there have been a few successful and not so successful attempts at automatically generating deep architectures. They are broadly based on neuro-evolution algorithms [17, 26, 40, 50, 64, 75, 76], Bayesian optimization [7, 39, 15] or reinforcement learning (RL) algorithms [**enas**, 4, 3, 81, 82, 80, 9]. Overall the RL based architecture search appears to be the most feasible and the most practical choice as of now in terms of the performance of the generated architectures, the applicability to a generic task and the algorithm scalability. Despite these

advantages, they have been applied only to completely supervised settings like supervised image classification on CIFAR-10 [30], CIFAR-100 [30], ImageNet [52], SVHN [43] and MNIST [34] datasets, and supervised text generation by using the PTB dataset [41]. In each of these supervised tasks, the algorithm has been executed for a sufficient amount of time, sometimes multiple days despite using multiple GPUs, for the search on a particular task. To mitigate this problem of the search being computationally intensive, [4] has provided a method to predict a network's validation accuracy at convergence using a regression model with the initial part of the validation curve as one of the inputs. This was an attempt at searching after only partially training a candidate network, while [13] has tried to devise a method for a similar prediction but without training the network at all. [4] still needs a partial training for the search and hence, is significantly computationally-expensive. [13] provides an approach that doesn't look very sound and convincing.

The RL based algorithms have been shown to work by using them on the tasks of supervised image classification and text generation only. Still, they have their own share of drawbacks. These search techniques are absolutely untested on unsupervised and semi-supervised tasks like image reconstruction, image generation etc. This keeps open the scope to devise methods to try and extend their applicability to unsupervised and semi-supervised tasks. In particular, the application of architecture search to the task of unsupervised image generation with variational autoencoders can provide multiple insights. Hand-designing variational autoencoder architectures is difficult for the purpose of the generation of good quality data, particularly the selection of the dimensionality of the latent space. Also, optimization of the joint loss function in a variational autoencoder is not trivial. Being relative more complex, an architecture search with variational autoencoders can give useful insights into the nature of the architecture search algorithms, the necessary heuristics for the search to work, and the dependence of the necessary heuristtics on the task at hand. Also, it can give an idea about the latent sizes that are good for a particular dataset from the image generation point of view.

1.2 Objective

This work is aimed at providing a detailed study of reinforcement learning based architecture search. It will try to clearly point out the applicability of RL based architecture search to supervised and unsupervised tasks, and will provide an in-depth description of the task-dependent heuristics that are necessary for the search to perform well. A comparative analysis will be put forward by showing the successful cases of architecture search for supervised classification and unsupervised reconstruction tasks by just using the heuristics as used in the current architecture-search literature, and the failure of such commonly used heuristics during the architecture search on unsupervised image generation tasks. The necessary changes in the heuristics for a search on

an unsupervised image generation task will prove to some extent the high dependence of the search heuristics on the task at hand. Specifically, this work to a large extent, will deal with the architecture search on variational autoencoders, the reformulation of its joint loss function for better optimization and the positive impact of such reformulation, the methods of formulating the reward from the validation loss metric, and the merits and demerits of such methods.

1.3 Organization

The report consists of ten chapters apart from the first introductory chapter.

The second chapter will discuss and explain the popular and relevant RL algorithms generally used for architecture search. A rough comparative analysis of the algorithms will also be done.

The third chapter is on the related work done in this aspect. It will very briefly cover the different publications which have used different techniques for architecture search, the performance of their methods, their advantages and shortcomings.

The fourth chapter will consist of the relevant baselines for architecture search using RL. The results from re-implementing a few of such baselines on benchmark datasets from different tasks will also be provided.

The fifth chapter will highlight the relevant experiments and improvisations that have been tried as of now. These experiments have broadly been aimed at two aspects, reducing the execution time a.k.a search time of baseline algorithms, and extending the baseline algorithms to other machine learning (ML) settings such as semi-supervised and unsupervised.

The sixth chapter will talk about the different image generation models in use and will highlight the possible insight and knowledge to be gained out of architecture search experiments on the task of completely unsupervised image generation using variational autoencoders.

The seventh chapter will be on the general theory of variational autoencoders, it's link with Bayesian inference and will also present rigorous mathematical expressions that are there behind the working of variational autoencoders.

The eighth chapter will present the crux of this work. It will describe in detail a new technique of reformulating the vanilla loss function for a variational autoencoder and the need for the reformulation.

The ninth chapter contains the results from a toy experiment with a random variational autoencoder architecture with a randomly picked 'bottleneck'/latent layer. It compares the MNIST performance of its vanilla version with the performance of the same network after using the proposed method of formulation in this work. This chapter basically acts as a proof of concept for the new loss formulation proposed in this work, and also builds the ground for the further experiments.

The tenth chapter will provide detailed results from the experimental work done with architecture search for variational autoencoders. It will also contain in-depth comparison of different results which will be help in providing valuable insights and drawing useful inferences.

The eleventh chapter provides a very short summary of the work done in this thesis. It covers the initial experiments, the new loss formulation proposed and the final architecture search experiments with the new formulation.

The twelfth chapter will draw a conclusion to the end-semester thesis report. It will also contain the work to be done in future and the expectations from such work.

Chapter 2

Reinforcement Learning Algorithms for Architecture Search

"Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning." - page 1, chapter 1, [66].

2.1 Reinforcement Learning in Simple Terms

The five primary key words of RL are

- ***Agent***
- ***Environment***
- ***State***
- ***Action***
- ***Reward***

In simple terms, the agent is the entity which plays the central role in an RL algorithm problem. It is both the doer i.e. takes actions according to a learned policy, and the receiver i.e. the recipient of rewards upon taking actions. The environment is the ambience or the surroundings in

which the agent acts and gets rewarded. The states are the different settings that the environment can be in at a particular instant in time. Any change in the setting of the environment can lead to a change in the state. The actions are the different possibilities of acting that an agent has when it is in a particular state in the environment.

The environment has fixed state-transition probabilities which are inherently characteristic of the environment and are function of the current state and action. Besides, it has fixed characteristic reward values which are also function of the state and the action. Whenever an agent takes an action, the environment may give back a reward or may not if it's delayed. The state-transition probabilities of the environment determine the next state that the environment will transition to based on the current state and the agent's action. Figure 3.1 illustrates the basic process of an agent acting on the environment from a certain state and getting a reward while the environment moves to a new state.

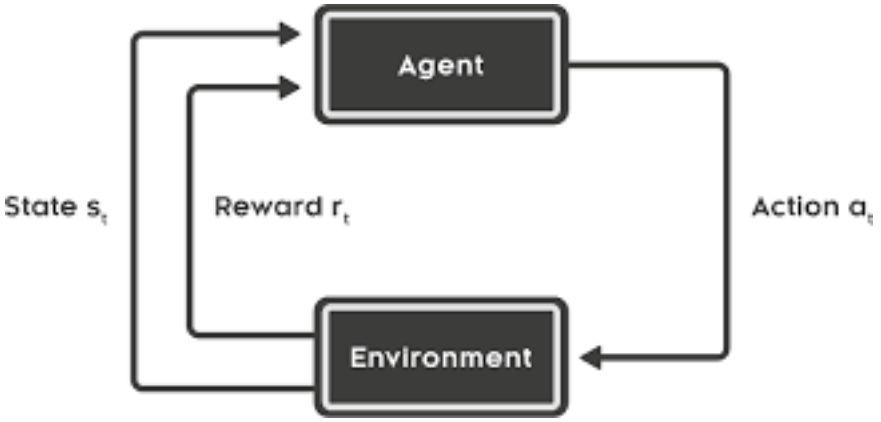


FIGURE 2.1: The whole cycle of an agent acting on the environment, getting rewarded by the environment, and the environment transition to a new state. ([51])

The ultimate goal of an RL agent is to maximize the total reward at the end of its action sequence. Thus, it attempts to take such a sequence of actions which will help it in realizing this goal. Hence, the objective of an RL algorithm is to efficiently generate a policy after sufficient learning/trial time that will produce trajectories or sequence of actions which maximize the total end reward. The RL algorithm essentially trains a controller to do this over multiple runs of trial/training. Figure 3.2 is an example RL problem. In this example, the goal is to find the shortest path from the start to the end of the maze. The reward value is -1 for each step taken. Therefore, the RL controller gets trained to produce a policy that generates the shortest path (the most optimal trajectory) from any state. The maze on the right graphically represents the most optimal trajectories from all possible states.

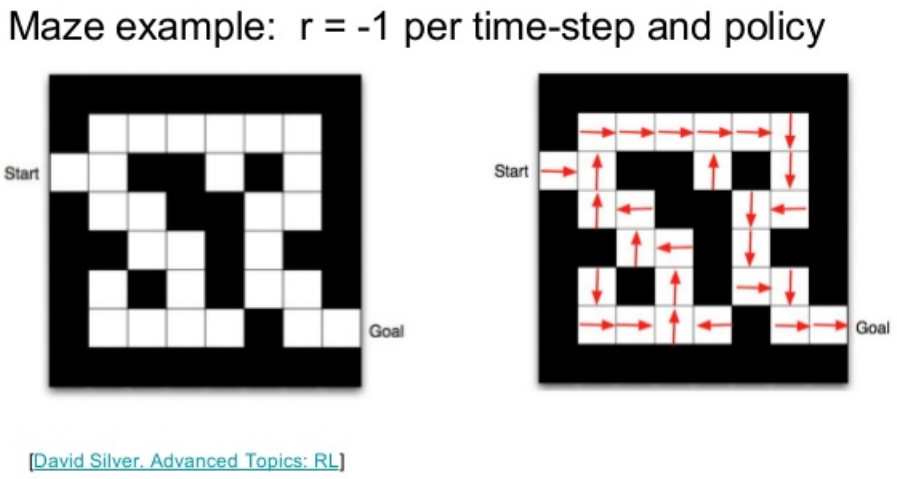


FIGURE 2.2: **Left:** A maze with a start and a goal is the environment, the RL problem is such that a reward of -1 is awarded per time step and hence, and the goal is to find the optimal path/trajectory to the goal of the maze from every possible position/state in the maze. **Right:** The optimal paths/trajectories corresponding to a trained policy, from every position/state. ([47])

2.2 Technical Details of Reinforcement Learning

The following technical summary of a generic RL algorithm has been adapted from [66].

The state of the environment at time t is denoted by s_t, the action taken by the agent at time t is given by r_t, and the reward at time t is $r(s_t, a_t)$. The state might not be always same as the way it is observed by the agent due to some sort of obstruction or occlusion, and the observation at time t is represented by o_t. If the state is exactly as observed, then the setting is a ***fully-observed*** setting or else it's a ***partially-observed*** setting. The policy in a fully-observed setting is given by $\pi_\theta(a_t|s_t)$ while in a partially-observed setting, it's given by $\pi_\theta(a_t|o_t)$. The parameters θ are the RL controller parameters which are tuned during the training of the controller by running RL algorithm such that at the end, the controller generates the most optimal trajectories. The parameters θ might be the parameters of a neural network based RL controller as shown in figure 3.3. Figure 3.3 also shows the basic RL loop. Besides, any RL setting can be thought of as a a Markov decision process (MDP) in which the Markov property is maintained because the state-transition probabilities, which give the chances of going to a particular state from a given state after taking a given action, are functions of the present state and the present action only as illustrated in figure 3.3.

The entire trajectory generated by a policy is given by the joint probability distribution in

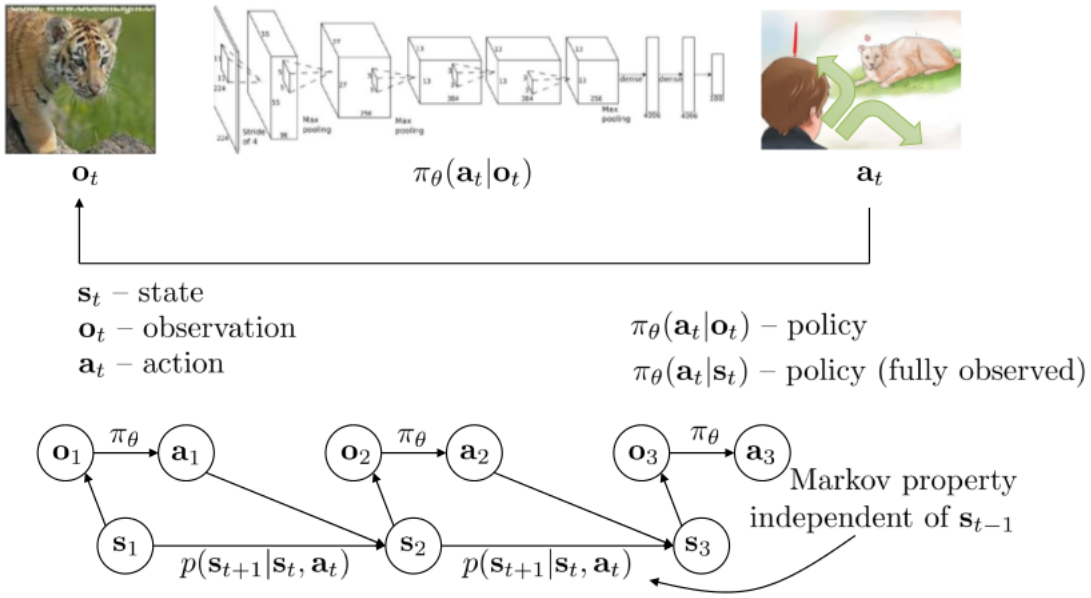


FIGURE 2.3: **Top:** The RL loop with the usual symbols and notations, showing an observation which may or may not be same as the state, a policy-generator in the form of a neural network with the network parameters being the parameters of the policy, and the possible actions from the particular state. **Bottom:** An MDP where the observation is not same as the state and each transition is dependent only on the current state and action while being independent of the history of state-action pairs. (slide 5, [36])

equation 3.1.

$$p_\theta(s_1, a_1, \ldots, s_t, a_t) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \tag{2.1}$$

The joint distribution in equation 3.1 is also sometimes denoted by $p(\tau)$ where τ is used in place of the sequence of the state-action pairs (trajectory).The goal of the RL controller/algorithm is to train the parameters of the policy θ such that the expectation of the total reward from any particular state is maximized. The maximization over the expectation of the total reward is mathematically represented by equation 3.2.

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right] \tag{2.2}$$

2.3 Ubiquitous RL Algorithms in Architecture Search

The most frequently used algorithms in neural architecture search are the Q-learning [74, 42] and policy gradients (REINFORCE) [67] algorithms. These algorithms are popular off-the-shelf algorithms in RL and apply very well to the architecture search setting primarily because these algorithms are found to work well for discrete cases. Still, there are a few shortcomings of these algorithms for discrete cases and problems might be encountered due to the drawbacks while

using these algorithms but on taking suitable remedial measures, these algorithms have been found to work decently. Though Q-learning was first put forward in [74] and policy gradients in [67], the following description and analysis of the algorithms will be as per their analysis in [66].

2.3.1 Policy Gradients (REINFORCE)

This algorithm is a gradient-based algorithm and is very similar to the gradient based back-propagation algorithm commonly used in the training of neural networks. It works as described in the following paragraph.

The training is started with a random policy initialization where the probabilities of an action given a state are picked from some random distribution and not optimized to give maximum reward. Subsequently, the untrained policy is used to produce trajectories and the rewards from the trajectory are accumulated or saved for training of the policy-generator. Next, the gradients of the log probabilities of the actions given the states of the trajectory are weighted by the corresponding stored reward values due to that particular action from that particular state and are added up to give a sum of products. Finally, the sum of products is used in taking a gradient ascent step to tune the policy-generator/controller parameters and this set of steps is repeated for a sufficient number of times to train the controller, and eventually, the optimal policy is achieved which produces highly optimal trajectories that in most cases, maximize the reward returns. The pseudo-code for the algorithm is as follows:

Algorithm 1 REINFORCE

top:
  sample $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (run it on the agent)
  $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\right)\left(\sum_t r(s_t^i, a_t^i)\right)$
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$.
goto *top* **N** times

In the pseudo-code, $\{\tau^i\}$ refers to a batch of sampled trajectories, $\nabla_\theta J(\theta)$ gives the estimated gradient, and α is the parameter learning rate while all the other symbols and notations are same as previously explained in section 2.2.

Though the REINFORCE algorithm is extremely intuitive and works well for most tabular discrete cases (the case of architecture search), there are few shortcomings which are listed hereafter. First of all, as seen in the pseudo-code of the algorithm, the algorithm is strictly ***on-policy***. This means that whenever a change is made to the parameters of the policy-generator/controller, a new trajectory needs to be sampled which means that the agent has to run all over again. Thus, the training of the controller using REINFORCE is pretty intensive as each iteration in the training loop requires fresh running of the policy. For the purpose of architecture search

using REINFORCE, the controller training also accounts for a substantial percentage of the total training time in this way. Secondly, the algorithm, in its original form, has a very high variance term in the form of the summation of the exact reward values over multiple trajectories and taking its mean in the gradient approximation step. Lastly, the use of the exact values of rewards makes the gradient depend on the actual reward values rather than the relative reward values. In simple terms, all the rewards might be negative as in the maze example of figure 3.2, yet the rewards can be maximized on that particular scale of rewards (in the maze example, the scale is the the scale of all possible negative reward sums). Thus, when a gradient step is taken by using the sum using the exact values, the step might go in a completely wrong direction and thus, leading the training to nowhere.

The remedies to the above-mentioned problems are as follows. To go around the problem of on-policy, ***importance sampling*** from Statistics can be used. Importance sampling makes possible emulating trajectories from an updated policy without actually generating the trajectories from the updated policy. But, this requires the knowledge of the conditional state-transition probabilities (not the fixed state-transition probabilities of the environment which are functions of both state and action but the state-transition probabilities of the policy which are only function of the action) of the new updated policy given those of the original policy. The mathematical representation of importance sampling in the context of the expectation of the reward sums which is a term in the gradient approximation step of the REINFORCE algorithm is

$$J(\theta) = \mathbb{E}_{\tau\sim\bar{\pi}(\tau)}\big[\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)} r(\tau)\big] \tag{2.3}$$

where $\bar{\pi}(\tau)$ represents the old set of state-transition probabilities according to the old policy while the $\pi_\theta(\tau)$ represents the updated ones. Next, in order to reduce the high variance from not using the actual expectation value in the gradient approximation step, the total reward-to-go can be used which is the mean over the reward sums from the present time step in place of the reward sums as calculated from the first time step. The gradient approximation step then becomes

$$\nabla_\theta J(\theta) \approx \sum_i \big(\sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\big)\big(\sum_{t'=t}^{T} r(s_{t'}^i, a_{t'}^i)\big) \tag{2.4}$$

Finally, for mitigating the problem of scale arising from using the exact reward values, a baseline can be subtracted from the reward sum which can be some sort of an average over the rewards. The gradient approximation step then becomes

$$\nabla_\theta J(\theta) \approx \sum_i \big(\sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\big)\big(\sum_t r(s_t^i, a_t^i) - b\big) \tag{2.5}$$

where b is the baseline

2.3.2 Q-learning

This algorithm works on the principle of maximization of reward sums and doesn't require any training per se in discrete search cases. It is the other algorithm that is popularly used in architecture search. In RL, Q-value is defined as the expectation over the total reward-to-go from a certain state when a certain action is taken:

$$Q^{\pi}(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t] \tag{2.6}$$

It is parameterized by the policy and is a function of the state and the action and for particular environment. The Q-value for the optimal policy has a unique value for each pair of state and action though it's often unknown at the start because of the untrained non-optimal policy at the beginning of 'training', but deterministic to a certain extent after a sufficient time has been spent in 'training' the policy. The Q-learning algorithm attempts to determine the most optimal Q-values (Q-values corresponding the optimal policy) at the time of training. It works as described hereafter.

At the start, an initial set of Q-values for every state and action pair is started with, corresponding to a randomly chosen initial policy random initial probabilities of an action given a state a trajectory is taken such that each action is according to the maximum Q-value from that particular state in an absolutely deterministic manner or sometimes the choice is made probabilistic as explained later. Thereafter, the Q-value for the present state and the action is updated corresponding to the maximum Q-value such that it's equal to the reward for that particular state-action pair and the maximum of the possible Q-values from the next state which is uniquely determined by the characteristic state-action probabilities of the environment, corresponding to different actions from the next state.The above mentioned steps are repeated for a sufficient number of times and at convergence, the approximately estimated Q-values corresponding to the optimal policy which generates the best reward sums most often, will be obtained.

So, the ideal policy according to an ϵ-greedy schedule after the approximation of the actual Q-values is done, is as follows

$$\pi(s_t, a_t) = \begin{cases} 1 - \epsilon & \text{if} \quad a_t = argmax_{a_t} Q_\phi(s_t, a_t) \\ \epsilon & otherwise \end{cases} \tag{2.7}$$

Such a schedule includes some scope of exploration of the search space in addition to exploitation by taking steps according to the principle of maximization of Q-values. Initially in the schedule, the ϵ value is kept high so as to favor more of exploration so that most of the state action pairs are visited almost uniformly for an efficient and proper 'training' of the policy-generator, and it's

gradually reduced so as to favor more of exploitation towards the end of the 'training' schedule. This ϵ-greedy schedule can also be used in the trajectory generation step during the 'training' of the controller to make the choice of step in the trajectory probabilistic and a little exploratory. This algorithm is particularly useful in discrete cases like architecture search and the version of the algorithm for continuous cases is called the ***fully-fitted Q-iteration*** algorithm. Since the state and action spaces are continuous in such cases, the maximization step for estimating the Q-values can't be applied and instead a function approximation is done to estimate the Q-values. The function approximator is the controller in this case which at convergence, generates optimal trajectories according to the ϵ-greedy schedule based on the precisely-estimated Q-values. The pseudo-code for the fully-fitted Q-iteration algorithm which is more generic, is as presented in algorithm 2.

Algorithm 2 Fully-fitted Q-iteration Algorithm

1: *top*:
2: collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy
3: set$y_i \leftarrow r(s_i, a_i) + \gamma max_{a'_i}(Q_\phi(s'_i, a'_i))$.
4: set$\phi \leftarrow argmin_\phi \frac{1}{2} \sum_i ||Q_\phi(s_i, a_i) - y_i||^2$.
5: **goto** *top* **K** times

In the pseudo-code, γ is a discount factor, $||X||^2$ gives the L2-norm of X, and all other symbols and notations have exactly same meaning as section 2.2. Step 4 in the pseudo-code can be done away with in discrete cases as discussed above.

The Q-learning is both theoretically and practically very fast because of no actual training of the controller and looks very suitable for discrete state and action spaces as in the case of architecture search . Also, as can be observed from step 1 in the pseudo-code, the algorithm is entirely off-policy and can working with sampling of trajectories from any policy as long as the policy has a wide support i.e. covers a wide and diverse set of state-action pairs. Still, it also comes with its own share of drawbacks. They are as follows.

Firstly, the guarantees of convergence are lost in non-discrete/continuous cases because of the need to use a function approximator to estimate the Q-values. The function approximator can be in the form of a neural network or a regression model but the targets/labels required during the training of the function approximator are moving targets and its values are not fixed. The y_is in step 3 and step 4 of the algorithm change with every step and thus, are not fixed. Thus, many times the function approximator doesn't converge. Besides, when the tuples of current state, current action, next state and reward, are collected from a certain policy as shown in step 1 of the pseudo-code, the samples might be highly correlated and the function approximator in a continuous case might overfit those samples due to high correlation. Also, such correlated samples lack broad support which are a criterion for the algorithm to work successfully in an off-policy manner.

To solve the problem of correlation, a replay buffer is used which is a huge memory bank that stores samples from different policies. So, the above-specified pseudo-code basically becomes the inner loop of the algorithm where the the sample tuples are taken from the replay buffer and thus, mostly do not belong to a particular policy. The addition of new policy samples to the replay buffer which is the outermost loop in the changed algorithm is done less frequently in comparison to the inner loop of Q-value estimation. An example buffer is shown in figure 3.3. The problem of moving targets in the training of the function approximator is dealt by the use

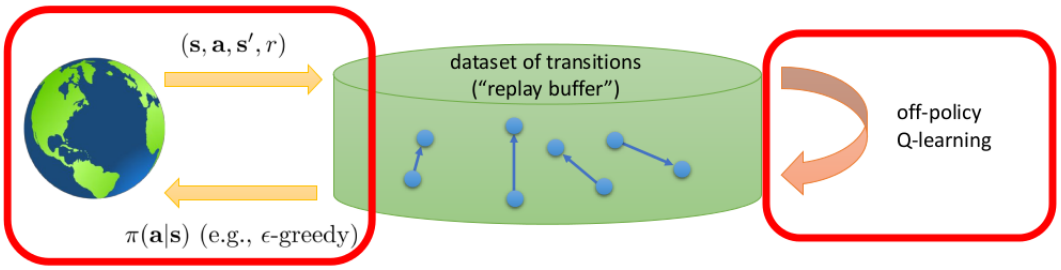


FIGURE 2.4: A sample replay paper where addition of trajectories to the buffer is done on the left while training of the policy is being done on the right independent of the trajectory generation. (slide 9, [35])

of target networks. The target networks set the targets for the function approximator algorithm and are parameterized by another set of parameters which are completely different from those of the policy generator. The update of this set of parameters becomes the outermost loop in the final changed algorithm thus guaranteeing that the targets during the run of the innermost loop (above-given pseudo-code) remain fixed. The final changed Q-learning algorithm also called ***DQN*** is given by algorithm 3.

Algorithm 3 DQN/Q-learning with Replay Buffer and Target Network

top:
  save target network parameters: $\phi' \leftarrow \phi$
  mid:
    collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\boldsymbol{B}$
    in:
      sample a batch (s_i, a_i, s'_i, r_i) from $\boldsymbol{B}$
      $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma max_{a'} Q_{\phi'}(s'_i, a'_i)])$
    goto *in* **K** times
  goto *mid* **N** times
goto *top* **O** times

2.4 A Comparative Analysis of REINFORCE and Q-learning Algorithms for the Purpose of Architecture Search

After a close scrutiny and a detailed analysis of the technical nature of the two algorithms in the way given above, a comparative study of the two can be done. A short comparative study is included in the immediately following paragraph.

Both the algorithms are effective and can be efficiently used in discrete cases if the necessary remedial measures are taken before using them. Neural architecture search being a discrete case can involve any of the algorithms. There shouldn't be any tangible difference in the performance if the algorithms are given their share of necessary training time and, breadth and support of state and action spaces. Despite the expected similarity in performance in discrete cases, the Q-learning algorithm doesn't have any controller/policy-generator training per se and an architecture search using Q-learning might turn out to be faster than that with REINFORCE in all probability if the number of architectures in both searches is kept same.

Chapter 3

Related Work

Hyperparamater optimization for finding architecture configurations has been in use for quite some time now [8, 5, 6, 7, 39], given that it is a very important topic in ML. But hyperparameter optimization is limited to searching networks from a fixed-length search space and fails to discover variable-length networks. Practically, it is expected to fare well if supplied with a good initial model. On the other hand, automatic variable-length architecture search broadly covers three types of algorithms:

- Bayesian optimization methods
- Neuro-evolution/Genetic algorithms
- RL algorithms

Work under these categories will be discussed hereafter.

The Bayesian optimization methods were used in [7, 39, 15]. They were used with the aim to generate variable-length architectures but they have been often found to be less general and less flexible. They are mainly based on sequential model-based optimization (SMBO) [24], Gaussian processes (GP) [61], Tree of Parzen Estimators [7], and neural networks [62]. In [7], Bayesian optimization was used for automatic selection of network architectures while in [61], it's used for the automatic selection of hyperparameters like the optimal values of learning rate, batch size etc. But they didn't quite match the performance of hand-crafted networks as expected. Mostly, random search or grid search is used for practical experiments [5]. Recently, Hyperband, a multi-armed bandit-based efficient random search method, was proposed in [37] which was found to outperform state-of-the-art Bayesian optimization methods.

The neuro-evolution algorithms were used in [17, 26, 40, 50, 64, 75, 76]. Genetic algorithm-based methods were tried out to find both architectures and weights in [55]. But, networks generated

using genetic algorithms, like the ones generated with the NEAT algorithm [65], are yet to succeed at improving over the hand-engineered networks on standard benchmarks [73]. Other bio-motivated approaches, inspired by screening method in genetics, as in [49] put forward a high-throughput network generation technique where thousands of architectures were searched over randomly and a few promising ones were finally trained. More flexible approaches were proposed in the recent neuro-evolutionary works like [75, 17, 64]. But they have not shown success at large scale as of now because they are mere search based methods and are either very slow due to their requirement of excessive computational power, or require many heuristics to perform well.

The RL algorithms have been shown to be the most promising choices for architecture search in the current scenario as seen in [**enas**, 4, 3, 81, 82, 80, 9]. [**enas**, 81, 82] claimed to devise methods to generate architectures that gave state-of-the-art performance in completely supervised image classification [30, 30, 52, 43, 34] and completely supervised text generation [41]. These works mostly used either the policy gradients [67] and proximal policy optimization (PPO) [56] as the RL algorithm for the search. The other works [4, 3, 80, 9] proposed methods for Q-Learning [74, 42] based search which produced competitive performance on benchmark datasets in image recognition, and policy gradients based network transformation methods which also gave better performance on benchmark datasets than the corresponding hand-designed state-of-the-art networks.

[4] designed a method to predict the final validation accuracy at convergence by using the initial time-series validation accuracies (along with their first-order and second-order differences), and other hyperparameters including those that are based on the network structure like the number of weights, number of layers etc., and others like initial learning rate, learning rate decay etc. It's method though shown to successfully search highly efficient networks by just using 25% of the training/validation curve, required the initial training of the candidate architectures and hence, was still computationally demanding. [**peephole**] claimed to predict validation accuracy at convergence without any training. It used a method similar to [4] and made a prediction based on the network structure but didn't require the initial training/validation curve as one of the inputs for the prediction model. Essentially, it trained a model to map the network architecture to a final validation accuracy. Issues might rise with this method due to the limited extent of the training dataset for the model since it's most unlikely to include the highly efficient architectures and their corresponding validation accuracies in the data since such architectures and their performances are very hard to imagine, and the prediction model might fail at extrapolating it's prediction to unforeseen architectures.

Chapter 4

Relevant Architecture Search Baselines and Re-implementation of such Baselines

There has been a number of publications in the past two years which have approached the problem of architecture search using reinforcement learning by either using Q-Learning or by using REINFORCE. Despite always using one of these two well-suited RL algorithms, the way to convert a candidate architecture from the search space into a candidate trajectory i.e. the way of mapping an actual architecture to the states and actions of the RL environment is different in different works. Some of these works have produced very competitive results while some have given state-of-the-art performances on some benchmark datasets for certain tasks like image recognition (MNIST, SVHN, ImageNet, CIFAR-10, CIFAR-100) and text generation (PTB). The others have aimed at reducing the search time by using different heuristics and algorithms. The works which stand out among all in terms of a well-defined, robust and intuitive work pipeline, and performance are as follows

- ***Designing Neural Network Architectures using Reinforcement Learning*** [3]
- ***Neural Architecture Search with Reinforcement Learning*** [81]
- ***Accelerating Neural Architecture Search using Performance Prediction*** [4]
- ***Efficient Neural Architecture Search via Parameters Sharing*** [48]

Among these, *Designing Neural Network Architectures using Reinforcement Learning*, *Neural Architecture Search with Reinforcement Learning*, and *Efficient Neural Architecture Search via Parameters Sharing* have been re-implemented with the purpose of further extending them to

other machine learning settings like unsupervised and semi-supervised learning or for further reducing the search time.

4.1 Analysis of *Designing Neural Network Architectures using Reinforcement Learning* (MetaQNN)

To the best of author's knowledge, this paper [3] was the first recent work that used Q-learning for the purpose of architecture search. The overall algorithm presented in this work was called *MetaQNN*. The way this work approached the search problem was as described in the next three paragraphs.

Each layer in a convolutional neural network (CNN) was treated as a state and the choice of the next layer as the action from that state. Accordingly, a completely discrete search space was designed where the discrete state space was fabricated by defining each state a.k.a layer by specifying the layer type from the choices of convolution, max-pooling, fully-connected, soft-max classification etc., the kernel size of the filters, the kernel stride, the number of filters, the layer number/depth, the net feature dimensionality, among others. The design of the discrete action space where each action stood for the next network layer to be inserted, was done in a very similar way. A layer limit and some other restrictions which dictated the connection of fully-connected and classification layers, were also specified to keep the trajectory finite. Figure 4.1, taken from the MetaQNN paper, shows the pipeline used for the search. This pipeline is nothing other than an MDP.

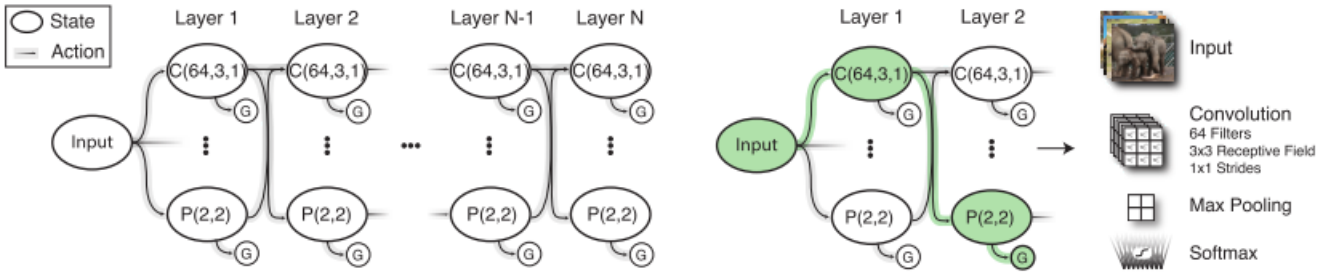


FIGURE 4.1: **Left:** The MDP in CNN generation where each layer/state is an ellipse, an arrow is an action a.k.a transition to the next layer/state, and the first/starting state is the image input. **Right:** A sample CNN/trajectory. (page 4, [3])

The basic Q-learning algorithm used in this work was mathemetically formulated as follows:

$$Q_{t+1}(s_i, u) = (1 - \alpha)Q_{t+1}(s_i, u) + \alpha[r_t + \gamma max_{u' \epsilon U(s_j)} Q_t(s_j, u')] \tag{4.1}$$

Method	**CIFAR-10**	**SVHN**	**MNIST**	**CIFAR-100**
MetaQNN (ensemble)	7.32	**2.06**	**0.32**	-
MetaQNN (top model)	**6.92**	2.28	0.44	**27.14***

TABLE 4.1: The error-rates of the top model and the top-five ensemble in MetaQNN on CIFAR-10, SVHN, MNIST and CIFAR-100 datasets.

where s_i stood for the state i, u for the action, Q_t stood for the Q-value at time t, α was the weighing term, U was the entire set of actions given the state, and γ was the discount factor. The updates to the Q-values were made in such a way that they were not completely overwritten during the update and a part of the previous Q-value was retained.

An ϵ-greedy schedule was followed to select architectures and the chosen architectures were trained until convergence. Every time after running an architecture and collecting the best validation accuracy from that runs, an update to the Q-values was made using the validation accuracies from a few other architecture randomly chosen from among the already-trained ones. When a new architecture was chosen next, it's chosen according to the updated Q-values as per the ϵ-greedy schedule. The whole algorithm can be represented in very simple terms as done in algorithm 4.

Algorithm 4 MetaQNN

1: *top*:
2: **sample an architecture** according to the **Q-values** as per an **ϵ-greedy** schedule
3: **train** the **sampled architecture** till **convergence** and **store** the **best validation accuracy** as the **reward**
4: *in*:
5: **sample a few already trained architecture randomly**
6: $\vee$ **t i.e. all layers/states** in the **sampled architectures** : $Q_{t+1}(s_i, u) = (1-\alpha)Q_{t+1}(s_i, u) + \alpha[r_t + \gamma max_{u' \epsilon U(s_j)} Q_t() s_j, u']$
7: **goto** *in* **K** times
8: **goto** *top* **N** times

Noticeable results from the paper were the highly competitive performances of the top architectures and top architecture-ensembles on the benchmark datasets (MNIST, CIFAR-10, CIFAR-100, SVHN). The best networks were found to beat the state-of-the-art hand-designed networks with similar layer types and produce competitive performance against state-of-the-art networks with more complex layer types. Table 4.1 contains the results (error-rates) of the top model and the top-five ensemble on MNIST and SVHN datasets without any data augmentation, and on CIFAR-10 and CIFAR-100 with moderate data augmentation. A serious issue with using MetaQNN was the huge search time. The paper mentions training of several thousand architectures until convergence on each dataset before finding the top model.

4.2 Analysis of *Neural Architecture Search with Reinforcement Learning* (NAS)

NAS [81] was most probably the first publication that used the REINFORCE algorithm for the architecture search. The pipeline in this work was partially similar to the one used for MetaQNN. But the use of REINFORCE algorithm forced the use of the RL controller in an altogether different way. The similarity lies in the fact that the discrete state and action space were designed in a similar manner by considering a layer in a CNN or a node of computation in a recurrent cell as the state and the choice of next layer in the CNN or the choice of the next node of computation in the recurrent cell as the action, and specifying the layer type, the kernel size, the kernel stride, the number of filters etc. to define the layer in a CNN, and the type of computation and the nodes involved to define the node for a recurrent cell. This discussion will solely focus on the generation of CNNs to elucidate the concept at work and for brevity. The generation of a recurrent cell was done in a very similar way. A brief description of their overall algorithm is included in the subsequent three paras.

The RL controller a.k.a. trajectory generator was in the form of a recurrent neural network (RNN) or a long short term memory (LSTM). The controller generated a sequence of states until a certain limit on the number of states, corresponding to the layer limit, was reached. Each state was produced by the sequential generation of the parameters defining the state by the RNN. So, if their were K parameters defining a state in a network and if their were S states in the network (S layers in a CNN or S nodes of computation in a recurrent cell or LSTM cell), then the RNN controller ran for a total of $S * K$ time steps. A graphical representation of this layer generation in a CNN is given in figure 4.2.

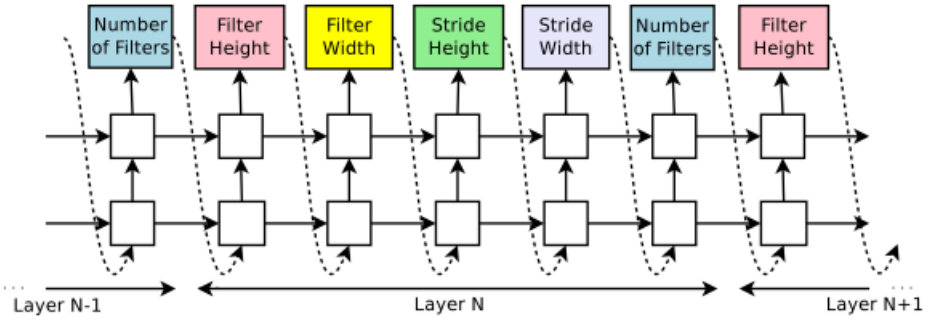


FIGURE 4.2: A particular CNN layer is parameterized by four values (filter height, filter width, stride height, stride width) which get generated sequentially by the RNN controller and thus, define a state. Such tuples of four values are also sequentially generated to define the whole CNN. (page 3, [81])

Once the architecture was generated in the above-mentioned way, it's trained until convergence and the best validation performance of the architecture was used to train the parameters θ of the controller using the REINFORCE algorithm by using the parameter update in the following mathematical expression:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T}\nabla_{\theta_c}\log P(a_t|a_{(t-1):1};\theta_c)(R_k - b) \tag{4.2}$$

where m was the number of architectures/trajectories to average over, T was the number of time-steps the RNN controller was run to generate a network, a_t was the action at time step t, θ_c were the controller parameters, R_k was the reward sum for the k^{th} architecture and b was the exponentially moving average baseline.

After training the controller with the best validation performances of the generated architectures for a sufficient number of runs, the controller parameters eventually got tuned to generate the high performing efficient networks. The overall algorithm which the paper calls NAS, has been compactly written in simple terms in algorithm 5.

Algorithm 5 NAS

1: *top*:
2: **sample an architecture** by running the **controller** for **S*K time steps**
3: **train** the **sampled architecture** till **convergence** and **store** the **best validation performance** to compute the **reward**
4: **update** the **controller parameters** using $\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T}\nabla_{\theta_c}\log P(a_t|a_{(t-1):1};\theta_c)(R_k - b)$
5: **goto** *top* **N** times

The paper reports networks which gave state-of-the-art performances for image classification on the CIFAR-10 dataset and text generation by using the PTB dataset while at the same time using fewer parameters. The top model gives a test error of 3.65% on the CIFAR-10 dataset which is 0.09% better and $1.05x$ faster than the previous state-of-the-art network that used the same architectural scheme. On the PTB dataset, the algorithm generates a novel recurrent cell (very different from the ubiquitous LSTM cell). This cell gives a test set perplexity of 62.4 which is 3.6 perplexity better than the previous state-of-the-art model. A compact representation of the results (test error-rates) from the CNN search using the CIFAR-10 dataset is given in table 4.2. The NAS algorithm also suffers from the issue of a huge search time. It trained $12,800$ CNN architectures and $15,000$ architectures on multiple GPUs to finally get such high-performing architectures.

Model	**Depth**	**Parameters**	**Error rate (%)**
NAS1 (no stride and pooling)	15	4.2M	5.50
NAS2 (predicting strides)	20	2.5M	6.01
NAS3 (max pooling)	39	7.1M	4.47
NAS4 (max pooling + more filters)	39	37.4M	3.65

TABLE 4.2: The error-rates of the NAS models of different types on the CIFAR-10 dataset.

4.3 Analysis of *Accelerating Neural Architecture Search using Performance Prediction*

This paper [4] will be discussed briefly because its content is not exactly in line with the main objective of the thesis. This work talks about predicting the validation performances of candidate networks at convergence, during the search by using a heuristic to bring down the search time. This work is the first work which dealt with the reduction of the search time in RL based architecture search scenarios, to the best of the author's knowledge. The prediction of the training/validation accuracies at convergence, was done using a regression model which takes as inputs the training/validation accuracies in the early stages of training/validation, and other architecture dependent features/hyperparameters. The important steps in the performance prediction mechanism and using the predicted performances for the search are as follows.

The set of input features for the regression model consisted of the architectural hyperparameters like number of weights, number of layers etc., the initial time-series validation accuracies (the exact accuracy values, their first order and second order differences), and other training/validation hyperparameters like initial learning rate, learning rate decay etc. Different regression methods which were tried were ordinary least squares regression, random forests, and support vector regression. By using the set of features as inputs and one of the regression methods, a function approximation was carried out in order to predict the validation accuracy of an architecture at convergence.

Support vector regression with radial basis function kernels was found to work best when chosen as a regression method. An architecture search speedup of 4X was noticed if performance prediction was done only with the initial 25% of the validation curve while the performance of the top architectures found after the search, remained almost same. These are some of the main observations and highlights of this work.

4.4 Analysis of *Efficient Neural Architecture Search via Parameters Sharing* (ENAS)

This paper [48] proposes a method of parameter sharing which as per the reported results, not only reduced the training time drastically but also gave competitive performance on image classification with the CIFAR-10 dataset (similar to the NASNet [82] performance) and state-of-the-art performance on the text-generation task with the PTB dataset. The reduction in search time was attributed to the shared parameters such that every time a new architecture was sampled, fresh training of the architecture shared parameters from scratch until convergence was no longer required. Instead, the shared parameters could be used as they were for getting the validation performance which was used as the reward to train the controller in a way similar to the one in the NAS algorithm. As a result, the training was bi-fold and interleaved. The first phase of training required the training of the shared parameters which determined all possible sets of parameters that might be required for a candidate architecture during validation to generate the reward for the training of the RL controller. The second phase of training involved the use of these shared parameter values to run the candidate architectures on the validation datasets and using the validation performance as the reward to train the controller. The overall algorithm is called ENAS. The discussion below will mostly be with respect to a CNN while a recurrent cell generation can be conceived in a similar way. ENAS has been summarized hereafter.

A network/architecture/cell(recurrent cell) was represented in the form of a directed acyclic graph (DAG). Each node in the DAG could be seen as a layer in a CNN or a node of computation in a recurrent cell. Each node could be connected to any other node in one particular direction with the restriction that the exit node (the last layer in a CNN or the last node of computation in a recurrent cell) shouldn't be connected back to the entry node (the first layer in a CNN or the first node of computation in a recurrent cell). This automatically implied skip connections in case of a CNN. Each connection from the set of all possible connections was accompanied by its corresponding weight values, the dimensionality of which was determined by the output dimensionality of the source node and the input dimensionality of the input node. The DAG structure completely covered all possible networks that could be generated using the same number of nodes (layers in CNN and computational nodes in a recurrent cell) by conditionally maintaining and removing node-to-node links as per the architecture requirement. The total number of possible architectures/cells that could be generated using the DAG was determined by the number of nodes in the DAG. A sample DAG and the generation of an architecture from the DAG is shown in figure 4.3.

The central point of this work was the claim that since the DAG covered all possible architectures, a single set of weights (each weight component corresponding to a particular node-to-node link)

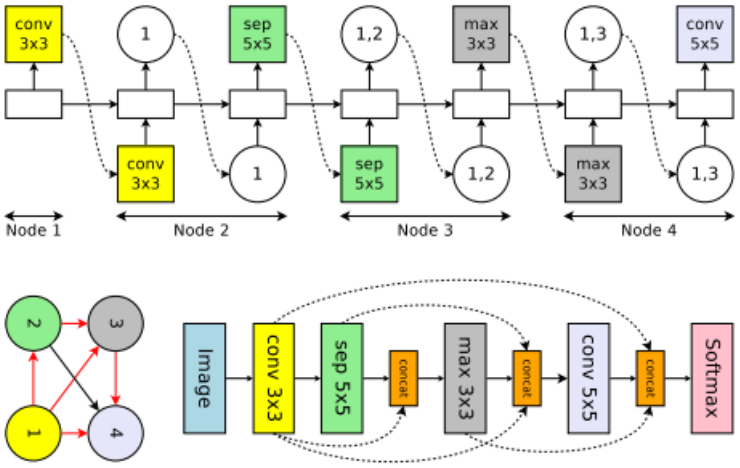


FIGURE 4.3: **Top:** A sample DAG shown on the bottom-left with node 1 as the input and node 4 as the output, rolled out in the form of a chain/sequence of RNN controller time steps with the controller action at each time step, each layer/node is defined by the set of previous layers/nodes it;s connected to (direct and skip connections), and the type of filtering to take place in the layer. **Bottom-Left:** A sample DAG. **Bottom Right:** The final CNN built from the DAG and the controller actions at each time step. (page 3, [48])

could also be used as the source set of weight values for any architecture whatsoever and hence, leading to shared weights. Thus, the initial phase of training required the training of the shared parameters wherein the controller was used to generate architectures randomly without it's own training, for a certain number of times and the validation performance due to each architecture was used to train the shared parameters using an update which was equal to the expectation over the validation performances of the child architectures. The mathematical expression for the update is was follows

$$\nabla_{\omega} \mathbb{E}_{m \sim \pi(m;\theta)}[L(m,\omega)] \approx \frac{1}{M} \sum_{i=1}^{M} \nabla_{\omega} L(m_i, \omega) \tag{4.3}$$

where m represented a candidate architecture generated during the tuning of shared weights, π gave the policy which is parameterized by θ (the controller parameters), L was the loss function (giving the validation loss in this context), M was the total number of samples/architectures/cells to calculate the expectation for the multi-sample update, and ω denoted the shared parameters. The paper also claimed surprisingly that $M = 1$ worked fine though in theory, it should lead to a very high variance update.

Once the shared parameters were trained, the training of the controller was carried out in a fashion exactly same as NAS in algorithm 5. The controller generated architectures/cells which used the shared weights, and their validation performance was used as the reward to train the controller using the REINFORCE algorithm.

Method	**GPUs**	**Times**(Days)	**Params**(million)	**Error**(%)
ENAS + macro search	1	0.32	34.9	4.23
ENAS + macro search + 512 filters in last layer	1	0.32	38.0	**3.87**
ENAS + micro search	1	0.45	4.6	3.54
ENAS + micro search + CutOut *C_cutout*	1	0.45	4.6	**2.89**

TABLE 4.3: The error-rates and search times for macro and micro search od different types in ENAS on the CIFAR-10 dataset.

The architectures generated from a DAG in the case of CNNs could be either used as the complete architecture for end-to-end training which was also called the *macro search*, or could be used as the constituent cell for a stacked-cell network where each cell was searched for and the overall configuration of the network (the stacking of the cells) was kept fixed for a particular task, which was termed as *macro search*. The overall algorithm for ENAS is described in algorithm 6.

Algorithm 6 ENAS

1: *shared parameter training*:
2: **sample** M architectures using **untrained controller** and get **weights** ω from **shared parameter set** Ω
3: **train shared parameters** Ω with the **validation performance** using $\nabla_\omega \mathbb{E}_{m\sim\pi(m;\theta)}[L(m,\omega)] \approx \frac{1}{M}\sum_{i=1}^{M} \nabla_\omega L(m_i,\omega)$
4: **goto** *shared parameter training* **N1** times
5: *do controller training using REINFORCE, exactly same as NAS in algorithm 4*

The results, according to the paper, were absolutely phenomenal with a $1000x$ reduction in architecture search time, and state-of-the-art results on the PTB dataset based text generation and near-state-of-the-art results on the CIFAR-10 dataset based image classification. The test perplexity for the PTB dataset from the resulting top recurrent cell was 55.8 which was 6 perplexity lesser than the previous state-of-the-art found in NASNet and took only *10 hours* for the search. The results (error rates on CIFAR-10) for the CNN search (both macro search and micro search) are presented table 4.3. As can be seen from the table, the best macro model got generated in only *7 hours* while the best micro search model got generated in only *11.5 hours*. Still, a potential drawback of this model is the greedy selection of architecture weights in order to substantially reduce the search time.

4.5 Re-implementation of *MetaQNN* Baseline for Image Classification

The MetaQNN was the first baseline to be re-implemented to test the effectiveness and utility of it in architecture search and to also test the stability and the guarantees of convergence with the Q-learning algorithm. The task of interest in this particular re-implementation was the image classification task with the MNIST dataset, a benchmark dataset for handwritten digit classification, and the AEROBI binary classification dataset, a dataset containing concrete bridge images of two categories, background and cracks. The implementation was done in PyTorch [46], an auto-differentiation package hosted by Facebook AI Research.

The discrete search space for MNIST handwritten digit classification was designed in the way specified in the MetaQNN paper. The convolutional filter size was chosen from among $[1, 3, 5]$, the number of convolutional filters from among $[64, 128, 256, 512]$, the max pooling kernel size from among $[5, 3, 2]$, the possible pool stride from $[3, 2]$, and the possible fully connected layer size from $[128, 256, 512]$. The layer limit was kept at 12 and the limit on the number of fully connected layers was 2. Besides, some other restrictions on the connection of a fully connected layer to a convolutional layer were imposed so as to not make the network too shallow. Each candidate was trained for 20 epochs and a total of 2700 architectures was trained. The training of each architecture was followed by the sampling of 128 random architectures for the Q-value update. The architecture training was done using the Adam [28] optimizer with the negative log-likelihood (NLL) validation loss. In this experiment, the width and height of the convolutional output feature space were kept same as the input feature space by using necessary padding during the convolution, it was noticed that by not using padding and by allowing a decrease in the two-dimensional space size the performance of the top model for MNIST doesn't change by much. The reward in this case was the best validation score from 20 epochs of validation. The architectures and the best top-1 validation accuracy of the top five models for this task are given in table 4.4[1].

The discrete search space for AEROBI binary classification dataset was quite similar to that for MNIST classification. The binary classification dataset initially had images of different dimensions and aspect ratios. This required a data pre-processing step of scaling the images

[1] *In result tables 4.4 - 5.4,* C(X,Y,Z) *represents a convolutional layer of kernel size* Y*, number of output filters equal to* X *and stride equal to* Z*,* D(X,Y) *represents a dropout layer of index* X *out of a total of* Y *dropout layers (the indexing is necessary to use an index dependent dropout rate as per the paper),* P(X,Y) *denotes a pooling layer of kernel size* X *and stride* Y*,* FC(X) *represents a fully connected layer with output feature number equal to* X*, and* SM(X) *represents a soft max classifier with* X *final output units. In result tables 9.1 - 9.16, BCE loss of the architecture search is the plain normalized binary cross-entropy loss calculated between the reconstructed batch and the input batch irrespective of whether the vanilla combined loss or the reformulated combined loss is being used and for the decoder architecture,* C(X,Y,Z) *represents a convolutional layer of kernel size* Y*, number of input filters equal to* X *in place of number of output filters as denoted everywhere else and stride equal to* Z*.*

Architecture	**Best Top-1 Accuracy**(%)
[C(64,5,1), C(256,3,1), D(1,6), C(128,3,1), P(5,3), D(2,6), C(512,5,1), P(5,2), D(3,6), C(128,1,1), FC(128), D(4,6), FC(128), D(5,6), SM(10)]	**99.70**
[C(256,3,1), C(128,3,1), D(1,5), C(128,3,1), P(5,3), D(2,5), C(128,5,1), C(128,1,1), D(3,5), P(5,2), FC(256), D(4,5), SM(10)]	**99.69**
[C(256,5,1), C(64,5,1), D(1,6), C(128,5,1), P(5,2), D(2,6), C(64,5,1), P(5,2), D(3,6), FC(128), D(4,6), FC(128), D(5,6), SM(10)]	**99.67**
[C(64,5,1), C(512,5,1), D(1,6), P(2,2), C(256,5,1), D(2,6), P(2,2), C(512,3,1), D(3,6), C(64,1,1), C(512,5,1), D(4,6), P(3,2), C(128,1,1), D(5,6), C(512,1,1), P(2,2), D(6,6), SM(10)]	**99.67**
[C(512,5,1), C(64,1,1), D(1,5), P(2,2), C(128,5,1), D(2,5), C(256,5,1), C(256,5,1), D(3,5), P(5,3), FC(128), D(4,5), SM(10)]	**99.66**

TABLE 4.4: The architecture configurations and their best top-1 validation accuracy for MNIST digit classification, sorted in the descending order of accuracy values.

such that the smallest dimension of the images remained same for all images while keeping the aspect ratios same as in the original dataset. Still, the images were of different sizes and this required the use of a spatial pyramidal pooling layer (SPP) [21] after the convolutional layers of the network. The SPP layer does a scale dependent pooling which brings down feature spaces of different dimensionality, originating from differently sized images, to the same linear dimension before feeding the same to the fully-connected (FC) layers which are followed by the classifier. The wide range of image sizes required the use of both small and large convolutional filters for proper feature learning. The convolutional filter size was chosen from $[3, 5, 7, 9]$ and the two-dimensional size of the feature space was kept same as the input image in order to use the spatial pyramidal pooling layer without error by the use of convolutional padding and by not using pooling. The fully connected part and the classifier were kept same for all candidate architectures by using only 2 FC layers and the softmax (SM) classifier. The reward in this case was the best validation F1 score from among the 50 epochs of training of each child model. The architectures and the best top-1 F1-score of the top five models for this task are given in table 4.5[1]. As can be seen from the table, the search generated shallower but high-performing networks in contrary to the deeper networks for MNIST digit classification and this shows that this type of search is very task specific. Also, all top five models performed far better than the best-performing hand-engineered model for this task which gave an F1-score of around 48%.

Architecture	**Best Top-1 F1-score**(%)
[C(64,5,1), C(128,3,1), C(96,5,1), C(128,3,1), C(96,7,1), C(96,5,1), SM(2)]	**72.19**
[C(128,7,1), C(64,7,1), C(128,7,1), C(96,7,1), C(96,3,1), C(32,7,1), SM(2)]	**65.15**
[C(32,3,1), C(96,3,1), C(96,7,1), C(128,5,1), C(96,7,1), C(64,3,1), C(128,3,1), SM(2)]	**62.30**
[C(64,5,1), C(128,7,1), SM(2)]	**60.23**
[C(96,3,1), C(64,5,1), C(32,5,1), SM(2)]	**58.98**

TABLE 4.5: The architecture configurations and their best top-1 F1-score for AEROBI binary classification, sorted in the decreasing order of F1-scores.

4.6 Re-implementation of *NAS* and *ENAS* Baselines

The next baselines to be re-implemented were the NAS baseline and the ENAS baseline. The target task for the NAS baseline was MNIST digit classification while that for the ENAS baseline was text generation using the PTB dataset. These code were also written in PyTorch but both of them suffered from the problem of falling into some local minimum during the controller training which resulted in the generation of the same architecture throughout the search. This bug needs to be fixed and hence, a detailed discussion and the results from these baselines are being considered to be a part of the future work.

Chapter 5

Initial Experiments and Modifications: Architecture Search with Random Weights

The different experiments and improvisations that were tried out initially were primarily based on the ideas and propositions of two publications, *On Random Weights and Unsupervised Feature Learning* [54] and *Understanding Deep Representations through Random Weights* [58]. The underlying principle in both these works is that ***the performance of any neural network model on any task is majorly dependent on the structure of the network rather than the learned weights, and that there is a strong correlation between the performance of an architecture with random weights and that of the same architecture with trained weights***. **This correlation can be exploited to use the random performance from architectures as the reward in the architecture search which in turn might lead to a substantial reduction in the search time.** The proposition looks very appealing but comes with its own share of caveats. These works, the applicability of their propositions, the caveats involved, and the experimentation done based on these propositions will be discussed at length in the first part of this chapter. Immediately following are the analyses of *On Random Weights and Unsupervised Feature Learning* and *Understanding Deep Representations through Random Weights*.

The proposition from *On Random Weights and Unsupervised Feature Learning* that's relevant to our experiments, is that the there is a strong correlation between the performance of a neural network with random weights and its performance after complete training. This correlation, also mentioned in [25], can serve as a fast heuristic for architecture search. A caveat to this is one random performance might not always correlate well with the trained performance but the average of many runs with fresh random initialization of weights has a very high chance

of correlating well with the performance of the trained model as mentioned explicitly in *"We note that it is insufficient to evaluate an architecture based on a single random initialization because the performance difference between random initializations is not generally negligible. Also, while there was a significant correlation between the mean performance of an architecture using random weights and its performance after training, a particular high performing initialization did not generally perform well after pretraining and finetuning"* under footnote 2 of page 6 of [54]. The paper suggests ranking the architectures based on performance, completely training the top few and picking the best architecture from them. Figure 5.1 shows the results from the paper where the correlations between the random network performance, and the performance of the same network with pretrained and finetuned weights, on the CIFAR-10-mono and NORB-mono datasets have been plotted.

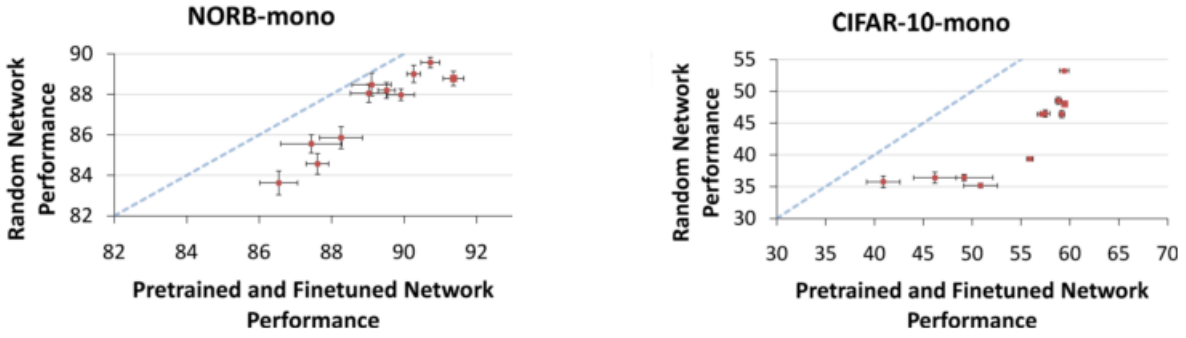


FIGURE 5.1: The classification performance of random-weight networks vs pretrained and finetuned networks where error bars represent a 95% confidence interval about the mean. **Left:** NORB-mono. **Right:** CIFAR-10-mono. (page 6, [54])

Understanding Deep Representations through Random Weights discusses the use of random weights in a convolutional autoencoder (CAE) [23] of the form as given in figure 5.2. They compare the performance of a CAE with both trained encoder and trained decoder, with that of a CAE with a trained decoder and an encoder with random weights, and a CAE with both random encoder and random decoder. Their results show that a CAE with a random encoder produces better performance and faster convergence than a fully trained CAE while the fully random CAE with both random encoder and random decoder, fare worse than the fully trained CAE and the CAE with only trained encoder, as the reconstructed images loose a lot of colorimetric information. A graphical comparison of performance between a random encoder CAE and a fully trained CAE is given in figure 5.3 where the comparison is drawn by using the benchmark AlexNet and VGG-16 architectures for reconstruction. The other inference that can be drawn from this work is that the reconstruction quality deteriorates with the depth of architectures if they are built using the same schema or in other words, they have the same inherent reconstruction ability i.e. a type of network which is inherently as good as another network in a task, where increasing the depth usually improves the performance, like classification, but deeper,

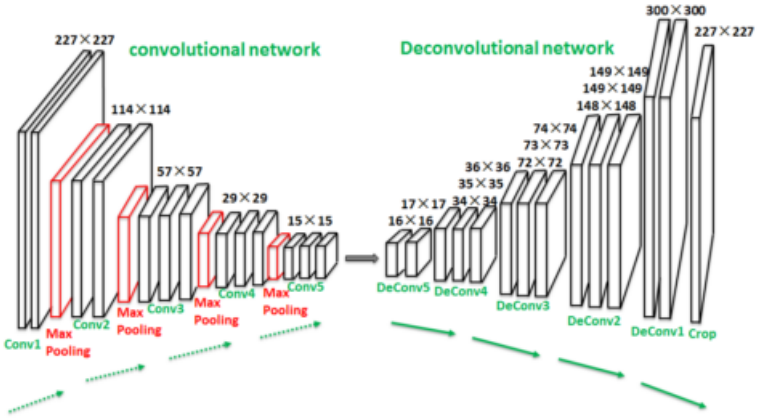


FIGURE 5.2: A generic CAE with only 5 layers of convolution and 4 layers of pooling, one between two layers of pooling, in the encoder and a corresponding asymmetric decoder, the asymmetry leading to an output image larger than the input which is finally center-cropped for further use. (page 2, [58])

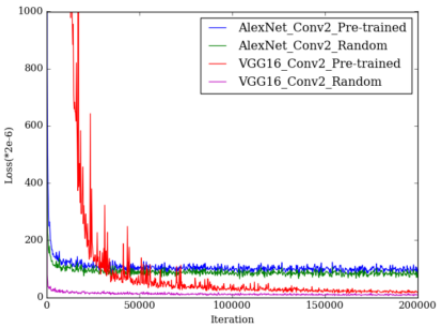


FIGURE 5.3: The training loss on reconstruction from the second layer of AlexNet and VGG-16. Training converges quicker and has lower loss for random encoder. Trend is more apparent with VGG-16. (page 3, [58])

will fare worse in reconstruction. It's shown with the examples of VGG-16 [60] and AlexNet [32] that reconstruction done from a deeper layer of any one of these networks is worse than that from a higher layer, and reconstruction done from a certain layer of VGG-16 is way better than that the reconstruction from a layer of the same depth in the AlexNet because VGG-16 is inherently better than AlexNet in vision tasks. Such a comparison is illustrated in figure 5.4.

The experiments/improvisations carried out to modify/extend the usual RL based architecture search were targeted at exploiting the correlation of the random weight performance and fully end-to-end-trained performance of architectures by using the random weight performance as the reward for the search and then training the top models end-to-end in the end to decide on the ultimate top model. Another intention was to observe the applicability of architecture search in a different task like image reconstruction. Given below is a broad outline of the experiments which will be discussed at length thereafter.

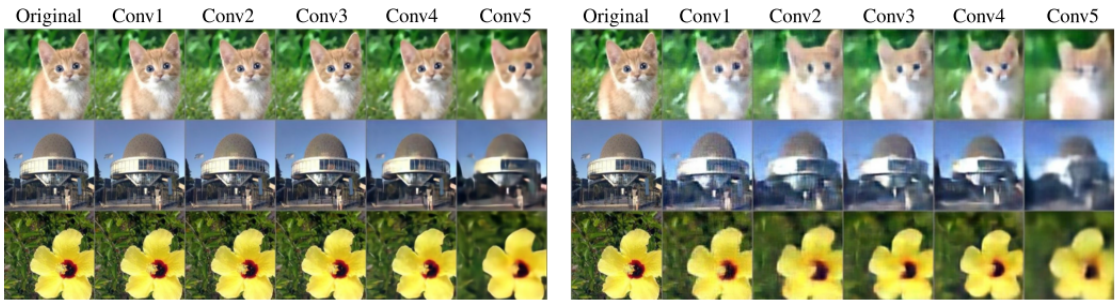


FIGURE 5.4: Reconstruction of images with VGG16 and AlexNet with random weights where the image reconstruction quality deteriorates with the depth and also, the reconstruction from a certain layer in a VGG-16 is better than that in the AlexNet because VGG-16 is inherently better than AlexNet. **Left:** Reconstruction with VGG16. **Right:** Reconstruction with AlexNet. (page 4, [58])

- Training only the classifier and the FC layers on the MNIST dataset while initializing the convolutional layers with random weights (inspired from [54])
  - *Search:* **Training only the FC layers and the classifier during search and using the validation accuracy as the reward**
  - *Evaluation:* **End-to-end training of the top few models for correlation**
- Image reconstruction using symmetric CAE with random encoder and trained decoder on MNIST, STL-10 datasets (inspired from [58, 54])
  - *Search:* **Training only the decoder during search and using the validation reconstruction error as the reward**
  - *Evaluation:* **End-to-end training of the top few architectures for correlation**

5.1 Experiment of Training Only the Classifier and the Fully-connected Layers

This was the first experiment tried out with the sole objective of a fast architecture search if the correlation between the average performance after multiple runs of training only the FC layers and the classifier with random convolutional weight initializations, and the performance after training the same architecture end-to-end was found to exist. The fully-connected and the classifier parts of all candidate architectures were kept exactly same (2 FC layers and an SM classifier) so that their is no uneven bias towards architectures with stronger classifiers. The search space and the other training/validation hyperparameters and settings were kept same as the initial Q-learning architecture search on the MNIST image classification task except for a change in the ϵ-schedule which now meant that the search is going to be carried out among 270

Architecture	**Average Best Top-1 Accuracy**(%)
[C(128,5,1), P(5,2), D(1,2), C(512,1,1), P(3,3), D(2,2), SM(10)]	**97.71**
[C(128,5,1), P(5,2), D(1,2), C(256,5,1), P(3,3), D(2,2), SM(10)]	**97.45**
[C(128,5,1), P(5,3), D(1,2), C(256,3,1), P(2,2), D(2,2), SM(10)]	**97.26**
[C(64,5,1), P(5,3), D(1,1), SM(10)]	**97.08**
[C(512,5,1), P(3,3), D(1,3), C(512,3,1), C(512,1,1), D(2,3), P(3,2), SM(10)]	**96.94**

TABLE 5.1: The architecture configurations and the average of best top-1 validation accuracies from 10 random initializations for MNIST image classification, sorted in the decrease order of accuracies.

architectures. The reduction in the number of candidate architectures was that for checking if the correlation exists, there is no hard requirement of an extensively long search with a single GPU over multiple days. The reward in this case was the average validation performance after using 10 different random weight initializations of the convolutional layers and training only the FC layers and the classifier. To test the existence of a correlation, the top 50 models from this search were completely retrained end-to-end. If there would have been a strong correlation, at least a top few models from among all 50 models shouldn't have changed much in their relative order. But unfortunately, that didn't seem to be the case, and after retraining, the order changed drastically. The results (architectures and rewards in descending order of reward value) from training only the classifier are given in table 5.1[1] and those from training the top 50 models are given in table 5.2[1]. In both the tables, the top 5 models are arranged in the descending order of their corresponding reward (average validation performance from 10 random convolutional weight initializations in table 5.1, and the validation performance after end-to-end training in table 5.2). This particular experiment failed to work because the top architectures found after only training the classifier and the FC layers are shallower, which should not be the trend in the case of image classification, while the top architectures from full end-to-end training are deeper which matches the expected trend.

5.2 Experiment of a Symmetric CAE Search by Training Only the Decoder and Randomly Initializing the Encoder

This experiment was carried out on both STL-10 [11] and MNIST datasets. The STL-10 dataset has 96×96 images and the images and the classes are very similar to the CIFAR-10 dataset, the only difference being the image resolution because in CIFAR-10 dataset, the images are 32×32. The search space and the training/validation settings/hyperparameters for STL-10

Architecture	**Best Top-1 Accuracy**(%)
[C(128,5,1), P(2,2), D(1,6), C(256,3,1), C(256,3,1), D(2,6), P(2,2), C(128,5,1), D(3,6), FC(512), D(4,6), FC(512), D(5,6), SM(10)]	**99.47**
[C(128,5,1), C(64,1,1), D(1,6), C(256,3,1), P(5,3), D(2,6), C(512,3,1), P(3,2), D(3,6), FC(256), D(4,6), FC(256), D(5,6), SM(10)]	**99.42**
[C(64,5,1), C(128,3,1), D(1,5), P(3,2), C(512,5,1), D(2,5), C(128,3,1), P(2,2), D(3,5), FC(128), D(4,5), SM(10)]	**99.40**
[C(128,5,1), P(3,2), D(1,4), C(128,1,1), C(512,5,1), D(2,4), P(5,3), FC(128), D(3,4), SM(10)]	**99.35**
[C(256,3,1), C(512,5,1), D(1,3), P(5,3), C(64,3,1), D(2,3), P(2,2), SM(10)]	**99.32**

TABLE 5.2: The architecture configurations and the best top-1 validation accuracies from end-to-end training of the top 20 models from the search whose results are in table 5.1, sorted in the decreasing order of accuracies. No correlation is seen to exist as the top five models from table 5.1 are nowhere among the top five in table 5.2.

reconstruction and MNIST reconstruction were very similar to those for MNIST classification. One difference for the STL-10 dataset was the size of the convolutional kernels which was now chosen from $[1, 3, 5, 7, 9]$. Also, no padding was used for convolution and no pooling layer was included in any architecture though the search could very easily be made to include the search for pooling hyperparameters by imposing slight constraints on their values. Besides, there was no weight-decay during the optimization step in training for both STL-10 and MNIST. Binary cross entropy (BCE) loss was used as the loss metric. The reward in this case was the inverse of the best/smallest validation reconstruction loss. The encoder was kept absolutely random and the decoder was trained fully. The number of architectures searched over was 75 for both datasets according to the ϵ-schedule of the search. The reason behind a short search was to check if the Q-learning based architecture search actually works in this case. Surprisingly, extremely promising results were obtained from such a short search. For MNIST, the top 20 CAEs were completely trained (both encoder and decoder) while for STL-10, the top 50 were completely trained to check if any correlation exists. The rationale behind the choice of fewer MNIST CAEs for complete retraining in comparison to STL-10 was that STL-10 is a much more complex and diverse dataset for image reconstruction.

Results will be provided only from the CAE search on STL-10 image reconstruction due to the reason of experimental importance and brevity. The results (architectures and rewards in descending order of reward value) from training only the decoder on the STL-10 dataset are given in table 5.3[1] and those from complete retraining of the top 50 models from the above search on STL-10 are given in table 5.4[1] (only the top 5 ar reported).

Architecture	**Inverse of Lowest Reconstruction Loss**
[C(256,1,1), C(256,3,1), SM(10)]	**1.9207**
[C(64,3,1), C(256,3,1), C(256,7,1), C(256,1,1), SM(10)]	**1.9127**
[C(256,3,1), C(64,7,1), SM(10)]	**1.9103**
[C(256,7,1), C(256,5,1), SM(10)]	**1.9086**
[C(64,3,1), C(256,3,1), C(256,3,1), C(256,1,1), C(128,7,1), SM(10)]	**1.9063**

TABLE 5.3: The architecture configurations and the inverse of best (lowest) validation reconstruction loss from training only the decoder for STL-10 image reconstruction, sorted in the decreasing order of accuracy values.

Architecture	**Inverse of Lowest Reconstruction Loss**
[C(256,1,1), C(256,3,1), SM(10)]	**1.9219**
[C(64,3,1), C(256,3,1), C(64,3,1), C(64,1,1), C(128,1,1), C(256,1,1), SM(10)]	**1.9173**
[C(256,3,1), C(64,7,1), SM(10)]	**1.9160**
[C(64,1,1), C(128,3,1), C(128,1,1), C(128,3,1), C(64,7,1), SM(10)]	**1.9159**
[C(256,1,1), C(256,7,1), C(64,1,1), C(64,1,1), C(128,1,1), C(256,1,1), SM(10)]	**1.9156**

TABLE 5.4: The architecture configurations and the inverse of best (lowest) validation reconstruction loss from the complete retraining of top 50 models from the search whose results are in table 5.3, sorted in the decreasing order of accuracy values. The top architecture in table 5.3 is exactly same as that in this table.

First of all, the top model found in training only the decoder is same as the top model found in training both the encoder and the decoder. Next, the top few models from training only the encoder can be found among the top few models after the complete retraining though always not in the same order. Also, the absolute difference of the reconstruction loss values between full training and training of only the decoder is minute despite using BCE loss function which is supposed to have more spread than the other more intuitive loss functions like mean square error (MSE), L1 or L2 loss which resemble the pixel-wise difference of the input and the reconstructed image more closely. Finally, the search actually produces shallow architectures which should be the case for image reconstruction both theoretically and practically. These are a few key inferences that can be drawn from the tables.

The claim of negligible difference in the reconstruction metrics after training only the decoder from those after complete re-training is substantiated by figure 5.6 which shows the reconstructed sets of images by using the top architecture from table 5.3 and table 5.4 which incidentally, are one and the same. Fig 5.5 gives the input set of images for the reconstructions shown in figure 5.6.

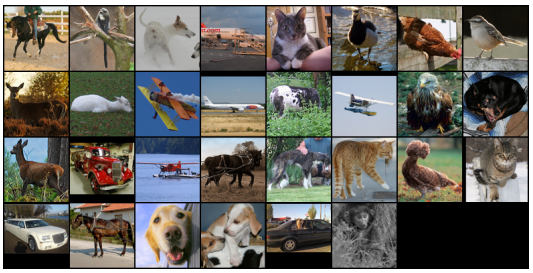
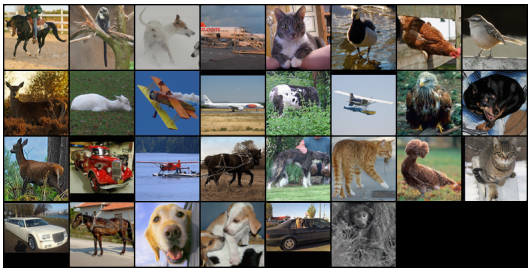
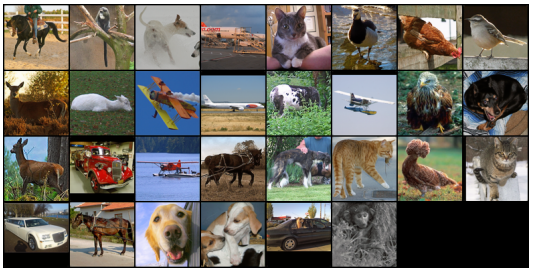


FIGURE 5.5: The input set from the STL-10 dataset containing images of all classes, for reconstruction.



(A) Reconstruction from Random Encoder



(B) Reconstruction from Full Training

FIGURE 5.6: The reconstructed image sets from random encoder CAE and fully-trained CAE where the difference in the quality of the two sets of reconstructed images is barely visible.

The effectiveness of using a trained decoder only for reconstruction was further supported when an interactive t-distributed stochastic neighbor embedding (t-SNE) [38], a low dimensional embedding generation technique for intuitive interpretations, plot where the perplexity value and the learning rate required for the t-SNE algorithm were user inputs, was plotted using the feature map from the last layer of the untrained encoder during reconstruction on the validation set. A sample plot with a perplexity of 30 and a learning rate of 20 is shown in figure 5.7. A few important observations from figure 5.7 which prove that the encoder, despite being untrained, when paired with the trained encoder, is fairly capable of learning image-class specific discriminating features which are extremely important for effective reconstruction, are

- colour based clustering can be seen in the plot
- object and class based clustering to a decent extent can also be noticed

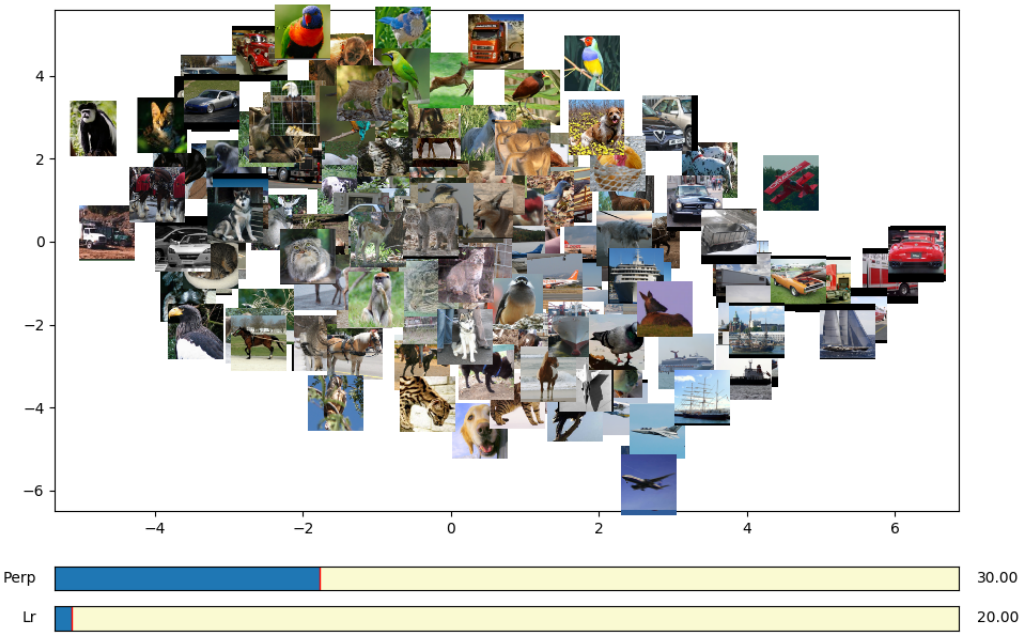


FIGURE 5.7: A sample t-SNE plot by plotting the feature space produced from the last layer of the random-weight encoder of the top model from the reconstruction in table 5.3, with perplexity = 30 and learning rate = 20.

Chapter 6

Architecture Search for Unsupervised Image Generation

This chapter will discuss image generation, the commonly used supervised and unsupervised image generation models, the difficulties associated with using the current image generation models and finally, will focus on the architecture search for unsupervised image generation with VAEs. It will provide justification for choosing unsupervised image generation using VAEs as the task for the architecture search and will highlight the benefits and insights that could possibly be obtained from such an architecture search.

6.1 Image Generation in Machine Learning

Image generation in ML has been a topic of interest for a very long time. The basic objective in image generation is to build an ML model that is able to produce samples that look similar to a certain image data distribution, from random noise samples. The applications of image generation are manifold. It has a significant use in computer graphics and data augmentation among others. For example image generation can be used to create numerous similar looking virtual entities in video games or a realistic piece of art. For example, figure 6.1 shows realistic art samples created using GANs.

It is theoretically well known that any random distribution can be transformed into a particular distribution, the original image data distribution in this case, through Markov Chains (MC) given a broad support of transformation operators and sufficient number of iterations for successive transformations [53]. End-to-end unsupervised neural network models in ML are popular choices for image generation nowadays because of their ability to learn long chains of sequential transformation operators for transforming a random probability distribution/noise to

FIGURE 6.1: Realistic art sample generation using GANs, can be used for aesthetic purposes. [16]

a meaningful data distribution, the target image data distribution in this case, and also due to the convenient end-to-end training capability of neural networks.

6.2 Neural Network Models for Image Generation

In recent times, generative adversarial networks (GAN) [18] and variational autoencoders (VAE) [29] are the most commonly used unsupervised neural network models for image generation. One other type of image generation model is the set of auto-regressive models [19, 44, 45] but auto-regressive models isn't the topic of study in this work, so they won't be described or explained. These models have come to prominence mainly because of their end-to-end training capability.

6.2.1 Image Generation with Generative Adversarial Networks

To put it simply, GANs basically consist of an image generator network and an image discriminator network running in parallel. The generator network tries to transform a random noise, Gaussian noise in most cases, to a target image data distribution that is very similar to the original data while the discriminator network tries to differentiate the generated data distribution from the original data distribution which is a supervised binary classification task. Initially, the generator is untrained and is not good at mapping the random noise to the target data and the discriminator is untrained at accurately classifying the input data as generated data or original image data. Gradually with training both the networks get better at their respective tasks and at convergence, the generator is supposed to produce output data that is very close to the original data.

This is an explanation of the working of a GAN which follows very closely the way of description and explanation in [18]. Here, X represents the original dataset and $p_{data}(x)$ gives the original data distribution. The generator network $G(z;\theta_g)$ is parameterized by θ_g. It takes as input a noise variable $z \sim p_z(z)$ where $p_z(z)$ is a prior distribution over the noise variables, and generates a distribution over data x given by $p_g(x)$. The discriminator network $D(x;\theta_d)$ acts as a data classification network which classifies the input data x as a sample from the original data X or as a sample generated by the generator. It basically outputs a scalar $D(x)$ which gives the probability that sample x comes from the original data rather than the generated data $p(g)$. So, $1 - D(x)$ gives the probability of the sample x coming from the generated data rather than the original data. The task of the generator network is to minimize the log probability of the samples generated when input with random noise samples, being classified as samples from the generated data rather than the original data. On the other hand, the task of the discriminator network is to classify the original data samples and the samples from the generated data as correctly as possible. Thus, the generator network G and the discriminator network D are trained simultaneously to gradually get better at their respective tasks. They essentially play the two-player minimax game with the combined value function $V(D,G)$ as given in equation 6.1

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{6.1}$$

This equation necessarily implies a training procedure as follows. For training a GAN, the discriminator network is trained a certain number of times for every training iteration of the generator though care should be taken that the number of times of training of D for every training step of G isn't such that D overfits the task of discrimination very early in training before the generator even starts to learn generating images that are close to the original data. One more preventive measure for the same problem is to train the generator network to maximize $\log(D(G(z)))$ in place of $\log(1 - D(G(z)))$ because for a decently powerful discriminator $\log(1 - D(G(z)))$ will saturate very quickly and there will be no gradient to train the generator. Figure 6.2 is a representative figure for a GAN. As already explained above, the two difficulties in training GANs are the training of the discriminator for every step of training of the generator which makes the overall training slower, and the instability of the vanilla training procedure because the discriminator has a very high tendency to overfit the binary classification task i.e. it will very accurately tell apart the generated data from the original image data distribution.

6.2.2 Image Generation with Variational Autoencoders

VAEs [29] are one more type of completely unsupervised models for image generation and they have a structure that is very similar to a regular autoencoder that is mostly used for completely unsupervised image reconstruction. The only difference is that the 'bottleneck' of a

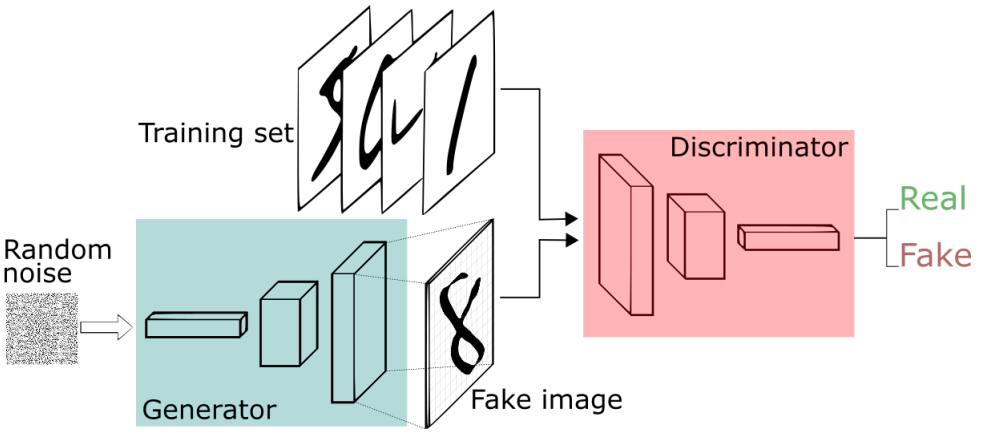


FIGURE 6.2: A rather pedagogical representation of a GAN which explains the ideas of generation and discrimination in a very intuitive way. [12]

VAE, also more commonly referred to as the latent layer, is trained to take the form of a known random distribution besides the task of training the network for a regular image reconstruction. Thus, the loss function for training a VAE is the sum of the regular reconstruction loss and the Kullback–Leibler (KL) divergence of the latent layer distribution from a known random distribution which is the prior in Bayesian inference parlance. VAEs are fully probabilistic models of Bayesian inference for the purpose of image generation. VAEs will be explicitly discussed in chapter 7. **Readers are requested to kindly refer to chapter 7 if they first want a detailed theoretical understanding of VAEs similar to the one on GANs in the previous subsection, before they continue with the rest of this chapter.** The merits and demerits of using VAEs for the image generation purpose will be mentioned in a subsequent section.

6.3 Utility of Architecture Search for Unsupervised Image Generation

GANs are found to be better at image generation than VAEs usually in terms of the visual quality of the generated images as shown in figure 6.3 but VAEs on the other hand are robust and probabilistic models of Bayesian inference which are more stable to train and far more interpretable than GANs as noted in [71, 10, 79]. The quality of the images generated by the VAEs being relatively worse is often attributed to the training of a VAE using the joint/combined sum loss wherein optimization of both losses jointly becomes very difficult which sometimes hurts the quality of the generated images. On the other hand, there is no apparent issue in the training of VAE that might lead to an unstable training unlike GANs which have the problem of the discriminating overfitting it's task of discrimination which leads to an unstable training very often. Besides, VAEs produce disentangled feature representations in the latent layer which are

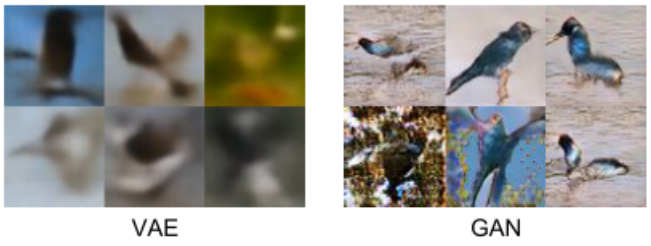


FIGURE 6.3: Comparison between complex 64×64 bird images generated by a VAE and those generated by a GAN. The difference in quality is well evident. [57]

of prime significance from the perspective of image generation. The disentanglement of features heavily depends on the the size of the latent space which should not be too high or too low. A very low latent size helps the model to overfit the task of optimizing the KL divergence of the approximated posterior from the prior which hurts the task of reconstruction while a very high latent size helps the task of reconstruction but hurts badly the task of optimizing the KL divergence of the approximated posterior from the prior. Besides, VAEs can be used in semi-supervised settings wherein they give highly competitive performance with the use of fewer training samples as seen in figure 6.2 where N gives the number of training samples, $M1$ represents a VAE for semi-supervised classification and $M2$ represents a VAE which uses the latent vector for classification during the same task of semi-supervised classification. As seen in the figure, the VAE performs better than other models for the same number of samples when they are fewer in number.

In spite of these advantages, the training of a VAE is quite difficult in general because of the combination of two loss functions which are very different both nature wise and scale wise. Usually, a VAE network ends up overfitting either of the two tasks of optimizing the error for image reconstruction and optimizing the KL divergence of the approximated posterior (encoder output distribution) from the prior (known random distribution). This makes an RL based search on VAE models with the search reward being formulated using the vanilla loss function, very difficult. Here, the commonly used loss function from [29] for the training of VAEs is being referred to as the vanilla loss and this will be explicitly derived in the next chapter (chapter 7). The reward function which is very important for RL based searches can't be formulated from the vanilla validation loss at convergence because the vanilla validation loss doesn't decrease with training and hence, doesn't reflect the true image generation potential of the trained network. Thus, not only individual VAEs are very difficult to train because of the loss function nature, an RL based search over such vanilla models is also extremely difficult because of the difficulty in reward function formulation again due to the nature of the validation loss.

One of the possible ways of dealing with the above mentioned problems is to reformulate the loss function without tampering the underlying mathematics so that the individual VAE

models can be trained more efficiently and also, so that the reward can be formulated for an RL based search. A successful RL based architecture search on VAEs with the reward formulation from the reformulated loss function will emphasize on the importance of reward formulation that is necessary for an RL based search algorithm like MetaQNN, REINFORCE etc. to work.

Also, the latent layer can be included in the search space and a successful search on top of it may generate the 'optimal' or the better latent sizes for a particular data distribution which may in a way reflect the complexity of the original data distribution that is being worked with.

So, the benefits of moving away from the vanilla formulation which is probably just one of the many possibilities of dealing with the current problems, are

- reformulation of the loss might lead to better training of individual VAE models
- it might also help in formulating a reward function which truly reflects the image generation potential of VAEs which would in turn result in a good RL based search
- such a search can also be used to search over the better or the so-called 'optimal' latent sizes for a particular data distribution

Table 1: MNIST Results

Model	N=100	N=500	N=1000	N=2000	N=5000
PCA + SVM	0.692	0.871	0.891	0.911	0.929
CNN	0.262	0.921	0.934	0.955	0.978
M1	0.628	0.885	0.905	0.921	0.933
M2	•	•	0.975	•	•

FIGURE 6.4: Table showing relative performance of different models on a semi-supervised classification task Using different amounts of training data. [27]

Chapter 7

Variational Autoencoders

Variational autoencoders (VAE) are networks for Bayesian inference and are commonly used for the purpose of image generation and for learning disentangled encoded feature representation for the purpose of semi-supervised learning. Disentanglement of features is the phenomenon in which a particular modality in the input image among its different modalities like colour, texture etc., activate a particular neuron or a particular set of neurons in a certain layer while the other units or groups are activated to a much lesser extent [22]. Disentanglement can help in better image generation and also for generating useful features for semi-supervised learning. They try to approximate the posterior distribution of the original image data distribution by minimizing the KL divergence of the approximated posterior from the original posterior which can be exploited for the purpose of image generation. This is achieved by jointly maximizing the expectation of the log-likelihood by minimizing the reconstruction error, and minimizing the KL divergence of the approximate posterior from the prior (known random distribution, to be further used for generating images). Once the VAE network is trained using this joint loss function, at convergence it's supposed to generate images close enough to the original image data distribution when input with random samples from the prior. The mathematical formulation of the VAE loss function and the reparameterization technique which is characteristic of VAEs will be discussed in detail. Both the discussions on the mathematical formulation and the reparameterization have been adopted from [29] and [14].

7.1 Mathematical Formulation of Variational Autoencoders

The original image data is represented by $X = \left\{x^{(i)}\right\}_{i=1}^{N}$ consisting of N i.i.d samples of continuous or discrete variable x. It's assumed that the data are generated by some random process using an unobserved continuous random variable z. The process is assumed to consist of the following two steps

- a sample of $z^{(i)}$ is produced from some prior distribution $p_{\theta^*}(z)$ or some arbitrary prior distribution $p(z)$
- a value $x^{(i)}$ is produced using a conditional probability distribution $p_{\theta^*}(x|z)$

Here, the prior distribution $p_{\theta^*}(z)$ and the likelihood $p_{\theta^*}(x|z)$ are assumed to originate from parametric families of distributions $p_\theta(z)$ and $p_\theta(x|z)$ respectively. They are also assumed to be differentiable almost everywhere with respect to θ and z. The marginal likelihood of X can be written as

$$p(x) = \int_z p_\theta(z)p_\theta(x|z)dz \tag{7.1}$$

But, unfortunately the marginal likelihood is intractable which further results in an intractable posterior distribution $p_\theta(z|x)$ according to the Bayes rule as given in the equation below

$$p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)} \tag{7.2}$$

Sampling based methods like Monte Carlo expectation maximization (EM) that uses Markov models, MCMC etc.can be used to go around the problem of intractability but they are too slow as they require extensive iterative sampling for every datapoint. VAEs solve this problem of intractable Bayesian inference in a unique way.

VAEs in a way imitate the sampling of a random variable from the prior and generating data similar to the original data distribution X by exploiting the conditional distribution/likelihood. Thus, they result in approximate inference of the marginal likelihood of the data $p_\theta(x)$. They also produce disentangled feature representations in the latent layer to facilitate image generation by approximately inferring the posterior $p_\theta(z|x)$ of the latent variable z given an original data sample x and the encoder parameters during training. The recognition model for producing the approximate posterior is represented by $q_\phi(z|x)$ which is nothing but the decoder part of the VAE. The training of a VAE ensures the joint training of the recognition parameters ϕ and the generative parameters θ. In practice, given an original data sample $x \epsilon X$ a probabilistic encoder $q_\phi(z|x)$ in a VAE produces a distribution over the latent variable z from which x can be generated back and the probabilistic decoder $p_\theta(x|z)$ produces a distribution over the corresponding values of x that are very similar to the original input sample, given a certain latent vector z.

To do the same it aims at minimizing the KL divergence measure from the true posterior $p_\theta(z|x)$ to the approximated posterior $q_\phi(z|x)$ which is expressed as $D_{KL}(q_\phi(z|x)||p_\theta(z|x))$. The same KL divergence can be expanded and the marginal likelihood can be expressed as follows in

the immediately next set of expressions and equations.

$$\begin{aligned} D_{KL}(q_\phi(z|x)||p_\theta(z|x)) &= \mathbb{E}_{q_\phi(z|x)}\left[\log q_\phi(z|x) - \log p_\theta(z|x)\right] \\ &= \mathbb{E}_{q_\phi(z|x)}\left[\log q_\phi(z|x) - \log\left(\frac{p_\theta(z,x)}{p_\theta(x)}\right)\right] \\ &= \mathbb{E}_{q_\phi(z|x)}\left[\log q_\phi(z|x) - \log\left(\frac{p_\theta(x,z)}{p_\theta(x)}\right)\right] \\ &= \mathbb{E}_{q_\phi(z|x)}\left[\log q_\phi(z|x) - \log p_\theta(x,z)\right] + \log p_\theta(x) \\ &= \mathbb{E}_{q_\phi(z|x)}\left[\log q_\phi(z|x) - \log(p_\theta(x|z)p_\theta(z))\right] + \log p_\theta(x) \\ &= \mathbb{E}_{q_\phi(z|x)}\left[\log q_\phi(z|x) - \log p_\theta(x|z) - \log p_\theta(z)\right] + \log p_\theta(x) \end{aligned} \tag{7.3}$$

$$\begin{aligned} \Rightarrow \log_\theta p(x) &= D_{KL}(q_\phi(z|x)||p_\theta(z|x)) + \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x) - \log p_\theta(z)] \\ &= D_{KL}(q_\phi(z|x)||p_\theta(z|x)) + \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}(q_\phi(z|x)||p_\theta(z)) \\ &\geq \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}(q_\phi(z|x)||p_\theta(z))}_{\text{Evidence Lower Bound (ELBO)}} \end{aligned} \tag{7.4}$$

Since the KL divergence term $D_{KL}(q_\phi(z|x)||p_\theta(z|x))$ is non-negative, maximizing the marginal likelihood of the original data distribution $\log p_\theta(x)$ implies maximizing the evidence lower bound (ELBO) as referred to in [29]. Maximizing the ELBO requires minimizing the KL divergence $D_{KL}(q_\phi(z|x)||p_\theta(z))$ from the prior $p_\theta(z)$ to the approximate posterior $q_\phi(z|x)$ and maximizing the expectation over the log-likelihood $\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]$. The ELBO is represented by $\mathcal{L}(\theta, \phi; x)$. Hence,

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}(q_\phi(z|x)||p_\theta(z)) \tag{7.5}$$

One problem with this expression is the difficulty in computing the gradients of the ELBO with respect to ϕ because of the expectation over the log-likelihood. The naive Monte Carlo gradient estimator is given in the equation below

$$\begin{aligned} \nabla_\phi \, \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] &= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z) \nabla_{q_\phi(z|x)} \log q_\phi(z|x)\right] \\ &\simeq \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(x|z^{(l)}) \nabla_{q_\phi(z^{(l)}|x)} \log q_\phi(z^{(l)}|x) \end{aligned} \tag{7.6}$$

where $z^{(l)} \sim q_\phi(z|x)$. But this is a very high variance Monte Carlo estimate although it is unbiased and is generally not used. Therefore, for choosing a low variance unbiased estimate for the ELBO the reparameterization trick is used which is a characteristic feature of VAEs.

7.2 Reparameterization Trick for Variational Autoencoders

With the objective of sampling from an approximate posterior $q_\phi(z|x)$, the random variable $z^{(l)} \sim q_\phi(z|x)$ is generated using a differential transformation $g_\phi(\epsilon, x)$ of a noise variable ϵ which acts as an auxiliary variable such that

$$\tilde{z} = g_\phi(\epsilon, x) \text{ where } \epsilon \sim p(\epsilon) \tag{7.7}$$

Here, $p(\epsilon)$ is the independent marginal distribution over ϵ. Now, the Monte Carlo estimate for computing the ELBO with respect to $q_\phi(z|x)$ is given by

$$\begin{aligned} \mathbb{E}_{q_\phi(z|x)}\left[p_\theta(x|z)\right] &= \mathbb{E}_{p(\epsilon)}\left[\log p_\theta(x|g_\phi(\epsilon, x))\right] \\ &\simeq \frac{1}{L}\sum_{l=1}^{L} \log_\theta(x|g_\phi(\epsilon^{(l)}|x)) \text{ where } \epsilon^{(l)} \sim p(\epsilon) \end{aligned} \tag{7.8}$$

After this given reparameterization, the new ELBO without the KL divergence from the prior, called as the Stochastic Gradient Variational Bayes (SGVB) estimator in [29], which is denoted by $\tilde{\mathcal{L}}^A(\theta, \phi; x)$ can be formulated as follows

$$\tilde{\mathcal{L}}^A(\theta, \phi; x) = \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(x, z^{(l)}) - \log q_\phi(z^{(l)}|x) \tag{7.9}$$

where $z^{(l)} = g_\phi(\epsilon^{(l)}, x)$ and $\epsilon^{(l)} \sim p(\epsilon)$. This estimate is approximately equal to the original form of the ELBO without the KL divergence from the prior. If the KL divergence $D_{KL}(q_\phi(z|x)||p_\theta(z))$ from the prior to the approximated posterior is also subtracted from the SGVB estimate of the ELBO, it yields a second version where the KL divergence term acts as a regularizer on the encoder parameters ϕ thus forcing the approximated posterior from the encoder to take the form of the prior albeit maintaining the required correlation among the encoded features of x which is very necessary for a good reconstruction. The second version of the SGVB estimate (equation 7.10) for the ELBO denoted by $\tilde{\mathcal{L}}^B(\theta, \phi; x)$ which is approximately equal to the ELBO and hence, unbiased but has way less variance than the original ELBO is given by

$$\tilde{\mathcal{L}}^B(\theta, \phi; x) = -D_{KL}(q_\phi(z|x)||p_\theta(z)) + \frac{1}{L}\sum_{l=1}^{L}(\log p_\theta(x|z^{(l)})) \tag{7.10}$$

where $z^{(l)} = g_\phi(\epsilon^{(l)}, x)$ and $\epsilon^{(l)} \sim p(\epsilon)$.

Usually the approximated posterior is made to take the form of a Gaussian over z for every x such that $z \sim p(z|x) = \mathcal{N}(\mu, \sigma^2)$. The valid reparameterization for this case is $z = \mu + \sigma\epsilon$ where the auxilary noise variable $\epsilon \sim \mathcal{N}(0, 1)$. Therefore, the expectation over the log-likelihood

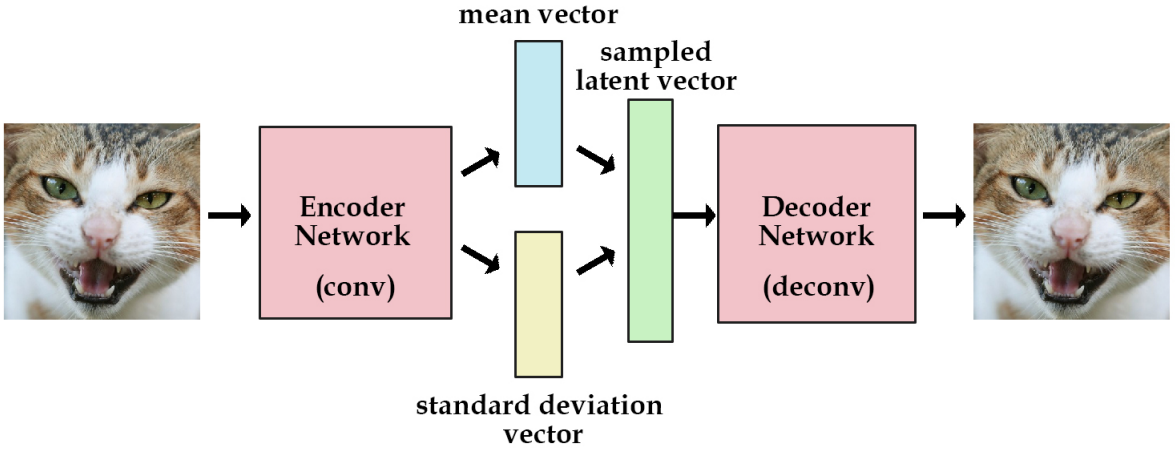


FIGURE 7.1: A sample VAE with reparameterization is shown with an input image and the reconstructed output. For the reparameterization, the last output feature map of the encoder is connected to a 'mean' vector and a 'standard deviation' vector which together learn an approximate posterior distribution of a Gaussian form. A sample from the Gaussian is drawn using the noise variable as shown in equation 7.11. ([33])

becomes

$$\begin{aligned}\mathbb{E}_{\mathcal{N}(z;\mu_\phi(x),\sigma^2_\phi(x))}\left[\log p_\theta(z|x)\right] &= \mathbb{E}_{\mathcal{N}(\epsilon;0,1)}\left[\log p_\theta(x|(\mu_\phi(x)+\sigma_\phi(x)\epsilon))\right] \\ &\simeq \frac{1}{L}\sum_{l=1}^{L}\log p_\theta(x|(\mu_\phi(x)+\sigma_\phi(x)\epsilon^{(l)})) \text{ where } \epsilon^{(l)} \sim \mathcal{N}(0,1)\end{aligned} \tag{7.11}$$

In this case, the negative of the KL divergence $D_{KL}(q_\phi(z|x)||p_\theta(z))$ of the prior to the approximated posterior for the second version of the SGVB (equation 7.10) can be computed as follows

$$\begin{aligned}-D_{KL}(q_\phi(z|x)||p_\theta(z)) &= \int q_\phi(z|x)(\log p_\theta(z) - \log q_\phi(z|x))dz \\ &= \frac{1}{2}\sum_{j=1}^{J}(1+\log\big((\sigma_{\phi,j}(x))^2\big) - (\mu_{\phi,j}(x))^2 - (\sigma_{\phi,j}(x))^2)\end{aligned} \tag{7.12}$$

where J is the dimensionality of the 'bottleneck' or the latent layer of the VAE. A sample VAE which uses the reparameterization trick is shown in figure 7.1.

The maximization of the mean over log-likelihood means the minimization of the log-likelihood reconstruction error between the reconstructed image output and the input image sample. The log-likelihood reconstruction loss is approximated by the reparameterization and sampling technique as explained above. This is exactly same as the reconstruction loss in an autoencoder. Practically, the first term of the joint loss for a VAE is the negative of the KL divergence from

the prior to the posterior acts as regularizer while the second term is the reconstruction error between the reconstructed output and the input sample. Thus, the optimization of the VAE essentially involves minimization of the sum of a reconstruction loss and a KL divergence term.

The actual optimization isn't very easy due to the following two factors apparently. Firstly, the two loss terms are not always on scale. Secondly, the loss terms are very different nature wise because one is usually a binary cross-entropy loss (the log-likelihood reconstruction loss) and the other is a KL divergence term which result in the optimization spaces of the individual losses being very different. Usually, the plain addition of these two losses results in the VAE network overfitting the KL loss very quickly for relatively more complex image datasets like CIFAR-10, STL etc. and thus, it can no more optimize the reconstruction loss. This is because fitting a prior which is usually a Gaussian is very easy for a deep VAE network while the process of reconstruction which involves the learning of meaningful correlated features is usually slower and more complicated. Different solutions to this have been proposed in literature. To the best of the author's knowledge, almost all of them involve weighing the two loss terms differently before adding them. The most significant one among them is the ***'warm-up'*** [78, 63]. The 'warm-up' involves the gradual increasing the weight on the KL divergence term to 1 as training proceeds so that the VAE network doesn't end up overfitting the of the latent layer to the prior distribution during the initial stages of training when the VAE is not good at reconstruction. This means that the weight on the KL term will start with small values and will be increased gradually according to a certain schedule. More evidence and more discussion on this phenomenon will be provided in the chapters following the next chapter on the reformulation of the ELBO. One other point to be noted is that the differential weighing of the two loss terms in 'warm-up' or in the ***'β-VAE'*** [22] for the disentanglement of the latent features, doesn't allow the new loss formulation with weights, to be the same as the original ELBO or the second version of the SGVB anymore (equation 7.10). This point is being highlighted because the reformulation technique to be proposed in the next chapter will also be along the lines of not retaining the exact ELBO for practical benefits. The version of the VAEs with the plain sum of the reconstruction loss and the KL divergence as the combined loss will be referred to as vanilla variational autoencoders (vanilla VAE) here onwards. In the vanilla VAE, the combined loss term for practical purposes that needs to be minimized for a pair (x, x') of a random input sample and the reconstructed output where x' is the reconstructed sample is as follows

$$\mathcal{J}(x, x', p(z); \theta, \phi) = BCE(x, x') + D_{KL}(q_\phi(z|x)||p(z)) \tag{7.13}$$

where $BCE(x, x')$ denotes the binary cross-entropy loss between x and x', and $\mathcal{J}(x, x', p(z); \theta, \phi)$ is the combined loss term. Here, a parameter independent prior $p(z)$ is used for convenience in sampling while image generation. The 'warm-up' and the 'β-VAE' versions have the combined

loss function as follows

$$\mathcal{J}(x, x', p(z); \theta, \phi) = BCE(x, x') + \omega * D_{KL}(q_\phi(z|x)||p(z)) \tag{7.14}$$

where ω is the weight/regularization factor which weighs the two loss terms differently. In 'β-VAE', the ω values are usually on the higher side (values between 10 and 500) to efficiently disentangle the encoded features in the latent space while in 'warm-up', ω follows a certain schedule during the training of the VAE and is usually increased from 0.1 to 1 in steps. The purpose of 'β-VAE' and 'warm-up' are completely different. While 'β-VAE' aims at producing disentangled representations for generating crisp representative images, 'warm-up' helps the VAE in optimizing the two losses of the combined loss equally well.

Chapter 8

Reformulation of the Loss Function from Vanilla VAE

The difficulties in the optimization of the vanilla VAE as mentioned in the last chapter led to the differential weighing ('warm-up', 'β-VAE'). Despite being a possible method to tackle the optimization problem, it's not possible to implement the different methods of weighing the loss terms differently because the schedule in 'warm-up' and the value of β in the 'β-VAE' can be both very task-specific and very architecture-specific. One option is to also include the 'weight' factor for the KL divergence of the approximated posterior from the prior (in β-VAE and warm-up) in the search space but in that case for a short and quick search (searching over a number of architectures where the number is usually between 50 and 100) a lot of architectures will be such that they will have a completely wrong value of the weight and will not train at all. Such architectures will not contribute at all to the search and the search might come up with very bad architectures. Besides, the weight factor is also dataset dependent and doing a search on a new dataset will require the practitioner to set a range of values for the weight that is well-suited for that dataset. This in turn requires the practitioner to have an initial about the nature of the dataset which to a certain extent defeats the purpose of an automatic architecture search. So, for the purpose of an architecture search over different datasets (MNIST, CIFAR-10) for image generation with VAEs, using such a task and architecture dependent schedule or value is being avoided here. But on the other hand, maintaining the exact second version of the SGVB (equation 7.10) or the ELBO doesn't help the optimization at all in many cases which will be discussed in the next chapter. This necessitated the exclusion of any weighing factor and the reformulation of the loss function (the second version of the SGVB in equation 7.10) in such a way that the two loss terms are easier to be optimized jointly. Basically, the reconstruction loss which is to be minimized along with the KL divergence to maximize the expectation over the log-likelihood, was reformulated as a KL divergence term which conveyed the same logical information as the original reconstruction/binary cross-entropy loss although it didn't retain the

ELBO as it is. A fool-proof mathematical link between the original loss and the reformulated loss is yet to be established which will further substantiate the fact that experimental results with the reformulated loss function are often better for individual VAE networks. These results and the comparisons will be provided in detail in the next chapter.

8.1 The Reformulated ELBO

As mentioned in the previous chapter, $x \epsilon X$ is a sample from the original high dimensional data distribution. For the task of image generation, X is the distribution over the images from a particular image dataset like MNIST,CIFAR-10, STL etc. The sample member x has a certain dimensionality and any operation or function over the individual dimensions of x is denoted with an extra asterisk over the symbol for the sample. For eg. the individual dimension wise sum of two high dimensional samples x and y from X will be given by $x^* + y^*$ in place of $x + y$. This notation will be used for the purpose of reformulating the loss function.

In the reformulation, the reconstruction loss is replaced by a KL divergence term from a Gaussian distribution of 0 ***mean and a very small standard deviation of*** $\mathcal{E}$ ***where*** $\mathcal{E} \rightarrow 0$***, to the distribution over the plain pixel wise difference of a particular original data sample and the reconstructed version. A high dimensional Gaussian of*** 0 ***mean and a very small standard deviation essentially implies that with high probability the individual dimensions of a high dimensional sample from the same distribution will be very close to*** 0***. In this context, it essentially means that the pixel wise difference of the original data sample and the reconstructed version will be forced to take a Gaussian with*** 0 ***mean and a very small standard deviation which in turn implies that each pixel wise difference will be individually close to*** 0***. This directly implies that minimizing the reformulated KL term of the pixel-wise error distribution for a particular image sample from the Gaussian with*** 0 ***mean and negligible standard deviation, will result in the minimization of the binary cross-entropy reconstruction loss.***. $\mathcal{E}$ can be treated as hyperparameter during the experiments and is always set to 0.01 during all the experiments that are mentioned in the next chapter.

Optimizing the sum of two KL divergence terms is similar optimizing the loss in equation 7.14. While the second KL divergence term i.e. the KL divergence of the approximated posterior from the prior, is being retained the binary cross-entropy reconstruction loss in equation 7.14 is being replaced with a KL divergence term which is not same as the BCE loss but it conveys a similar meaning and is highly correlated with the BCE loss. One important point to be noted in this reformulated loss is that both the KL divergences are normalized i.e. the KL divergence of the dimension wise difference between the input sample and the reconstructed output from

the Gaussian of 0 mean and a standard deviation of $\mathcal{E}$, is scaled down by the dimensionality of the input/reconstructed output and the KL divergence of the approximated posterior from the prior is scaled down by the dimensionality of the latent vector, J (equation 7.12). The loss in equation 7.14 when reformulated becomes as follows

$$\mathcal{J}'(x, x', p(z); \theta, \phi) = \underbrace{D_{KL}((x^* - x'^*)||\mathcal{N}(0, \mathcal{E}))}_{\text{Reformulated portion}} + D_{KL}(q_\phi(z|x)||p(z)) \tag{8.1}$$

where $\mathcal{E} \to 0$ and $\mathcal{J}'(x, x', p(z); \theta, \phi)$ is the reformulated loss. The loss in equation 7.14 is being rewritten here for easy reference and visual comparison.

$$\mathcal{J}(x, x', p(z); \theta, \phi) = \underbrace{BCE(x, x')}_{\text{Changed to KL divergence in equation 8.1}} + D_{KL}(q_\phi(z|x)||p(z)) \tag{8.2}$$

Also, a graphic representation of the reformulation, similar to the figure 7.1 but showing the loss reformulation in addition to that, is given in figure 8.1.

It was reasoned that reformulating the loss this way with normalization has the following advantages. First of all, both the loss terms are KL divergences which makes them computationally and intrinsically similar that might lead to similar optimization spaces for the individual losses and hence might help the optimization of the combined loss. Secondly, the normalization might bring the two KL divergences on the same scale since the normalization averages the both the KL divergences over the dimensionality of the feature/pixel-spaces for which they are computed. These intuitions are further supported by the usual better performance in terms of generation, of individual VAE networks when used with this particular formulation of the loss function as will be seen in the next chapter on results.

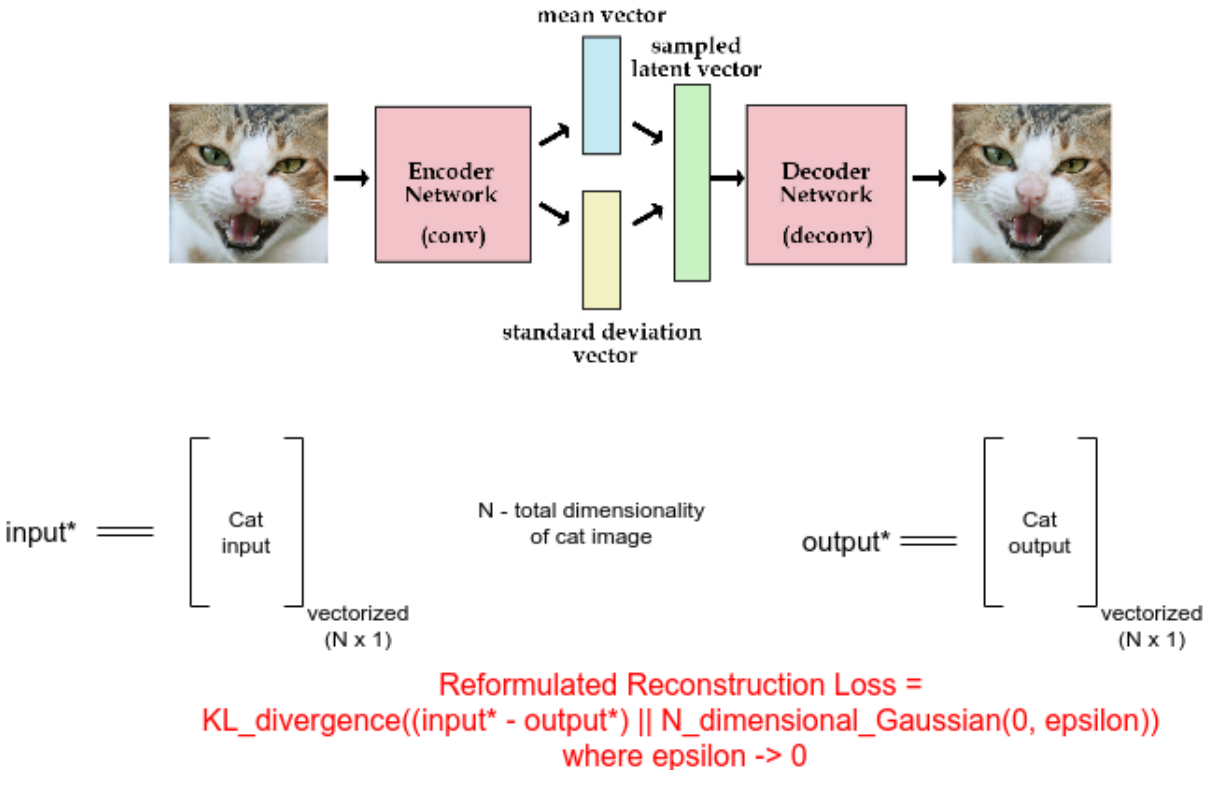


FIGURE 8.1: A sample VAE with reparameterization and reformulation of the loss function for training, is shown with an input image and the reconstructed output ([33]). Below that, a representative expression for the reconstruction loss after the reformulation is also being shown. For computing the reformulated loss, the input image and the reconstructed image are reshaped into a one dimensional array where the number of elements in the array is N (N is the total image dimensionality) and the KL divergence of the element wise difference of the input and the reconstructed output from an N dimensional Gaussian with 0 mean and very small standard deviation, is used to replace the BCE reconstruction loss. For example, if a dataset with $28 \times 28 \times 3$ images is being used, the input and the reconstructed output are reshaped into 2352×1 vectors and their element wise difference of dimensionality 2352×1 is used to compute the KL divergence from the 2352×1 Gaussian of 0 mean and $\mathcal{E}$ standard deviation where $\mathcal{E} \to 0$.

Chapter 9

Random Vanilla VAE vs Reformulated VAE with Same Architecture for MNIST

Before going into comparing results from the architecture search with the vanilla formulation and with reformulation on different datasets (CIFAR-10, MNIST), the training and validation performance of a random VAE architecture with and without reformulation are being provided as a proof of concept. Here, MNIST was used as the dataset for training and validation.

The random architecture with a randomly picked latent size of 5 used with or without reformulation was as given in table 9.1. For both the vanilla formulation and the reformulation, the architecture was trained and validated for 20 epochs and the following metrics were printed out during training and validation

- **Reformulation**
  - **Training/Validation:** *reformulated reconstruction loss, corresponding BCE loss, KL divergence from prior*
- **Vanilla Formulation**

Encoder	**Latent Size**	**Decoder**
[C(64,6,2), C(128,4,2), C(256,2,2), C(512,2,2)]	5	[C(128,8,2), C(256,2,2), C(512,4,2)]

Table 9.1: A random architecture with a latent size of 5 used on MNIST with vanilla formulation and with reformulation.

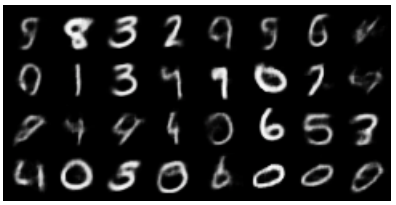


FIGURE 9.1: MNIST images generated by the VAE in table 9.1 in the 20th validation epoch when trained with the reformulated loss.

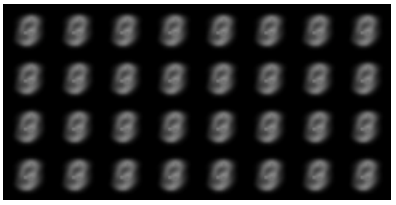


FIGURE 9.2: MNIST images generated by the VAE in table 9.1 in the 20th validation epoch when trained with the vanilla loss.

– **Training/Validation:** *BCE loss, KL divergence from prior*

The **loss metrics from every** 5 **epochs are summarized in table 9.2 for the training/-validation with reformulation** and **those for the vanilla formulation are presented in table 9.3.**

Figure 9.1 contains a set of 32 **generated images in the last validation epoch with random samples from a unit Gaussian as inputs to the network with reformulation. Figure 9.2 presents a set of** 32 **generated images in the last validation epoch by the same network network but with the vanilla formulation.**

As apparent from the figures and the tables, for a randomly selected VAE architecture with a randomly picked latent size of 5, the reformulation appears to work better than the vanilla formulation for the purpose of image generation. In the vanilla formulation, the netowrk overfits the task of optimizing the KL divergence from the prior and fails to optimize the reconstruction loss at all. This experiment acts a proof of concept and creates a base to perform the subsequent architecture search experiments.

Epoch	Train/Val	Reformulated Reconstruction Loss	BCE	KL
1	Training	380.058	0.471	3.360
	Validation	262.645	0.221	2.340
5	Training	127.006	0.146	4.027
	Validation	124.518	0.143	4.128
10	Training	111.456	0.135	3.212.
	Validation	111.097	0.134	3.155
15	Training	103.720	0.130	3.006
	Validation	105.758	0.130	3.028
20	Training	99.188	0.127	2.983
	Validation	103.134	0.129	3.101

TABLE 9.2: Training/validation metrics of every 5th epoch with reformulation on MNIST

Epoch	Train/Val	BCE	KL
1	Training	0.293	0.073
	Validation	0.263	$\sim 10^{-4}$
5	Training	0.263	$\sim 10^{-6}$
	Validation	0.263	$\sim 10^{-6}$
10	Training	0.263	$\sim 10^{-7}$
	Validation	0.263	$\sim 10^{-7}$
15	Training	0.263	$\sim 10^{-7}$
	Validation	0.263	$\sim 10^{-7}$
20	Training	0.263	$\sim 10^{-7}$
	Validation	0.263	$\sim 10^{-7}$

TABLE 9.3: Training/validation metrics of every 5th epoch with vanilla formulation on MNIST

Chapter 10

Comparison of the Results from Architecture Searches with the Vanilla Loss and the Same with the Reformulated Loss

This chapter contains the comparative study between the performance of architecture searches with the vanilla loss (equation 7.14) with that of architecture searches with the reformulated loss (equation 8.1) both in terms of visualization of the generated images and the latent embeddings, and in terms of numerical metrics (reconstruction loss, KL divergence, search reward etc.). An important point to be noted is that all the architecture search experiments employed the MetaQNN algorithm (algorithm 4) because it was a functional baseline algorithm that performed decently well for the initial experiments on completely supervised classification and completely unsupervised reconstruction tasks. Also, this chapter mentions in detail the different methods of formulation of the reward from the loss metrics for different searches before going into performance comparison.

10.1 Different Methods of Reward Formulation

Altogether, two different methods were used to formulate the reward for the different search experiments. In each of the search experiments, the combined/sum loss, the vanilla sum or the sum after reformulation, was used to train the candidate networks but the reward for the search was formulated in two completely different ways. It was experimentally observed that depending on the task/image dataset, a few VAE architectures would optimize the reconstruction loss

well while a few others would perform well on the task of optimizing the KL divergence of the approximated posterior from the prior. In some cases, the architecture absolutely overfit the task of optimizing the KL divergence of the posterior from the prior. Taking into consideration all these possible cases, the two different intuitive but somewhat empirical reward formulations were used. It was observed that in order to generate images that look decently similar to the original image data distribution a candidate architecture should be able to optimize both the losses from the combined loss moderately well without overfitting either of them. For generating images that are very close to the original image data, the architecture should be able to optimize both the losses very well.

The first method used the inverse of the lowest combined/sum loss, vanilla or reformulated, on the validation dataset from among all the validation epochs as the reward. This method was the more obvious method between the two and was exactly same as the reward formulation for the completely supervised classification tasks and the completely unsupervised reconstruction tasks from the initial experiments wherein the inverse of the average validation classification accuracy and the inverse of the average validation reconstruction loss were used as the respective rewards. A search with such a reward formulation was expected to work well if for the majority of the candidate architectures the two losses of the combined loss were optimized more or less equally well i.e. both losses more or less went down with training.

The second method used the inverse of the validation KL divergence from the validation epoch that gave the lowest combined loss, vanilla or reformulated, on the validation dataset, as the reward. This would work well if most of the architectures from the task would be able to decently perform on the reconstruction task or overfit the same but would not be able to optimize the the KL divergence of the approximated posterior from the prior well enough. The rationale behind such a search which uses the KL divergence as the reward is two-fold. First of all, the tendency to overfit the KL divergence of the approximated posterior from the prior, depends very much on the dataset and hence, a search with this reward might provide additional insight. Secondly, a VAE with reformulation of the combined loss function no longer overfits the KL divergence but for a good image generation a good optimization of the KL is required and hence, the search using such a reward might come up with architectures that optimize the KL divergence very well in addition to the combined loss optimization and are very good at generating images.

Finally, there are four different search methods which are to be noted carefully by the author for understanding the results in the subsequent sections:

1.
   - **Training/Validation Loss:** *Reformulated loss*
   - **Reward:** *Inverse of lowest validation combined loss*

2. • **Training/Validation Loss:** *Vanilla loss*
   • **Reward:** *Inverse of lowest validation combined loss*
3. • **Training/Validation Loss:** *Reformulated loss*
   • **Reward:** *Inverse of validation KL divergence from the prior, of validation epoch with lowest validation combined loss*
4. • **Training/Validation Loss:** *Vanilla loss*
   • **Reward:** *Inverse of validation KL divergence from the prior, of validation epoch with lowest validation combined loss*

Through the results from the architecture search , the current methods of reward formulation don't work well and the top architectures produced definitely are not the best among all the architectures that the search comes across. Also, the top architectures do not have latent sizes that could in any way represent the dataset complexity i.e. the latent sizes of the top networks are such that only the rewards per se are maximized. This again indicates the failure of lack of use of the current method of formulating the reward. The overall decent performance of the handpicked architectures supports the author's claim of the reformulated loss function being useful and the methods of reward formulation not being of great use. On the other hand, this led to the comparison of not only the performance of the top architectures but also the performance of a few 'cherry-picked' or handpicked architectures. More discussion and analysis is included below.

10.2 Performance Comparison for Image Generation on CIFAR-10 and MNIST Datasets

For both the vanilla formulation of the combined loss and the reformulation of the combined loss, each loss term (the reconstruction loss (binary cross-entropy/KL divergence), the KL divergence of the approximated posterior from the prior) was normalized with respect to the dimensionality over which the particular loss is computed i.e. the reconstruction loss (reformulated/vanilla) was normalized with respect to the dimensionality of the pixel space and the KL divergence term was normalized with respect to the dimensionality of the latent space. The first subsection under this section will contain the results for the MNIST dataset while the second one will contain the results for the CIFAR-10 dataset. All the search experiments searched over architectures with fully connected (FC) latent layers and included the latent size in the search space. The search space was the same for both MNIST and CIFAR-10. The possible latent sizes for both these datasets were $[2, 8, 16, 32, 64, 128, 256, 512]$. The possible convolutional filter sizes were $[2, 4, 6, 8]$, the possible convolutional filter numbers were $[64, 128, 256, 512]$, the layer limit for both the

encoder and the decoder was 12 and each candidate architecture was trained for 20 epochs using the Adam optimizer with a starting learning rate of 0.001. The other search hyperparameters were also kept the same for the two datasets. Besides, the only fully-connected layers in the networks were the 'mean vector' layer, the 'standard deviation vector' layer and the latent layer. All other layers were purely convolutional. This measure was taken solely because of the limitations of the hardware for training the networks.

10.2.1 Performance Comparison for Image Generation with MNIST

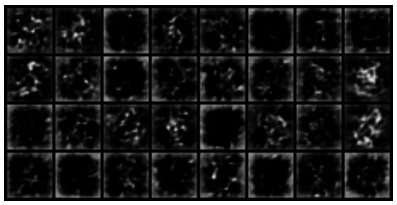
Apart from the four search experiments (search 1, 2, 3 and 4), four more experiments were conducted with MNIST wherein the latent size was fixed to 2 and the two dimensional latent embeddings for the top architectures from the searches were visualized. They are as listed below

5. search 1 with two dimensional latent space
6. search 2 with two dimensional latent space
7. search 3 with two dimensional latent space
8. search 4 with two dimensional latent space

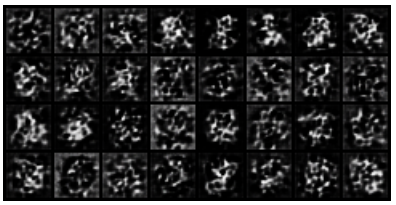
In these four experiments, the MNIST embeddings of the two dimensional latent spaces were visualized wherein all possible two dimensional points from a certain square grid, centered at the origin and discretized using a certain step value, were fed as inputs to the latent vector and the decoded/generated images were plotted at those discrete points in the form of a grid. The discretization was done by choosing a certain step value (usually < 1; often 0.25 or 0.5) and a certain range (usually $[-10, 10]$ or $[-5, 5]$) for sampling the discrete grid points for the square grid is defined. Now the all possible discrete points are sampled every step size from a square grid centered at 0 and having each side length equal to the range. For these experiments, the range or each side of the square grid was $[-5, 5]$ and the step size was 0.5. The output grid is supposed to show clustering of different classes of the dataset in the embedding space. In the MNIST context, the ten digits should form crisp clusters with as less overlap as possible for an architecture that is good at producing correlated but disentangled two dimensional features i.e. good at generating images using a two dimensional latent space. The search space and other training/validation hyperparameters were set as mentioned above.

10.2.1.1 Search 1 (Reformulated Search) vs Search 2 (Vanilla Search)

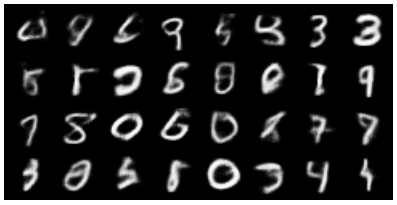
Figure 10.1 presents a comparative visualization of the generated images in the last validation epoch with random samples from a unit Gaussian as inputs for the **top two architectures**



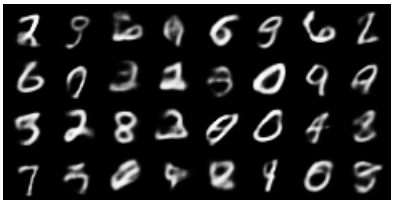
(A) Best Architecture with Reformulation



(B) Second Best Architecture with Reformulation



(C) Best Architecture without Reformulation



(D) Second Best Architecture without Reformulation

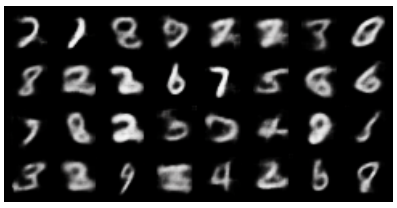
FIGURE 10.1: Images generated by the top two (maximum reward) architectures from each of the searches on MNIST i.e. vanilla and with reformulation. Here, the reward is the inverse of the combined validation loss with or without reformulation. **Top:** Top two architectures with reformulation. **Bottom:** Top two architectures without reformulation.

Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(256,2,2), C(64,4,2)]	512	[C(128,8,2)]	0.0664	2.0472
[C(128,2,2)]	128	[C(256,4,2)]	0.0663	2.7878

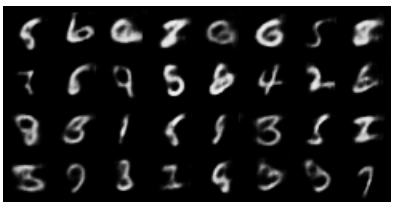
TABLE 10.1: The top two architectures from the search on MNIST with the inverse of the combined reformulated loss from the best validation epoch as the reward.

from search 1 and those for the **top two networks from search 2**. **Table 10.1**[1] **shows the top two architectures from the same reformulated search** while **table 10.2**[1] **gives the top two architectures from the above-mentioned vanilla search**.

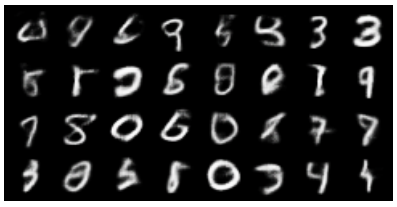
Figure 10.2 contains a similar comparative visualization of the generated images during the last validation epoch with random samples from a unit Gaussian as inputs for **two hand-picked architectures from the reformulated search** and **two more hand-picked networks from the search with the vanilla formulation**. Two architectures from each of the searches were handpicked purely on the basis of manually viewing the quality of the generated images i.e. the images generated in the last validation epoch by all the architectures in a certain search were looked at and the ones that generate the best looking images, images that are closer to the original image data than those generated by the other architectures, were handpicked. **Table 10.3**[1] **shows the handpicked architectures from the reformulated search** while **table 10.4**[1] **contains the handpicked architectures from the vanilla search.**



(A) First Handpicked Architecture with Reformulation



(B) Second Handpicked Architecture with Reformulation



(C) First Handpicked Architecture without Reformulation

(D) Second Handpicked Architecture without Reformulation

FIGURE 10.2: Images generated by the two handpicked architectures from the reformulated and vanilla searches on MNIST respectively. Here, the reward is inverse of the lowest validation combined loss with or without reformulation. **Top:** Two handpicked architectures with reformulation. **Bottom:** Two handpicked architectures without reformulation.

Encoder	**Latent Size**	**Decoder**	**BCE Loss**	**KL Loss**
[C(256,4,2), C(256,6,2)]	512	[C(512,4,2), C(512,2,2)]	0.1075	0.0384
[C(256,6,2), C(64,6,2)]	512	[C(512,4,2), C(256,8,2)]	0.1117	0.0368

TABLE 10.2: The top two architectures from the search on MNIST with the inverse of the combined vanilla loss from the best validation epoch as the reward.

Encoder	**Latent Size**	**Decoder**	**BCE Loss**	**KL Loss**
[C(256,2,2), C(64,4,2), C(64,4,2)]	8	[C(128,2,2), C(64,8,2), C(128,4,2)]	0.1249	3.9940
[C(256,2,2), C(64,4,2), C(64,4,2)]	8	[C(128,2,2), C(64,8,2), C(128,4,2)]	0.1180	4.0013

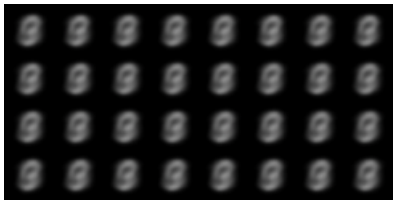
TABLE 10.3: Two handpicked architectures from the search on MNIST with the inverse of the combined reformulated loss from the best validation epoch as the reward.

Encoder	**Latent Size**	**Decoder**	**BCE Loss**	**KL Loss**
[C(256,6,2), C(64,6,2)]	128	[C(512,2,2), C(128,4,2), C(128,8,2)]	0.1685	0.0437
[C(256,4,2), C(256,6,2)]	512	[C(512,4,2), C(512,2,2)]	0.1075	0.0384

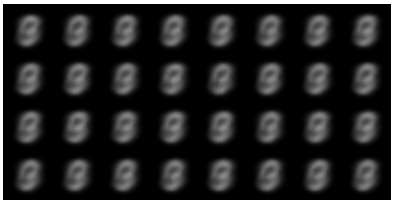
TABLE 10.4: Two handpicked architectures from the search on MNIST with the inverse of the combined vanilla loss from the best validation epoch as the reward.

(A) Best Architecture with Reformulation

(B) Second Best Architecture with Reformulation



(C) Best Architecture without Reformulation



(D) Second Best Architecture without Reformulation

FIGURE 10.3: Generation of images by the top two architectures from each of the reformulated and vanilla searches on MNIST with the inverse of the KL divergence from the best validation epoch as the reward. **Top:** Top two architectures with reformulation. **Bottom:** Top two architectures without reformulation.

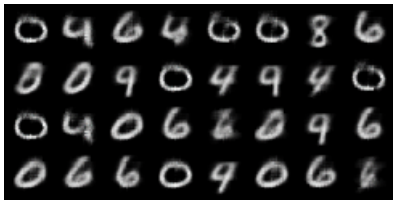
10.2.1.2 Search 3 (Reformulated Search) vs Search 4 (Vanilla Search)

Figure 10.3 contains another comparative visualization of the generated images in the last validation epoch with random samples from a unit Gaussian as inputs for the **top two architectures from search 3** and those for the **top two networks from search 4. Table 10.5**[1] **gives the configuration of the top two architectures from the above-mentioned reformulated search** while **table 10.6**[1] **contains the top two architectures from the vanilla search mentioned above**.

Figure 10.4 shows a comparative visualization of the generated images during the last validation epoch with random samples from a unit Gaussian as inputs for **two hand-picked architectures from the reformulated search** and those generated by **two more hand-picked networks from the vanilla search**. The handpicking was done in the way mentioned in sub-sub-section 10.2.1.1. **Table 10.7**[1] **presents the same two handpicked architectures from the reformulated search** while **table 10.8**[1] **shows the handpicked architectures from the reformulated search**.

10.2.1.3 Search 5 vs Search 6; Search 7 vs Search 8

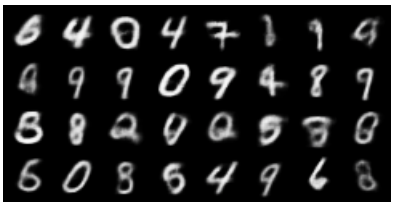
Figures 10.5 and 10.6 present the visualization of the two dimensional latent embeddings of the top architectures from the last four searches with the fixed latent size of 2. **Figure 10.5** compares the embeddings of the **top two architectures from search 5 with those of the top two architectures from search 6**. On the other hand, **figure 10.6** compares the embeddings of the **top two architectures from search 7 with those of the top two from search 8**.



(A) First Handpicked Architecture with Reformulation

(B) Second Handpicked Architecture with Reformulation

(C) First Handpicked Architecture without Reformulation



(D) Second Handpicked Architecture without Reformulation

FIGURE 10.4: Generation of images by the two handpicked architectures from the reformulated and vanilla searches on MNIST respectively. Here, the reward is inverse of the KL divergence from the validation epoch with the lowest combined loss with or without reformulation. **Top:** Two handpicked architectures with reformulation. **Bottom:** Two handpicked architectures without reformulation.

Encoder	**Latent Size**	**Decoder**	**BCE Loss**	**KL Loss**
[C(128,2,2), C(512,4,2), C(64,2,2), C(256,2,2)]	32	[C(64,6,2), C(128,8,2)]	3.6471	$\sim 10^{-7}$
[C(512,4,2), C(128,8,2)]	8	[C(128,6,2), C(512,4,2)]	3.4801	$\sim 10^{-7}$

TABLE 10.5: The top two architectures from the search on MNIST with the reformulated sum loss and the inverse of the KL divergence from the best validation epoch as the reward.

Encoder	**Latent Size**	**Decoder**	**BCE Loss**	**KL Loss**
[C(128,2,2), C(512,4,2), C(64,4,2)]	8	[C(512,2,2), C(512,2,2), C(256,2,2), C(128,8,2)]	0.2625	$\sim 10^{-7}$
[C(256,2,2), C(64,6,2)]	2	[C(128,8,2)]	0.2626	$\sim 10^{-7}$

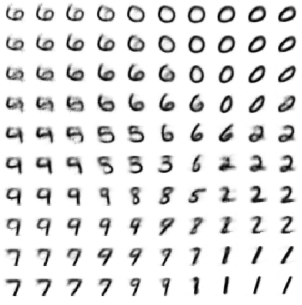
TABLE 10.6: The top two architectures from the search on MNIST with the vanilla sum loss and the inverse of the KL divergence from the best validation epoch as the reward.

Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(128,6,2), C(512,2,2)]	2	[C(512,2,2), C(64,2,2)]	0.2212	4.4004
[C(128,6,2), C(512,6,2), C(256,2,2)]	8	[C(512,2,2), C(256,2,2), C(256,4,2)]	0.1782	3.8372

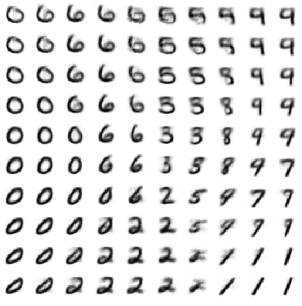
TABLE 10.7: The two handpicked architectures from the reformulated search on MNIST with the inverse of the KL loss from the best validation epoch as the reward.

Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(64,4,2), C(512,8,2)]	128	[C(512,4,2), C(256,4,2), C(512,2,2)]	0.2592	0.0023
[C(512,2,2), C(256,4,2), C(512,4,2)]	512	[C(512,2,2), C(64,2,2), C(512,6,2), C(128,4,2)]	0.2600	0.0016

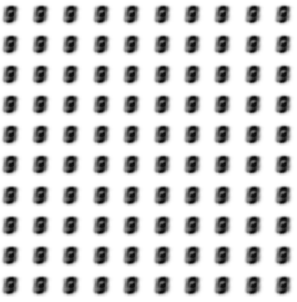
TABLE 10.8: The two handpicked architectures from the vanilla search on MNIST with the inverse of the KL loss from the best validation epoch as the reward.



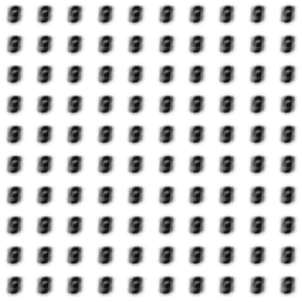
(A) Best Architecture with Reformulation



(B) Second Best Architecture with Reformulation

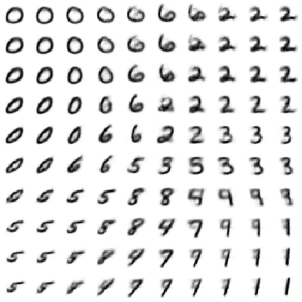


(C) Best Architecture without Reformulation

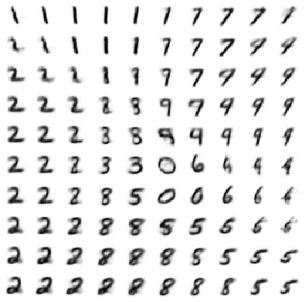


(D) Second Best Architecture without Reformulation

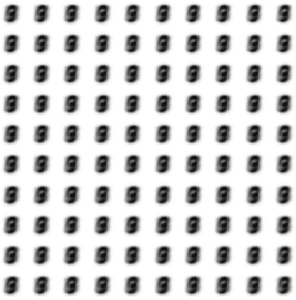
FIGURE 10.5: Two dimensional embeddings of the top two architectures from each of the reformulated and vanilla searches on MNIST with the inverse of the combined loss from the best validation epoch as the reward. **Top:** Top two architectures with reformulation. **Bottom:** Top two architectures without reformulation.



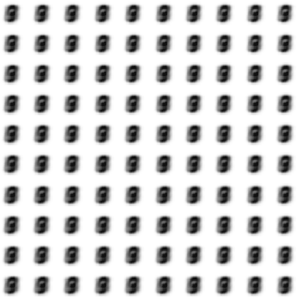
(A) Best Architecture with Reformulation



(B) Second Best Architecture with Reformulation



(C) Best Architecture without Reformulation



(D) Second Best Architecture without Reformulation

FIGURE 10.6: Two dimensional embeddings of the top two architectures from each of the reformulated and vanilla searches on MNIST with the inverse of the KL loss from the best validation epoch as the reward. **Top:** Top two architectures with reformulation. **Bottom:** Top two architectures without reformulation.

10.2.1.4 Concluding Remarks for MNIST Performance Comparison

As evident from figures 10.1 and 10.3, the top architectures from the searches with the vanilla formulation perform distinctively better than those from the searches with reformulation although figures 10.2 and 10.4 prove the performance of the handpicked architectures from the searches with reformulation are comparable to that of the handpicked architectures from the searches with vanilla formulation. Although the figures 10.1-10.4 don't explicitly make evident the utility of reformulating over using the vanilla loss function, tables 10.3 and 10.7 show that the handpicked architectures from the searches with reformulation tend to mostly have latent sizes of 8 and 2.

After showing the results for the CIFAR-10 dataset it will be empirically shown that the handpicked architectures which are performance wise the best architectures from the search, for different tasks tend to have latent sizes from within a certain range (the range for MNIST $\sim [2, 8]$ but for CIFAR-10 the results will show that this approximate range is widely different) and the range is somewhat characteristic of the task or the dataset at hand.

Also, as seen in figure 10.3, the inverse of the KL divergence of the approximated posterior from the prior of the validation epoch with the minimum combined loss, vanilla or reformulated, clearly fails to act as even a reasonable reward formulation for the search because the top architectures that are found, clearly overfit the task of optimizing the KL divergence from the prior but can't optimize the reconstruction loss at all.

On the other hand, the reformulation clearly outperforms the vanilla loss formulation when the latent dimensionality is restricted to 2 for MNIST as made apparent by figures 10.5 and 10.6. Irrespective of the architecture, for very small latent sizes (for eg. 2 in this case) the vanilla version of the VAE overfits the task of optimizing the KL divergence from the prior while the reformulated version overfits none of the tasks. The top architectures with the vanilla formulation end up overfitting the task of optimizing the KL loss of the approximated posterior with the prior while they fail to optimize the reconstruction loss at all. This again demonstrates the usefulness of the reformulation in this aspect.

The final useful conclusions from the MNIST results are as follows

- **the vanilla formulation works better in certain cases**
    1. *top architectures from search 1 vs top architectures from search 2 (figure 10.1)*
- **the reformulation works better for very small latent spaces where the vanilla VAE completely overfits the task of optimizing the KL divergence from the prior**
    1. *top architectures from search 5 vs top architectures from search 6 (figure 10.5)*
    2. *top architectures from search 7 vs top architectures from search 8 (figure 10.6)*
- **the reformulation and the vanilla formulation are comparable in performance in certain instances**
    1. *top architectures from search 3 vs top architectures from search 4 (figure 10.3)*
    2. *handpicked architectures from search 1 vs handpicked architectures from search 2 (figure 10.2)*
    3. *handpicked architectures from search 3 vs handpicked architectures from search 4 (figure 10.4)*
- **the current methods of reward formulation can't be fully relied upon since the top architectures from the searches are not the best/handpicked ones**
- **the handpicked/best performing architectures from the searches with reformulation have latent sizes in the range $\sim [2, 8]$ which somewhat reflect the complexity of the MNIST dataset**

1. *handpicked architectures from search 1 (table 10.3)*
2. *handpicked architectures from search 3 (table 10.7)*

10.2.2 Performance Comparison for Image Generation with CIFAR-10

In total, only the first four search experiments (1, 2, 3, 4 as mentioned above in section 10.1) were done for the CIFAR-10 dataset. The search space, the possible latent sizes, the architecture layer limit, the optimizer, the initial learning rate, and the other training and validation hyperparameters were exactly same as in the case of MNIST. Keeping the search spaces and other hyperparameters same for two completely different datasets would result in searches that are not additionally favoured by search spaces. The search spaces were very generic and were definitely not fine-tuned for particular tasks. Such generic searches would help in doing a better analysis of the utility of the reformulation of the combined loss over the combined vanilla loss and the utility of having the two current methods of formulating the reward function i.e. using the inverse of the lowest validation combined loss or the inverse of the KL divergence from the best validation epoch as the reward. A detailed analysis of the CIFAR-10 results (both figures and tables) will be provided in the sub-sub-section that concludes this section on the CIFAR-10 results.

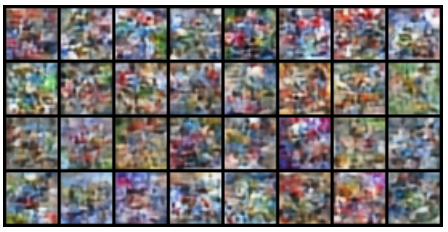
10.2.2.1 Search 1 (Reformulated Search) vs Search 2 (Vanilla Search)

Figure 10.7 presents a comparative visualization of the generated images in the last validation epoch with random samples from a unit Gaussian as inputs for the **top two architectures from search 1** and those for the **top two networks from search 2**. **Table 10.9[1] shows the top two architectures from the same reformulated search** while **table 10.10[1] gives the top two architectures from the vanilla search**.

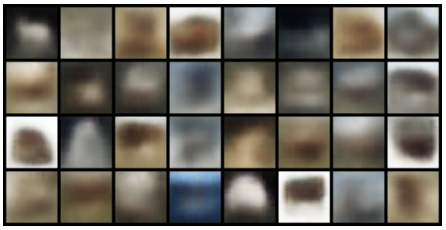
Figure 10.8 contains a similar comparative visualization of the generated images during the last validation epoch with random samples from a unit Gaussian as inputs for **two hand-picked architectures from the reformulated search** and **two more hand-picked networks from the search with the vanilla formulation**. **Table 10.11[1] shows the handpicked architectures from the reformulated search** while **table 10.12[1] contains the handpicked architectures from the vanilla search.** The handpicking method used was same as the one for MNIST.

A detailed analysis of the results along with the reasons for showing the results from handpicked architectures will be provided at the end of this section. Such analysis will be very useful to provide the reader valuable insight on the utility of the reformulation performance wise, the

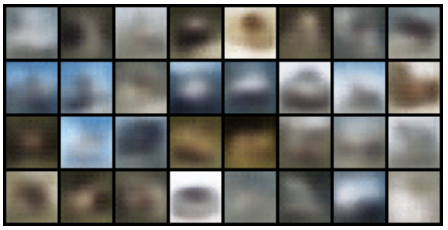
(A) Best Architecture with Reformulation



(B) Second Best Architecture with Reformulation

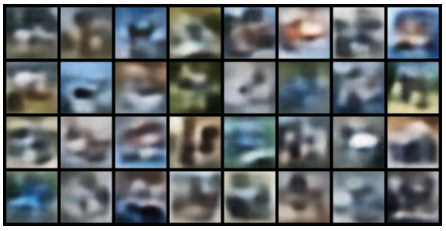


(C) Best Architecture without Reformulation

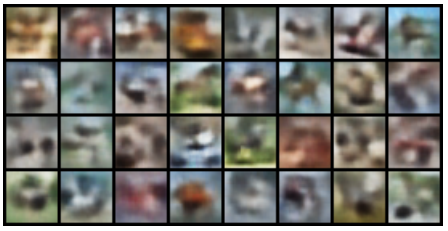


(D) Second Best Architecture without Reformulation

FIGURE 10.7: Images generated by the top two (maximum reward) architectures from each of the searches on CIFAR-10 i.e. vanilla and with reformulation, with inverse of the best validation combined loss as the reward. **Top:** Top two architectures with reformulation. **Bottom:** Top two architectures without reformulation.

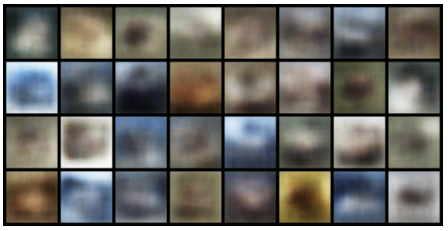


(A) First Handpicked Architecture with Reformulation



(B) Second Handpicked Architecture with Reformulation

(C) First Handpicked Architecture without Reformulation



(D) Second Handpicked Architecture without Reformulation

FIGURE 10.8: Images generated by the two handpicked architectures from the reformulated and vanilla searches on CIFAR-10 respectively. Here, the reward is inverse of the lowest validation combined loss with or without reformulation. **Top:** Two handpicked architectures with reformulation. **Bottom:** Two handpicked architectures without reformulation.

effectiveness of forming the reward function in these two particular ways and their effect on the MetaQNN based search.

Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(256,2,2)]	512	[C(64,2,2), C(512,6,2), C(256,6,2)]	0.5540	2.2444
[C(64,8,2), C(256,2,2), C(512,2,2)]	512	[C(512,4,2), C(128,2,2)]	0.5548	2.2380

TABLE 10.9: The top two architectures from the search on CIFAR-10 with the inverse of the combined reformulated loss from the best validation epoch as the reward.

Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(512,2,2), C(128,8,2), C(256,2,2), C(512,2,2)]	512	[C(512,4,2), C(512,2,2), C(256,6,2)]	0.6096	0.0184
[C(512,4,2), C(256,6,2), C(256,2,2)]	512	[C(512,6,2), C(512,2,2)]	0.6103	0.0180

TABLE 10.10: The top two architectures from the search on CIFAR-10 with the inverse of the combined vanilla loss from the best validation epoch as the reward.

Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(256,8,2), C(256,4,2)]	32	[C(64,2,2), C(512,6,2), C(256,6,2)]	0.5778	3.2622
[C(256,2,2)]	64	[C(64,2,2), C(512,6,2), C(64,2,2)]	0.5683	3.049

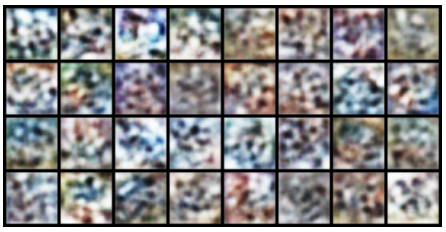
TABLE 10.11: Two handpicked architectures from the search on CIFAR-10 with the inverse of the combined reformulated loss from the best validation epoch as the reward.

Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(128,6,2)]	512	[C(256,2,2), C(512,2,2)]	0.6109	0.0183
[C(256,2,2)]	512	[C(256,8,2)]	0.6092	0.0196

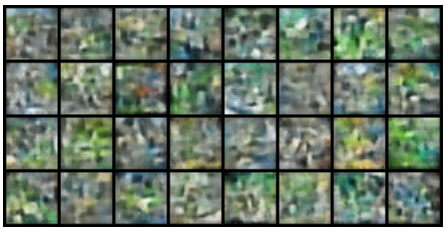
TABLE 10.12: Two handpicked architectures from the search on CIFAR-10 with the inverse of the combined vanilla loss from the best validation epoch as the reward.

10.2.2.2 Search 3 (Reformulated Search) vs Search 4 (Vanilla Search)

Figure 10.9 contains another comparative visualization of the generated images in the last validation epoch with random samples from a unit Gaussian as inputs for the **top two architectures from search 3** and those for the **top two networks from search 4**. **Table 10.13**[1] **gives the configuration of the top two architectures from the above-mentioned reformulated search** while **table 10.14**[1] **contains the top two architectures from the vanilla search** mentioned above.



(A) Best Architecture with Reformulation



(B) Second Best Architecture with Reformulation

(C) Best Architecture without Reformulation

(D) Second Best Architecture without Reformulation

FIGURE 10.9: Generation of images by the top two architectures from each of the reformulated and vanilla searches on CIFAR-10 with the inverse of the KL divergence from the best validation epoch as the reward. **Top:** Top two architectures with reformulation. **Bottom:** Top two architectures without reformulation.

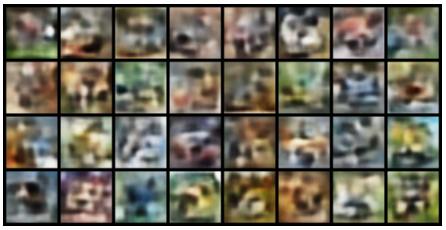
Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(512,4,2), C(128,8,2)]	512	[C(64,4,2)]	0.5714	1.5786
[C(512,4,2), C(128,8,2)]	512	[C(512,4,2), C(512,2,2)]	0.5772	1.6029

TABLE 10.13: The top two architectures from the search on CIFAR-10 with the reformulated sum loss and the inverse of the KL divergence from the best validation epoch as the reward.

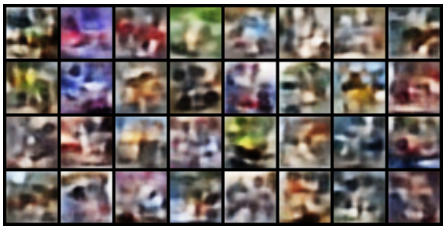
Encoder	Latent Size	Decoder	BCE Loss	KL Loss
[C(128,8,2), C(512,2,2), C(512,4,2)]	8	[C(256,6,2), C(64,6,2)]	0.6894	$\sim 10^{-7}$
[C(64,6,2), C(512,6,2), C(128,4,2)]	2	[C(64,2,2), C(64,6,2)]	0.6894	$\sim 10^{-7}$

TABLE 10.14: The top two architectures from the search on CIFAR-10 with the vanilla sum loss and the inverse of the KL divergence from the best validation epoch as the reward.

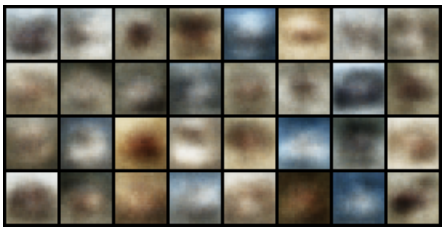
Figure 10.10 shows a comparative visualization of the generated images during the last validation epoch with random samples from a unit Gaussian as inputs for **two hand-picked architectures from the reformulated search** and those generated by **two more hand-picked networks from the vanilla search**. The handpicking was done in exactly the same way as always. **Table 10.15**[1] **presents the same two handpicked architectures from the reformulated search** while **table 10.16**[1] **shows the handpicked architectures from the vanilla search**.



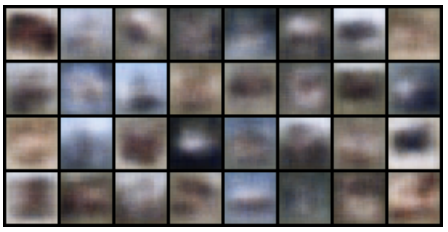
(A) First Handpicked Architecture with Reformulation



(B) Second Handpicked Architecture with Reformulation



(C) First Handpicked Architecture without Reformulation



(D) Second Handpicked Architecture without Reformulation

FIGURE 10.10: Generation of images by the two handpicked architectures from the reformulated and vanilla searches on CIFAR-10 respectively. Here, the reward is inverse of the KL divergence from the validation epoch with the lowest combined loss with or without reformulation. **Top:** Two handpicked architectures with reformulation. **Bottom:** Two handpicked architectures without reformulation.

Encoder	**Latent Size**	**Decoder**	**BCE Loss**	**KL Loss**
[C(128,2,2)]	64	[C(512,6,2), C(512,2,2), C(128,8,2)]	0.5790	2.7911
[C(64,8,2), C(256,4,2), C(64,4,2)]	64	[C(512,6,2), C(256,4,2)]	0.5861	2.3920

TABLE 10.15: The two handpicked architectures from the reformulated search on CIFAR-10 with the inverse of the KL loss from the best validation epoch as the reward.

Encoder	**Latent Size**	**Decoder**	**BCE Loss**	**KL Loss**
[C(512,2,2), C(512,8,2)]	512	[C(512,2,2)]	0.6365	0.0140
[C(64,2,2), C(64,8,2)]	512	[C(512,6,2)]	0.6212	0.0166

TABLE 10.16: The two handpicked architectures from the vanilla search on CIFAR-10 with the inverse of the KL loss from the best validation epoch as the reward.

10.2.2.3 Concluding Remarks for CIFAR-10 Performance Comparison

As can be seen in figure 10.8, 10.9 and 10.10, the reformulation of the combined loss evidently works better in the case of CIFAR-10 for handpicked architectures with both types of reward formulation, inverse of the combined loss and inverse of the KL divergence, and for the top architectures with the inverse of the KL divergence from the best validation epoch as the reward. The images from figure 10.7 aren't very conclusive and hence no inference on the utility of reformulation in the particular case of using the inverse of the reformulated sum loss from the best validation epoch as the reward can be drawn. From tables 10.9 and 10.10, it can be seen that the top architectures from each of the searches with the inverse of the sum loss as the reward have the highest possible latent size. Higher latent sizes are helpful in maximizing the reward in this case. Similarly, the top architectures in tables 10.13 and 10.14 tend to have latent sizes that help in optimizing the KL loss very well thus maximizing the reward in the case of the searches where the reward is the inverse of the KL loss from the best validation epoch.

On the contrary, the handpicked architectures from the searches with reformulation, which generate images that are the closest to the original CIFAR-10 dataset when compared to all the other architectures from the searches, have latent sizes of 32 and 64 as seen in tables 10.11 and 10.15 while for MNIST, the latent sizes of the handpicked architectures with reformulation were in the range $\sim [2, 8]$. Thus, it somewhat shows that the best-performing architectures for a dataset have their latent sizes in a certain range. There is no such apparent pattern for the searches with the handpicked architectures from the searches with the vanilla formulation. Hence, it can be loosely guessed that the handpicked networks which are actually the best performing architectures for the reformulated searches have latent sizes which somewhat represent the dataset complexity and to some extent can be seen as the optimal latent sizes for the particular task/dataset.

As evident in figure 10.9, the inverse of the KL divergence of the approximated posterior from the prior of the validation epoch with the minimum combined loss, in the vanilla formulation clearly fails to function as even a potential reward formulation for the search because the top architectures such searches end up finding, clearly overfit the task of optimizing the KL divergence from the prior and can't optimize the reconstruction loss at all. But, the reformulated search with the inverse of the KL loss from the validation epoch with the most optimal combined loss, with reformulation ends up finding architectures that at least generate some features and colours though they are clearly not up to the expectations. This partially shows again that the reformulation is a little more efficient than the vanilla formulation.

Finally, the list of conclusions that can be drawn from the CIFAR-10 results are as follows

- **reformulation apparently works better than the vanilla formulation**
  1. *top architectures from search 3 vs top architectures from search 4 (figure 10.9)*
  2. *handpicked architectures from search 3 vs handpicked architectures from search 4 (figure 10.10)*
  3. *handpicked architectures from search 1 vs handpicked architectures from search 2 (figure 10.8)*
- **the current methods of reward formulation can't be fully relied upon since the top architectures from the searches are not the best/handpicked ones**
- **the handpicked/best performing architectures from the searches with reformulation have latent sizes in the range $\sim [32, 64]$ which somewhat reflect the complexity of the CIFAR-10 dataset**
  1. *handpicked architectures from search 1 (table 10.11)*
  2. *handpicked architectures from search 3 (table 10.15)*

10.3 Concluding Remarks for Overall Performance Comparison

The reformulation of the loss function appears to work emphatically better for the image generation task on a relatively more complex dataset like CIFAR-10. For a relatively simpler dataset like MNIST which is more often used as a benchmark for comparing image generation algorithms using VAEs, the results are not very conclusive though loosely the reformulation and the vanilla formulation can be considered to be at par with each other taking into account the better results of the vanilla search when the latent sizes are also searched over and the far better performance of the reformulated search when the size of the latent layer is fixed to 2. One important observation to be made is that the handpicked architectures for both the datatets with any one of the two reward formulations have latent sizes that apparently reflect the task/dataset complexity. On the other hand, the two present methods of forming the search reward from the loss metrics are far from being acceptable because the top architectures that such searches produce are nowhere near the best ones among the candidates they search over (the handpicked architectures are not even among the top five architectures from the searches).

A few important overall conclusions and insights from the comparison of the results for CIFAR-10 and MNIST are

- **the reformulation appears to work evidently better for CIFAR-10**

- **for MNIST, the performance of the reformulated VAE is more or less comparable to the vanilla VAE if not better**
- **the current two methods of defining the reward from the combined loss need to be modified**
- **for MNIST, the better latent sizes fall in the approximate range** $[2, 8]$ **while for CIFAR-10, they fall in the approximate range** $[32, 64]$

Chapter 11

Modification of Q-learning Update Rule to Greedy Version

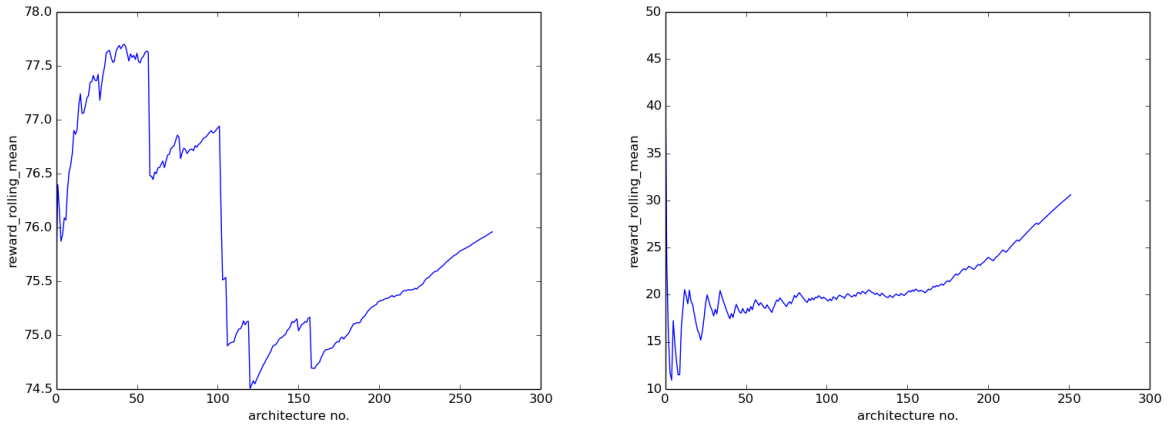
While using MetaQNN for searching for architectures in different vision tasks, it was observed that the search fails to converge when the search schedule is short i.e. the architectures produced towards the end of the search were not consistently better than the architectures produced during the initial phase of the search. This was in direct conflict with the main purpose of doing an architecture search because one would expect the search algorithm to produce functionally better architectures towards the end of the schedule. In the original MetaQNN paper [3], as discussed before, they produce results for extremely long search schedules in the task of image classification for benchmark vision datasets and they showed the convergence of their search Q-learning based algorithm for long search schedules that included 2000+ architectures. For more practically executable and shorter search schedules, I modified the Q-learning update [4.1] to a greedy update rule similar to greedy tree search.

$$Q_{t+1}(s_i, u) = max(Q_t(s_i, u), r_t) \tag{11.1}$$

Two advantages from this modification are

- the initial Q-values can be set to 0 always irrespective of the reward design for the task because the update is always greedy and the nature doesn't depend on the initial values of the Q-values as long as they are 0, however it's risky to initialize them to other non-zero values because that might hurt the update as evident from the update rule
- convergence can be expected even for shorter search schedules

Time didn't permit me to rigorously compare the two update rules for different tasks with different benchmark datasets but I compared them for the same-length search schedule of 270



(A) Original Q-learning update

(B) Modified MetaQNN update

FIGURE 11.1: Comparative analysis between rolling mean plots for original Q-learning update, and modified greedy MetaQNN update AEROBI architecture search with 270 architectures. **Left:** Rolling mean plot for original Q-learning update rule. **Right:** Rolling mean plot for modified MetaQNN update rule.

architectures for the AEROBI multi-label classification task.I plotted the rolling mean of the architecture rewards from the searches against the architecture number. The plots are displayed side by side in figure 11.1.

Chapter 12

Summary of Work Done

This chapter provides a summary of the work done in this thesis. The insightful take-aways from thes work are as follows.

Firstly, the MetaQNN (or other RL) based architecture search works for completely supervised classification tasks and completely unuspervised reconstruction. For supervised classification, table 4.4 from chapter 4 shows a validation accuracy of 99.70% on the MNIST dataset while table while table 4.5 shows a validation F1-score of 72.19% on the AEROBI dataset which is far better than the F1-score for the hand-engineered architectures. For unsupervised image reconstruction, fig 5.6 shows reconstructed STL images from the top architecture which are very clear and are very close to the input images, both visually and from the point of loss metrics. The top architecture gives a validation BCE loss of 0.5203 on the STL dataset which is a very low BCE loss value for STL. In these two tasks, the reward can be formulated as the inverse of the validation performance metric of the architectures and the reward reflects to a large extent the potential of a candidate architecture for that particular task.

Second of all, the same naive MetaQNN (or other RL) based search techniques can't be applied for an architecture search on the task of image generation using VAEs. There are two reasons for this which are as follows

1. individual VAEs find it very hard to jointly optimize the two loss components of the combined loss function when used in the vanilla fashion
2. formulating a reward from the validation performance metrics of a VAE which truly reflects the image generation potential of a VAE, is not very trivial

Figure 9.2 shows an example architecture with a randomly selected latent size of 5 doesn't generate MNIST images at all which supports the first point.

Thirdly, the proposed technique of reformulating the combined loss used to train the VAE, is found to work better than the vanilla combined loss both for individual VAE networks and for architecture search experiments in most cases. The results supporting this observation are as listed below

- for CIFAR-10, the reformulation works far better than the vanilla formulation as seen in figures 10.7, 10.8 and 10.9
- for MNIST their performances are comparable
  - the top architectures produced from the vanilla architecture search with the inverse of the sum loss as the reward, produce better results than the top architectures from the corresponding reformulated architecture search (figures 10.1)
  - the top architectures from the vanilla search with the inverse of the KL divergence as the reward, and the handpicked architectures from the reformulated searches are comparable with those from the corresponding vanilla searches performance wise (figures 10.2, 10.3 and 10.4)
  - the handpicked architectures from the reformulated searches have latent sizes that look like representative latent sizes for the particular dataset
    - for MNIST, the representative latent sizes are in the range of $\sim [2, 8]$ (tables 10.3 and 10.6)
    - for CIFAR-10, they are in the range of $\sim [32, 64]$ (tables 10.11 and 10.14)

Next, the current two methods of reward formulation from the validation performance metrics of individual architectures for a search on VAEs, need to be worked upon and modified. The top architectures from the searches according to the reward metrics are not the best architectures among all the other architectures that have been searched over. Figures 10.1 and 10.3 for MNIST, and figures 10.7 and 10.9 for CIFAR-10, are proofs to this fact because from the visual fidelity point of view, the handpicked architectures from all the searches produce better quality images.

Finally, the modified MetaQNN update rule as give by equation 11.1 might in better convergence of the search MetaQNN search algorithm. Although, the report doesn't do a comprehensive comparison of the two algorithms in diverse search scenarios but produce one illustrative analysis for a particular task in the form of figure 11.1.

Chapter 13

Future Work and Conclusion

This chapter will draw a conclusion to the end-semester thesis report by discussing the key takeaways from the work done and will also provide the outline and the prospects of the work to be done in future.

13.1 Future Work

Overall, there are four aspects of this work that need to be worked upon as part of the future work. They are as follows.

Primarily, even though the way things stand now the reformulation of the combined loss tends to work somewhat better than the vanilla formulation, there is no concrete mathematical link between the reconstruction loss reformulated as a KL divergence term in equation 8.1 and the reconstruction loss in 7.14 which was formed directly out of equation 7.10 (SGVB/ELBO) even though the substitution is quite intuitive and logical. If the mathematical proof can't be established, since the intuitive link between the vanilla combined loss (SGVB/ELBO) and the reformulated combined loss is quite apparent more experimental results will have to be provided i.e. image generation results with other datasets will have to be shown.

Next, as evident from the results of both MNIST and CIFAR-10, the current reward formulations of using the inverse of the combined loss or the inverse of the KL divergence from the prior, as the reward are not very sound. The top architectures produced by the MetaQNN based searches using these rewards are definitely not the best architectures image generation wise among all the other architectures the algorithm searches over. The handpicked architectures generate far better images than the top architectures.

The third aspect of the future work is to test how sensitive the apparently optimal latent sizes (found in the handpicked architectures from searches with reformulation) are to the complexity of the dataset. The handpicked architectures from the reformulated searches for both CIFAR-10 and MNIST have latent sizes that are apparently dependent on the dataset but if one creates very minute perturbations in the data of any one of these datasets i.e. introduces very small noise in a controlled way, the best architectures from reformulated searches should not have latent sizes that are widely different from those for the original unperturbed data from a logical point of view. Having similar latent sizes will further substantiate the author's claim of the reformulation being more efficient in comparison to the vanilla formulation.

Also, a rigorous analysis and testing of the convergence guarantees for the modified MetaQNN update rule is necessary. The update rule should be used for searches on different tasks with different datasets and only then, it would provide a better understanding of its efficacies and drawbacks.

Lastly, to produce results for some more comparative study, an architecture search over GANs should be conducted where the discriminator network would have a fixed architecture but would be powerful enough, and the validation accuracy of classifying the images generated by the generator network as images from the original data distribution, will be used as the reward.

13.2 Conclusion

The RL based architecture search produces highly efficient architectures in completely supervised settings of image classification and the completely unsupervised settings of image reconstruction and the choice of RL algorithm, if chosen from Q-learning or REINFORCE, doesn't make much difference due to the commonly used discrete nature of search space. Despite their excellent performance on such tasks, the naive use of RL algorithms on far more complex tasks of image generation using VAEs doesn't fare well. A successful architecture search on such a setting would not only require a very sound reward formulation from the loss function but also a reformulation of the loss function in general. The reason behind reformulating the loss function is that the vanilla loss where the reconstruction loss and the KL divergence from the posterior are plainly summed together, as in equation 7.14, has been experimentally found to be very architecture and data dependent i.e. the trainability of a particular network for jointly learning the task of reconstructing and minimizing the KL divergence from the prior is highly dependent on the network architecture used and the dataset that one is working with. The reformulation technique proposed in this work is a step in that direction although as of now it can be regarded only as a promising technique but not as a foolproof or sound one. Also, be it with or without reformulation, a particular way to form the search reward from the loss metrics need to be developed which in a way reflects the image generating potential of a candidate model. Lastly,

one other untested avenue is to move away from the commonly used pixel-by-pixel loss (mean squared error, BCE loss or the reformulated pixel wise loss in our case) and to try to obtain a divergence measure of a distribution over the total error as a whole or errors from local neighbourhoods that more adequately tie to the features/concepts. Moreover, the modified greedy MetaQNN update rule shows promise in terms of better search convergence but it's at a very nascent state and requires more firm and thorough analysis of its possibilities.

Bibliography

[1] Marcin Andrychowicz et al. "Learning to learn by gradient descent by gradient descent". In: *CoRR* abs/1606.04474 (2016). arXiv: 1606.04474. URL: http://arxiv.org/abs/1606.04474.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *CoRR* abs/1409.0473 (2014). arXiv: 1409.0473. URL: http://arxiv.org/abs/1409.0473.

[3] Bowen Baker et al. "Designing Neural Network Architectures using Reinforcement Learning". In: *CoRR* abs/1611.02167 (2016). arXiv: 1611.02167. URL: http://arxiv.org/abs/1611.02167.

[4] Bowen Baker et al. "Practical Neural Network Performance Prediction for Early Stopping". In: *CoRR* abs/1705.10823 (2017). arXiv: 1705.10823. URL: http://arxiv.org/abs/1705.10823.

[5] James Bergstra and Yoshua Bengio. "Random Search for Hyper-parameter Optimization". In: *J. Mach. Learn. Res.* 13.1 (Feb. 2012), pp. 281–305. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2503308.2188395.

[6] James Bergstra and Yoshua Bengio. "Random Search for Hyper-parameter Optimization". In: *J. Mach. Learn. Res.* 13 (Feb. 2012), pp. 281–305. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2188385.2188395.

[7] James Bergstra, Daniel Yamins, and David Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 1. Atlanta, Georgia, USA: PMLR, 2013, pp. 115–123. URL: http://proceedings.mlr.press/v28/bergstra13.html.

[8] James S. Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-Taylor et al. Curran Associates, Inc., 2011, pp. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[9] Han Cai et al. "Reinforcement Learning for Architecture Search by Network Transformation". In: *CoRR* abs/1707.04873 (2017). arXiv: 1707.04873. URL: http://arxiv.org/abs/1707.04873.

[10] Xi Chen et al. "InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets". In: *CoRR* abs/1606.03657 (2016). arXiv: 1606.03657. URL: http://arxiv.org/abs/1606.03657.

[11] Adam Coates, Andrew Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 215–223.

[12] DeepLearning4j. *DeepLearning4j GANs*. 2018. URL: https://deeplearning4j.org/img/GANs.png (visited on 04/30/2018).

[13] Boyang Deng, Junjie Yan, and Dahua Lin. "Peephole: Predicting Network Performance Before Training". In: *CoRR* abs/1712.03351 (2017). arXiv: 1712.03351. URL: http://arxiv.org/abs/1712.03351.

[14] Carl Doersch. "Tutorial on Variational Autoencoders". In: *CoRR* abs/1606.05908 (2016).

[15] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. "Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves". In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 3460–3468. ISBN: 978-1-57735-738-4. URL: http://dl.acm.org/citation.cfm?id=2832581.2832731.

[16] Ahmed Elgammal. *Generating "art" by Learning About Styles and Deviating from Style Norms*. 2018. URL: https://medium.com/@ahmed_elgammal/generating-art-by-learning-about-styles-and-deviating-from-style-norms-8037a13ae027 (visited on 04/30/2018).

[17] Dario Floreano, Peter Dürr, and Claudio Mattiussi. *Neuroevolution: from architectures to learning*. 2008.

[18] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[19] Karol Gregor et al. "DRAW: A Recurrent Neural Network For Image Generation". In: *CoRR* abs/1502.04623 (2015). arXiv: 1502.04623. URL: http://arxiv.org/abs/1502.04623.

[20] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[21] Kaiming He et al. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". In: *CoRR* abs/1406.4729 (2014). arXiv: 1406.4729. URL: http://arxiv.org/abs/1406.4729.

[22] Irina Higgins. *beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*. 2018. URL: https://openreview.net/forum?id=Sy2fzU9gl (visited on 04/30/2018).

[23] Geoffrey E Hinton and Richard S. Zemel. "Autoencoders, Minimum Description Length and Helmholtz Free Energy". In: *Advances in Neural Information Processing Systems 6*. Ed. by J. D. Cowan, G. Tesauro, and J. Alspector. Morgan-Kaufmann, 1994, pp. 3–10. URL: http://papers.nips.cc/paper/798-autoencoders-minimum-description-length-and-helmholtz-free-energy.pdf.

[24] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-based Optimization for General Algorithm Configuration". In: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*. LION'05. Rome, Italy: Springer-Verlag, 2011, pp. 507–523. ISBN: 978-3-642-25565-6. DOI: 10.1007/978-3-642-25566-3_40. URL: http://dx.doi.org/10.1007/978-3-642-25566-3_40.

[25] Kevin Jarrett, Koray Kavukcuoglu, and Yann Lecun. *What is the Best Multi-Stage Architecture for Object Recognition?*

[26] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. "An Empirical Exploration of Recurrent Network Architectures". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 2342–2350. URL: http://dl.acm.org/citation.cfm?id=3045118.3045367.

[27] Brian Keng. *Semi-supervised Learning with Variational Autoencoders*. 2018. URL: http://bjlkeng.github.io/posts/semi-supervised-learning-with-variational-autoencoders/ (visited on 04/30/2018).

[28] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[29] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes." In: *CoRR* abs/1312.6114 (2013). URL: http://dblp.uni-trier.de/db/journals/corr/corr1312.html#KingmaW13.

[30] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*, p. 2012.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[33] kvfrans. *Variational Autoencoders Explained*. 2018. URL: http://kvfrans.com/variational-autoencoders-explained/ (visited on 04/30/2018).

[34] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[35] Sergey Levine. *Deep RL with Q-Functions*. 2018. URL: http://rll.berkeley.edu/deeprlcourse/f17docs/lecture_7_advanced_q_learning.pdf (visited on 04/30/2018).

[36] Sergey Levine. *Introduction to Reinforcement Learning*. 2018. URL: http://rll.berkeley.edu/deeprlcourse/f17docs/lecture_3_rl_intro.pdf (visited on 04/30/2018).

[37] Lisha Li et al. "Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits". In: *CoRR* abs/1603.06560 (2016). arXiv: 1603.06560. URL: http://arxiv.org/abs/1603.06560.

[38] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

[39] Hector Mendoza et al. "Towards Automatically-Tuned Neural Networks". In: *Proceedings of the Workshop on Automatic Machine Learning*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Vol. 64. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, pp. 58–65. URL: http://proceedings.mlr.press/v64/mendoza_towards_2016.html.

[40] Risto Miikkulainen et al. "Evolving Deep Neural Networks". In: *CoRR* abs/1703.00548 (2017). arXiv: 1703.00548. URL: http://arxiv.org/abs/1703.00548.

[41] Mary Ann Marcinkiewicz Ann Taylor Mitchell P. Marcus Beatrice Santorini. *Treebank-3*. 2018. URL: https://catalog.ldc.upenn.edu/ldc99t42 (visited on 04/30/2018).

[42] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[43] Yuval Netzer et al. "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*. 2011. URL: http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

[44] Aäron van den Oord et al. "Conditional Image Generation with PixelCNN Decoders". In: *CoRR* abs/1606.05328 (2016). arXiv: 1606.05328. URL: http://arxiv.org/abs/1606.05328.

[45] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. "Pixel Recurrent Neural Networks". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, pp. 1747–1756. URL: http://proceedings.mlr.press/v48/oord16.html.

[46] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[47] Nikolay Pavlov. *Deep Q-Learning*. 2018. URL: https://www.slideshare.net/nikolaypavlov/deep-qlearning (visited on 04/30/2018).

[48] Hieu Pham et al. "Efficient Neural Architecture Search via Parameter Sharing". In: *CoRR* abs/1802.03268 (2018). arXiv: 1802.03268. URL: http://arxiv.org/abs/1802.03268.

[49] Nicolas Pinto et al. "A high-throughput screening approach to discovering good forms of biologically inspired visual representation". In: *PLoS Computational Biology* 5.11 (2009). PMID: 19956750, e1000579. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1000579. URL: http://dicarlolab.mit.edu/sites/dicarlolab.mit.edu/files/pubs/pinto-doukhan-dicarlo-cox_plos_2009.pdf.

[50] Esteban Real et al. "Large-Scale Evolution of Image Classifiers". In: *CoRR* abs/1703.01041 (2017). arXiv: 1703.01041. URL: http://arxiv.org/abs/1703.01041.

[51] *Reinforcement learning*. 2018. URL: https://reinforce.io/reinforcement-learning/ (visited on 04/30/2018).

[52] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[53] Tim Salimans, Diederik Kingma, and Max Welling. "Markov Chain Monte Carlo and Variational Inference: Bridging the Gap". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 1218–1226. URL: http://proceedings.mlr.press/v37/salimans15.html.

[54] Andrew M. Saxe et al. "On Random Weights and Unsupervised Feature Learning". In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA: Omnipress, 2011, pp. 1089–1096. ISBN: 978-1-4503-0619-5. URL: http://dl.acm.org/citation.cfm?id=3104482.3104619.

[55] J. D. Schaffer, D. Whitley, and L. J. Eshelman. "Combinations of genetic algorithms and neural networks: a survey of the state of the art". In: *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on* (1992), pp. 1–37. DOI: 10.1109/COGANN.1992.273950. URL: http://dx.doi.org/10.1109/COGANN.1992.273950.

[56] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[57] Wenling Shang et al. "Channel-Recurrent Variational Autoencoders". In: *CoRR* abs/1706.03729 (2017). arXiv: 1706.03729. URL: http://arxiv.org/abs/1706.03729.

[58] Yao Shu et al. "Understanding Deep Representations through Random Weights". In: *CoRR* abs/1704.00330 (2017). arXiv: 1704.00330. URL: http://arxiv.org/abs/1704.00330.

[59] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[60] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[61] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 2951–2959. URL: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

[62] Jasper Snoek et al. "Scalable Bayesian Optimization Using Deep Neural Networks". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 2171–2180. URL: http://proceedings.mlr.press/v37/snoek15.html.

[63] Casper Kaae Sø nderby et al. "Ladder Variational Autoencoders". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 3738–3746. URL: http://papers.nips.cc/paper/6275-ladder-variational-autoencoders.pdf.

[64] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. "A Hypercube-based Encoding for Evolving Large-scale Neural Networks". In: *Artif. Life* 15.2 (Apr. 2009), pp. 185–212. ISSN: 1064-5462. DOI: 10.1162/artl.2009.15.2.15202. URL: http://dx.doi.org/10.1162/artl.2009.15.2.15202.

[65] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies". In: *Evol. Comput.* 10.2 (June 2002), pp. 99–127. ISSN: 1063-6560. DOI: 10.1162/106365602320169811. URL: http://dx.doi.org/10.1162/106365602320169811.

[66] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[67] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Denver, CO: MIT Press, 1999, pp. 1057–1063. URL: http://dl.acm.org/citation.cfm?id=3009657.3009806.

[68] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning". In: *CoRR* abs/1602.07261 (2016). arXiv: 1602.07261. URL: http://arxiv.org/abs/1602.07261.

[69] Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[70] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

[71] Casper Kaae Sønderby et al. "How to Train Deep Variational Autoencoders and Probabilistic Ladder Networks". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016)*. 2016.

[72] Sebastian Thrun and Lorien Pratt, eds. *Learning to Learn*. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0-7923-8047-9.

[73] Phillip Verbancsics and Josh Harguess. "Generative NeuroEvolution for Deep Learning". In: *CoRR* abs/1312.5355 (2013). arXiv: 1312.5355. URL: http://arxiv.org/abs/1312.5355.

[74] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning*. 1992, pp. 279–292.

[75] Daan Wierstra, Faustino J. Gomez, and Jürgen Schmidhuber. "Modeling Systems with Internal State Using Evolino". In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. GECCO '05. Washington DC, USA: ACM, 2005, pp. 1795–1802. ISBN: 1-59593-010-8. DOI: 10.1145/1068009.1068315. URL: http://doi.acm.org/10.1145/1068009.1068315.

[76] Lingxi Xie and Alan L. Yuille. "Genetic CNN". In: *CoRR* abs/1703.01513 (2017). arXiv: 1703.01513. URL: http://arxiv.org/abs/1703.01513.

[77] Saining Xie et al. "Aggregated Residual Transformations for Deep Neural Networks". In: *CoRR* abs/1611.05431 (2016). arXiv: 1611.05431. URL: http://arxiv.org/abs/1611.05431.

[78] Shengjia Zhao, Jiaming Song, and Stefano Ermon. "InfoVAE: Information Maximizing Variational Autoencoders". In: *CoRR* abs/1706.02262 (2017). arXiv: 1706.02262. URL: http://arxiv.org/abs/1706.02262.

[79] Shengjia Zhao, Jiaming Song, and Stefano Ermon. "Towards Deeper Understanding of Variational Autoencoding Models". In: *CoRR* abs/1702.08658 (2017). arXiv: 1702.08658. URL: http://arxiv.org/abs/1702.08658.

[80] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. "Practical Network Blocks Design with Q-Learning". In: *CoRR* abs/1708.05552 (2017). arXiv: 1708.05552. URL: http://arxiv.org/abs/1708.05552.

[81] Barret Zoph and Quoc V. Le. "Neural Architecture Search with Reinforcement Learning". In: *CoRR* abs/1611.01578 (2016). arXiv: 1611.01578. URL: http://arxiv.org/abs/1611.01578.

[82] Barret Zoph et al. “Learning Transferable Architectures for Scalable Image Recognition”. In: *CoRR* abs/1707.07012 (2017). arXiv: 1707.07012. URL: http://arxiv.org/abs/1707.07012.