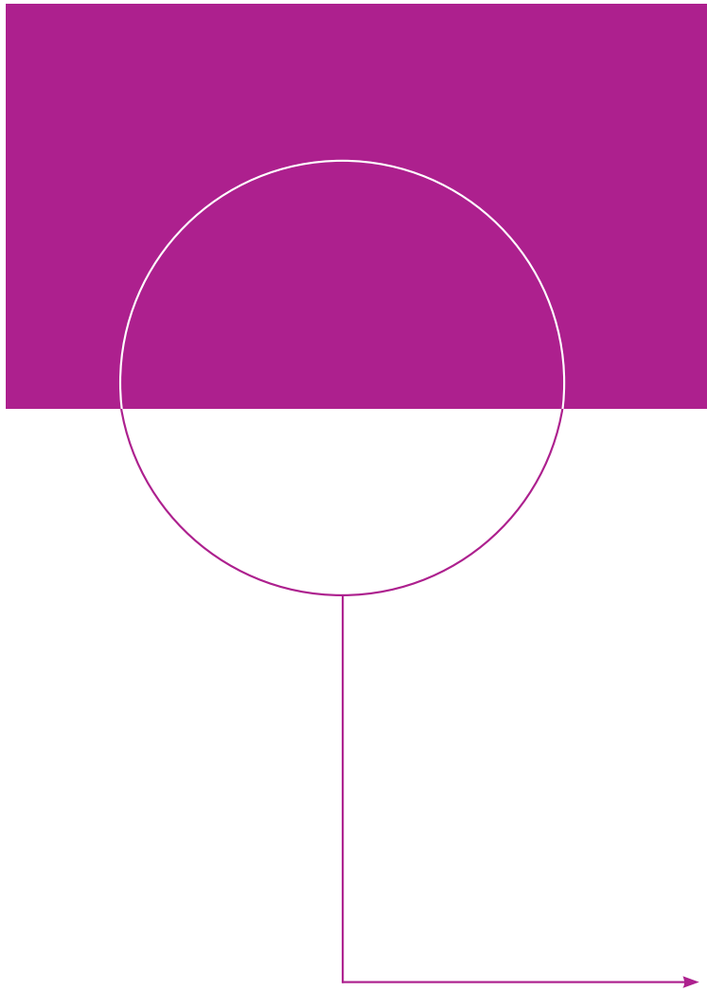


Master's thesis

2020

Johannes Austbø Grande

**NTNU**
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Johannes Austbø Grande

# Analyzing the influence of policy architecture choice on generalizing to new environments in deep reinforcement learning

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

# Analyzing the influence of policy architecture choice on generalizing to new environments in deep reinforcement learning

**Johannes Austbø Grande**

Datateknologi
Submission date: June 2020
Supervisor: Keith Downing
Co-supervisor: Arjun Chandra

Norwegian University of Science and Technology
Department of Computer Science

**Abstract**

Generalization is the ability to transfer knowledge from a seen problem to an unseen problem. It is something that has recently gained a lot of attention in deep Reinforcement Learning (RL), and has shown to both speed up training and improve performance. In machine learning it can be measured by having one training and one evaluation set with similar, but different, problems. Self-attention architectures, such as transformers, have recently shown breakthroughs in sequential modeling fields such as Natural Language Processing (NLP). RL also relies on sequential modeling, and should be able to make use of these breakthroughs to improve generalization. Long Short Term Memory (LSTM), which uses recursion, has been the go-to method for sequential data, and still is in deep RL. Transformers do not have the same rigid temporal dependencies as LSTM built into their structure, and this might be an advantage when it comes to generalization. This thesis investigates the influence of transformers and LSTMs, which model agent policies, has on the generalization ability of agents. This is done using procedurally generated environments to be able to generate the large amount of different problems necessary for trying to measure generalization. The results show that LSTM is still mostly superior to transformers in performance on seen problems, but transformers show that they can match LSTMs in performance on unseen problems, in some circumstances.

**Samandrag**

Generalisering er evnen til å overføre kunnskap fra et kjent problem til et ukjent problem. Det er noe som i det siste har fått mye oppmerksomhet innen dyp "Reinforcement Learning" (RL), og har ført til både raskere trening og bedre løsnigner. I maskinlæring kan det måles ved å ha et trenings- og et evalueringssett med lignende, men forskjellige, problemer. "Self-attention"-arkitekturer, som for eksempel "transformers", har nylig hatt gjennombrudd innen sekvensmodelleringsfelt som "Natural Language Processing" (NLP). RL er også avhengig av sekvensmodellering, og bør kunne benytte seg av disse gjennombruddene til å forbedre generalisering. "Long Short Term Memory" (LSTM), som bruker rekursjon, har vært go-to-metoden for sekvensiel data, og er det fremdeles i dyp RL. "Transformers" har ikke de samme innebygde tidsavhengighetene som LSTM har, og dette kan være en fordel når det gjelder generalisering. Denne oppgaven undersøker påvirkningen transformers"og LSTM, som modell for en agent, har på agenters generaliseringsevner. Dette gjøres ved bruk av tilfeldig genererte problemer for å kunne generere det antallet forskjellige problemer som er nødvendige for å prøve å måle generalisering. Resultatene viser at LSTM fremdeles stort sett er bedre enn "transformers" på kjente problemer, men "transformers" viser at de kan matche LSTM i ytelse på ukjente problemer, under visse omstendigheter.

# Contents

# List of Figures

# List of Tables



## Abbrevations

| | |
|---|---|
| RL | Reinforcement Learning |
| NLP | Natural Language Processing |
| LSTM | Long Short Term Memory |
| GRU | Gated Recurrent Unit |
| RNN | Recurrent Neural Network |
| CNN | Convolutional Neural Network |
| MDP | Markov Decision Process |
| MC | Monte Carlo |
| TD | Temporal Difference |
| PPO | Proximal Policy Optimization |
| seq2seq | Sequence to Sequence |
| ALE | Arcade Learning Environment |

# Chapter 1

# Introduction

Deep reinforcement learning is a mix of deep learning and Reinforcement Learning (RL), and has proven to be very effective at solving some hard tasks. The goal in RL is to maximize cumulative reward in an environment based on a sequence of observations, actions and rewards. Sometimes a good decision cannot be made with only the current observation, in which case some form of memory is needed. The most common way of implementing memory in a deep RL agent is with recurrent network-structures, such as Long Short Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997]. Recently, transformers [Vaswani et al., 2017], which use self-attention, have been used with great success in sequential tasks like Natural Language Processing (NLP) [Devlin et al., 2018]. A version of the transformer has also been used with success in deep RL, outperforming LSTM architectures in both memory-based and more reactive environments [Parisotto et al., 2019]. Another structure that can be used as memory is temporal convolutional networks, which have been shown to outperform recurrent structures in tasks like audio synthesis and machine translation [Bai et al., 2018].

When presented with a sequence of observations, transformers do not make an assumption of which previous observations are more important, although there are methods to insert spatial bias into them. LSTMs do, as they assume spatially close data are more relevant to each other. In addition, LSTMs makes an assumption that any sequence of states can be encoded into a vector of fixed length. This difference in assumptions might affect their ability to model agents in certain environments and their ability to generalize. Recurrent neural networks (RNN) like LSTM can also be difficult to train due to vanishing or explosion of gradients, and have to be trained sequentially. Self-attention, used in transformers, does not have this requirement [Vaswani et al., 2017]. My goal for this is thesis is:

**Research Goal:** Compare the influence transformers and LSTMs have on the generalization ability of agents in reinforcement learning.

Deep RL tends to be geared towards optimizing a policy in a complex environment, and not generalize to unseen examples. It can be extremely easily affected by small changes in the environment [Zhang et al., 2018]. It overfits easily, and effort has to be made in order to avoid this. If you want to learn some very specific environment, like traditional chess, overfitting might be acceptable. But an agent trained on traditional chess can fail when presented with problems like chess960, which has the exact same rules as normal chess, but with different starting positions. Research also shows that training on more general environments can improve performance in the base environment [Packer et al., 2018]. For example, training on chess960 might give better performance in traditional chess than just training on traditional chess.

In contrast to supervised learning, RL often trains on the same data as it tests on. It also has to generate the data itself, which means that data from the same environment might be differently correlated depending on the optimizing algorithm and the model. Deep RL agents often fail to generalize to unseen levels when trained to near optimality on a set of training levels [Justesen et al., 2018]. A human has no problem with this. Some of the reason for this might come from the fact that humans have access to a lot of other information and experience from other tasks, or priors, and access to different and maybe better neural net structures than a deep RL agent currently has. Multi-task learning, where different environments are tried optimized with one agent, and architectures specifically designed for this [Liu et al., 2019] [Teh et al., 2017], is another branch of RL that is being explored. The goal is to learn multiple tasks at once and do all of them better than when only learning one. This sounds like a beneficial strategy, and is something humans do. Measuring generalization of RL agents and finding ways to improve it is an important part of making RL agents solve more difficult and diverse tasks simultaneously. My first research question is:

**Research Question 1:** How much better can transformers generalize than LSTMs in single- and multi-task reinforcement learning?

Generating episodes for RL training on thousands of machines is useful for scalability. This makes stable, off-policy learning important as well, as there might be significant lag between a machine generating an episode and the agent being able to train on it if thousands of episodes are generated every second. Off-policy learning might further skew the distribution of training data and affect generalization. My second research question is:

**Research Question 2:** How does off-policy correction affect the generalization ability of LSTMs and transformers?

## 1.1 Structure

First I will first present some background information about reinforcement learning and some algorithms necessary to understand the rest of the thesis. After that some relevant research articles will be reviewed before I explain my methodology for the experiments. I will then present some interesting results before I give a higher level conclusion about the research goal and questions.

This introduction have been adapted from the project preceding this thesis [Grande, 2019b].

# Chapter 2

# Background

Most background information have been adapted from the project preceding this thesis [Grande, 2019b].

## 2.1 Markov Decision Processes (MDP)

The interaction between an agent and an environment in RL can be described by an MDP, consisting of a set of possible states $S$, a set of possible start states $S_0$, a set of possible actions $A$, dynamics describing the state transition probabilities $p$ (shown in equation 2.1), a reward function $R(s, a, s')$ describing rewards given after transitioning from state $s$ to $s'$ after taking action $a$, and a discount factor $0 <= \gamma <= 1$, describing how much future rewards should be discounted for each time-step taken. An episode in RL is a sequence of these states, actions and rewards. The agent is in a state, takes an action, gets a reward, and repeats this process until a terminal state or some set time-horizon is reached.

$$p(s'|s, a) = Pr[S_t = s'|S_{t-1} = s, A_{t-1} = a] \tag{2.1}$$

The goal of an RL agent, is to maximize the expected, discounted, cumulative reward, also called expected return, for some set of environments. The return of an episode is expressed in equation 2.2, where T is the end of the episode.

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.2}$$

### 2.1.1 Partial Observability

If the best action in an environment only depends on the current state, the environment is Markovian and can be solved with only access to the current state at any given time. In environments used in deep RL, this is usually not the case, and often an agent will benefit from using the entire history of states. Some environments are also partially observable, which means that there are certain features of the environment that the agent can never directly observe. In both cases the agent needs memory to represent the previous states. This memory can be used to keep some kind of model of the unobservable features, or just to make a better action in a state. They are also both usually modeled by the same kind of architectures, which are described in greater detail in section 2.3.

### 2.1.2 Policy and Value Functions

A policy ($\pi(a|s) = Pr[A = a|S = s]$) is a mapping from a state to a probability distribution describing the probability of taking each action in that state. All RL agents has a policy, and it's this policy that we are interested in improving.

A value function is a function that outputs the expected return of a state given a policy $\pi$. There are two kinds of value functions that are used. One is the state-value function seen in equation 2.3, which gives the expected return of a state, and the other is the state-action-value function seen in equation 2.4, which gives the expected return of taking an action in a state.

$$v_\pi(s) = E_\pi[G_{t:T}|S_t = s] \tag{2.3}$$

$$q_\pi(s, a) = E_\pi[G_{t:T}|S_t = s, A_t = a] \tag{2.4}$$

Some RL agents learn models of the environment, which is a probability distribution of the next state given the current state and action ($pr[S_t + 1|S_t, A_t]$). If a model is learned, the state-value function can be used to find a good policy, since it can use the model to look one step ahead, pick the best state, and take the action with the highest probability of leading to that state. If no model is learned, a state-action-value function is needed to find the best action in a state.

An optimal policy is a policy that maximizes the value function

$$v_{optimal}(s) = v_*(s) = \max_\pi v_\pi(s) \tag{2.5}$$

$$q_{optimal}(s, a) = q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.6}$$

for all $s \in S$, $a \in A$.

Usually optimality is not achievable, and approximations are needed. A balance of exploration of not seen or rarely seen before states, and exploitation of well known, high value states, are needed in order to make an effective search algorithm. An agent will probably make bad moves for states that never or rarely arises during training.

### 2.1.3 Monte Carlo (MC)

MC learning methods works by gathering experience and updating a value function. The simplest version consists of gathering experience by sampling the state-action-space by getting a starting state, following the current policy until a terminal state is reached, thus generating a sequence of states, actions and rewards. This sequence can be used to update a state- or state-action-value function, as described in equation 2.7, where $\alpha$ is the learning rate. Each episode is called a rollout, and $\epsilon$-greedy policies, where $0 < \epsilon < 1$ is the probability of taking a random action, are often used to improve exploration [Sutton and Barto, 2017].

$$v(S_t) = v(S_t) + \alpha[G_{t:T} - v(S_t)] \tag{2.7}$$

### 2.1.4 Temporal Difference (TD)

One of the problems with pure MC methods is that they must wait until the end of the episode to update the value function and policy. TD learning instead updates the value function based on the $n > 0$ future rewards and current value function estimates. Updating an estimate based on another estimate is called bootstrapping. Equation 2.8 describes the 1-step value function update, which updates the value estimate based on the reward gotten, and the estimate for the next state. This has a different drawback, since it has to iteratively update the estimates to propagate any changes in the estimates. Equation 2.9 is the n-step generalized version of the TD update, which updates the value function based on the n next rewards and the estimate of the n'th state. This re-introduces some of the problems from MC with having to wait for the entire rollout [Sutton and Barto, 2017].

$$v(S_t) = v(S_t) + \alpha[R_{t+1} + \gamma v(S_{t+1}) - v(S_t)] \tag{2.8}$$

$$v(S_t) = v(S_t) + \alpha[G_{t:n-1} + \gamma^{n-1} v(S_{t+n}) - v(S_{t+n-1})] \tag{2.9}$$

### 2.1.5 Off-policy and Importance Sampling

In RL, a policy is needed to generate training data. If the RL algorithm is trying to learn a different policy (target policy $\pi$) than the one generating the data (behavioural policy $b$), the learning is happening off-policy. When training on old data, which can happen if there is lag between the generation of data and the use of that data in training, some way to weight the training data based on relevance to the target (or current) policy might be useful. Importance sampling is one such strategy, and it's goal is to reach the target policy more efficiently.

Importance sampling is a technique for sampling some distribution of interest from a different distribution. To be able to do this, there has to be some overlap between the two distributions. Importance sampling in RL use some form of weighting of the data based on how relevant it is to the target policy. This weighing usually use some form of equation 2.10 to weight data based on how likely it is to occur in the target policy compared to the behavioural policy.

$$p_t = \frac{\pi(a_t|s_t)}{b(a_t|s_t)} \tag{2.10}$$

The further the two policies are from each other, the higher variance this will have because of finite sampling. In the extreme case, all the importance weights will be close to zero or wandering off to infinity, and nothing will be able to be learned because the policies are scarcely overlapping.

### 2.1.6 Q-learning

Q-learning uses the action-state-value, or Q-value, to learn a model-less policy as shown in equation 2.11. It is an off-policy method as it uses the best Q-value in a state to estimate all the Q-values values in the state, seen by the $\max_a q(s_{t+1}, a)$ in the equation.

$$q(s_t, a_t) = q(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t)] \tag{2.11}$$

Deep Q-learning, which use a neural network to estimate the Q-values, can be unstable during training, because of the combination of off-policy learning, function approximation and bootstrapping [Sutton and Barto, 2017]. There are ways to make deep Q-learning more stable, for instance by storing a large amount of episodes, as done in the APE-X architecture [Horgan et al., 2018].

## 2.2 Policy Gradient

A policy gradient method learns a parameterized policy, which is a probability distribution over actions $a$ given a state $s$ and a parameter vector $\theta$; $\pi(a|s,\theta) = Pr(A = a|S = s, w = \theta)$. A policy gradient method might also learn a value function, $v(s,w)$, where some of the parameters might be shared between w and $\theta$ to help learn the policy faster. The policy is updated based on some differentiable performance measure $J(\theta)$, where the policy parameters are updated using stochastic gradient ascent as shown in equation 2.12 [Sutton and Barto, 2017].

While the state-action-value function $Q(a,s)$ is used in Q-learning to learn a policy, how good an action is also depends on how good the other actions are in the state it's taken in. The advantage function, shown in equation 2.13, makes it easier to differentiate between actions by subtracting the average state-value from the state-action-value. This makes learning more stable.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.12}$$

$$A_t(a_t, s_t) = Q(s_t, a_t) - V(s_t) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.13}$$

The most commonly used performance measure gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}[A(a,s)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}] \tag{2.14}$$

where $A$ is the advantage function and $\mathbb{E}$ is an average based on the sampled data. Intuitively, the numerator can be thought of as the gradient maximizing the value of the policy, based on the samples collected. The denominator scales the numerator based on how often it's sampled, so it doesn't point in a direction simply because it's sampled more often. The advantage function scales the direction based on how good the sampled action is, making the gradient point in an overall better direction. Additionally, some entropy might also be added to the update to stop the policy from getting stuck in a local minimum and encourage exploration.

### 2.2.1 Proximal Policy Optimization (PPO)

PPO [Schulman et al., 2017] maximizes a clipped surrogate function. The gradient of the surrogate function is shown in equation 2.15. It's similar to equation 2.14, but uses an older policy in the denominator. The policies' probability ratio ($\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_{\theta old}(a|s)}$) is also limited by "clipping", which means restricting this ratio to some interval between $1-\epsilon$ and $1+\epsilon$, where $\epsilon$ is a hyperparameter. It can also also be done with for instance Kullback-Leibler divergence, which is a way to

measure differences between two probability distributions, limiting the update if the difference is large.

As long as the new policy is close to the old, maximization of this surrogate function will also maximize the real objective, seen in equation 2.14. Clipping the surrogate function makes it flat if the policies are too different from each other. How different depends on the $\epsilon$-parameter. This makes it possible to run many iterations of gradient ascent on the same data, updating the policy weights and the policy probability ratio with each update, until such a flat area is encountered. This results in a policy that is not too different from the old one. PPO therefore allows for multiple iterations of gradient ascent on the same data without much increase in variance.

$$\nabla_\theta J(\theta) = \mathbb{E}[A(a,s)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_{\theta old}(a|s)}] \tag{2.15}$$

### 2.2.2 Asynchronous Advantage-Actor-Critic (A3C)

2.14 An advantage actor-critic (A2C) [Clemente et al., 2017] is a policy-gradient method that learns both a parameterized policy and a value-function and uses the advantage function to update the policy, as shown in equation 2.14. Asynchronous advantage-actor-critic (A3C) [Mnih et al., 2016] uses a set of actors to generate training data and calculate gradients using this data. The actors send the gradients to a centralized learner which updates the policy- and value-functions, which in turn sends the updated parameters to the actors.

### 2.2.3 Impala

Impala [Espeholt et al., 2018] builds on the actor-critic model. It has multiple, decentralized actors and one or more centralized learners, although one learner is usually used. In A3C each individual actor calculates the update gradients from the it's episodes and sends these directly to a learner that uses them to update the weights. In contrast, Impala actors send the whole episode directly to a learner that calculates the update gradients. This allows for higher throughput, since the data can be sent to a central server as soon as it is collected, the gradient calculation can happen centrally on a GPU, and the actors can send episode trajectories and update their weights whenever they want. If each actor calculate their own gradients the entire process needs to be synchronized to avoid actors calculating updates for old policies. If multiple learners are used, the learners shares gradients with each other with regular intervals.

Since the episodes are sent to a learning queue, and not used to calculate gradients directly as in A3C, there might be lag between the data generation and the gradient calculation. Therefore, the policy used to generate the data might

be different than the current one, which means the learning is happening off-policy. Impala uses v-trace, an off-policy actor-critic algorithm using truncated importance sampling to try to correct for this.

V-trace works, as any importance sampling algorithm, by increasing the weight of rewards that are more likely to happen in the current policy than the behavioral policy, and decreasing them if the opposite is true. This increases variance, so v-trace clips this scaling to assure the variance is not increased by too much.

## 2.3 Function Approximation / Neural Networks

In deep RL we seek a function that outputs the best action in a state, the value of a state, or both. For large and complex environments, function approximation is needed. The most common one being artificial neural networks.

Artificial neural networks use neurons, consisting of an input, an output, and an activation function, organized into different kinds of layers. The most simple being a fully connected feed forward layer, where a layer of neurons are all connected to the previous layer's outputs, and the next layer's inputs. This simplest form form of layer is referred to as a dense layer.

### 2.3.1 Sequence to sequence (seq2seq)

A seq2seq model takes a sequence as input and outputs another sequence. An English-to-German translator is an example of a seq2seq model. An RL agent is similar to this, as it often uses a sequence of states to output an action. However, after outputting an action, the agent gets another state it can use to output the next action. This is not the case in a machine translator, where all the elements in the output sequence are dependent on the same input sequence. Seq2seq modeling can be done in many ways, for example with recurrent units, to sequentially encode a sequence, then decode it into a different sequence.

### 2.3.2 Recurrent units and LSTM

In RL it's often not possible to make a good decision when only the observation from the current state is available. Previous observations might contain important information. In these cases, recurrent units that update their hidden state for each sequential input can be used. The simplest kind of these take their output as additional input for the next state.

It turns out that this simple architecture, while theoretically having the capability of detecting long-range dependencies, does so poorly. This is why Long-Short-Term-Memory (LSTM) is usually used instead, which is capable of remembering long-range, temporal dependencies efficiently.

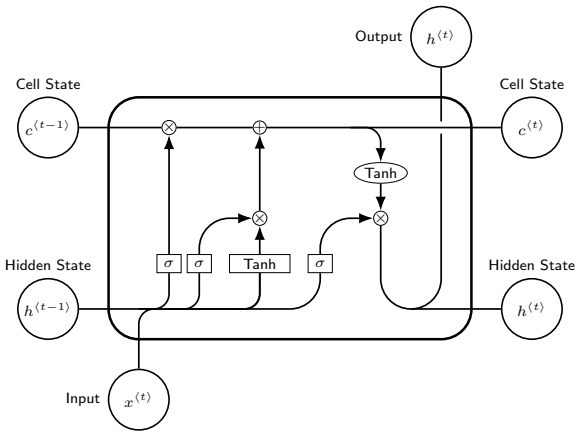



Figure 2.1: LSTM cell

The major difference between an LSTM cell and a basic RNN cell is that an LSTM cell has a cell state in addition to the hidden state. The cell state is what allows the LSTM cell to remember over longer periods of time than a basic RNN cell [Hochreiter and Schmidhuber, 1997]. Both of these states are recurrent, and like an RNN cell the output and the hidden state are the same. In figure 2.1 you can see a typical LSTM cell, altough there are different variants. The squares are layers, which takes two values as input and has learnable weights. These values are added together before being passed through some activation function, either the logistic function ($\sigma$) or hyperbolic tangent. The small circles are pointwise operations without learnable weights. You can see that all the layers takes the input and the hidden state as their inputs.

The LSTM works by using the cell state as a kind of memory. The cell state is first multiplied by some value calculated from learned weights, the input, and the hidden state. This allows the LSTM to forget information by multiplying by a number close to zero. Another value, also calculated by learned weights, the input, and the hidden state, is then added to the cell state. This allows the cell to add new memories. Finally, the new hidden state and output state are calculated by the input, hidden state and the cell state. Multiple of these cells are combined into layers.

Multiple versions of LSTM architectures have been designed both by hand and by computers, and some of them perform better in specific environments, but nothing found is universally a better design [Chung et al., 2014] [Greff et al., 2016]. A Gated Recurrent Unit (GRU) is a version without a hidden state and with the cell state being used as the output.

### 2.3.3 Self-attention

The goal of self-attention is to calculate the relevance between elements in a sequence and use these values to help the neural network focus on the right input in order to speed up convergence and peak on an overall better performance.

One way of calculating self-attention that has had success in natural language processing [Vaswani et al., 2017] [Devlin et al., 2018], is expressed in equation 2.16 and in figure 2.2. With a sequence as input it calculates three matrices from each element in the sequence by learned weights; a query (Q), a key (K) and a value (V). An attention score is calculated by multiplying Q and K, and dividing by the square root of the number of dimensions to scale it. Softmax is used over all the attention scores to give you a probability distribution. The final multiplication by V gives you an attention matrix between all elements in the sequence.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.16}$$

This can be done multiple times, in parallel, for each input sequence. You then concatenate the outputs and multiply them with a matrix to reduce them to the desired dimension. This is called multi-head attention. Multi-head attention scores allow more nuanced attention, and are empirically better than calculating only one.

Since self-attention is permutation independent, positional encoding is sometimes used to induce temporal information into a sequence, which makes temporal close elements give each other higher attention scores. There are also other ways to make self-attention mechanism take temporal information into account, and remove the need for positional encoding [Shaw et al., 2018].

### 2.3.4 Transformers

A transformer is an architecture used in seq2seq modeling. In figure 2.2 you can see the basic structure of a transformer. Positional encoding is often added before the input to add spatial information to it. Attention scores are then calculated using multi-head attention, which output is concatenated and normalized [Ba et al., 2016] before it's passed through a dense network and normalized again. Residual connections are added around both the multi-head attention layer and the dense layer.

The input into a transformer is of dimensions T x D. T is the temporal dimension, which is how many elements there are in the input sequence. This affects the time horizon for which the architecture can detect dependencies. D is the spatial dimension, which is how large one element is. Unlike recurrent networks, which can in theory detect any time dependency, no matter the distance

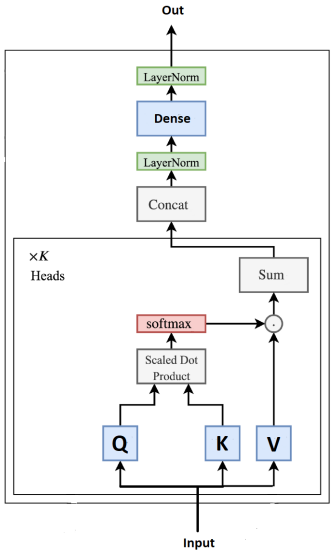



Figure 2.2: Transformer

between them, transformers can only detect time dependencies that happens T time steps or less apart.

Unlike LSTM, a transformer does not have a recurrent connection and therefore can't form memories in the same way LSTM does. It instead takes a sequence as input, which make them able to act on previous information, like LSTM. Some versions memorize the input from previous time-steps making them more efficient as the input does not have to go through all the previous layers. This is done for efficiency.

### 2.3.5 Temporal Convolutional Networks

A convolutional neural network (CNN) layer is based on two assumptions. The first is that spatially close data has more in common than data that are far away from each other. The second one is that features that are found in one part of the input, are also useful in other parts of the input. This is done by having multiple filters with the same weights striding over the input.

Convolutional networks are state-of-the-art in image classification. It's therefore also used in end-to-end RL when the input is an image or a sequence of images. Convolutional networks can also be used in seq2seq models in addition to, or instead of, recurrence and self-attention. This is called a temporal convolutional network, as its done over a temporal structure and not a spatial structure.

## 2.4 Generalization

The difference between the performance on the test data and the training data is called the generalization gap. In supervised learning, different regularization methods are used to tighten this gap. If the testing set and the training set are the same, there is no need for these regularization methods. Using the same training and testing data is common in RL, and until a couple of years ago regularization methods had rarely been used. However, it has been shown that deep RL can take advantage of regularization techniques from supervised learning, like dropout, when the train and test sets are not the same [Cobbe et al., 2018]. Since the agent's policy changes with time in RL, the distribution of data can change during training. This coupled with often sparse rewards, makes generalization in RL and the measuring of it more difficult than in supervised learning.

Say you are training an agent on a video game. If you are training on one level, there is little to no generalization needed. It's very likely that the agent will just remember the level. When an agent remembers a level it's overfitted to that level. This is sometimes fine if all you care about is that level. If you are training on multiple levels of the same video game, there is generalization between the levels. The generalization increases with the number of levels as it becomes infeasible to just remember the levels. If you are training on levels from different video games, there is also generalization between games. This last one is called multi-task learning.

Interpolation (also called in-distribution generalization) is generalization to data in the same distribution. Extrapolation (also called out-of-distribution generalization) is generalization to data in a different distribution. Extrapolation is trickier than interpolation and has more pitfalls, as it is statistically riskier to do.

### 2.4.1 Multi-Task Learning

In multi-task learning an agent is trained on multiple different tasks/environments. The goal is that features found in one environment are also useful in other environments, thus improving performance in all environments. However, if the environments are too different, the agent might perform worse than if it was just trained at each one individually [Espeholt et al., 2018].

DeepMind Lab (DmLab) [Beattie et al., 2016] is an environment suite containing 30 3D-environments designed for multi-task learning. In all the environments an agent is controlling a robot from a first-person perspective, and has to learn how to do so with visual input in the form of RGB images. The goals vary from game to game, but they usually include some kind of maze navigation and puzzle solving.

The Arcade Learning Environment (ALE) [Bellemare et al., 2013] is another environment suite containing 57 popular Atari games. In all the environments an agent is playing the game using a simulated joystick and a button. The games are very diverse, which makes it difficult to train one agent to do well on all of them. Multi-task learning on these environments usually results in worse performance than single-task learning on any individual environment.

# Chapter 3

# Related Work

Most of the literature reviews have been adapted from the project preceding this thesis [Grande, 2019b].

## 3.1 Structured Literature Review

A structured literature review was conducted to find interesting ideas to build upon. The first step was to read some papers presented by my supervisor and go through the citations made by these papers. The next step was to search google-scholar and arxiv for keywords. The keywords used where generalization, multitask learning, singletask learning, LSTM and transformer. The abstract of the most interesting and seemingly relevant papers where then read, and if the paper was indeed relevant, the references made by that paper would be checked. The most relevant papers found will be presented in the literature review.

## 3.2 Training algorithms

In this section two articles that present new training algorithms for RL and tries to use the algorithms presented to improve generalization is reviewed.

**"Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures"** [Espeholt et al., 2018] presents Impala, a method designed for high throughput and distributed multi- and single-task learning, mentioned in section 2.2.3.

The paper evaluates 3 aspects of the algorithm. Sample efficiency, performance, and the effectiveness of v-trace. They compare it to A3C and A2C on some

DmLab and ALE environments. Impala approximately produce 1.5 times the amount of frames per second of batched A2C on a single machine, and double that of A3C on a distributed setup. With the same number of training-data, it also outperforms A2C on 2 out of 5 DmLab tasks tested, and get similar scores on 3 of them.

To test the effect of Impala with v-trace, they compare it to Impala with no off-policy correction, $\epsilon$-correction, which means adding a small value to the gradient calculation to avoid numerical instabilities, and 1-step importance sampling. V-trace is an n-step importance sampling method, and the difference between the two is similar to the difference between 1-step and n-step TD-learning. 1-step is comparable or a bit better in 2 of the tasks, and both no correction and $\epsilon$-correction performs significantly worse. V-trace also benefits more from a replay-buffer, and is the only correction which performance is consistently improved by the addition of such. This makes sense since a replay-buffer makes the learning more off-policy, as there is more lag between data generation and gradient calculation. The positive effect of v-trace when training on off-policy data is clearly seen.

Finally the paper tests multi-task performance with the DmLab and ALE environments. Impala beats A3C by 46.5% to 23.8% where 100% is average human performance, when trained on all 30 environments of DmLab. Impala also performs better than when trained on them individually (46.5% to 44.5%). This shows positive transfer between these tasks. This is not the case in ALE, where when trained on all 57 games, performance decreases. This is interesting, and says something about the need for better approaches for positive transfer in such environments. It would also be interesting to explore the difference in distributions of environments where positive transfer occurs versus ones where it does not.

**"Off-Policy Actor-Critic with Shared Experience Replay"** [Schmitt et al., 2019] explores the stability of off-policy actor-critic learning and the effects of replay-buffers in Impala.

The truncating of importance weights in v-trace does reduce variance, but introduces a bias in the training data. Even though v-trace is designed for off-policy learning, it struggles when the data is too much off-policy. To mitigate this, they propose mixing off-policy from a replay-buffer with on-policy data during training. Their experiments shows that mixing off-policy and on-policy in a 7/8 ratio is stable and improves the performance of Impala. Ratios of 1/2 and 3/4 are also stable, but results in lower sample-efficiency. They also compare uniform and prioritized replay-buffers, and find that a prioritized replay-buffer gives little benefits and introduces bias and more hyper-parameters for weighting schemes.

It's worth mentioning that APE-X [Horgan et al., 2018] implements a prioritized replay-buffer that works well in Deep Q-learning.

## 3.3 Neural Net Structures

In this section some articles that present new neural net structures, such as different versions of transformers, with promising performance for RL are reviewed.

**"Attention is all you need"** [Vaswani et al., 2017] explores the use of self-attention in natural language processing (NLP). It introduces the transformer, which relies on self-attention, as a component of an encoder-decoder structure, compares this analytically to recurrent and convolutional models, and empirically to other state-of-the-art models.

The paper mentions three reasons why self-attention might be better at NLP than recurrent and convolutional models. The first is the computational complexity. Both convolutional networks and recurrent network are generally slower than self-attention for this type of problem. Recurrent networks are faster than self-attention when the temporal dimension of the input is shorter than the spatial dimension of each individual element in the input, but that's usually not the case. The second is parallelizability, which recurrent networks are bad at, since they are dependent on the output of the previous element in a sequence to calculate the next output.

The third is the path length between two elements. This is important in order to be able to learn long-range dependencies. In a recurrent network, the path length is the number of elements between the two elements, as the cell state is updated sequentially, and has to keep some information from processing the first element to processing the second, to be able to detect a dependency. In a convolutional network it's the number of elements divided by the size of the kernel, as a kernel smaller than the sequence does not directly connect the elements, but have to use a stack of these kernels. In a self-attention network it's one, as the attention score between all elements are calculated directly. The paper tests an encoder-decoder structure consisting of 6 encoders and 6 decoders made up by one transformer each, seen in figure 2.2. The goal is to encode a sentence from an actual language into a learned language, only understood by the agent, then decode it into another actual language.

Their experiments show that this structure performed better than the state of the art NLP in 2017. The model required less than 1/4 of the computation of other models, and using the Bilingual Evaluation Understudy (BLEU) score, it beat them by 2 (28.4 to 26.36) in the WMT 2014 English-to-German translation task, and 41.8 to 41.29 in the WMT 2014 English-to-French translation task. They also tested the structure on an English constituency parsing (parsing text to syntactic trees) test. It performed well, but was beaten by a recurrent architecture (Recurrent Neural Network Grammar), showing that it's not necessarily better than recurrent structures.

A version of this architecture was also used in the state-of-the-art BERT archi-

tecture for NLP [Devlin et al., 2018].

**"Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context"** [Dai et al., 2019] introduces the transformerXL (TrXL) which improves the design of the base transformer. It removes the need for fixed-length input by splitting up a variable-length input-sequence into fixed-length segments, and using recursion by reusing the encoding of a segment as input when encoding the next segment. To speed up training, they avoid doing recursion during training, and only during testing. This makes training 1800 faster than the original transformer.

To measure how well their structure learns long-range dependencies, they introduce Relative Effective Context Length (RECL). Its goal is to measure the length of learned dependencies. Based on RECL, TrXL learns dependencies 80% longer than RNNs, and 450% longer than vanilla transformers. TrXL also outperforms other transformers on data with only short-term dependencies.

**"Stabilizing Transformers for Reinforcement Learning"** [Parisotto et al., 2019] builds on the TrXL architecture, and tries to stabilize them for deep RL, since transformers have been unstable when used there. The paper introduces the gated transformer XL (GTrXL), shown in figure 3.1. GTrXL moves layer normalization before the multi-head attention and uses gating mechanisms for the residual connection instead addition.

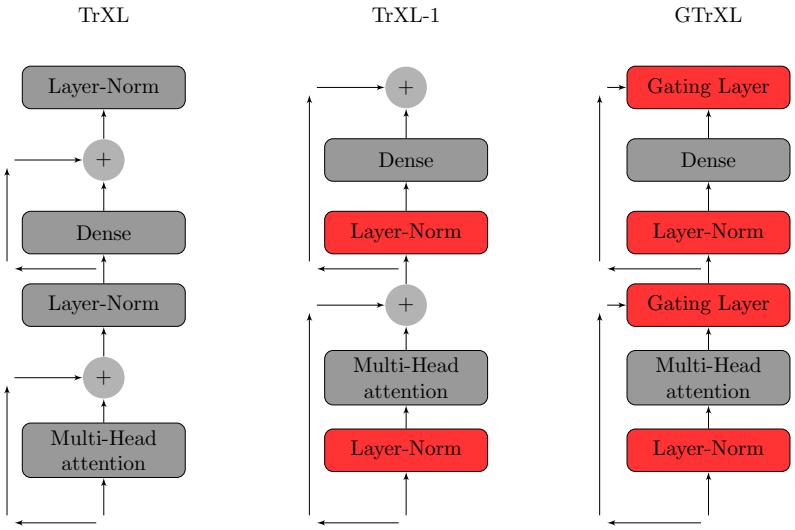



Figure 3.1: 3 different transformer architectures

The moving of layer normalization allows the output to be close to 0 at initialization time, as it's not normalized. This makes it easier for an agent to learn

behavior based only on the current state right after weight-initialization, which improves learning of more reactive behavior that often needs to be learned before memory-based behavior. For example, an agent needs to learn how to walk before it can learn where to walk or where it has walked. Their experiments show that this does indeed improve stability.

The paper also compares a bunch of different gating mechanisms. The gating mechanism from GRU works better, or is at least as good as the other gating mechanisms tested.

To test the memorization capability, they compare GTrXL to LSTM architectures on the "Numpad" environment which consist of a controllable robot on a numpad (x times x numbers) and a hidden sequence of numbers. The robot can move on a platform between the numbers and press them, and gets rewarded each time it presses the next number in the sequence. After it has pressed all of them, the sequence starts over again. This means that an agent that remembers the sequence well gets more rewards. Their experiments show that GTrXL (with GRU gating) trains faster and performs better than LSTM structure in this environment. Approximately 2 times better at a 4x4 numpad, and  4 times at a 2x2 numpad.

They further tests GTrXL against LSTM, TrXL and TrXL1 on 30 DmLab tasks which requires both reactivity and memory. GTrXL outperforms LSTM by 10%, but uses  2 times as many parameters. TrXL does very poorly, and TrXL1 scores between GTrXL and LSTM with as many parameters as GTrXL. The presented architecture offers a new architecture to RL agents that need memory.

**"A Simple Randomization Technique for Generalization in Deep Reinforcement Learning"** [Lee et al., 2019] proposes a way of training deep RL agents in visual environments by adding a randomization layer to the agent's network. The layer is initialized with a defined distribution, so it only alters the visuals and not the actual goal. By randomizing the layer for each iteration, the agent should be able to converge to a policy robust to irrelevant visual changes, like a different background, by finding invariant features. The layer is tried added at multiple places in the network, but adding it as the first layer has the best performance for both environments seen and not seen during training.

They compare their method against more conventional data augmentation and regularization methods using the Coinrun environment with a different set of textures during training and testing [Cobbe et al., 2018]. All methods do well when tested with textures used during training, but their method doubles the performance of any other method when textures not seen during training are used.

They also test their network against a network without a randomization layer in the DmLab [Beattie et al., 2016] and Surreal Robotics [Fan et al., 2018] environments. The randomization network beats the non-randomization network

by 6 times the score when training with one texture, and by 1,5 - 2 times the score using 16 - 25 textures. This shows that training with multiple textures might indeed work, but you need a lot of them. This makes sense as using a different texture for each input is kind of what the randomization layer is doing.

This paper shows that deep RL agents using conventional regularization and data augmentation methods may fail when faced with completely irrelevant changes in the environment.

## 3.4 Measuring Generalization

In this section some articles exploring ways to measure generalization are reviewed.

**"Assessing Generalization in Deep Reinforcement Learning"** [Packer et al., 2018] explores how well some RL methods interpolate and extrapolate. To do this the paper defines 3 types of environment parameter settings. Deterministic (D): fixed parameters. Random (R): parameters are sampled from a hypercube with the deterministic parameters as the centre. Extreme (E): parameters are sampled from hypercubes anchored at the vertices of the random hypercube. This makes for 9 different training and testing pairs. Training on D and testing on D is how a lot of RL are done. Extrapolation is measured by training on D and testing on R or E and training on R and testing on E. Interpolation is measured by training on R and testing on R. A set of environments from OpenAI gym are used.

Ensemble policy optimization (EPOpt) is a method for RL training that aims to create policies that are robust to out-of-distribution environments, which should give good extrapolation performance. At each iteration it generates a number of episodes, switches environment parameters between each episode, and uses some fraction of the episodes with the lowest return to update the policy. RL2 tries to do the same as EPOpt, but does it by using a recurrent network with inputs being a sequence of states, actions and rewards instead of just states which is more common. It also samples multiple episodes from the environment without switching parameters. Both Epopt and RL2 are run on top of another RL optimization method, which in this case is A2C and PPO. The algorithms that are compared are: A2C, PPO, EPOpt-A2C, EPOpt-PPO, RL2-A2C, RL2-PPO.

The experiments performed shows that A2C and PPO, agents with no correction for generalization, performs well on both interpolation and extrapolation. A2C does best, beating most EPOpt and RL2 methods and performing as well as EPOpt-PPO. It seems that training on the same environment with different parameters, is a decent strategy for generalization in itself.

Sometimes performance on D when trained on R is better than when trained on D. This might be because it finds better representations of features in the environment. It shows that adding variation to a fixed environment can help stabilize training. EPOpt and RL2 also performs poorly when trained on D. This makes sense as these methods are not designed to do so. It also raises the question of whether training on D and testing on R or E actually tests extrapolation fairly. This is not mentioned in the paper.

**"A Study on Overfitting in Deep Reinforcement Learning"** [Zhang et al., 2018] explores how overfitting in deep RL occurs, tests the effects of some methods that are used to reduce overfitting and increase generalization, and how inductive biases, e.g. architectural choices favoring nature of data being modeled, in neural networks can affect deep RL.

The paper uses A3C to solve a grid-world maze environment with three difficulty levels. The mazes are procedurally generated, and they sample a number of different seeds for training and testing sets. They define generalization as the difference between performance on these sets. The experiments are run with 4 different agents training on 10, 100, 1000 and 10000 different seeds. Unsurprisingly, more training mazes result in better performance on the test set. Agents with 10 training levels also experience decreasing performance on the test set after training for a while, which hints at overfitting to training mazes.

Furthermore, the paper tests the memorization-ability of the neural networks by flipping the sign of the reward (reward-flipping) in some fraction of the mazes. Reward-flipping in half of the mazes makes it impossible to achieve a good score, except if agents memorize the maze. Agents with 10 training levels achieve optimal scores on training set, and agents with 10000 training levels still achieve a non-trivial score. This shows that networks are able to remember quite a few levels, and adding stochasticity by procedural generation, does not prevent overfitting.

When an agent memorizes, it ignores states and performs various action sequences, which can in a lot of cases be a good strategy. This is overfitting, as the agent fails when small variations are added to the environment, but it can be useful. Humans do this. However, the agent needs to be able to figure out when the sequence didn't work, and change it's policy accordingly. That memorization leads to a poor generalization is expected.

The paper goes on to discuss how inductive biases in the neural network needs to be compatible with the environment. CNNs outperform dense networks when used on spatially correlated input. They test how these two structures compare in the maze environment, and find that dense networks are better at when trained on one maze, but CNNs generalize better. The neural network model used is very important, and finding structures that fit the environment might be a good way to achieve better generalization.

**"Reasoning and Generalization in RL: A Tool Use Perspective"** [Wenke et al., 2019] explores tool use as an RL problem. Tool use requires multiple levels of generalization, and is an important part of the success of humans. The paper introduces a 2D grid-world environment containing an agent, a tool, two tubes (walls) opposite each other, two one-way trap-doors that open the same way between the tubes, and a piece of food enclosed between the tubes and trap-doors which the agent has to use the tool to get. It gets 50 actions to do so, or it loses. The action space consist of move actions, which moves the agent in the specified direction, and grasp actions which moves the agent and a tool, but only if the tool is adjacent.

They define 3 environment transfers. A perceptual transfer, which randomizes shape, position and rotation of various objects. The food is still in an enclosed room that the agent must use a tool to get. A structural transfer, which randomizes colors. And a symbolic transfer, which switches the position of the tool with the position of a tube or a trap door. None of these transfers change the agent or the food, and any number of them can be applied to the base environment.

The agent is trained with PPO and PPO with intrinsic curiosity module (ICM) [Pathak et al., 2017] to encourage exploration since the rewards are sparse as the agent only gets rewarded when winning. Agent trained with PPO-ICM manages to solve the basic task, but not any transfer tasks after training on only the basic task. Agents trained with only PPO fails to solve anything, which is probably because of the sparse environment as PPO struggles in sparse environments.

Their generalization experiment results shows that PPO-ICM reaches 40% win-rate when trained and tested on symbolic transfer, 2% at perceptual transfer and 0% at structural transfer. Combinations of the transfers are between 40 and 0%. This shows that the agent generalizes poorly in most cases. The paper concludes that algorithms or models with priors that makes structured exploration and tool use more intuitive for the agent, like a CNN makes 2D and 3D space more intuitive, are probably needed to solve these tasks effectively. Just getting a reward at success without any prior knowledge requires a lot of exploration, and close to random exploration is inefficient.

**"Quantifying Generalization in Reinforcement Learning"** [Cobbe et al., 2018] explores the generalization of deep RL by designing two new environments called Coinrun, seen in figure 3.2, and Coinrun Platform. Coinrun consists of a procedural generated, fully observable environment in which an agent is to collect a bunch of coins to win. Coinrun Platform has the same design principle, but the entire environment is not fully observable at any given time. They're both generated based on a seed of 32 bits, which makes it very unlikely for any two generated environments to be the same.

The paper presents two ways of measuring generalization. One way is to train on an unbound set of levels, measure the performance during training on a

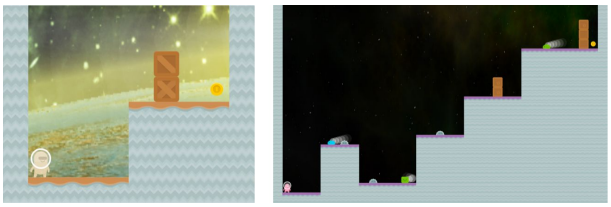


Figure 3.2: Two instances of the Coinrun environment [Cobbe et al., 2018]

separate test-set, and compare the performance curves. An agent that needs fewer training levels to achieve the same test performance generalizes better than one that needs more. Another way is to train on a fixed set of levels then measure the performance on a separate test-set. In both cases the agent can't simply remember how to get all the coins in the levels, but has to find a generalized way of achieving that goal.

The experiments are run with a convolutional structure and trained with PPO. They find that training on a small set of levels, even when there are thousands of them, results in significant overfitting. They also show that that L2 regularization, dropout, data augmentation, and batch normalization increase generalization. In addition, it seems that a larger neural network is both better at solving individual levels and better at generalization than a smaller one.

In Coinrun Platform the agents also have an LSTM layer. It is observed that these agents need to train for a long time to tighten the generalization gap. It is not clear why, but LSTMs might not be the best structure for generalization in memory based agents.

**"Leveraging Procedural Generation to Benchmark Reinforcement Learning"** [Cobbe et al., 2019] presents a set of 16 procedural generated environments, named Procgen seen in figure 3.3, built with the same design principles as the Coinrun environment, and is designed to be used to test generalization in RL. All the environments are procedurally generated to create as much diversity as possible, although no measure of diversity is given. The goal is to control a robot with low level motor actions (walk, jump, etc.) based on visual input. The state and action spaces are the same across all environments, and there are two difficulty levels, hard and easy, to make it possible to use without as much computational power available.

To test generalization they recommend training on 500 instances of an environment with 200 million time-steps, then test on a different set of environments. They find that the generalization gap is usually pretty large, and in the cases where it's not, the training performance is also low. The paper also trains agents on a deterministic sequence of instances, that means they start training on one instance of an environment, then when the agent solves it, it proceeds to the next. If it at any point fails, it goes back to the first instance. The agents rarely

(a) bigfish (b) bossfight (c) caveflyer (d) chaser

(e) climber (f) coinrun (g) dodgeball (h) fruitbot

(i) heist (j) jumper (k) leaper (l) maze

(m) miner (n) ninja (o) plunder (p) starpilot

Figure 3.3: The 16 environments from the Procgen environment suite [Cobbe et al., 2019]

solve more than 20 instances, and when tested on the same instances but in random order, they fail, suggesting that the agents don't generalize well when trained like that, but just memorize some action sequence.

**"Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation"** [Justesen et al., 2018] introduces the "General Video Game AI" framework (GVG-AI). It's capable of procedural generation of levels with different difficulties ranging from 0 to 1 for four games (Boulderdash, Frogs, Solarfox and Zelda), all of which are two-dimensional grid-worlds. The goal of the paper is to explore how procedural generated levels during training, and it's distribution, affects generalization. 4 human generated levels are also included for perspective.

The paper compares two ways of generating training levels. Progressive procedural content generation (PPCG), which is a method increasing training level difficulty as the agent wins and decreasing it as it loses, and PCG X, which is a method that generates training levels with a fixed difficulty. The testing is done on a new set of generated levels likely not encountered during training, as there are $10^8$ possible levels at the lowest difficulty and $10^{24}$ at the highest.

The results shows that in some games (Boulderdash and Solarfox) PPCG achieves good scores, but rarely wins, and thus continues training on low difficulty levels. PCG X at max difficulty outperforms PPCG in these games. In games where PPCG reaches max difficulty (Frogs and Zelda) it outperforms a PCG X agent at max difficulty. The most notable difference was that PPCG was able to achieve a win-rate of 57% on difficult Frogs levels, compared to 0% for PCG X. This presumably is caused by learning a better representations of game features early on when training on easy levels, and thus having an easier time of generalizing. In some games, PPCG and PCG X did not manage to solve the human generated levels (Boulderdash and Frogs), hinting at a discrepancy in the level distribution of the procedural generated levels and the human levels.

The paper goes on to explore the distribution of procedural generated levels by principal component analysis and clustering techniques. 1000 levels with max difficulty and the 4 human generated levels are used for the clustering. They discover that in the distribution of Boulderdash levels, the human levels are outliers, which is presumably why PPCG and PCG X did not manage to solve those. The Frogs distribution consists of a lot of sizeable clusters, some of which contains human levels, and the reason why PPCG did not manage to solve the human Frogs levels, might just be a coincidence, as it only got a 57% win-rate in this game. The clustering reveals that care has to be taken when procedurally generating levels to get a good distribution.

## 3.5 Motivation

The main theme in most of these articles is to measure and improve generalization. There have been multiple attempts at doing both at the same time, as they are affected by each other. This is done through new neural network structures, new training policies, or new ways to measure generalization. The most common way to improve measuring is by the construction of some new procedural environment, as having a large amount of instances of the same or similar tasks is necessary for measuring generalization. There is no standardized way to do this in reinforcement learning, which is probably why so many new environments have been constructed. This leads to a lot of the research being difficult to compare.

Some of the articles focuses on transformers. This had great success in NLP, and one article ("Stabilizing Transformers for Reinforcement Learning") managed to stabilize it for reinforcement learning. At the time of writing this, no-one has been able to reproduce this results. The environments they are using are manually constructed 3D environments, DMLab30. It would be interesting to see how transformers performs on different environments. A three dimensional environment is also likely quite hard to train on. A presumably simpler environment would be the environment used in "Quantifying Generalization in Reinforcement Learning" and "Leveraging Procedural Generation to Benchmark Reinforcement Learning". They are both built on the same principles of procedural generation, and are all two dimensional. They still require a quite large convolutional network, which does possibly make the task more difficult than necessary.

Most of the articles seems to focus most of it's attention on convolutional and fully connected network structures that don't require memory. This might be because it's simpler. It would be interesting to compare different memory-based structures, and see how they performs on these environments.

This also raises the question if measuring generalization for memory-based agents require different environments than for non-memory-based agents. It is not unthinkable that a comparison of memory-based agents in an environment where memory should not be necessary is yielding. It's also possible that memory-based agents perform better at generalization.

When training RL on procedurally generated environments, we need scalable methods, like Impala. This means dealing with off-policy data. Another question is therefore what off-policy correction does for generalization. Some of the articles has tried to answer what off-policy correction does for training performance, but not specifically for generalization. It is of course possible that these are very similar, but it would be interesting to have a closer look at that. Both off-policy correction and memory could also have different impacts in single- and multi-task environments.

# Chapter 4

# Methodology

In the previous section I went over some attempts to measure and some ways to improve generalization in reinforcement learning. In this section I will go over how I'm going to measure it and the architecture and experiments I will use to do so.

There are three things you need for RL. An environment, a trainable model, in this case a neural network representing a policy and value function, and a training algorithm. The environment is part of the problem, and the the neural net and training algorithm is part of the solution. Changes to the network and the training algorithm can improve the performance and generalization and changes to the environment can help measure different aspects of the network and algorithm such as generalization.

## 4.1 Measuring Generalization

In supervised learning you have a training set and a test set. Usually the goal is to achieve high performance on the test set. Having a high performance on the training set and a low performance on the test set means the agent is just remembering the different cases, which is not desirable. Having roughly the same performance on both means the agent has learned the underlying distribution and is able to perform well on new data. This is important if we want to train agents to solve tasks where the number of variables are to large to train on every permutation, which is most tasks.

In RL the data is generated by interaction with an environment. Unlike supervised learning there is no optimal labels on this data. The agent only knows the actions taken, the states those leads to, and the rewards gotten. The agent does not know what the optimal action in any state is. The goal is therefore different. In supervised learning the goal is to find the underlying distribution

that allows the agent to differentiate between the known optimal labels. In RL the goal is to find the optimal labels with the help of the rewards, and then find the underlying distribution to differentiate between the found optimal labels.

One naive approach to generate a training and test set would be to split the episodes gathered from the environment into a training and test sets. This might seem like it would work since the agent would traverse the environment multiple times, thus being able to generate a lot of data. However, there's only one episode we are interested in the agent learning for any given instance of an environment: the optimal one. The agent should not learn all the suboptimal solutions discovered during training and an optimal agent shouldn't be able solve most of the data in the training set after training.

With only one instance of an environment it's therefore not possible to create a training and test set. In order to construct a training and test set and measure generalization in RL we therefore need an environment with multiple instances, multiple environments, or both. This allows us to split the instances into training and test sets consisting of different instances. The optimal solutions that we are interested in learning should be different in each of these, and the agent needs to find the underlying distribution shared between these instances to be able to solve them.

### 4.1.1 Single-task Generalization

The purpose of single-task generalization is to achieve good performance on multiple instances of one environment, including instances not seen during training. The goal is to train an agent that is better at solving all instances of this environment. If an agent is able to solve instances not seen during training it means that it must have found some general knowledge about the environment, which is what we want.

### 4.1.2 Multi-task Generalization

A single-task environment contains multiple different instances of an environment while a multi-task environment consists of multiple single-task environments. The measuring of both single- and multi-task generalization is an open problem in reinforcement learning. Some approaches have been mentioned in section 3. Later papers have focused a lot on procedural generation of environments. Having a large number of environment-instances makes it infeasible for an agent to remember an action sequence to solve the task. Doing so anyway will result in poor performance on a separate test set.

In addition to finding underlying structures between instances of the same environment, multi-task generalization also has to find underlying structures between different environments. In theory this should make the agent able to

perform even better on all environment since it has more possible knowledge to base it's actions on. However, in practice the opposite is sometimes the case.

### 4.1.3 Creating an Environment Suitable for Measuring Generalization

With only one instance of an environment, remembering an action-sequence often leads to pretty good performance. If this one environment instance is deterministic too, remembering can often be an easy way to achieve an optimal solution. Making the environment non-deterministic removes some of the value in remembering action sequences, but it often still yields good results [Bellemare et al., 2013]. Purely making an environment instance non-deterministic does not remove the ability of an agent to achieve good results by just remembering. Remembering an action sequence does not lead to good generalization, and having an environment where remembering is encouraged in training would therefore make it difficult to measure generalization.

Even though each instance of the environments is deterministic, creating a training set of multiple instances makes the environment, consisting of all the individual instances, non-deterministic. The agent does not know what the initial condition it's starting in, and can't just remember an action sequence. However, if the amount of instances is small, the agent can still remember action sequences by identifying these initial conditions. This becomes less of a good strategy as the number of instances increases. Having a separate set of instances only used for testing makes it possible to see how well the agent is generalizing to the underlying distribution.

One early approach was the Atari learning suite [Bellemare et al., 2013] mentioned in section 2.4. One problem with this is the linear structure of the games. All games start in the exact same state, and if the agent performs the same action sequence twice, the results will be the same. This makes the task more of a memorization or exploration task and less of a learning task. Any generalization obtained in one of the single-task environments would also be difficult to measure, as there is only one instance of each environment. Combining the environments into a multi-task environment, thus allowing us to split the environments into a training and testing set, would allow us to see if an agent is able to find some general knowledge between the games. One problem here is that the games are very different, and it would probably be difficult for an agent to learn general knowledge from one game and apply it to a different game.

It has been shown that getting the exact same sequence of environment instances lead to worse generalization and, maybe a bit more surprising, worse training than getting them in a random order each time.

To be able to measure generalization in a reliable way you therefore need an environment that is capable of generating multiple instances of itself. A common way of accomplishing this is by making the environment procedural. Procedural

generation can be used to generate different initial conditions for an environment, and therefore makes it possible to create a training and testing set. This is an important step to be able to measure generalization. In such an environment the agent needs a better understanding of the underlying workings of the environment to be able to solve all instances of it efficiently. It can also by manually creating a lot of different instances, but this doesn't scale as well and is more time consuming.

In the paper [Cobbe et al., 2018] they introduce a procedurally generated environment and a couple ways of measuring generalization by using this environment. The first one is training on a fixed set of different environment-seeds and using the rest for testing. The second one is training and testing on a new random seed for every episode. This does not guarantee that the training and test set is completely different, but if the number of possible seeds is high, it becomes likely. Assuming testing is done on the same number of seeds makes some overlap almost unavoidable but most of the seeds should still be different as the number of possible seeds in the environment $2^{32}$ and the number of episodes used are in the thousands.

[Cobbe et al., 2019] builds on this to test multi-task generalization as well. It introduces 15 additional environments with the same observation- and action-space. This makes it possible to train the same agent on all the 16 environments without any advanced action processing. Running the same tests described above should give a measure of the generalization-abilities of the network and training algorithm. The paper concludes that 500 test seeds is a good number for testing generalization abilities for both single- and multi- task learning. The environment suite is publicly available and is called Procgen, and was explained further in section 3.4.

### 4.1.4 How to measure generalization

When measuring generalization in supervised learning you want to know the difference between performance on a training and a test set consisting of data from some unknown distribution. While both sets come from the same distribution, their individual distributions are different due to variance when sampling a finite amount of data. The performance on the test set is almost always going to be worse than on the training set, but with enough training data and if the agent has good generalization capabilities, the performance difference between the two should decrease. This difference is called the generalization gap, and is often used to say something about how good an agent is at generalizing.

Let's say you have two agents trained on the same data. One with a performance of 90% of max and a testing performance of 50%, and one with a training performance of 50% and a testing performance of 40%. Which is best at generalization? This is not obvious. Only looking at the testing performance says the first one is the best, while only looking at the performance gap says the second

one is best. The first agent has a very good training performance, but has only managed to transfer some of this knowledge, around 55% ($50/90 \sim 0.55$), to the testing set. The second agent has a worse training performance, but has managed to transfer most of this knowledge, 80%, to the testing set. When thinking about it like this it becomes more obvious that the second one has better generalized it's knowledge. It might have acquired more wrong knowledge, but it's better at generalizing whatever knowledge it has.

One exception is with very low or trivially achievable training performance. An agent that scores say 1% on both training and testing has not managed to learn anything and therefore can't have generalized anything. The same score on both training and testing sets could also indicates that the score was achieved due to randomness. A random policy would achieve the same score on any sufficiently large sets of data.

Intuition says that the only difference between testing generalization for memory-based agents should be the need for memory to be useful in the environment. If memory is not useful it could be the case that you're not going to see any difference. However, it could also be the case that you see a difference anyway. This is going to be further explored in the experiments.

## 4.2 Improving Generalization

I've went over some ways to improve generalization in section 3.2 and 3.3. There are two main ways to improve generalization. One is to change the training algorithm and the other is to change the neural network. Changes to the network include transforming the environment with noise, like adding different textures to the environment, and adding a noise layer, either between the input and the first layer, or between two layers. Different training algorithms might result in different levels of generalization. Both of these are related to each other and the environment, and might perform differently when the other parts are changed. My main focus is going to be on the LSTM and Transformer neural net structures and how these interact with the different parts.

Often, things that improve training performance also improve generalization. This is not a surprise as in complex environments it's very useful to find some underlying structures as opposed to memorizing everything. Generally the generalization gap decreases as the number of training instances of an environment increases. The relationship between training performance, testing performance and the number of environment-instances can vary a lot with different training algorithms, networks and environments.

## 4.3 RLlib

To accomplish exploring differences between LSTM and Transformers, I've decided to implement a flexible system capable of running multiple training algorithms, environments, and neural networks using RLlib [Liang et al., 2017]. RLlib is a modular library for reinforcement learning that allows for different environments and networks being used with already implemented training algorithms like Impala and PPO, explained in section 2.2.3 and 2.2.1.

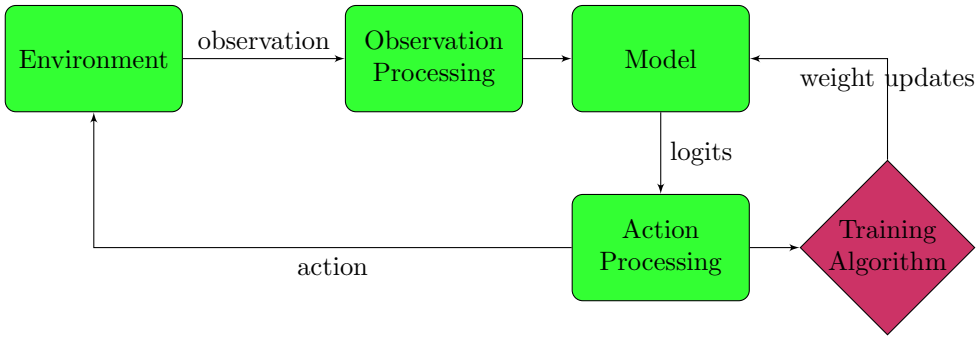



Figure 4.1: Structure of rllib, green is easily modifiable elements

In figure 4.1 you can see the structure of Rllib. The environment sends its output to an observation processor to make the environment output ready to be used by the network. The network in turn sends its output to an action processor which makes the network output ready to be used by the environment and the current training algorithm for weight updates.

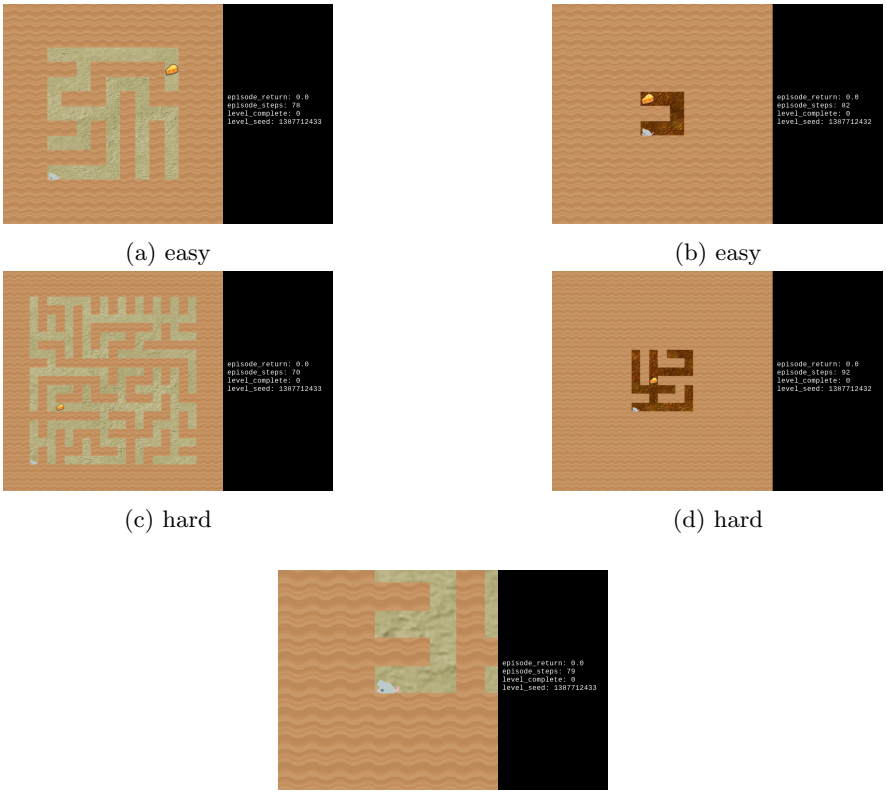
## 4.4 Environment

I've decided to use the Procgen environment, which I discussed in section 3.4. It consists of 16 different procedurally generated environments with the same observation- and action- space. Having the same observation- and action-space makes it easy to combine the 16 single-task environments, or a subset of them, into a multi-task environment, as I'm interested in generalization for both single- and multi-task learning. Each time-step of an environment consist of a 64 by 64 RGB picture. This gives a 3 dimensional input space of 64 by 64 by 3. There are 15 different actions that can be taken in all the environments, however, not all actions do anything in every environment. This is done to make it easy to combine them.

The environments also has the ability to use randomly generated textures. This should make it easier for an agent to generalize as the textures are irrelevant to

the task, and should force the agent to not focus on them. If an agent knows how to play chess, but becomes unable to play chess once the black tiles change color to green, it doesn't really know how to play chess. The color of the tiles does not matter, and an agent should not learn a policy that relies on irrelevant observations. This is a feature to improve generalization, and not measure it, but it's interesting to see how LSTM and transformers perform with and without them.

Another important aspect is that you can specify the number of environment-instances available for the agent to train on. This was looked at in the paper presenting it, and the agent performs differently with different amount of instances. In some environments agents start with high performance, gradually lowering the performance as number of instances decreases. Some are the other way around, and some goes down then up again as the number increases beyond a certain threshold.




(a) easy (b) easy

(c) hard (d) hard

(e) memory

Figure 4.2: The maze environment from the Procgen environment suite

In addition it offers different distribution modes. This changes the way instances are generated and presented to the agent. Only three of these are interesting to us. The two basic ones are easy and hard. The hard distribution is designed to be harder to solve than easy, with generally larger environments. The third one is memory, which is only available for 6 of the environments (Caveflyer, Dodgeball, Miner, Jumper, Maze and Heist). This distribution generates partially observable environments that are generally more difficult then hard. This makes it necessary for an agent to have some kind of memory of the previous actions in an episode in order to solve the environment-instance. The three distribution modes of the maze environment is seen in figure 4.2e with two different seeds for easy and hard. You can see that there is quite a bit of variation between individual instances of easy and hard. You can also see that the agent can only see part of the memory environment. In general the memory distribution is a lot harder than the hard distribution.

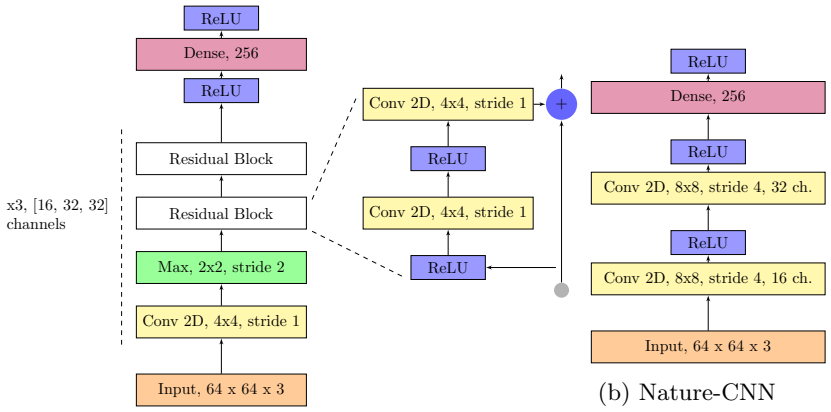
One downside with these environments are that they take a lot of resources to train on. They are vision based, meaning they need a convolutional network to be learned efficiently, which is more complicated than a dense network. They are not specifically designed for agents with memory, and only 6 of the enviornments have a memory distribution. In short, Procgen allows agents to train on both single- and multi-task environments with a lot of customizability in how to do so.

## 4.5 Neural Network

As mentioned earlier, a convolutional network is needed for good training performance. [Cobbe et al., 2018] and [Cobbe et al., 2019] both uses two different convolutional networks. The smaller one is referred to as nature-CNN, presented in [Mnih, 2015] and the larger one is referred to as Impala-CNN, presented in [Espeholt et al., 2018]. The reason for the usage for these networks is that they worked well in both the papers that presented them and in a lot of papers presented afterwards. They make it more straight forward to compare results, as a large part of the network is the same. Using two different convolutional networks can tell us something about how much information about an environment each layer needs to be able to generalize successfully. A structure that is able to perform with a smaller network might sometimes be desirable.

In figure 4.3 you can see the Impala-CNN and Nature-CNN. As you can see, Impala-CNN makes use of residual blocks and is significantly larger. This will probably results in both better training and testing performance, but will also probably require more resources to train.

On top of the CNN-network there will be either a fully connected layer, to have a baseline to compare to, a transformer-layer, or an LSTM-layer. This makes it easier to compare the different architectures with each other and with other




(a) Impala-CNN

(b) Nature-CNN

Figure 4.3: CNN networks

| **network** | dense | lstm | transformer |
|---|---|---|---|
| **#variables** | 750K | 1.5M | 2.1M |

Table 4.1: Number of trainable variables for different architectures

papers using the same CNN networks. After this top layer there will also be two parallel dense layers, one with 15 outputs, one for each action, and one with 1 output to be used by the value function needed by the training algorithms.

The LSTM and dense layer each have 256 cells. LSTM cells are however much more complicated than dense cells and contains more trainable variables (8 times more because $inputsize = outputsize$, $(LSTM : 4(mn+n^2))$). The transformer layer consists of 8 heads, each with a size of 32. These are concatenated and resulting in an output shape of $8 * 32 = 256$. In table 4.1 you can see the approximate amount of variables of each architectures when the output shape of the top layer is 256. They are not equal, and going for one similarity reduces another one. I've decided to make the output shapes of the different top layers the same. In comparison [Parisotto et al., 2019] uses 8 heads of size 64, which are then ran through a dense layer to make the output size 256.

To construct the networks I'm going to use Keras [Chollet et al., 2015]. LSTM and dense layers are already implemented there. For the transformer structure I'm going to be using the gated transformer architecture presented in [Parisotto et al., 2019], since that's the only transformer structure that has managed to outperform LSTM in RL. No code was released for the paper, so it is of course possible, and probably the case, that differences are present. This architecture uses the gating mechanism from Gated Recurrent Unit (GRU), which is similar to the gating architectures LSTM uses but a bit simpler. Unlike both GRU and LSTM, this gating mechanism is used as a residual connection and not a

recurrent connection.

## 4.6 Training Algorithm

Different training algorithms might give certain architectures an advantage. I'm therefore going to try multiple training algorithms. These are all already implemented in RLlib. However, configurations of the algorithms are still needed and is important for good results. The training algorithms I'm going to use is Impala, PPO and Asyncrhonous PPO (APPO). APPO uses a learning queue, like Impala, but uses the same loss functions as PPO.

All training algorithms has a loss function that they try to minimize. The training algorithms I'm going to use have a three part loss function. A policy loss, a value loss and a entropy loss. Policy and value loss are calculated based on the policy and value output from the neural network and the rewards gotten. The entropy loss is there to stop the agent from getting stuck at a local minimum. The three part loss is then added together and propagated through the network to update the weights.

Both APPO and Impala have the ability to take advantage of a replay buffer. The replay buffer is just a storage for generated data. With a replay buffer the agent usually becomes more sample efficient as it uses the same data more than once for training.

Any explicit exploration policy other than entropy is not used, which is standard practice in gradient methods. Some environments require more advanced exploration algorithms, but Procgen should work fine without. In all of the training algorithms, transformers are trained as a recurrent network with sequences of observations being sent to them.

## 4.7 Experiment Plan

There are multiple parameters I want to test the effect of for LSTM and transformer networks. To have a baseline I will also include a dense network. To be able to compare them I will use the convolutional networks explained previously with these different top layers.

**Different networks:** The first variable is this top layer. As mentioned earlier, there will be three different top layers, dense, LSTM and transformer. I will also test the effect Nature-CNN and Impala-CNN has when used in conjunction with these top layers.

**Single- and multi-task:** I want to test the performance of the networks on both single and multi task environments. For the singletask environment I will use Coinrun, since a lot of data is available on this environment from two papers

[Cobbe et al., 2018], [Cobbe et al., 2019]. As the multitask environment I will be using a combination of Caveflyer, Dodgeball, Miner, Jumper, Maze and Heist, since these are the six multitask environments in Procgen that has a memory distribution. Multitask environment are generally harder than singletask ones and the performance are expected to be worse on these environments.

**Number of environment instances:** The number of instances in the training set might affect performance. If a network is better at memorizing you would expect it to perform better with a lower number of instances. If it's better at generalizing the opposite might be true. I plan to start with a low number of instances to make make sure everything's stable, and then increase it.

**Training algorithms:** Different training algorithms might give different networks an advantage. I will use PPO, APPO and Impala, as these are all able to train using the exact same network.

**Environment distribution:** The different environment distributions (easy, hard and memory) available in the Procgen environment suite should show differences in performance based on the quality of memory in a network. Networks with good memory should perform better at the memory distribution, but might perform worse on the other distributions. The memory distribution is partially observable and needs memory, while the easy distribution shouldn't need memory, tough it sill might benefit from it. Any difference in performance between these two distributions should show us how good the different memory architectures are. I will focus on memory and easy, as hard is, at least in theory, just a harder version of easy.

**Replay buffer:** A buffer should in theory stabilize the training process. However, it also makes the data more off-policy as it's used for more training iterations after it's generated. Checking how networks or their generalization performance are affected by this will help to answer the research question about how they are affected by off-policy data and correction.

**Off-policy correction (vtrace):** To answer the question about how off-policy correction affects the different networks, I'm going to test the effect of vtrace has on training and test performance.

**Procedurally generated textures:** Procedurally generated textures should in theory help agents to generalize. A network better at generalization might see a larger increase in performance if these textures are added, giving information about the generalization capabilities of the different networks.

The plan is test as many permutations of these variables as possible, given the time and resources available, with a focus on the difference between LSTM and Transformers on any given setting.

| Parameter | Value |
|---|---|
| $\gamma$ (reward discount factor per time-step) | 0.999 |
| learning rate | $5 \cdot 10^5$ |
| workers | 6 (1 base worker + 5) |
| environment per worker | 12 |
| training batch size | $50 \cdot 12 \cdot 8 = 4800$ |
| minibatches per epoch | 8 |
| total time steps | easy: 25M, memory: 200M |
| epochs per rollout (called number_sgd_iter in RLlib) | 3 |
| entropy coefficient | 0.01 |
| value coefficient | 0.5 |
| **parameters only if buffer** | |
| replay proportion | 0.8 |
| training batches in replay buffer | 120 |
| **parameters only in PPO and APPO** | |
| lambda | 0.95 |
| kl coefficient | 0.5 |
| clip parameter | 0.1 |

Table 4.2: Configurations for training algorithms

## 4.8 Experiment setup

In table 4.2 you can see the parameters used for the experiments. I have chosen to use the settings from the paper [Cobbe et al., 2018], for better comparisons and since these setting worked for them, though some modifications had to be made to be able to run them on the available cluster.

The training batch size is set to one tenth of the value because of memory limitations on the GPU used, and the learning rate is also set to one tenth to compensate for this. This change could lead to the training being less stable due to the higher variance caused by the smaller batch size.

The worker and environment per worker count also had to be changed to use less RAM. The number of workers is the number of parallel processes generating training data. This is set to 6, each with 12 parallel environments. RLlib has one base worker, so the worker parameter in RLlib is set to 5. These changes should only affect the wall clock time.

The total time steps is the number of time steps generated for training. This is smaller for the easy distribution since it's easier to learn. The number of time steps for the memory distribution is the same as used for the hard distribution in [Cobbe et al., 2018]. This might be too little data for good performance, but we should see some performance increase during the first 200 million time steps.

The replay proportion is the percentage of data from the replay buffer used for each training batch, with the rest coming from the learning queue.

The experiments was performed on the NTNU IDUN computing cluster [Själander et al., 2019]. Each run used 100 GB of memory, 6 Intel Xeon cores (one for each worker), and a NVIDIA Tesla V100 GPU. The code for the experiments can be found at: $https://github.com/nummer1/Masters$ [Grande, 2019a].

# Chapter 5

# Results And Analysis

All graphs in this chapter have the parameters in the description text, and any parameters shared between runs within the same figure have their shared parameters in the shared description. All graph's x-axis is the number of time-steps and the y-axis is the mean episodic reward.

For all graphs labeled with singletask, the Coinrun environment was used. This has a maximum reward of 10 and a minimum of 0. According to [Cobbe et al., 2019], it has a trivial reward of 5. This trivial reward was calculated by training an agent by giving it only the reward achieved, and the action taken.

For all graphs labeled multitask, 6 singletask environments from the Procgen suite are used: Caveflyer, Dodgeball, Miner, Jumper, Maze and Heist. These environments also have a minimum of 0 and a maximum of respectively 12, 19, 13, 10, 10 and 10. This causes the multitask environment to have a maximum reward of 12.3, the average of these 6 tasks. The trivial reward here is around 2 for all environments. For this multitask environment, the number of training instances is given for each individual task. That means that a training set of the multitask environment consisting of 10 instances, consists of $10 * 6 = 60$ instances in total.

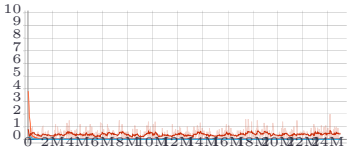
When trained on an unbound set of instances, a new environment instance is created for each episode. The actual number of instances created are dependent on the length of the average episode.

All lines in the graphs are also smoothed to be easier to read. The more transparent lines are the unsmoothed data, and gives an indication of how stable the performance is during training and testing.

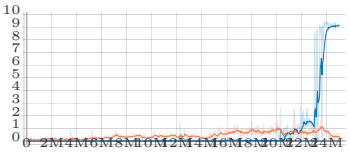
## 5.1 Experiment Results

The limiting factor for the number of experiments is the capacity of the Idun cluster. One run with the easy environment distribution takes around 6 hours to complete, and with memory distribution it takes around 40 hours. This is a shared resource, and only so many experiments can be run in a limited time frame.

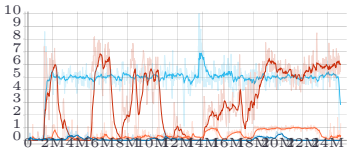
### 5.1.1 Training algorithms



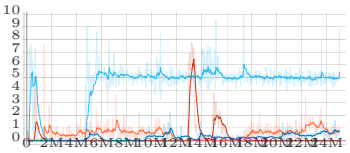
(a) PPO, dense, lstm and transformer, training result



(b) APPO, dense, training result



(c) APPO, lstm, training result



(d) APPO, transformer, training result

Figure 5.1: easy, singletask, no buffer, no vtrace.

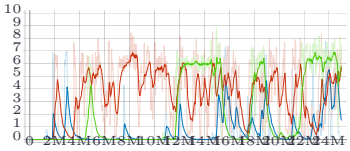
The experiment shown in figure 5.1 is designed to highlight differences between PPO, APPO and Impala. 4 runs was done for each network with each training algorithm. One training on multitask with 2 (dark blue) instances, one training on multitask with 10 instances (orange), one training on singletask with 2 instances (light blue) and one training on singletask with 10 instances (red). It is ran without a buffer and without vtrace as the PPO algorithm implemented in RLlib supports neither of these options.

Figure 5.1a shows all 12 runs with the PPO training algorithm. As you can see it achieves minimal training performance for all of them. This might be because of poor hyperparameters. It uses mostly the same as in [Cobbe et al., 2018] with a smaller training batch size and smaller learning rate to compensate for that. This is done due to memory limitations on the GPU used for training.

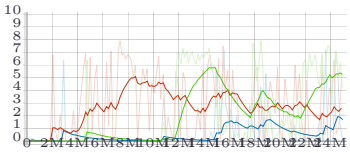
APPO performs seemingly better, although none of the runs reaches a performance significantly above 5, which should be trivial. The exception being the dense one trained on multitask with 2 instances, seen in figure 5.1b, which

spikes around 24 million time steps. This shows that it is possible for APPO to reach higher performance. An increase in performance is not seen in the testing set, so this training performance is entirely achieved though memorization of the training instances. It's possible better tuned hyperparameters would make APPO achieve higher performance more reliable.

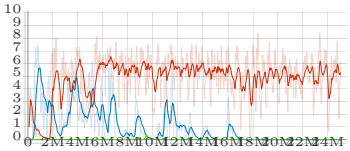
### 5.1.2 Off-policy correction



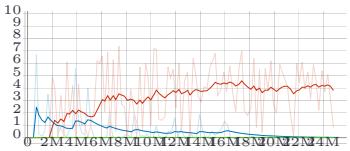
(a) no buffer, vtrace, training result



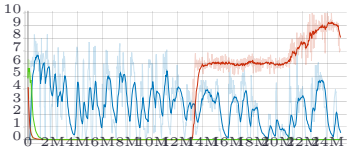
(b) no buffer, vtrace, test result



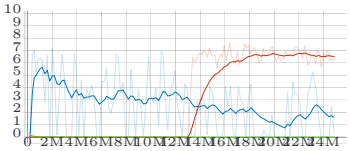
(c) buffer, vtrace, training result



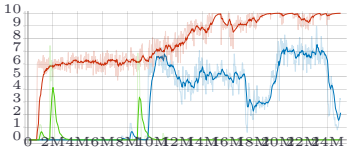
(d) buffer, vtrace, test result



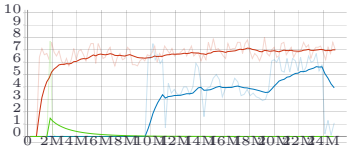
(e) no buffer, no vtrace, training result



(f) no buffer, no vtrace, test result



(g) buffer, no vtrace, training result



(h) buffer, no vtrace, test result

Figure 5.2: Impala, easy, singletask, 10 instances. Red lines are lstm, blue are transformers, and green are dense

The experiment shown in figure 5.2 is designed to test the effect of off-policy correction. One run for each of the networks was run with a buffer and vtrace turned on and off. Coinrun with 10 instances is used as the training set, as previous experiments have shown that the multitask environment is likely more

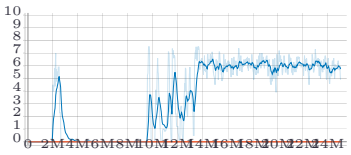
difficult. Since vtrace is used to correct off-policy data, and a buffer causes the training data to become more off-policy, it seemed logical to test these parameters together.

Without buffer and with vtrace, seen in figure 5.2a, all networks seem unstable, as seen by the constant increase and decrease in performance. With the addition of a buffer, seen in figure 5.2c, all networks still seem to be unstable, although the LSTM network seems to be performing better.

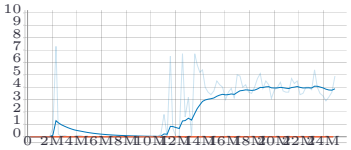
Without vtrace, as seen in figure 5.2g and 5.2e, the LSTM and transformer network seems to be more stable, but only the LSTM network reaches close to peak performance on the training set. It seems that no vtrace causes more stability than vtrace, and that a buffer causes more stability than no buffer. It's therefore not surprise that buffer without vtrace seems to be the most stable configuration. This is the same result [Schmitt et al., 2019] found.

### 5.1.3 Differences between multi- and single-task

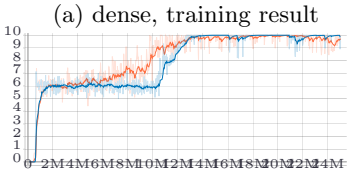
The experiments shown in figure 5.3 and 5.4 are designed to show differences between single- and multi-task generalization. A buffer without vtrace was used as that was the seemingly more stable configuration.



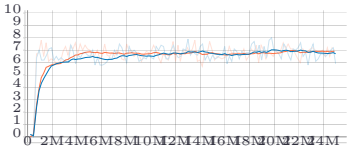
(a) dense, training result



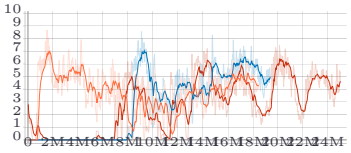
(b) dense, test result



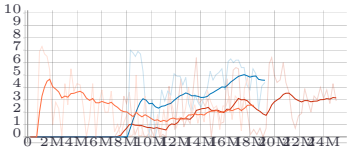
(c) lstm, training result



(d) lstm, test result



(e) transformer, training result

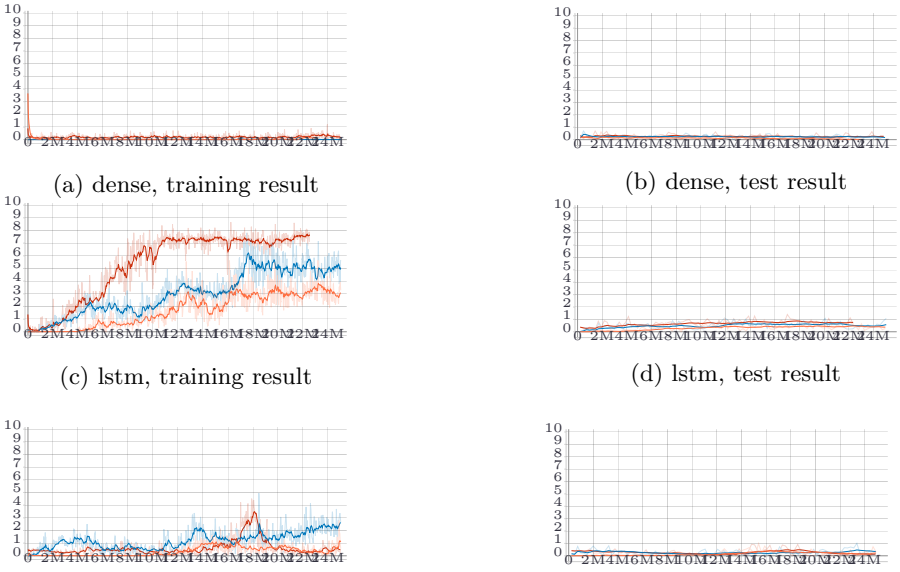


(f) transformer, test result

Figure 5.3: Impala, easy, singletask, 10 instances, buffer, no vtrace

Figure 5.3 shows three runs for each LSTM, transformers and dense networks on Coinrun with 10 instances. LSTM seems to perform better than the other two, although one of the three runs crashed. Two of the dense runs achieved a score of zero, and the transformer runs achieves trivial performance. Both dense and transformers seems to be less stable than LSTM, which is what was seen earlier too.

Both the LSTM runs achieves achieves slightly better than trivial performance at around 1 million timesteps. This performance increase is also seen in the testing set, indicating that the increase was largely due to generalized knowledge about the environment. At around 12 million timesteps they both find an almost optimal policy for the training instances. This improvement is not seen in the test performance, indicating that this improvement is due to memorization of the training instances, since a deeper understanding would also increase the test performance.



(a) dense, training result

(b) dense, test result

(c) lstm, training result

(d) lstm, test result

(e) transformer, training result

(f) transformer, test result

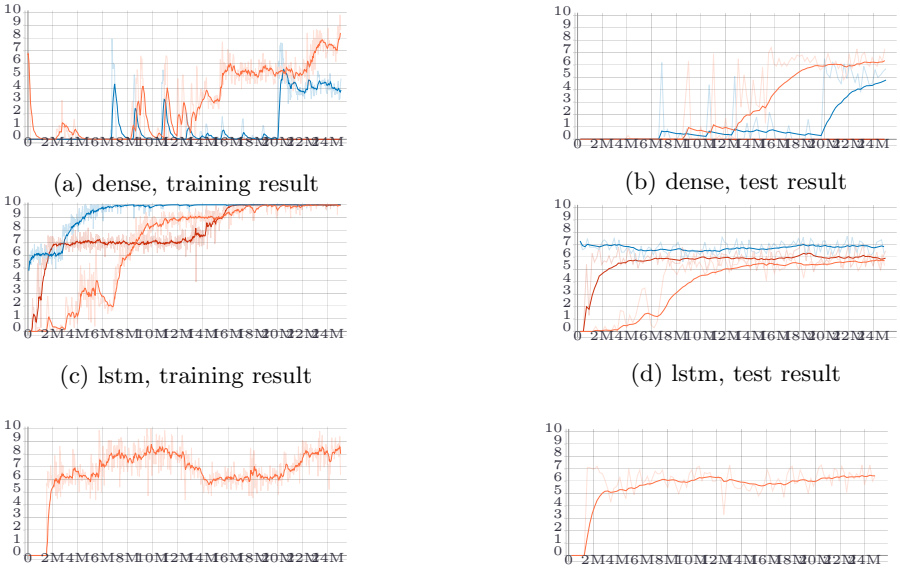
Figure 5.4: Impala, easy, multitask, 10 instances, buffer, no vtrace

Figure 5.4 shows performance on the multitask environment with 10 instances. LSTM is the only network to achieve a decent performance. However, it only does so by memorizing the training instances as no increase is seen in the test performance. The transformer network also seems to be memorizing some of the training instances as no increase is seen in test performance here either, but does so worse than LSTM.

Overall it seems that the multitask environment is more difficult than the singletask environment. This is not a surprise as multitask environment usually are more difficult. There is also no generalization happening in the multitask environment indicating that there are either too few instances to generalize from, or that the environments are too different from each other for the agents to find shared patterns with the current settings.

### 5.1.4 Training set size and procedural generated textures

The experiments shown in figure 5.5 and 5.6 are designed to show the effect of an increased training set size and of procedurally generated textures. As Coinrun seemed easier than the multitask environment, the addition of procedurally generated textures, and a larger training set of 50 instances is tested in this environment.



(a) dense, training result

(b) dense, test result

(c) lstm, training result

(d) lstm, test result

(e) transformer, training result

(f) transformer, test result

Figure 5.5: Impala, easy, singletask, 10 instances, buffer, no vtrace, procedurally generated textures

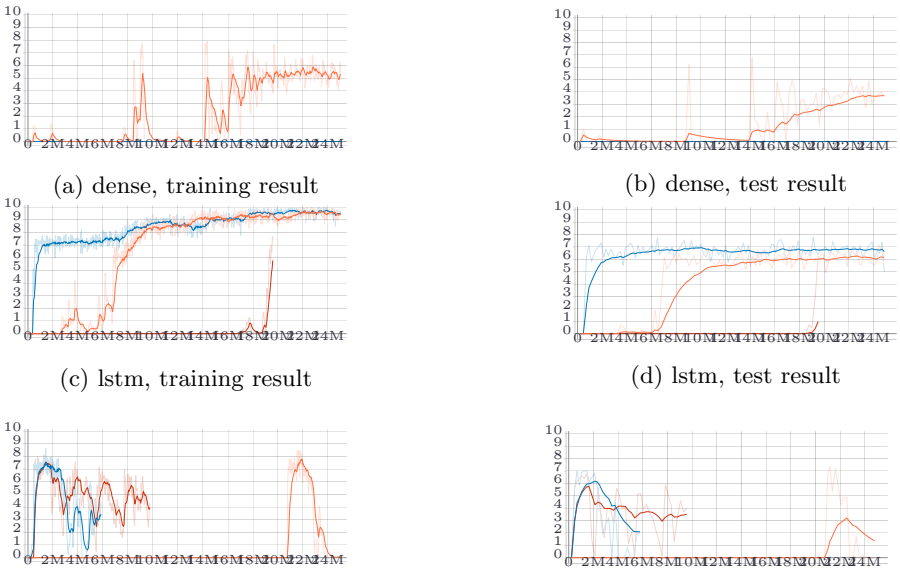
Figure 5.5 shows runs from LSTM, transformers and dense networks with the same configurations as in figure 5.4, but with the addition of procedurally generated textures. The procedural textures seems to stabilize the networks even more. Two of the transformer runs crashed, but the third seems to be stable and achieves a better training and test performance than without procedural

textures, seen in figure 5.3e and 5.3f.

We see the same increase in LSTM around 12 million timesteps as without procedurally generated textures, further supporting the idea that the network remembers the instances after a while. The addition of procedural textures does not seem to limit the networks ability to remember when only 10 training instances are used.

The training performance of the transformer run is worse than LSTM, but the test performance seems to be very similar. This tells us that transformers do have the ability to find some generalized knowledge about the environment. In this case it seems that the LSTM network is just better at memorizing the training instances, which is not something you want.

You do see that the training performance of the transformer varies more than the test performance. This might indicate that it tries to memorize some instances but forgets them again. Dense still seems to be performing the worst.



(a) dense, training result

(b) dense, test result

(c) lstm, training result

(d) lstm, test result

(e) transformer, training result

(f) transformer, test result

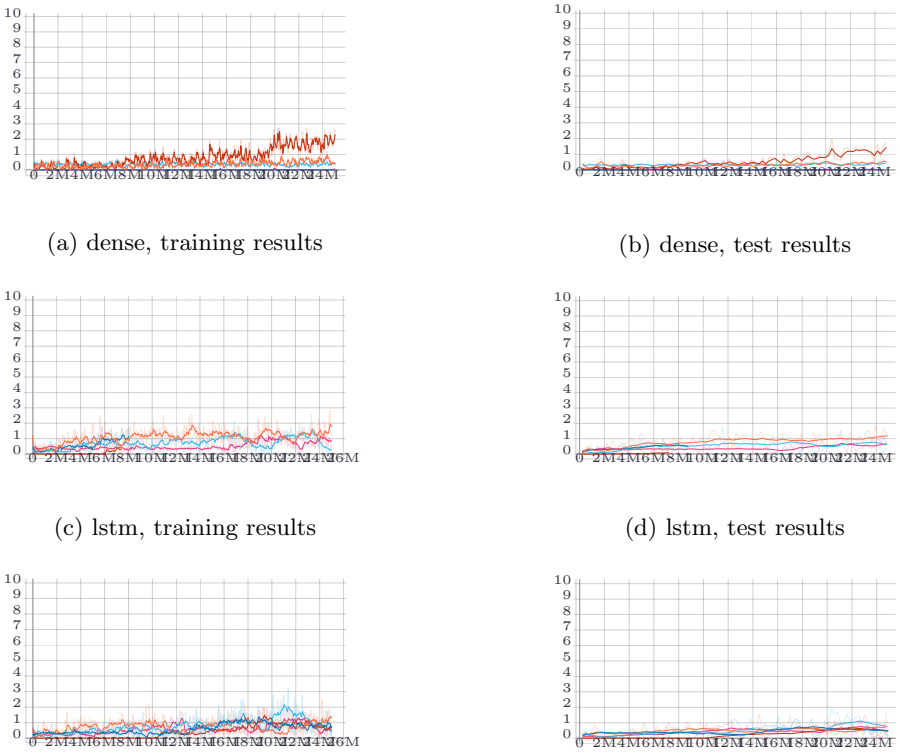
Figure 5.6: Impala, easy, singletask, 50 instances, buffer, no vtrace

Figure 5.6 shows performance on Coinrun with 50 instances. Only LSTM seems to perform here, and two of the transformer runs crashes early, though they do manage to learn something before doing so. The transformer do not seem to be able to remember what it has learned though, as both the training and test performance decreases after the first increase in all three cases.

The test performance for LSTM is mostly flat while the training performance is increasing, hinting at the same pattern of finding some generalized knowledge early, and then memorizing the training instances. The increase in the training performance after the first jump is not as sudden here as seen earlier, likely because it takes more time to remember 50 instances than 10.

### 5.1.5 Increasing the training set size

The experiments in figure 5.7, 5.8, 5.9, and 5.10 were designed to show how generalization of transformers and LSTM is affected by a larger training set, as seen in both [Cobbe et al., 2018] and [Cobbe et al., 2019]. A buffer and procedurally generated textures are used as that seemed to improve stability.



(a) dense, training results

(b) dense, test results

(c) lstm, training results

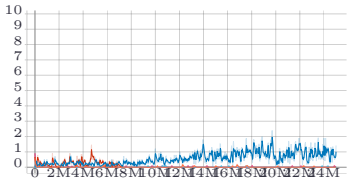
(d) lstm, test results

(e) transformer, training results

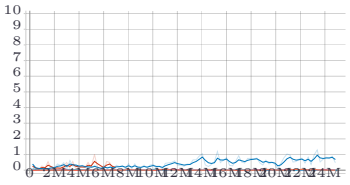
(f) transformer, test results

Figure 5.7: Impala, multitask, 500 instances, procedural textures, buffer, no vtrace

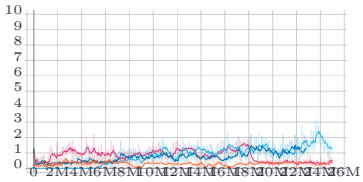
Figure 5.8 and 5.7 shows performance on 5 runs with each of the networks on the multitask environment with respectively and unbound set of training instances



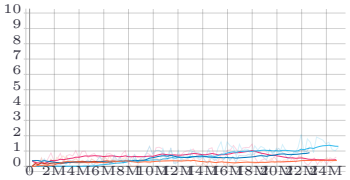
(a) dense, training results



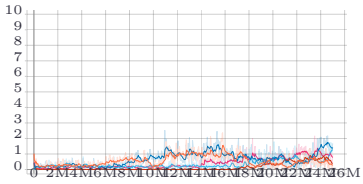
(b) dense, test results



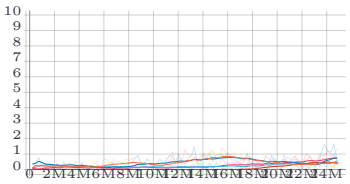
(c) lstm, training results



(d) lstm, test results



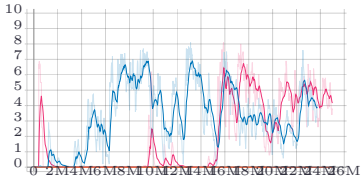
(e) transformer, training results



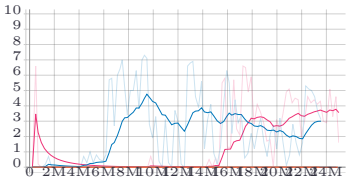
(f) transformer, test results

Figure 5.8: Impala, multitask, unbound instances, procedural textures, buffer, no vtrace

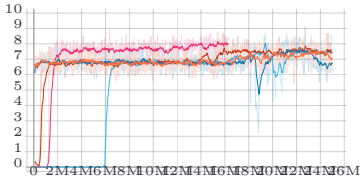
and 500 training instances. The lack of both training and test performance indicates that no instances where memorized, and no generalized knowledge was found. Only LSTM managed to achieve trianing performance on multitask with 10 instances, and it did so by memorization. It does not seem like this larger training set fixes any of the problems seen earlier.



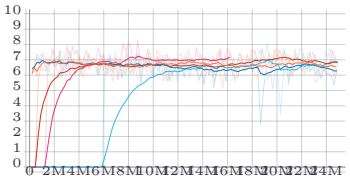
(a) dense, train results



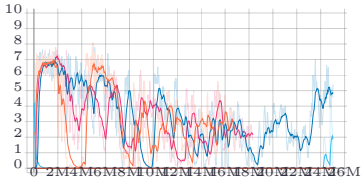
(b) dense, test results



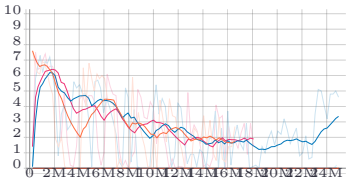
(c) lstm, training results



(d) lstm, test results



(e) transformer, training results

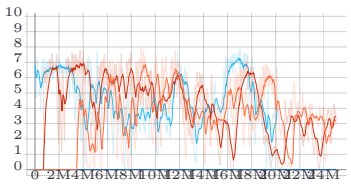


(f) transformer, test results

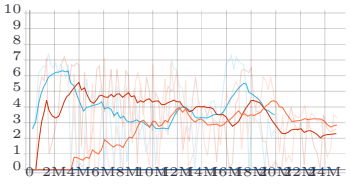
Figure 5.9: Impala, singletask, 500 instances, generate textures, buffer, no vtrace

Figure 5.9 shows performance of 5 runs with each of the networks on the Coinrun environment with 500 training instances and procedural generation. LSTM seems to be the only network performing well here. The transformer network seems to perform similarly to when trained on a training set size of 10 and 50 instances without procedural textures. The same goes for the dense network. However, transformers seems to be performing worse than with 10 instances and procedural textures. It seems that that procedural generation helps transformers with fewer instances, but when the number of training instances becomes to large the transformer again fails. This is not the case for LSTM. The training and

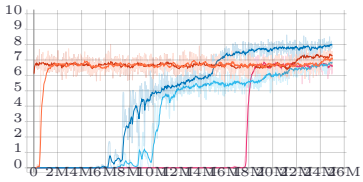
test performance for LSTM, seems to be roughly the same, indicating that no memorization is done here.



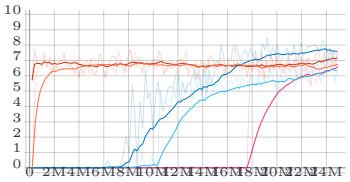
(a) dense, training results



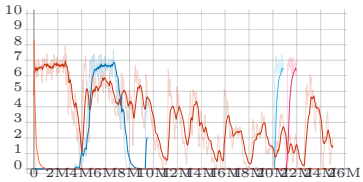
(b) dense, test results



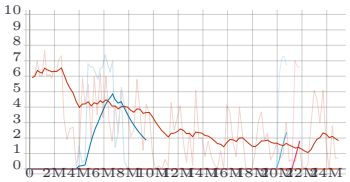
(c) lstm, training results



(d) lstm, test results



(e) transformer, training results



(f) transformer, test results

Figure 5.10: Impala, singletask, unbound instances, generate textures, buffer, no vtrace

Figure 5.10 shows performance of 5 runs with each of the networks on the Coinrun environment with an unbound set of training instances and procedural generation. The results are very similar to when trained on 500 instances. This indicates that 500 instances, like unbound, are too many to remember, which is supported by the multitask results with 500 and unbound instances too.

### 5.1.6 Nature-CNN and memory distribution

When the Impala-CNN was replaced with the Nature-CNN, it resulted in poor training and test performance for all settings. Since [Cobbe et al., 2018] did achieve some performance with Nature-CNN, the reason for this might be that

this simpler network requires better hyperparameter tuning to achieve anything, and that the changes done to accommodate for less GPU memory was enough to upset the balance.

No performance was achieved on the memory distribution with either single- or multi-task with 2, 10, 500 or an unbound set of instances. Buffer, vtrace and procedurally generated textures had no effect on performance either. All experiments on the memory distribution was ran with 200 million time steps. This was the same amount of time steps as [Cobbe et al., 2019] used for the hard distribution, and the memory distribution might require more time steps to train properly. However, you would expect some performance increase in the first 200 million time steps even if more was needed for good performance. The lack of any improvement leads us to believe that something else is needed to train successfully on this distribution.

## 5.2 Experiment Analysis

From the results we can see that better training performance usually results in better test performance. In some cases with a small number of environment instances in the training set, this is not the case. This is what we would expect, as a smaller training set makes the agent more likely to overfit. The poor performance on the test set is in these cases likely due overfitting and memorization of the training instances. When the number of training instances increases we see that networks manages to generalize some of the knowledge which we see with the increase in the test performance. This is also expected.

Looking at the graphs, test performance is usually less stable than training performance. This might be because all test instances are usually different from each other since they are sampled from all $2^{32}$ possible instances not used as training instances. The test set therefore has more variance than the training set.

Often we see a sudden jump in training performance after being stuck at 0 for a while. This might be because no exploration policy other than entropy was used. This makes the agent explore more or less randomly until it suddenly hits a promising path, resulting in a spike in performance. Sometimes there is also a sudden drop in performance. When this happens the agent obviously forgets what it has previously learned. It might be caused by a promising path that turns out to not actually be good.

Overall, LSTM seems to be the superior network. Transformers seems to be a bit better than dense, but overall worse than LSTM. LSTM also seems to be more stable than both transformers and dense networks, especially as the number of training instances increases. It's worth noting that I use 1 LSTM layer and 1 Transformer layer. [Parisotto et al., 2019] uses 3 LSTM layers and 12 transformer layers.

Additionally, procedurally generated textures seems to stabilize performance for all networks. The technique is meant to improve generalization, and it seems to do so as it results in both improved training and test performance.

### 5.2.1 Analysis of singletask, easy distribution

Agents trained on Coinrun with 10 and 50 instances seems to be learning some generalized knowledge before they remember the training instances which results in an increased training performance without an increase in test performance. This is similar to overfitting, when training on the same data too much results in increased training performance and decreased test performance. However, the overfitting in Coinrun does not seem to result in a decrease in test performance, only a increase in training performance. The agent does not forget the generalized knowledge acquired early in the run, otherwise we would see a decrease in test performance.

When trained on Coinrun with 500 and unbound instances, LSTM shows no sign of the increase in training performance without increase in test performance like it showed with both 10 and 50 instances. This indicates that 500 instances is too many to memorize. This is also supported by the fact that with both 500 and unbound instances LSTM has roughly the same performance in the training and test set. Everything, or at least almost everything, learned about the environment is generalized information in these cases.

Transformers often seem to have similar training and test performance on the Coinrun environment. This might mean it finds some generalized knowledge without memorizing training instances. It might also mean that it doesn't use the observations properly, and only learns with the help of rewards and actions. An agent trained with only access to the actions taken and the reward gotten, would achieve the same performance on both training and test set if the training set was sufficiently large enough.

It seems that memory is very useful in the Coinrun with easy distribution, as the LSTM network performs better than the dense network, and transforms at least as good as it. This is a bit surprising as memory should not be needed. It might be because the dense network is too small. It's possible it would be fairer to use a larger dense layer as both transformers and LSTM have more parameters per cell, as seen earlier in table 4.1.

### 5.2.2 Analysis of multitask, easy distribution

When training on 10 instances from the multitask environment, only LSTM achieved any notable training performance, and did so by memorization since it did not achieve any test performance. This tells us that 10 instances is not enough to find any generalized information from this environment with the

settings used.

When training on a set of 500 multitask instances, all networks performed poorly. As we saw with 10 instances, only memorization was used to learn. The reason for failing to learn anything with 500 instances might be because 500 is too many instances to memorize. We also see roughly the same performance with an unbound set of training instances.

The lack of test performance on the multitask environment with 10, 500 and an unbound set of environment instances tells us that it is too difficult to find a generalized solution to this multitask environment with the current settings. [Cobbe et al., 2019] managed to train agents on a multitask environment with 500 instances. However, this environment consisted of all the 16 Procgen single-task environments. It might be the case that the removal of 10 of the environments causes the agent to have a harder time finding shared patterns between the environments.

In short, only LSTM managed to achieve any performance in this multitask environment, and it only did so through memorization which is undesirable.

### 5.2.3 Analysis of memory distribution

All the experiments with the memory distribution achieved poor performance. In a lot of runs with the easy distribution we see that the performance is close to 0 before it spikes. This is as mentioned probably caused by exploration. The reason that the memory distribution achieves poor performance might simply be that no path is found in the first place. This makes sense as this distribution generally makes much larger instances that are more difficult to explore. With the size of the instances it's possible that the memory distribution is more of an exploration challenge than a memory challenge. Without any specific exploration policy other than entropy, which is more or less random exploration, this challenge might just be too difficult.

### 5.2.4 Analysis of off-policy correction

All networks seems to be less stable with vtrace. This instability caused by vtrace is likely due to an increase in variance caused by the algorithm. Previous work [Schmitt et al., 2019] has shown that this might be the case in some environments, and it seems to be the case in the Coinrun environment.

Even with the addition of a buffer, vtrace still seems unstable. A buffer should in theory make learning more stable as it reduces variance. It also makes the data more off-policy, and since vtrace is designed to correct off-policy data, it might sound like vtrace should perform here. This is not the case. All vtrace seems to do is just making the training process less stable.

It does make some intuitive sense that importance sampling performs worse than just using a buffer. As mentioned earlier, in section 2.1.5, importance sampling weigh data more likely to happen in the current policy and unlikely to happen in the behavioral policy higher than data unlikely to happen in the current policy and likely to happen in the behavioral policy. This inevitable increases the variance as more weight are placed on a smaller percentage of the data than without importance sampling. Truncating it reduces some of the variance, but not all. We see this in the results where the graphs have a lot of spikes when vtrace is used.

Of course, it might always be poor hyperparameters for vtrace that is causing the instability. The two clipping parameters of vtrace was both 1, which is what the Impala paper [Espeholt et al., 2018] used and found was best. As the variance is higher, the learning rate might also need to be decreased. This would however limit some of the usefulness as a lower learning rate would require more data and thus become less sample efficient.

# Chapter 6

# Conclusion

The research questions introduced in the introduction were:

**Research Question 1:** How much better can transformers generalize than LSTMs in single- and multi-task reinforcement learning?

**Research Question 2:** How does off-policy correction affect the generalization ability of LSTMs and transformers?

All the experiments in this thesis where ran on procedural environments, and there might of course be differences that this kind of environment does not show.

From the experiments it seems that transformers do not perform better than LSTM on the training data. It might perform better than a dense network in some situations, but in a lot of situations it's just as unstable and unreliable. It does however perform similar to LSTM on the test data in some situations. It also seems to be unable to memorize training data to the same extent LSTM is. This is a good property, as we don't want to remember the training data.

However, this inability to memorize does not result in an improved test performance over LSTM. When the training set size increases, the performance of the transformer also seems to decrease relative to LSTM. When faced with a large training set, LSTM loses the ability to memorize the training set, but retains the ability to generalize, while transformer loses the ability to generalize. Transformers also becomes more unstable as the training set increases, unlike LSTM.

The generalization ability of the transformer can therefore not be said to be consistently better than that of LSTM, but it can be said that it is sometimes equal, and even better when the size of the training set is small, as it does not memorize the training data. As generalization is the important part of any RL

algorithm that wants to solve more complex problems, this is promising for the transformer.

Unlike the results presented in [Parisotto et al., 2019], transformers are not outperforming LSTM in any environment tested. The biggest difference between the transformers used is the number of layers. 12 layers are used in the paper, while one is used here. Only 3 layers of LSTM are used in the paper, while one is used here. It looks like transformers might be a more promising architecture as the networks increase in size, and that LSTM still is the superior memory architecture with smaller networks.

Transformer do seem at least as good as dense networks, which means they are doing something right. It seems that the rigid, temporal structure of LSTM is better than the more context based structure of transformer on the tested environments. The more context based structure of transformers might do better in even more complex problems than the one tested in the thesis, as simple problems are often best solved using simple solutions.

The off-policy correction algorithm; vtrace, seems to make both LSTM and transformers unstable. This is not very surprising as previous papers [Schmitt et al., 2019] have shown that vtrace sometimes makes training less stable. The introduction of a buffer also seems to have the same effect on both LSTM and transformers, making them more stable. So does the introduction of procedural generated textures. The training to test performance ratio seems unaffected by these changes, which means an equal amount of the information learned is transferred from the training data to the test data, e.g. is generalized. In short, techniques that should make RL more stable, seems to have the same effect on both LSTM and transformers.

The failing of all the networks to achieve any performance when presented with a complicated environment, shows that it's not straight forward to design an environment for measuring performance. It would be a significant advantage if the environment where easy to train on, as it would require less resources to experiment on, and is something that should be investigated.

LSTMs are still superior to transformers, but transformers show a lot of properties that could make them better than LSTMs given the right circumstance.

# Bibliography

[Ba et al., 2016] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

[Bai et al., 2018] Bai, S., Kolter, J. Z., and Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

[Beattie et al., 2016] Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., et al. (2016). Deepmind lab. *arXiv preprint arXiv:1612.03801*.

[Bellemare et al., 2013] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

[Chollet et al., 2015] Chollet, F. et al. (2015). Keras. `https://keras.io`.

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

[Clemente et al., 2017] Clemente, A. V., Martínez, H. N. C., and Chandra, A. (2017). Efficient parallel methods for deep reinforcement learning. *CoRR*, abs/1705.04862.

[Cobbe et al., 2019] Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. (2019). Leveraging procedural generation to benchmark reinforcement learning.

[Cobbe et al., 2018] Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2018). Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*.

[Dai et al., 2019] Dai, Z., Yang, Z., Yang, Y., Cohen, W. W., Carbonell, J., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Espeholt et al., 2018] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*.

[Fan et al., 2018] Fan, L., Zhu, Y., Zhu, J., Liu, Z., Zeng, O., Gupta, A., Creus-Costa, J., Savarese, S., and Fei-Fei, L. (2018). Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In *Conference on Robot Learning*.

[Grande, 2019a] Grande, J. A. (2019a). Masters. `https://github.com/nummer1/Masters`.

[Grande, 2019b] Grande, J. A. (2019b). Midterm report. Project report in TDT4501, Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology.

[Greff et al., 2016] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Horgan et al., 2018] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*.

[Justesen et al., 2018] Justesen, N., Torrado, R. R., Bontrager, P., Khalifa, A., Togelius, J., and Risi, S. (2018). Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*.

[Lee et al., 2019] Lee, K., Lee, K., Shin, J., and Lee, H. (2019). A simple randomization technique for generalization in deep reinforcement learning. *arXiv preprint arXiv:1910.05396*.

[Liang et al., 2017] Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I. (2017). Rllib: Abstractions for distributed reinforcement learning.

[Liu et al., 2019] Liu, S., Johns, E., and Davison, A. J. (2019). End-to-end multi-task learning with attention. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1871–1880.

[Mnih, 2015] Mnih, V. (2015). Human level control through deep reinforcement learning.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

[Packer et al., 2018] Packer, C., Gao, K., Kos, J., Krähenbühl, P., Koltun, V., and Song, D. (2018). Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*.

[Parisotto et al., 2019] Parisotto, E., Song, H. F., Rae, J. W., Pascanu, R., Gulcehre, C., Jayakumar, S. M., Jaderberg, M., Kaufman, R. L., Clark, A., Noury, S., et al. (2019). Stabilizing transformers for reinforcement learning. *arXiv preprint arXiv:1910.06764*.

[Pathak et al., 2017] Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17.

[Schmitt et al., 2019] Schmitt, S., Hessel, M., and Simonyan, K. (2019). Off-policy actor-critic with shared experience replay.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[Shaw et al., 2018] Shaw, P., Uszkoreit, J., and Vaswani, A. (2018). Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*.

[Själander et al., 2019] Själander, M., Jahre, M., Tufte, G., and Reissmann, N. (2019). EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure.

[Sutton and Barto, 2017] Sutton, R. S. and Barto, A. G. (2017). *Reinforcement Learning: An Introduction*.

[Teh et al., 2017] Teh, Y., Bapst, V., Czarnecki, W. M., Quan, J., Kirkpatrick, J., Hadsell, R., Heess, N., and Pascanu, R. (2017). Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4496–4506.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

[Wenke et al., 2019] Wenke, S., Saunders, D., Qiu, M., and Fleming, J. (2019). Reasoning and generalization in rl: A tool use perspective. *arXiv preprint arXiv:1907.02050*.

[Zhang et al., 2018] Zhang, C., Vinyals, O., Munos, R., and Bengio, S. (2018). A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*.