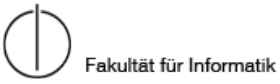
Robotics, Cognition, Intelligence

Technische Universität München



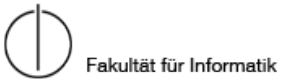
Master's Thesis

A Closed-Loop Control System via Deep Sequence Modeling and Reinforcement Learning

Guillermo González de Garibay Barba

Robotics, Cognition, Intelligence

Technische Universität München



Master's Thesis

A Closed-Loop Control System via Deep Sequence Modeling and Reinforcement Learning

Ein Regelkreis durch Deep Sequenz Modellierung und Reinforcement Learning

Author:	Guillermo González de Garibay Barba
1st examiner:	Prof. Dr.-Ing. Klaus Diepold
Assistant advisor:	Dominik Meyer MSc.
Submission Date:	September 15th, 2017

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

September 15th, 2017 Guillermo González de Garibay Barba

Acknowledgments

Thanks to my parents and my sister for the push and support they have given me. Thanks to Mariana for accompanying me in this journey. And thanks to everyone that openly shares work and knowledge, because without them this work would have been impossible.

"If I have seen further, it is by standing on the shoulders of giants."

-Isaac Newton

Abstract

Deep Learning has become the state of the art in many fields. From Computer Vision, passing by generation of realistic images, to Natural Language Processing, long established methods in these fields are being surpassed by Deep Learning methods or combinations of traditional algorithms with Deep Learning methods. Much of what is currently possible with these methods is limited by the amount and type of data available, but this is not the only limitation. Generating sequences of actions in interaction with an environment, such as in robotics, or for maintaining a conversation, are tasks that generally escape Deep Learning methods yet. Having a model of how actions affect an environment seems to be something potentially useful in this context, independently of how this model gets used.

Learning always requires some kind of supervision, be it a label indicating a category, an observation telling how wrong a prediction is, or a reward signal positively or negatively reinforcing some action or series of actions. Learning to generate sequences of actions can be done by explicitly indicating actions for each possible situation, which is not practical in general. It can also be done by providing some simple metric of the *goodness* of the realized actions. Reinforcement Learning provides an useful framework for analyzing this last type of supervision signal. The process of teaching what actions to perform can be focused in multiple ways.

There exist multiple model-free Reinforcement Learning approaches. These approaches try to directly learn a mapping from observations to actions. Making sure that the learning process is stable is one of the focus points of model-free approaches. Exploring the consequences of actions in all possible situations is also a hard problem. An alternative to studying stabilization procedures is following a different approach that might intrinsically be more stable. Following previous work, a model-based Reinforcement Learning system is proposed. The main idea behind it is having an action-generating controller that simultaneously exploits and improves a model, all with the goal of maximizing a series of rewards. This simultaneously provides a solution to the exploration problem, by reinforcing exploration by the controller.

An essential concept for a model-based reinforcement learning system is credit assignment. A good model should not only be able to predict future observations, but also to trace responsibility for some event as far backwards in time as it is possible. In this work a proposal for full credit assignment under a correctly predicting model is made. This is achieved by closing some feedback loops in the model. In this work, some variations of the structure of models and feedback loops are tested, leading to a final conclusion about the right structure for the model and controller. Validating the capabilities of this proposed model and controller will be part of future work.

Contents

Part I.

Introduction and Background Theory

1. Introduction

Deep Learning (DL) in neural networks (NN) studies a way of solving the *fundamental credit assignment problem* (Minsky, 1961), answering these questions (Schmidhuber, 2014): "Which modifiable components of a learning system are responsible for its success or failure? What changes to them improve performance?". There exist multiple approaches for modeling neural networks. Some of these models try to imitate biological neurons faithfully. In this work I focus instead on a model which arranges neurons in layers, where each of these layers is mathematically similar to a linear regression model. The high expressiveness of these models is achieved by stacking many of these layers, thus *deep* learning.

These DL models are currently trained *end-to-end*. This means the learning process consists of presenting the model with some input data, having the model generate outputs, and using some loss function penalizing wrong outputs from the model. DL systems learn to extract hierarchical features (i.e. features composed of simpler features hierarchically) from data in the process of minimizing some loss function. However, defining this loss function requires deciding what the target values are for the outputs of the network. This process of defining the loss function is called *supervision*. A way of supervising a model is having humans label input data with the desired outputs. Labeling many pieces of input data and randomly training once and again over them is bringing state-of-the-art results in multiple fields, such as Computer Vision or Natural Language Processing.

Many large human-labeled data-sets exist nowadays, which have greatly helped getting increasingly good results from DL systems, but the process of labeling is quite costly and thus minimizing this cost is of great interest. The existing amount of unlabeled data greatly exceeds the labeled data, so systems that can learn useful models from unlabeled data could greatly increase the capabilities of DL models. Training on unlabeled or scarcely labeled data still requires some form of supervision, and this is what Auto-encoders, Generative Adversarial Networks and Reinforcement Learning do.

Both Auto-encoders (AEs) and Generative Adversarial Networks (GANs) require nothing aside from the input data. Auto-encoders (AE) are DL systems that learn a lower-dimensional representation of their input. AEs consist of high-dimensional input and output layers with a low-dimensional layer in the middle of the deep network. The loss function for training AEs is defined as to minimize the difference between the input and output data, thus creating a low-dimensional representation. In this way, they are conceptually close to Principal Component Analysis, with the difference that AEs apply non-linear transformations.

On the other hand, Generative Adversarial Networks (GAN) are formed by two networks where one network generates data and a second network discriminates if the generated data

is real or generated. The *supervision* data of the second network is the given information that the training data is *real*, having the network learn a binary classification. The generating network has a loss function that is minimized by maximizing the error of the discriminating network.

Contrasting with AEs and GANs, which can be trained on raw data (with its implicit information as *supervision*), Reinforcement Learning (RL) systems require an extra piece of data, additional to the sequences of observations of the environment where they try to learn to act. The extra piece of data is a scalar signal representing a reward over time. The target of these systems is maximizing the sum of the rewards over time by generating a sequence of actions.

Recent successful approaches to Deep Reinforcement Learning[1] capture temporal information by concatenating consecutive frames. This strategy allows capturing temporal variables such as velocity but cannot capture temporal dependencies over times longer than the time windows provided as input. Even simple things such as periodic behaviours of environments cannot be captured if the temporal window of the input is too small. The concatenation of frames can be avoided and generalized by using a Recurrent Neural Network. Recurrent Neural Networks have special layers that also receive the outputs of previous sequence steps as inputs, additionally to the outputs of previous layers. This means that (e.g. in a temporal sequence) outputs depend on all previous inputs. Some Deep Reinforcement Learning models use this kind of networks with a temporal series of observations as input, and actions as outputs (e.g. (Heess et al., 2015)).

Recurrent Neural Networks (RNNs) (in particular Long-Short-Term Memory (LSTM) Networks, a variant that achieves longer term propagation of errors, see section 3.1.3) have shown to be able to learn complex temporal patterns (e.g. for speech and hand script recognition and generation). RNNs can also be used to build models predicting future observations in sequences (something like a temporal auto-encoder). Predicting future inputs requires learning to recognize all temporal features that can affect the future. These features can also be used for learning optimal sequences of actions. If a network can make optimal predictions about the future, then it must also have extracted all features necessary for acting optimally, since information not affecting the future cannot possibly be relevant for acting in the future. Of course, the problem of learning to act on those features still remains.

Another critical component of RL systems is exploration. Typically, DL systems are trained via Stochastic Gradient Descent on randomized mini-batches of data. However, the observations captured by RL systems depend on the actions they output, so collecting unbiased training data is harder. Only some particular sequences of actions lead the system to certain observations which might be critical for learning to solve the task. Thus a RL system needs to explore the action space in order to find a sequence of actions that solves the task. The standard methods for exploration are randomly choosing actions from the

[1] Deep Reinforcement Learning refers to using Deep Learning for solving Reinforcement Learning tasks.

action space or adding some noise to chosen actions. This is however not efficient in many cases, since random actions don't necessarily lead to informative observations for learning. A proposed heuristic for improving exploration is rewarding the RL system for being *curious*. What *curiosity* means in this context is not clear and there are multiple approaches to it. Most approaches are based on building some some kind of novelty metric (e.g. gradient norm). Having a network explicitly model an environment provides a way of measuring novelty (e.g. as the prediction error).

Summing up, RL systems are of interest because they can learn from little labeling, and potentially in a fully unsupervised way. Applying RNNs to RL systems is of interest because they can capture mid- and long-term patterns from observations, which is critical for solving some tasks. Building models that learn to predict future observations is of interest for three reasons. Firstly, during the learning process they also capture all possibly future-relevant information from the observations. Secondly, they allow implementing some kind of curiosity reward, which makes exploring environments easier. And thirdly, coming back to the credit assignment problem, having a model of the environment means being able to assign long-term responsibility for predictions, actions and rewards.

2. Background

There are two main research lines directly related to this work. The first one is about the set of model-free approaches (DQN (Mnih et al., 2015) (Riedmiller, 2005), DDPG (Lillicrap et al., 2015), A3C (Mnih et al., 2016), TRPO (Schulman et al., 2015), PPO (Schulman et al., 2017), and others) that constitute the state of the art in Deep Reinforcement Learning. A common problem found in this approaches is that learning becomes unstable if the state transitions sampled from the environments are not randomly mixed before iterating on the parameters, so that not all transitions used are consecutive.

The second line of work is about building models of an environment. Of special interest are environments where observations are images, due to the modeling difficulty outside of DL. Some approaches estimate individual pixel movement explicitly, which does not generalize for *all* possible video sequences, and some others take a more general modelling approach, allowing for *any* transformation between consecutive frames. One of the more general approaches inspiring this work applies LSTMs (see section 3.1.3) for modeling text one character at a time.

2.1. Model-free Reinforcement Learning

Model-free Reinforcement Learning consists in learning a policy. This means, a function that produces a certain probability distribution on an action space from state observations. This can be achieved in multiple ways, depending on the geometry of the action space. As stated in (Schulman et al., 2017):

> In recent years, several different approaches have been proposed for reinforcement learning with neural network function approximators. The leading contenders are deep Q-learning (Mnih et al., 2015), "vanilla" policy gradient methods (Mnih et al., 2016), and trust region / natural policy gradient methods (Schulman et al., 2015). However, there is room for improvement in developing a method that is scalable (to large models and parallel implementations), data efficient, and robust (i.e., successful on a variety of problems without hyperparameter tuning). Q-learning (with function approximation) fails on many simple problems and is poorly understood, vanilla policy gradient methods have poor data effiency and robustness; and trust region policy optimization (TRPO) is relatively complicated, and is not compatible with architectures that include

noise (such as dropout) or parameter sharing (between the policy and value function, or with auxiliary tasks).

The main difference between these methods is if an advantage/action-value function can be evaluated for multiple actions in order to choose an action (e.g. greedily). In general (with continuous action spaces, or very high dimensional discrete action spaces) this is not possible due to computational complexity. For low-dimensional and discrete action spaces, DQN/Q-learning (Mnih et al., 2015) seems to work well. It consists in learning to predict an expectation of the discounted sum of rewards for each possible action at a certain state. The alternative is learning a policy through Policy Gradient methods, with algorithms similar to Deep Deterministic Policy Gradients (DDPG) (Lillicrap et al., 2015) or more elaborate variants such as Proximal Policy Optimzation (PPO) (Schulman et al., 2017), Trust-region Policy Optimization (TRPO) (Schulman et al., 2015) or Natural Policy Gradients (NPG) (Kakade, 2002). All these policy gradient methods target learning a function (policy) that produces optimal actions from observations. This function is trained by maximizing another function, an action-value function, which estimates the expectation of the discounted sum of rewards for the observation and action pair. The focus of the variants is generating updates to the policy without destabilizing the learning process. An interesting thought is if a different system architecture would make learning stable instead of needing to tune these algorithms.

There is also some work replacing the fully connected networks of the previously discussed approaches by Recurrent Neural Networks (in particular Long-Short Term Memory networks, a variant with better performance on long sequences; see section 3.1.3). For example both the policy function and the action-value function of the DDPG algorithm (which are Feed-forward Neural Networks) are replaced by RNNs in the RDPG algorithm (Heess et al., 2015). Also in (Bakker, 2002), an algorithm quite close to DQN is presented but with an LSTM network instead of a Convolutional Neural Network.

2.2. Model building through Deep Learning

In this context we talk about *models* as systems predicting some kind of future observation. There is a variety of work concerned with building models outside of the Reinforcement Learning context.

Some work uses models for semantic segmentation in video. For example in (Patraucean et al., 2015) the authors use LSTM networks for predicting optical flow, in turn used for semantic segmentation, while in (Neverova et al., 2017) the authors predict future semantic segmentation frames using Convolutional Neural Networks (CNNs). In this last approach the authors propose extending their approach with Recurrent Neural Networks in future work.

Building models on video sequences can also be used for anomaly detection, as discussed in (Medel and Savakis, 2016). In this case a Convolutional LSTM network is used. This

kind of network uses Convolutional layers instead of fully connected layers inside the LSTM cells.

Other work focuses on audio generation (Mehri et al., 2016), handwriting generation (Graves, 2013) or high quality video prediction (Kalchbrenner et al., 2016), all using RNNs.

2.2.1. Related to Reinforcement Learning

There are also multiple model-building methods related to Reinforcement Learning. In contrast to the algorithm proposed in this work, most model-based approaches to RL focus on exploiting a model for planning. For example (Finn and Levine, 2016) (Finn et al., 2016) propose using pixel motion prediction for planning robotic motion tasks. Pixel motion prediction is not general enough for developing an algorithm able to learn to act in arbitrary environments. Also, successful planning requires having a model that can be faithfully simulated many time steps forward. Training this kind of model on environments where little long-term dependencies occur requires explicitly training long-term predictions. Extensive work on this kind of action-conditioned models has been performed in (Chiappa et al., 2017), following the work of (Oh et al., 2015). In these cases a CNN network (see subsection 3.1.2) encodes observations into a lower-dimensional code, while an LSTM predicts action-conditioned codes of future time steps. The output of the prediction is then decoded into the predicted observation image in order to compute the error and the loss. According to the results in (Chiappa et al., 2017), explicitly training long-term predictions is required for producing accurate long-term simulations. In (Chiappa et al., 2017) there is also a proposal for an architecture that does not require running the encoder/decoder networks, achieving higher computational efficiency. However, it achieves that by doubling the number of parameters. Such concept is also interesting for developing an agent that can learn to exploit a model-network, as discussed in (Schmidhuber, 2015).

2.2.2. Char-RNN

The `char-rnn` algorithm is of special interest for this work because of the analyses that have been done on the resulting models. The model is a simple LSTM (or similar) network learning to predict the next character in text. These analyses show that significant features of sequences can be learned by LSTM networks. The inputs are just *one-hot* encodings of the previous characters. A first analysis of these models (Karpathy et al., 2015) shows that LSTMs learn relatively complex sequential features, such as counting the number of characters since the last line break or keeping track of opening and closing brackets, which allow them to generate convincing texts in multiple formats (Karpathy, 2015). In (Radford et al., 2017) the authors show that this kind of unsupervised text modeling can learn to extract such a complex feature as the *feeling* embedded in some text. This is done without providing the network with any emotional labels in the training phase. The main learning phase consists in training a very wide LSTM network in an unsupervised way on Amazon product reviews. Later a single layer is trained on top of the hidden states of the LSTM

network. This layer is trained to predict if the product review is positive or negative (which is part of the given data). By analyzing the result of this training the authors realize that a single hidden unit has learned to reflect the *feeling* of the clients.

These analyses on `char-rnn` are specially interesting because they make evident that the state cells of LSTM networks trained on a primary task, contain information that can be easily exploited by other networks. This is also argued in (Schmidhuber, 2015).

2.2.3. Schmidhuber's Algorithmic Information Theory Argument

In the following, C denotes a RL controller and M a predictive world model. This is a quote from (Schmidhuber, 2015).

> *While FNNs [Feed-forward Neural Networks] are traditionally linked [...] to concepts of statistical mechanics and information theory [...], the programs of general computers such as RNNs call for the framework of Algorithmic Information Theory (AIT) [...] (...). Given some universal programming language [...] for a universal computer, the algorithmic information content or Kolmogorov complexity of some computable object is the length of the shortest program that computes it. Since any program for one computer can be translated into a functionally equivalent program for a different computer by a compiler program of constant size, the Kolmogorov complexity of most objects hardly depends on the particular computer used. Most computable objects of a given size, however, are hardly compressible, since there are only relatively few programs that are much shorter. Similar observations hold for practical variants of Kolmogorov complexity that explicitly take into account program runtime [...]. Our RNNAIs are inspired by the following argument.*
>
> *[...]* ***Basic AIT Argument***
>
> *According to AIT, given some universal computer, U, whose programs are encoded as bit strings, the mutual information between two programs p and q is expressed as $K(q|p)$, the length of the shortest program $\bar{w}$ that computes q, given p, ignoring an additive constant of $O(1)$ depending on U (in practical applications the computation will be time-bounded [...]). That is, if p is a solution to problem P , and q is a fast (say, linear time) solution to problem Q, and if $K(q|p)$ is small, and $\bar{w}$ is both fast and much shorter than q, then asymptotically optimal universal search [...] for a solution to Q, given p, will generally find $\bar{w}$ first (to compute q and solve Q), and thus solve Q much faster than search for q from scratch [...].*
>
> *[...]* ***One RNN-Like System Actively Learns to Exploit Algorithmic Information of Another***
>
> *The AIT argument [...] above has broad applicability. Let both C and M be RNNs or similar general parallel-sequential computers [...]. M's vector of*

> *learnable real-valued parameters w_M is trained by any SL [Supervised Learning] or UL [Unsupervised Learning] or RL [Reinforcement Learning] algorithm to perform a certain well-defined task in some environment. Then w_M is frozen. Now the goal is to train C's parameters w_C by some learning algorithm to perform another well-defined task whose solution may share mutual algorithmic information with the solution to M's task. To facilitate this, we simply allow C to learn to actively inspect and reuse (in essentially arbitrary computable fashion) the algorithmic information conveyed by M and w_M.*

This fragment introduces the framework of Algorithmic Information Theory for the analysis of learning complexity of Recurrent Reinforcement Learning algorithms. The argument states that learning a model of the world simplifies the task of learning a policy. This reinforces the idea that a model-based approach might solve some of the stability problems present in model-free algorithms.

2.2.4. A Controller integrated with the Model

The model explored in this work is very closely described in the following section of that same work (Schmidhuber, 2015):

> In one of the simplest cases, C is just a linear perceptron FNN (instead of an RNN like in the early system [...]). The fact that C has no built-in memory in this case is not a fundamental restriction since M is recurrent, and has been trained to predict not only normal sensory inputs, but also reward signals. That is, the state of M must contain all the historic information relevant to maximize future expected reward, provided the data history so far already contains the relevant experience, and M has learned to compactly extract and represent its regular aspects. This approach is different from other, previous combinations of traditional RL [...] and RNNs [...] which use RNNs only as value function approximators that directly predict cumulative expected reward, instead of trying to predict all sensations time step by time step. The CM system in the present section separates the hard task of prediction in partially observable environments from the comparatively simple task of RL under the Markovian assumption that the current input to C (which is M's state) contains all information relevant for achieving the goal.

The system discussed in a later section is very similar but regarded from a different perspective: The CM system is not regarded as a Controller and a Model interacting, but they are integrated into a single interdependent structure.

3. Theory

Deep Learning and Reinforcement Learning are the two main pillars of theory this work is based on. In the next two sections you can find an introduction to the concepts of these fields employed in this work. In the third —and last— section of this chapter a description of the design of the system proposed in this work can be found.

3.1. Deep Learning

This section is an introduction to the theory behind the techniques used for this work. For a comprehensive explanation of the Machine Learning and Deep Learning theory, please refer to (Goodfellow et al., 2016). For an exhaustive overview of the historical progress of Deep Learning, please refer to (Schmidhuber, 2014).

Deep Learning is the subset of Machine Learning concerning the study of Deep Artificial Neural Networks. Here I focus on a particular model of Artificial Neural Networks where artificial neurons are organized in layers. Each layer operates on an input by applying a linear transformation followed by an activation function. According to (Hornik, 1991), under certain conditions, a model with just two of this layers can approximate many functions:

> We show that standard multi-layer feed-forward networks with as few as a single hidden layer and arbitrary bounded and non-constant activation function are universal approximators with respect to $L_p(\mu)$ performance criteria, for arbitrary finite input environment measures μ, provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and non-constant, then continuous mappings can be learned uniformly over compact input sets. We also give very general conditions ensuring that networks with sufficiently smooth activation functions are capable of arbitrarily accurate approximation to a function and its derivatives.

It must be noted that this does not say anything about the learnability (i.e. the fact that a function exists does not imply that it can be found).

3.1.1. Basics

The simplest layer operating on an input $\boldsymbol{x} \in \mathbb{R}^n$ producing an output $\boldsymbol{y} \in \mathbb{R}^m$ is what is called a fully connected (FC) layer. In the following equation, $\boldsymbol{w}$ denotes a parameter matrix $\in \mathbb{R}^{n \times m}$ and $\boldsymbol{b}$ a bias parameter $\in \mathbb{R}^m$.

$$y_j = f(\sum_i w_{ij} x_i + b_j), \tag{3.1}$$

where f is typically one of the following activation functions: *sigmoid* (σ), tanh, Rectified Linear Unit (`ReLU`), Exponential Linear Unit (`ELU`), or (used in this work) Scaled Exponential Linear Unit (`SELU`).

The first two have the advantage of bringing the values inside a certain range ($(0, 1)$ and $(-1, 1)$ respectively), they however have the problem that when the values are close to the extremes of the intervals, the derivatives are very close to 0, making learning slow.

The $\texttt{ReLU}(x) := \max(0, x)$ function fixes this for $x > 0$ with a constant derivative; but for $x < 0$ the derivative is exactly 0, which makes learning stop if pre-activation values become negative permanently.

The `ELU` function

$$\texttt{ELU}(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

improves the `ReLU` function for $x < 0$ by making the derivative not 0 (it converges to 0 though).

The `SELU` function is a variant of `ELU`

$$\texttt{SELU}(x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}, \text{ with } \lambda = 1.0507 \text{ and } \alpha = 1.67326$$

which is a component of Self-Normalizing Neural Networks (Klambauer et al., 2017). This scheme keeps activations implicitly normalized through multiple layers as long as the statistical distribution of the parameters and inputs fall inside certain ranges. `SELU` activations get used in this work with no adverse effect on performance expected (i.e. no expected loss of generality).

As can be observed in Equation 3.1: $\boldsymbol{w} \in \mathbb{R}^{n \times m}$. This implies that $\boldsymbol{w}$ can already become of very high dimensionality for high dimensional inputs and relatively low dimensional outputs. A clear example of high dimensional inputs are images. The naive way of applying a neural network to an image with color channels would be flattening the image into a `1-D` array and applying the layer defined in 3.1. By training multiple layers like this, they would learn to extract the features relevant for the corresponding task, but they would take much longer than what is necessary.

3.1.2. Convolutional Neural Networks

Images can be processed much more efficiently by realizing that *large* images present the same structures at multiple locations in the image (a face can appear *anywhere* inside an image). This means that, instead of learning the same features multiple times on different

parts of the images, we can tie some values in $\boldsymbol{w}$ so that the number of parameters is reduced (and thus, easier to learn). We should also note that pixels related to each other tend to be close to each other (e.g. the pixels representing a face are all together inside a picture, and unrelated to the dog in the other corner of the picture). We can also use this for reducing the number of parameters.

In a naive approach, each component of the output is a feature that depends on *all* of the input values. If the features we want to extract are local (as opposed to global), as we have argued, we can use our observation to set most of the values in $\boldsymbol{w}$ to zero (i.e. force independece).

What all this means in practice is that the huge parameter matrix corresponding to Equation 3.1 can be considered sparse and with repeated groups of values. This is normally represented by defining multiple small (e.g. $3 \times 3 \times N$, $4 \times 4 \times N$, $5 \times 5 \times N$; where N is the number of channels the image has) square filters that are applied at regular intervals (or every position) across an input image image. This operation is actually applied as a cross-correlation operation, but in this field is called a convolution (this just implies reversing the filters).

When M filters (also called *kernels*) of the same size $K \times K$ are applied on a `2-D` image with N input channels, they generate a `2-D` output with M channels. The set of filters is represented by $\boldsymbol{w} \in \mathbb{R}^{M \times N \times K \times K}$; $\boldsymbol{b} \in \mathbb{R}^m$ is a bias parameter.

$$y_{ijm} = f\Big(\sum_{k_1,k_2,n} x_{i+k_1,j+k_2,n} w_{mnk_1k_2} + b_m\Big) \tag{3.2}$$

Note that in order to get `2-D` arrays of the same shape, the input needs to be padded. This has been obviated in the previous equation.

This operation can be also applied at regular intervals (instead of to every position) on the input so that the size of the `2-D` output is reduced. The distance between points where this operation is applied is typically called *strides*. When using *strides* larger than 1, the sizes of output layers are divided in each dimension by the size of the *strides*.

There exists a reciprocal operation where the value of the *strides* increases the size of the output. This operation is called Transposed Convolution, and it is actually the transpose of the parameter matrix when considering it as a huge sparse matrix with repeated values. When using this operation for generating images from lower dimensional feature vectors, one should pay attention to the relative sizes of the *strides* and the kernels (Odena et al., 2016).

When multiple convolutional layers are stacked, the higher layers combine the local features extracted by lower layers into increasingly complex features (e.g. features of lower layers represent simple features such as edge orientations, and features of higher layers can represent a dog).

It should be noted that for a kernel size larger than 1×1, the higher layers have an increasingly larger *dependence region* on the input. For example, for a kernel size of 3×3,

the activations of the first layer depend only on values in regions of $3 \times 3 \times N$. Then, the activations of the second layer depend on $3 \times 3 \times M$ patches of the activations of the first layer, and in turn on a $5 \times 5 \times N$ region of the input. This effect is increased when using *strides* larger than 1.

This kind of computation is very efficiently done by GPGPU hardware (General-Purpose computing on Graphics Processing Units) due to their processor and memory architecture. This kind of hardware has greatly accelerated developments in the Computer Vision field.

Another increment in efficiency can be achieved by operating the convolution in the frequency domain. This is however most efficient for large sized kernels, which is typically not the case, since there is no clear empirical advantage for larger kernels. Thanks to Deep Learning frameworks (section 4.1) this kind of efficiency problems are generally abstracted away and algorithm developers can focus on other details.

It should be noted that this kind of convolutional operations can also be applied on `1-D` (multi-channel scalar signals), `3-D` (video data), or higher dimensional data with similar properties.

However, when using convolutions on temporal data, the outputs can only depend on a limited set of consecutive values of the temporal sequence. The size of the dependence window on the input is a function of the number of convolutional layers and the sizes of the kernels used, as explained earlier. In typical temporal models, outputs of temporal sequences can depend on *all*[1] past values (and even all future values sometimes, see bi-directional RNNs in (Schuster and Paliwal, 1997)), and for this kind of dependencies, Recurrent Neural Networks are more appropriate.

3.1.3. Recurrent Neural Networks

Just like CNNs take advantage of properties intrinsic to images, Recurrent Neural Networks (RNNs) take advantage of properties intrinsic to sequential data. Specifically, RNNs are appropriate for modelling when the output of a network after input number t should depend on all inputs until t, and when it makes sense to apply the same transformation at each step of the sequence. Because computing the influence of every previous input gets intractable very quickly for long sequences, a Markov assumption is made. This means that the output of a recurrent network after input t depends only on the input t and the output of the network at $t-1$. In this sense, a RNN is often defined as in Equation 3.3. This type of basic RNN is often referred to as *vanilla*[2] RNN. In the following equation, $\boldsymbol{w}^{\text{input}}$ and $\boldsymbol{w}^{\text{recurrent}}$ are matrices of parameters just like those of fully connected layers. In fact, this RNN layer is mathematically equivalent to two fully connected layers (Equation 3.1) interacting additively before the non-linearity.

[1] If not all, at least an undetermined number within an indefinitely long period of time.
[2] The term comes from the standard flavor of ice cream, vanilla.

$$h_j^t = f(\sum_i x_i^t w_{ij}^{\text{input}} + \sum_k h_k^{t-1} w_{kj}^{\text{recurrent}} + b_j) \tag{3.3}$$

Even being theoretically able to capture long term dependencies, the structure of vanilla RNNs makes it in practice hard to train such networks to carry information across many recurrent transformations. LSTM networks provide a way of overcoming this difficulty.

Long Short-Term Memory Networks

As presented in (Hochreiter and Schmidhuber, 1997):

> Learning to store information over extended time intervals via recurrent back-propagation takes a very long time, mostly due to insufficient, decaying error back flow. We briefly review Hochreiter's 1991 analysis of this problem, then address it by introducing a novel, effcient, gradient-based method called "Long Short-Term Memory" (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing constant error flow through constant error "carrousels" within special units. [...]

LSTMs introduce an error carousel C that does not go through any linear transformation. It just receives additive and multiplicative operations as described in Equation 3.4. This requires computing 4 different values for each cell: forget, input, output, update. Each of these values is computed just like the output of a vanilla RNN, but the key is how they interact.

At each sequence step, the values of the state C are updated via 2 operations. A forget gate (f) with values in the range $(0, 1)$, multiplies the values of C as the first operation. This allows erasing the contents of the cell. In parallel, the update value is multiplied by the input gate (i), with values in the range $(0, 1)$, and added to C as the second operation. These two operations result in an updated value for C. The output of the layer is finally produced by first applying a non-linearity to the output of the layer and then multiplying the result by the output gate (o).

A representation of these interactions can be found in Figure 3.1.

$$
\begin{aligned}
f_j^t &= \sigma(\sum_i x_i^t w_{ij}^{if} + \sum_k h_k^{t-1} w_{kj}^{rf} + b_j^f) \\
i_j^t &= \sigma(\sum_i x_i^t w_{ij}^{ii} + \sum_k h_k^{t-1} w_{kj}^{ri} + b_j^i) \\
o_j^t &= \sigma(\sum_i x_i^t w_{ij}^{io} + \sum_k h_k^{t-1} w_{kj}^{ro} + b_j^o) \\
C_j^t &= f_j^t C_j^{t-1} + i_j^t \tanh(\sum_i x_i^t w_{ij}^{iC} + \sum_k h_k^{t-1} w_{kj}^{rC} + b_j^C) \\
h_j^t &= o_j^t \tanh(C_j^t)
\end{aligned}
\tag{3.4}
$$

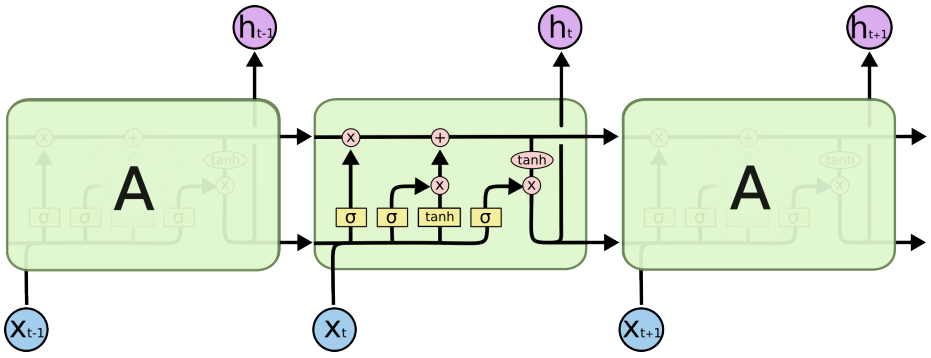


Figure 3.1.: A representation of an LSTM network (Olah, 2015). A thorough explanation of LSTM networks can be found at this same source.

LSTM Search Space Odissey It can be argued that certain small variations of the LSTM architecture can be beneficial for certain tasks. This opens a discussion about what precise architecture an LSTM should have. In (Greff et al., 2015), many LSTM variations have been systematically tested on multiple tasks. The forget and output gates seem to be the most critical components. Aside from that no variation seems to provide any general advantage.

3.1.4. Training

Deep Learning models are currently trained via Stochastic Gradient Descent (SGD). This means that a small set of data (*mini-batch*) is evaluated through the model. This is frequently called the *forward pass* of the network. Then a loss is computed for this small set of data, according to some supervision data. Next, the derivative of the loss with respect to the model parameters is computed via *backpropagation*. This is often called *backward pass*. The *backpropagation* algorithm consists in differentiating the model using the chain rule

(reverse-mode differentiation in particular), but taking advantage of cached values from the *forward pass*. Finally, the model parameters are updated by some magnitude in the direction opposite to the computed stochastic gradient, as described here, updating $\boldsymbol{w} \in \mathbb{R}^{N \times M}$ at step s in the following way:

$$w_{ij}^{(s+1)} = w_{ij}^{(s)} - \alpha \nabla_{w_{ij}^{(s)}} \text{Loss}. \tag{3.5}$$

The parameter α scaling the gradient is called *learning rate*, and needs to be adjusted depending on the problem, normally taking *small* values. The super-index $\cdot^{(s)}$ specifies a particular set of values for the corresponding variable at step s of an iterative process.

In order to avoid computing all the corresponding derivatives by hand, automatic differentiation software is widely used (see section 4.1).

There are other approaches for training these networks (e.g. random search, pre-training or genetic algorithms), but currently SGD is the most favoured method.

It should be noted that the optimization process is not convex in models deeper than one layer, even if the loss function is convex. Thus, a convergence to a global minimum cannot be guaranteed. However, the empirical evidence of the field shows that most times a "good enough" local minimum can be found. Finding these local minima normally requires initializing the model parameters within a certain range. This range is different depending on the non-linear activation function used (Glorot and Bengio, 2010) (He et al., 2015a), but it is defined so that the statistical distribution of the activations is close to normal (a close to normal distribution for the input data is also frequently assumed). This weight initialization is also often abstracted away by Deep Learning frameworks (section 4.1), although one should not take that the default initialization will work well.

Another technique often used in gradient descent methods is computing a *momentum* for the gradients, modifying the update of the parameters in the following way:

$$\begin{aligned} \mu_{ij}^{(s)} &= \beta \mu_{ij}^{(s-1)} + \nabla_{w_{ij}^{(s)}} \text{Loss} \\ w_{ij}^{(s+1)} &= w_{ij}^{(s)} - \alpha \mu_{ij}^{(s)} \end{aligned} \tag{3.6}$$

Here β, the momentum parameter, normally takes values of 0.9 or 0.99. It causes the parameter updates to be proportional to an exponential moving average of the gradient values. A theoretical justification for *momentum* can be found in (Goh, 2017).

Some other empirically useful modifications of SGD exist (e.g. RMSProp, Adam, gradient clipping) and are useful in different circumstances. For simplicity they have been kept out of this work.

3.2. Reinforcement Learning

For an in depth introduction to Reinforcement Learning, please refer to (Sutton and Barto, 1998) or (Sutton and Barto, 2017-draft).

Reinforcement Learning is the branch of machine learning which concerns learning to act by maximizing some cumulative reward function. Or, as put in (Sutton and Barto, 2017-draft):

> Reinforcement learning [...] is simultaneously a problem, a class of solution methods that work well on the class of problems, and the field that studies these problems and their solution methods. Reinforcement learning problems involve learning what to do —how to map situations to actions— so as to maximize a numerical reward signal. In an essential way these are *closed-loop* problems because the learning system's actions influence its later inputs. Moreover, the learner is not told which actions to take, as in many forms of machine learning, but instead must discover which actions yield the most reward by trying them out.

The process of training an agent how to act requires learning a *policy*. We call a *policy* to a function mapping a state to a probability distribution over actions. Learning a policy often requires learning some auxiliary function (e.g. action-value or advantage function for policy gradient algorithms). In general, an exact solution is not available and the exact functions are replaced by approximations. *Deep* Reinforcement Learning just refers to using Deep Neural Networks to approximate any of these functions.

In this work, the Reinforcement Learning problem is focused in the following way: An agent A receives at time t an observation $\boldsymbol{o}^t \in \mathbb{R}^n$ from the environment E (with a current state of the environment $\boldsymbol{e}^t \in \mathbb{R}^x$) and a reward $r^t \in \mathbb{R}^1$. The agent uses the received information to update its internal state to $\boldsymbol{s}^t \in \mathbb{R}^m$ and produces an action $\boldsymbol{a}^t \in \mathbb{R}^l$. This action together with the intrinsic properties of the environment updates the internal state of the environment to $\boldsymbol{e}^{t+1}$, and the loop continues.

It should be noted that the reward can be generated by the environment but is mostly intrinsic to the agent (i.e. different agents acting in the same way in the same environment might receive different rewards, whereas the observations they receive should be exactly the same).

It is also important to note that an observation $\boldsymbol{o}^t$ provides, in general, only partial information of $\boldsymbol{e}^t$ (the environment's *true* state). Thus, the agent must deal with partial information in order to act optimally. The most complete information the agent can "decide" on for generating $\boldsymbol{a}^t$ is the full history of observations and actions $\{\boldsymbol{o}^0, \boldsymbol{a}^1, \ldots, \boldsymbol{a}^{t-1}, \boldsymbol{o}^t\}$. Storing and processing all this information for each $\boldsymbol{a}$ is, in general, too expensive; and, in many cases, redundant (e.g. see lossless compression).

Thus, what an efficient agent might do, is updating its internal state $\boldsymbol{s}^t$ with the information it receives at each loop step to optimally approximate $\boldsymbol{e}^t$. As previously discussed,

this might be achieved by training a model with an internal state $\boldsymbol{s}$ to optimally predict every observation $\boldsymbol{o}$ after seeing all previous actions and observations. For this, a Recurrent Neural Network seems to be a good candidate.

A comment on environments Part of the difficulty of learning how to act in each state lies on *exploring* what consequences each action has. Without any prior knowledge of an environment, this problem is a random search problem. In fact, many solutions to the exploration problem rely on random search (e.g. (Mnih et al., 2015)), random noise added either to the output of a policy (e.g. (Lillicrap et al., 2015)) or to the parameter space of the algorithm (e.g. (Plappert et al., 2017)). Providing with *supervision* via a more informative reward function can also help (i.e. complex continuous *real* rewards vs simple discrete binary rewards). There is an interesting recent publication showing that training a single agent simultaneously on multiple related tasks with simple rewards can substantially help exploration (Heess et al., 2017).

3.3. A Closed-Loop Deep Reinforced Controller and Model

The goal of this section is presenting a design for a Deep Reinforcement Learning system combining many of the ideas presented in this work. The description increases the complexity of the model step-by-step, motivating each increment.

The starting point is quite close to the design used in previous environment modelling work (Chiappa et al., 2017) (Oh et al., 2015). The model can be seen in Figure 3.2.

The forward evaluation of the model is represented top to bottom; time progresses left to right. In the figures, o denotes observations, a actions, and r rewards. The sub-index t denotes a particular time step. The symbol $\sim$ over a variable denotes that the value is an approximation subject to optimization. The recurrent connections across time steps are represented by the variables h, for the LSTM output, and C, for the error carousel. k and d represent input and output codes respectively. In the models where the output code is a prediction of the next input code, and not a different code, the symbol d is replaced by $\tilde{k}$. A super-index on k or d denotes if the code specifically represents an observation o, action a, or reward r. The symbol $\frown$ denotes concatenation, and $|$ denotes the opposite operation, splitting. The horizontal line below the concatenations has no specific meaning, it reinforces the representation of the concatenation.

The model consists of a LSTM network with a feature encoder (CNN) attached at the input and a fully connected (FC) layer and a decoder (Transposed CNN) at the output. This basic model is a generalization of `char-rnn` to image inputs [3].

The core of the model (see the block LSTM&FC in Figure 3.2) is an LSTM layer with a fully connected layer at its output, so the output of the block is not the same as the output of the LSTM network. The FC layer is not present in previous work modelling environments, but is a critical component in the `char-rnn` algorithm (where the encoder and decoder are

[3] It might be interesting to train it on images of characters.

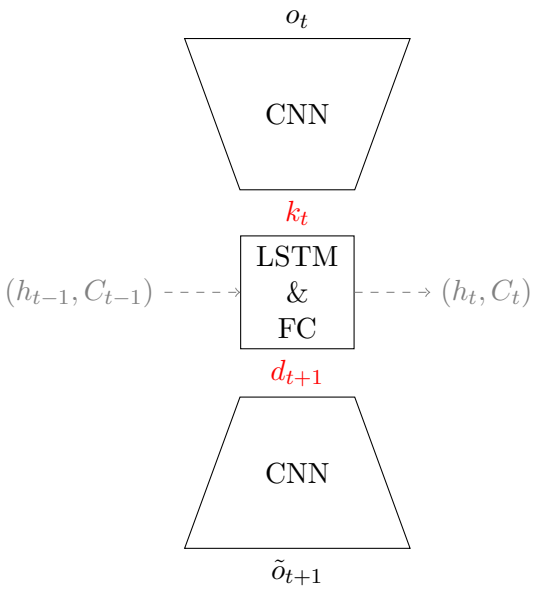


Figure 3.2.: Recurrent model trained end-to-end. The inputs are observations, represented at the top. They are encoded to a low dimensionality representation through a CNN, combined with previous observations in the LSTM cell, and decoded through a FC layer and a Transposed CNN. The output, at the bottom, is a prediction of the next input. Referred to as the "main" model in the Results chapter.

omitted). It is easy to argue for this layer in the case of the `char-rnn` algorithm, where the encoding and decoding functions are not learned (a one-hot representation of letters is used). If no FC layer was connected at the output of the LSTM cells, the output of the cells would need to directly represent the one-hot outputs, restricting the number of LSTM cells, and thus tying their activations to represent the characters directly, instead of e.g. *feelings* (Radford et al., 2017). This layer is thus critical for having flexibility in choosing the number of LSTM units, and for removing restrictions on the feature spaces of shallow models. It is represented as FC in the LSTM&FC block in the figures.

An interesting extension of this basic model might be mixing multiple temporally-aligned observation sources. Each observation source (e.g. image, audio, accelerometers) would be encoded by a different function, and all the low dimensional representations could be concatenated before entering the LSTM&FC block.

A clear example of such combination of observations is combining an action and an observation. We are considering observations as partial-information samples from the environment state. Actions are just observations about the agent, so there is in principle no reason for dealing with them differently from any other observation. Actions are dealt with

in this way in a variation of action incorporation discussed in (Chiappa et al., 2017), with the difference that, in this thesis, it is considered that actions should be encoded (just like observations are) before concatenation. A representation of this concept can be found in Figure 3.3.

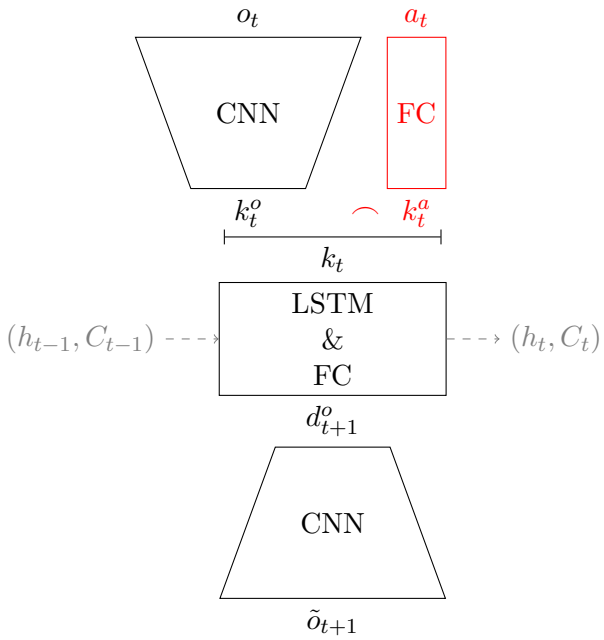


Figure 3.3.: Recurrent model trained end-to-end. Actions are incorporated as an additional observation.

To this point we have only discussed a minor variation of the work done in (Chiappa et al., 2017). This constitutes the structure of a world model (i.e. a version of the M component of the RNNAI system of (Schmidhuber, 2015)). The next component to be added is a controller C, which in the simplest symmetric case takes the mirrored form of the action-encoder. You can see a representation of this in Figure 3.4

The simplest case discussed in (Schmidhuber, 2015) for a CM (controller/model) system is having the controller be a single FC layer. Here, the controller is also considered as a decoder (thus the symmetry with the action-encoder), translating feature spaces for the action. In this sense, what the LSTM&FC block outputs in Figure 3.4 is the concatenated code of the predicted observation and the action to be done.

Note that, since the action is both interpreted and generated by the network, a new credit assignment path naturally emerges from this design. Namely, by letting errors backpropagate across time steps through the generated actions. This implies the observation loss can backpropagate through the decoded actions, effectively leading to a system that can learn

to generate actions minimizing a prediction error (i.e. a system that *stays in control*).

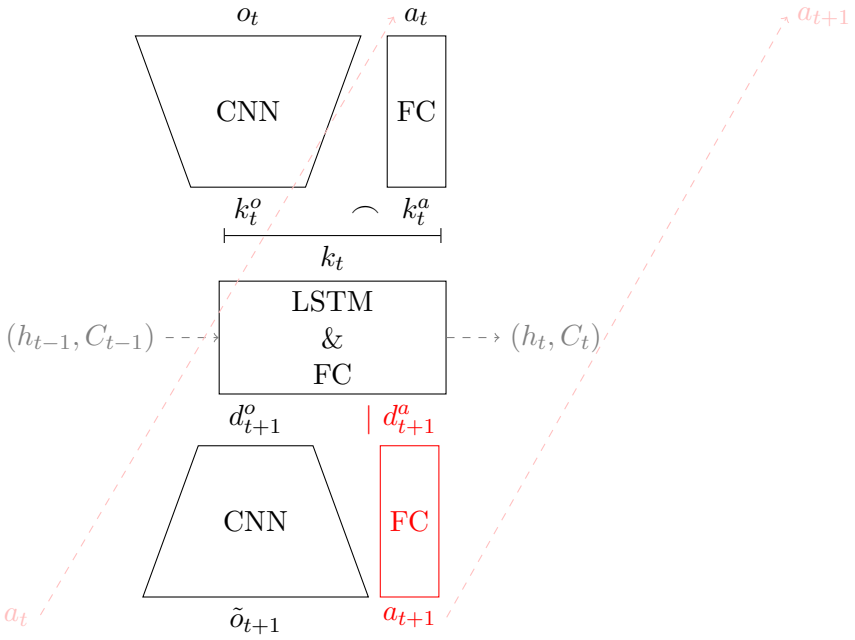


Figure 3.4.: Recurrent model trained end-to-end. Actions are incorporated as observations and generated as self-fulfilling predictions. This allows closing the action-loop for credit assignment.

A simple extension to the model in Figure 3.4 is including a prediction of the reward signal. The design can be seen in Figure 3.5.

This design makes an implicit non-binding assumption about the rewards. Namely that rewards can be predicted from observations and actions alone, independently from previous rewards [4].

This model could now be trained as to maximize a series of rewards by reinforcing (through gradient ascent) the predicted rewards. Since the model is trained end-to-end to predict the rewards, a credit assignment path can be found everywhere that backpropagation can occur. There is however a major block in the backpropagation loop, specifically the observations.

Even if the predicted observations are correct, this model has no way of directly assigning credit through observations. This is critical for *easy* environments where predictions $\tilde{o}_{t+1}$ are only dependent on the current observation o_t. In this case, the prediction does not

[4]This might seem unintuitive to someone thinking about electric shocks as a negative reward, but it is not the case since the pain originated by the shock can be considered an observation with an associated negative reward.

need to backpropagate errors through time through the LSTM network, since the recurrent dependence chain can be ignored and still successfully predict observations.

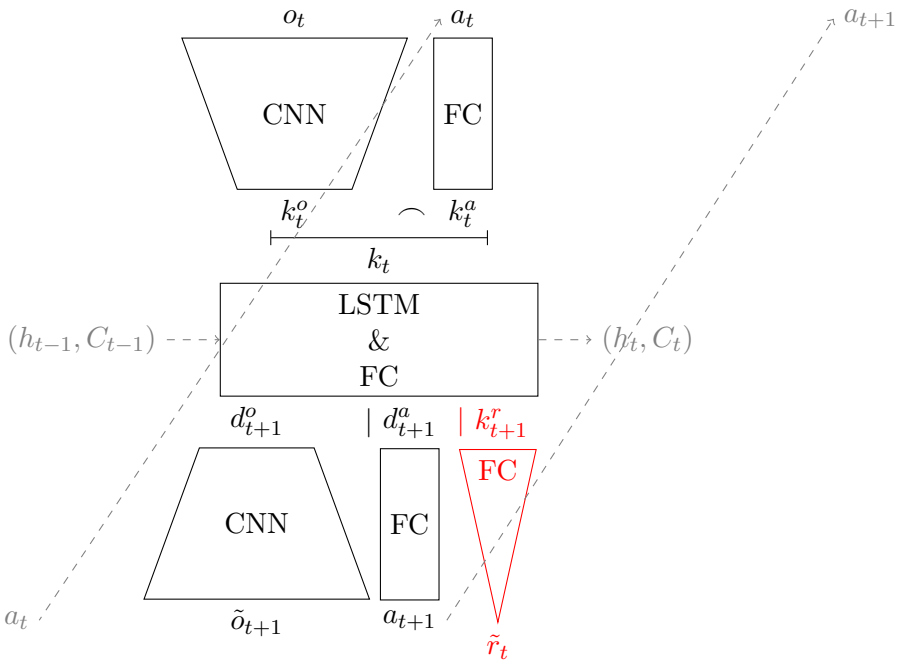


Figure 3.5.: Recurrent model trained end-to-end. Actions are incorporated as observations and generated as self-fulfilling predictions. Rewards are predicted just like observations.

Summing up, in this section we have discussed a Closed-Loop Deep Reinforced Controller and Model, that can be trained end-to-end both as a model and as a reinforced controller, thanks to a recurrent credit-assignment path. There is however a major roadblock in the credit-assignment path, which is one of the focus points of the experiments developed in the next chapter.

Part II.

Experiments

4. Methodology

This chapter contains a description of the tools used and the experiments performed. In the first two sections a discussion on DL frameworks and the environment chosen can be found. The last section describes the implementation details and the tested model variations.

4.1. Deep Learning frameworks

For this work, the `pytorch` (pytorch, 2017) Deep Learning framework has been used. Using a DL framework is convenient for mainly two reasons: GPU acceleration and automatic gradient computation.

Hardware acceleration (essentially through GPGPU) is often critical in the DL field. Reducing training times from months to days, in many cases makes many research lines feasable. Computations using CNNs profit specially from GPU acceleration, but large RNNs (with large enough matrix multiplications) also get an advantage vs CPUs. The programming model of CPUs and GPUs differs substantially, so a specific implementation is required in order to run operations on GPUs. Deep Learning frameworks generally include enough operations implemented for GPUs so that any DL function can be easily derived from these basic building blocks. This greatly reduces the programming effort required for running DL experiments.

The second reason (i.e. automatic gradient computation) removes the need for the experimenter to symbolically compute all the derivatives of the loss function. Automatic gradient computation systems record variables, intermediate values, and operations performed when evaluating models. This allows an efficient differentiation (through reverse mode differentiation) of the loss function with respect to the parameters of the model.

The framework `pytorch` was chosen for ease of debugging and enabling a straightforward implementation in `Python`. The framework `TensorFlow` (Abadi et al., 2015) was also considered, but the implementation (as of September 30, 2017) is more verbose and thus more demanding to quickly iterate on. DL programming in `pytorch` follows an *imperative* approach (i.e. operations are recorded for backpropagation as they are evaluated), versus the *symbolic* approach in `TensorFlow` (i.e. all operations are first defined, and then evaluated as a block) [1].

A broader discussion of all these concepts can be found in (MXNet, 2017) or (Chen et al., 2015).

[1] However, there is ongoing work on both `pytorch` and `TensorFlow` for enabling each other's approach.

4.2. Environment

Figure 4.1.: A capture from the online game `slither.io`

The environment `internet.SlitherIO-v0` from OpenAI Universe (OpenAI, 2017) and Gym (Brockman et al., 2016) has been used for the experiments performed. The main reason for choosing this environment is that it guarantees rich observations under (almost) any policy (i.e. exploration is *easy*). It is also a visually complex `2-D` partially-observable environment in which most pixels change at each time step. Additionally, it is an online game, so it is quite appropriate for testing the capabilities of an algorithm against human players, which might be desired in future work.

The game is playable through mouse or keyboard inputs. In this work, keyboard inputs are used. There are 3 possible actions an actor can do in the game: turn right, turn left, and accelerate. Each of these actions is binary (i.e. either action or no-action) and they can occur in any combination at every time step. Turn right and left respectively make the snake progressively turn to the right or to the left. These two actions can be input at the same time, resulting in a slithering forward movement. When no direction action is chosen, the snake just moves forward in the last direction it was moving. "Accelerate" makes the snake reduce its length and advance faster in whichever direction the other actions command.

The observations from this environment are pre-processed before becoming inputs to the model. The environment generates RGB observations of size 500×300 at up to 60fps. These observations are sampled at a rate of $\sim 10-12$fps. From these observations, only the central section of size 300×300 is processed. This section gets downscaled to 150×150 by averaging every 4 pixels. Finally the RGB image is transformed into one channel by taking the maximum brightness across the RGB channels of each pixel. Taking the maximum instead of the average makes a red pixel and a white pixel get the same final brightness, instead of a third of the brightness for the red pixel. In this way, all snakes roughly are equally bright in the 1-channel image, independently of color. This 1-channel 150×150

image is what the encoder receives at its input.

Some of the discussed models take only observations as inputs, some others take actions too. A model taking actions as inputs can predict observations for any sequence of actions. This is however not possible if actions are not an input, so for those models to work, sequences of actions need to be implicitly predictable.

For the experiments where the action is not an input, the acceleration action is never taken, for simplicity. Around 450 episodes of varying lengths have been collected. About 10% is split as test data. In these episodes, at each time step, the snake has a 1% chance of choosing a new action out of 4 (i.e. go straight, turn right, turn left, slither) with equal chance. While no new action is chosen, the snake just repeats the same action. In this way the model just needs to learn to recognize which of the 4 behaviours the snake is following in order to predict correctly. Of course, there will be an increased error when a random transition occurs.

For the experiments where actions are inputs, 3 random values are uniformly sampled from the interval $[0, 1]$ at each time step. The 3 values correspond to the 3 possible actions, and they are the input of the network. In this sense each action is represented by a value in the range $[0, 1]$. However, actions are binary, so if the value for an action is larger than 0.5, the action is performed, and vice versa. 500 episodes of varying lengths have been collected, from which 10% is split as test data.

4.3. Experiments

In this section a detailed description of the experiments performed can be found.

4.3.1. Implementation details

The observation-encoder consists of 6 convolutional layers. The kernel (filter) of each convolutional layer (see subsection 3.1.2) is of size 4×4 for the first 5 layers and size 5×5 for the last layer (this is so because the last convolution is actually a FC layer shaped as convolution). The input has 1 channel, and the subsequent layers have 16, 16, 32, 64, 128 and 512 channels respectively. The strides[2] are $(2, 2)$ for all the layers but the last one, where they are $(1, 1)$ (although this last value is irrelevant because this layer is a fully connected layer). To keep certain sizes for the outputs of the layers, some layers are padded with zeros around their edges. A padding of (n, n) indicates n rows/columns of zero-valued pixels are added on each edge. The paddings used are $(2, 2)$, $(1, 1)$, $(2, 2)$, $(1, 1)$, $(1, 1)$, $(0, 0)$ for each layer.

[2] The convolution of the filter is not always evaluated at every location of the input image, resulting in smaller outputs. The horizontal and vertical distances between adjacent evaluations are called strides (e.g. strides of (1,1) mean evaluation at every location).

The observation-decoder is exactly symmetric with the encoder. The transposed convolutional layers of the decoder are defined by the same hyper-parameters[3] but in reverse order.

The action-encoder has 2 hidden layers of size 128. The code size for the action has arbitrarily been chosen to be 16, which should be more than enough to represent all combinations of 3 possible actions.

All activation functions are `SELU`, except the output of the observation-decoder (which is a *sigmoid* function), and the activations inside the LSTM cell (which are either *sigmoid* or tanh). The *sigmoid* at the output of the decoder is chosen so that the values are constrained inside the possible range for brightness values $[0, 1]$.

The hidden size of the LSTM is 512, and the input and output sizes depend on which codes are concatenated.

The random distribution from which the initial parameters are sampled follows (Klambauer et al., 2017) and (Glorot and Bengio, 2010).

The loss function computed on the predicted observations measures Binary Cross Entropy (BCE) and is averaged over *all* values. This loss is used instead of an L_2 loss because it better reflects the fact that brightness never takes values outside the $[0, 1]$ interval.

Stochastic Gradient Descent (see subsection 3.1.4) with a momentum β of 0.9 and a learning rate α of 0.1 is used for optimization. The learning rate is set by starting with a high value and progressively decreasing it until the training process does not diverge. The non-divergence is measured by computing, for each independently initialized parameter group, the ratio between the gradient norm and the parameter norm. If these ratios take values $\gg 1$, the training is considered to be diverging.

Training is done with a batch size of 1 and with the LSTM unrolled for 32 time steps. Inside each episode, the hidden state at the end of every 32 time steps is the initial state for the next 32. The hidden state for each new episode is set to 0.

4.3.2. Tested models

The main model under focus is the one in Figure 3.2. Some variations of this model are to be experimentally tested. The experiments target two goals. One is testing some small variations of the main model and see how they affect performance. The second goal is finding a design where credit assignment can happen even if temporal dependencies in the model do not occur.

As a first step the main model is contrasted with two baselines. The first baseline, in Figure 4.2, is a non-recurrent model. This might perform relatively well on our data because the sequences of actions used are simple, but should not perform well in a general case Additionally, it lacks a proper mechanism for incorporating actions.

[3] The term "hyper-parameters" reflects the fact that the term "parameters" is typically reserved for referring to the weights of the model.

The second baseline, in Figure 4.3, is a simple recurrent model where an auto-encoder generates codes independently of predictions, and a recurrent network learns to generate code predictions. The recurrent network is trained with a Mean Squared Error (MSE) loss, averaged for all dimensions.

It should be noted that this kind of training is not optimal because of two reasons. The first reason is that the low dimensional representation learned by the auto-encoder might not be optimal for the LSTM to learn from. The second reason is that it gives equal weight to all the components of the code, which is in general not correct. In contrast, when we compute losses on pixels with the BCE loss, we also implicitly assign equal importance to each pixel, which is a reasonable assumption in this case. If these assumptions about non-optimality were proven wrong by experiments, it could be considered to close the loop by feeding the predicted codes in an additive or subtractive way to the input code.

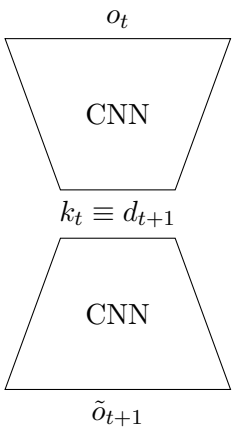


Figure 4.2.: Feed-forward baseline. Input encoder above, output decoder below. Referred to as the "baseline feed-forward" model in the Results chapter.

A first variation of the main model can be found in Figure 4.4. This variation, instead of having the recurrent network compute a predicted code, only computes additions to the input code. This concept is also used in Residual Networks (He et al., 2015b), and inside the LSTM cell itself. One clear advantage is that the encoder can be trained with errors flowing in parallel to the LSTM cell (i.e. gradients from the decoder do not need to backpropagate through the LSTM cells in order to generate gradients in the encoder). This potentially speeds up learning.

A side effect is that the feature spaces of the input and output of the LSTM&FC block are now forcefully shared. However, that does not necessarily mean that $\boldsymbol{d}^{t+1} \simeq \boldsymbol{k}^{t+1}$, since it might be possible to have two different codes representing the same observation: one encoded and one to be decoded.

Assuming $\boldsymbol{d}^{t+1} \simeq \boldsymbol{k}^{t+1}$ is true, an interesting experiment is feeding the LSTM network

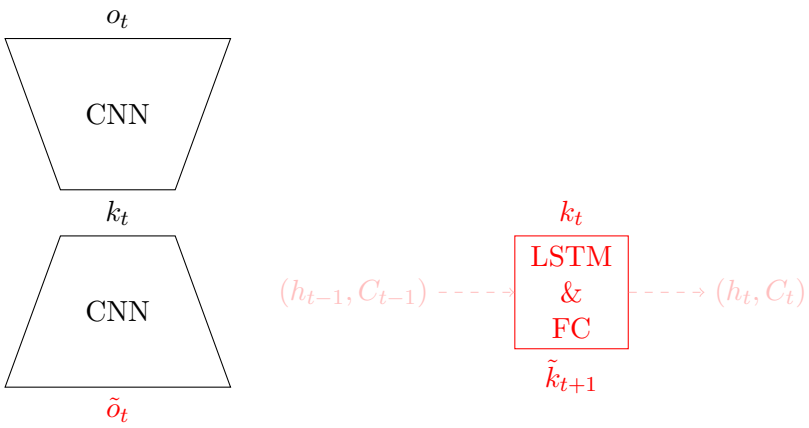


Figure 4.3.: Recurrent model baseline. The auto-encoder and LSTM cell are trained independently. Referred to as the "baseline recurrent" model in the Results chapter.

with the errors in the code prediction instead of with the codes themselves. This is something that the input gates of the LSTM cells can learn to compute themselves through the recurrent connection, but giving these values directly might make learning easier (since it is optimal information). You can see a representation in Figure 4.5. However, the meaning of this subtraction is only clear if the feature spaces are completely shared and $\boldsymbol{d}^{t+1} \simeq \boldsymbol{k}^{t+1}$ is true. If this is the case is something to be analyzed from the experimental results.

Until now we have only considered sharing feature spaces in the code representation, but there is a trivial alternative for sharing feature spaces. Namely, the inputs and outputs of the model are always shared since this is what is being optimized in the first place. A simple way of having the outputs of the model contribute to the input of the models is through addition or subtraction. This can be considered *closing the loop*. These two options are represented in Figure 4.6 and Figure 4.7.

Although, from a gradient propagation perspective, these two variants are equivalent, it is not the case from an informational perspective.

In the additive case (equivalent to averaging output and input), the outputs just add noise to the inputs, reducing the information content. On the other hand, the subtraction provides the inputs with optimal information; i.e. only what could not be predicted becomes the input.

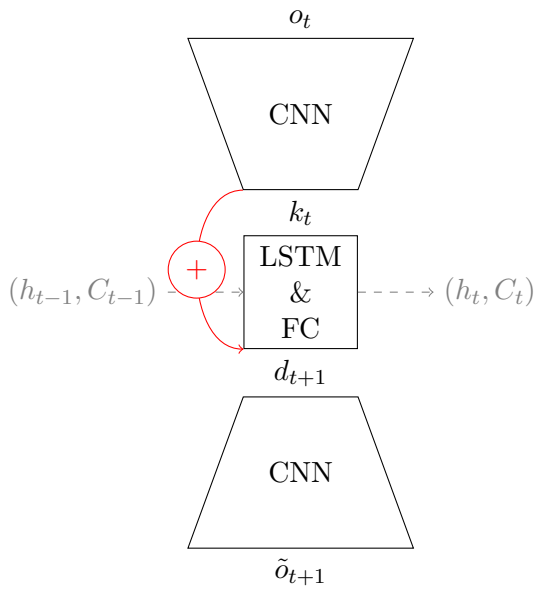


Figure 4.4.: Recurrent model trained end-to-end. The recurrent cell only computes updates to the code, alike residual networks. Referred to as the "residual" model in the Results chapter.

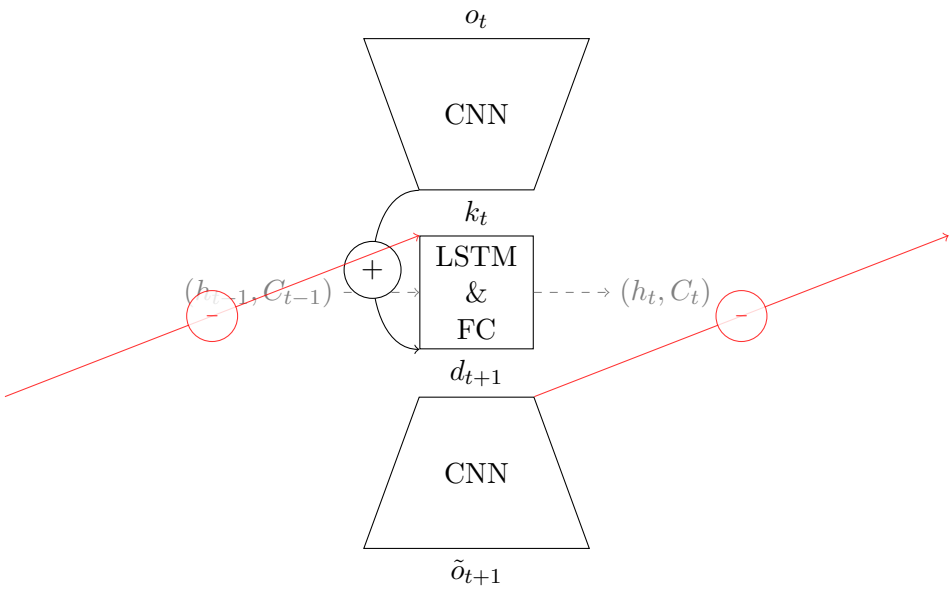


Figure 4.5.: Recurrent model trained end-to-end. The recurrent cell only computes updates to the code and only receives the errors from the previous prediction. Referred to as the "residual, error input" model in the Results chapter.

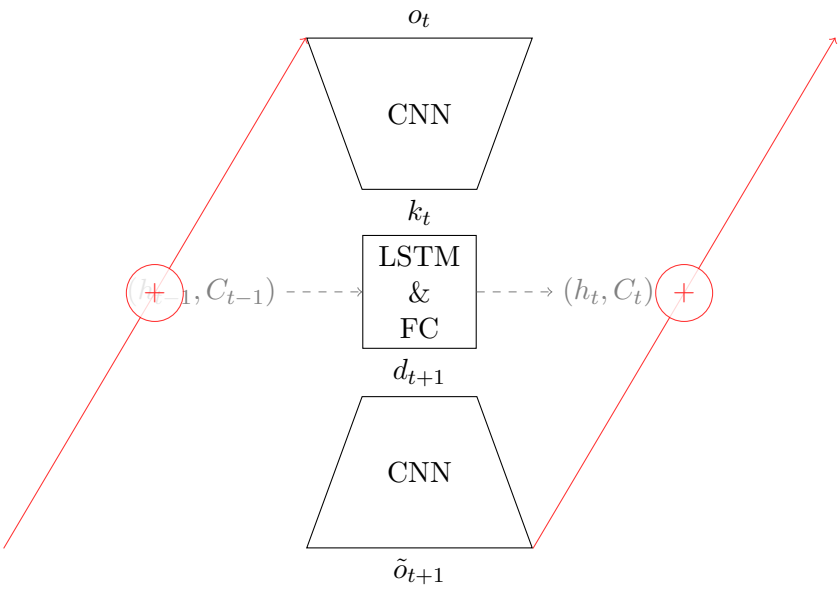


Figure 4.6.: Recurrent model trained end-to-end. The predicted observation is added to the observation. Referred to as the "closed loop +" in the Results chapter.

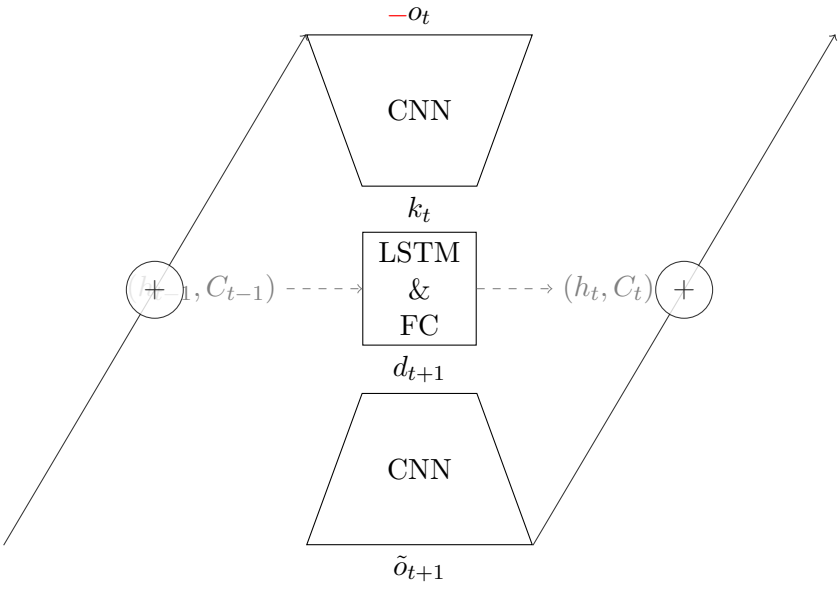


Figure 4.7.: Recurrent model trained end-to-end. The observation is substracted from the predicted observation. Perfect predictions result in zeroed inputs. Referred to as the "closed loop −" model in the Results chapter.

Part III.

Results and Conclusion

5. Results

In this chapter the results of the performed experiments are discussed.

All models have been trained for roughly the same number of training steps. The training of each model has taken about 2-3 days. The goal of the experiments was seeing which model variations learn faster and which have properties that allow full credit assignment under correct predictions. Because of that, the focus for *goodness* of the models is not on their final performance (which has not been optimized for), but on the decay rate of the loss and the running average of performance during training.

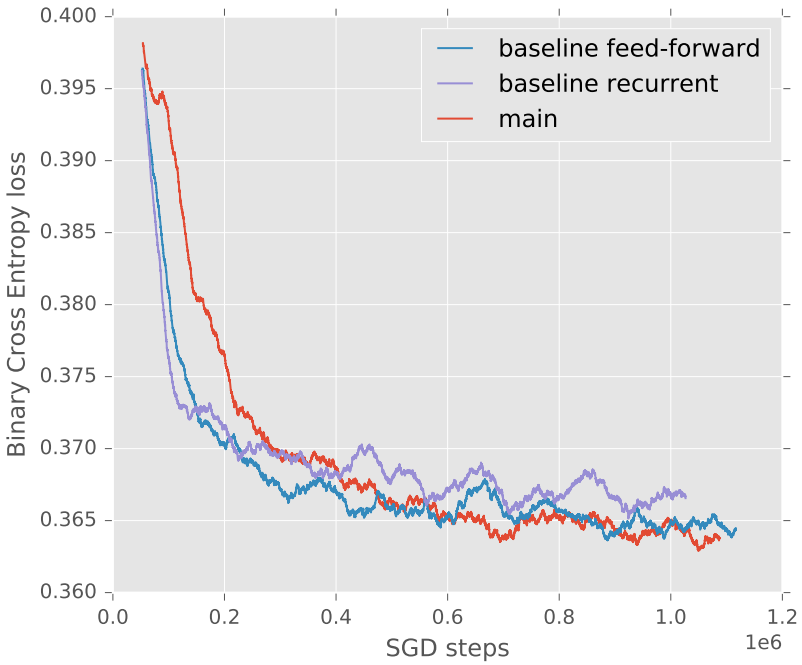


Figure 5.1.: Training losses of the "main" model and the two baselines. "main", "baseline feed-forward" and "baseline recurrent" refer to Figure 3.2, Figure 4.2, and Figure 4.3. Each plotted data point represents an average of the previous 4096 samples.

In Figure 5.1 I compare the training results of the "main" model with the two baselines in consideration. The "main" and "baseline feed-forward" models perform better than the "baseline recurrent" model, although not by a large margin.

The fact that the feed-forward baseline performs as well as the main model reflects that the training data is mostly predictable by a network without recurrence. Further analysis should be done studying if the recurrent model has an advantage in particular game situations where enemies go off screen and then back on screen.

The fact that the recurrent baseline performs worse than the main model reinforces the (previously discussed) idea that either the loss function is not appropriate, or the features extracted by an auto-encoder are sub-optimal for the performance of the LSTM network. Training the system end-to-end brings a performance advantage, as should be expected. Thus, closing the loop in the recurrent baseline model is sub-optimal and is, in principle, discarded.

It is also worth noting that, at the beginning of training, the loss decreases much faster for both baselines. This is expected since in both cases the errors backpropagate directly from the decoder to the encoder, allowing for a faster training of the encoder.

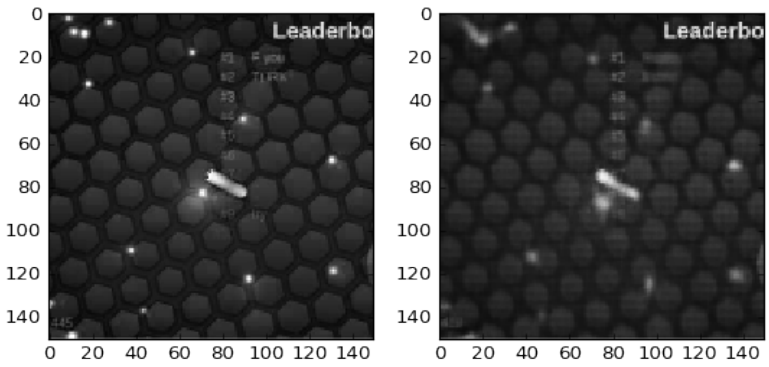


Figure 5.2.: Comparison of an input frame (left) and a prediction of the next frame (right) for the feed-forward baseline.

In Figure 5.2 the quality of a predicted frame is contrasted with the quality of an input frame. This corresponds to the feed-forward baseline. Since the achieved performance of

all runs is similar, further quality comparisons for other variants are omitted.

Figure 5.3.: Training losses of the "main" model and the two models using a residual network. "main", "residual" and "residual, error input" refer to Figure 3.2, Figure 4.4 and Figure 4.5. Each plotted data point represents an average of the previous 4096 samples.

This idea is reinforced by the next comparison, shown in Figure 5.3. In this figure a comparison of the "main" model with the two *residual* variants can be seen. The *residual* structure brings a significant performance gain at the beginning of the training, just like for the two baselines. In this case the gradient of the decoder can also backpropagate directly into the encoder. The residual models seem to have a small performance advantage, although these small differences might be reduced by a longer training time.

Having seen the good performance of the feed-forward baseline in Figure 5.1, it can be argued that these residual models are just using the feed-forward components of their models for achieving its performance. This is supported by Figure 5.5, where a qualitatively similar training pattern can be appreciated for these 3 runs.

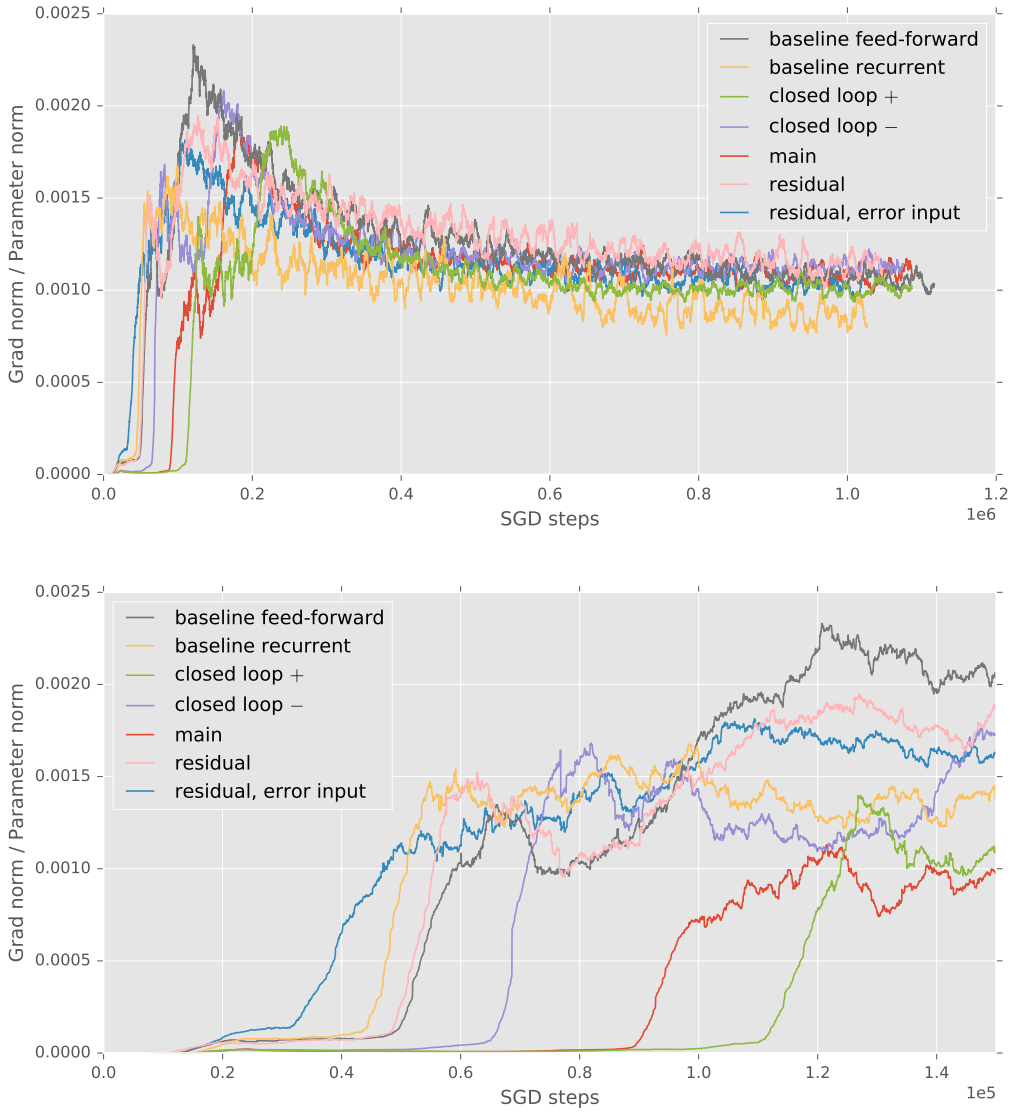


Figure 5.4.: L_2 norm ratio: gradient norm divided by parameter norm. This corresponds to the data for the convolutional kernel of the fourth layer of the encoder (parameters of size $64 \times 32 \times 4 \times 4$). Each plotted point represents a running average of 256 data points. Top: all data points. Bottom: first $\sim 10\%$.

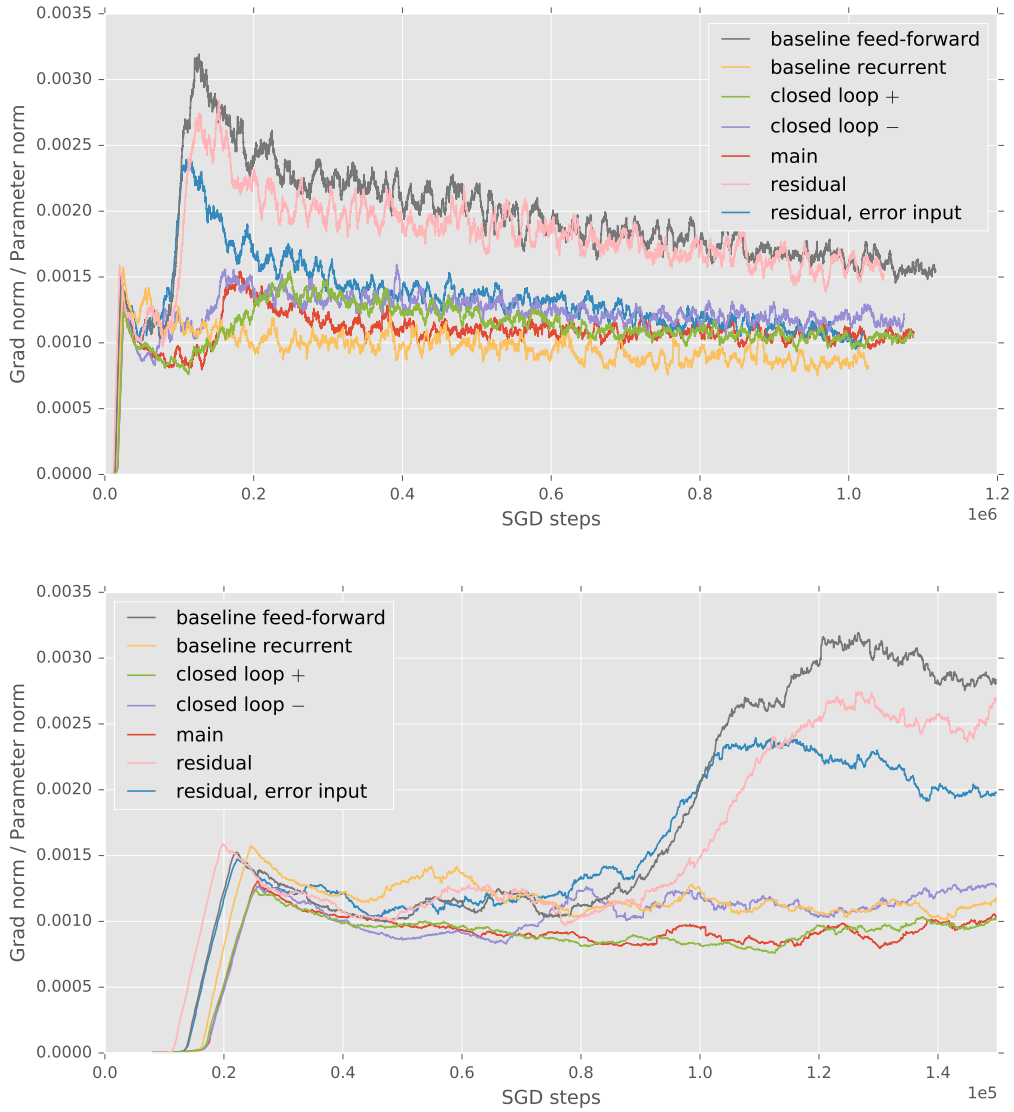


Figure 5.5.: L_2 norm ratio: gradient norm divided by parameter norm. This corresponds to the data for the convolutional kernel of the third layer of the decoder (parameters of size $64 \times 32 \times 4 \times 4$). Each plotted point represents a running average of 256 data points. Top: all data points. Bottom: first $\sim 10\%$.

Figure 5.4 and Figure 5.5 represent the ratios of gradient norm and parameter norm over the training process for a certain group of parameters. Groups of parameters present qualitatively similar behaviours inside the same sections of the model. All parameter groups

of the decoder behave similarly to the ratios in Figure 5.5. All parameter groups of the encoder and LSTM cell behave similarly to Figure 5.4. These two figures have been chosen as representative of all. A couple of interesting observations can be made about these two figures.

It can be seen that the gradient sizes of the decoder (Figure 5.5) grow simultaneously at the beginning of the training and only later diverge. This behaviour is reasonable since all runs receive qualitatively similar backpropagated errors (directly derived from the BCE loss) at the beginning of training.

On the other hand, there is a wider spread for the take-off point of the encoder gradient sizes (Figure 5.4). In this figure it can also be noted that both baselines and residual models are the first growing, also supporting the idea that the residual models might just be exploiting the feed-forward connections for performance.

These ideas about the behaviour of the learning process could be interpreted in terms of information compression, as presented in (Shwartz-Ziv and Tishby, 2017), which talks about two optimization phases that we can also observe in these figures, but this interpretation remains a task for future work.

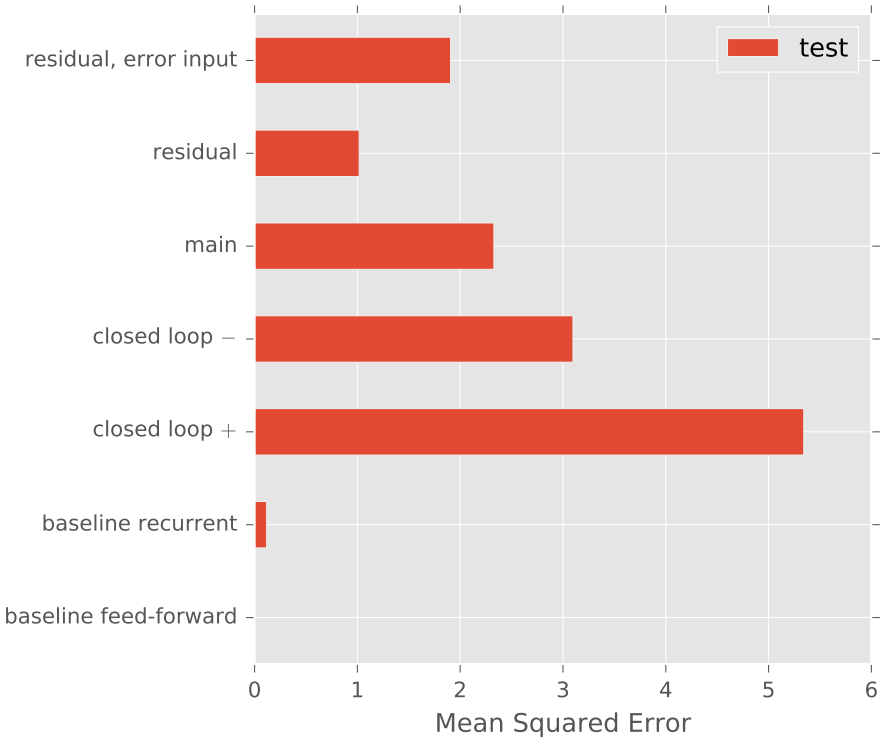


Figure 5.6.: Measure of the difference between the predicted output code at time step t and the true input code at time step $t + 1$.

Figure 5.6 represents the average difference between predicted codes at a certain time step and the next input code. The difference is measured through a Mean Squared Error. This figure shows that the assumption that $\boldsymbol{d}^{t+1} \simeq \boldsymbol{k}^{t+1}$, made when designing the experiments, does not hold (the error is significantly larger than 0 in both residual models). This means that different representations are learned for encoding and decoding. This makes these models not be appropriate as a solution for the credit assignment problem.

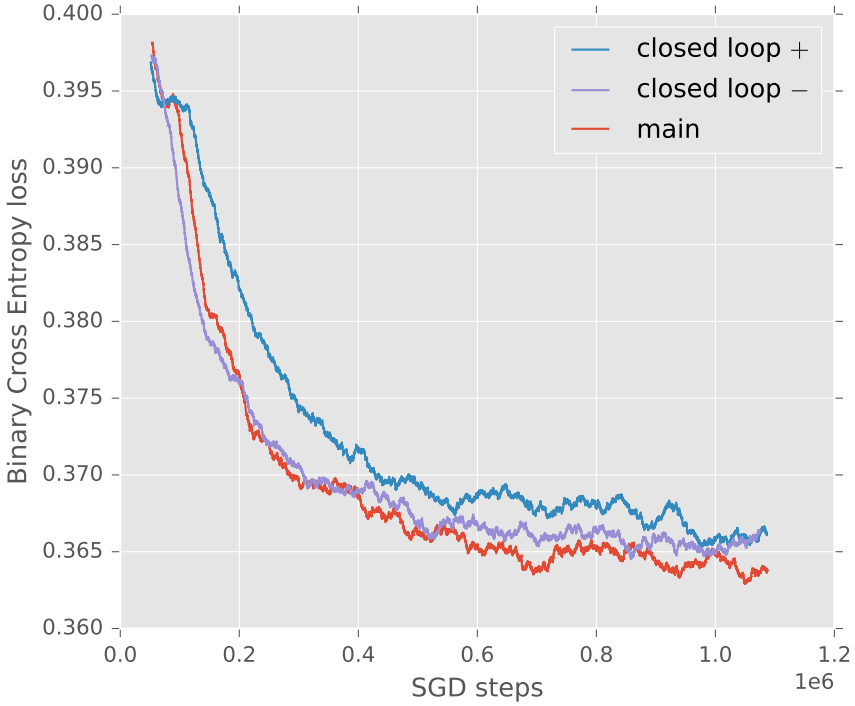


Figure 5.7.: Training losses of the "main" model and the two variants with a closed loop through the observation space. "main", "closed loop +" and "closed loop −" refer to Figure 3.2, Figure 4.6 and Figure 4.7. Each plotted data point represents an average of the previous 4096 samples.

In Figure 5.7 the training of the "main" model and the two closed loop variants is represented. As can be seen there, the performance difference between the subtractive closed-loop and the main model is small. Unsurprisingly, the additive closed-loop performs significantly worse. This could be expected since the additive closed-loop produces noisy inputs, contrasting with the optimal-information inputs from the subtractive model. Note that for this subtractive model to work, the recurrent connections are critical. This was not realized during the design of the experiments, but thanks to the subtractive closed-loop for obser-

vations, backpropagating through the inputs is actually not needed for credit assignment, as long as the model is predicting correctly. This is because correct predictions make the input to the model effectively zero (i.e. the only information it contains is that the recurrent connection contains *all* the information about the next observation).

This forces the subtractive closed-loop system to use the recurrent values for modeling, ensuring a consistent credit assignment path through the recurrent connections. Thus, including this subtractive connection for observations on the model of Figure 3.5, solves one of the open questions of previous chapters: how to guarantee full credit assignment under correct predictions. It seems that this can be done without losing performance. The final closed-loop model can be seen in Figure 6.1 and will be analyzed in the conclusions.

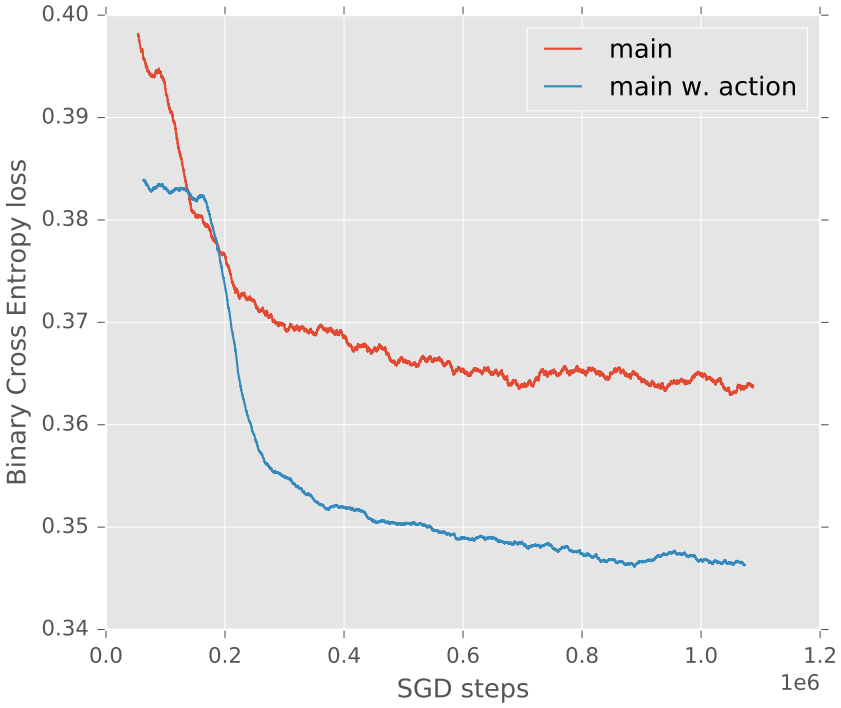


Figure 5.8.: Training losses of the "main" model and the "main" model incorporating actions. "main" refers to Figure 3.2 and "main w. action" to Figure 3.3. Each plotted data point represents an average of the previous 4096 samples.

In Figure 5.8 a rather inconclusive representation can be appreciated. It is a comparison of the training loss of the main model and the analogous variant incorporating encoded actions. The losses represented are hardly comparable since they correspond to different data sets, sampled at slightly different rates. However, the fact that the action reduces the

error, would fit the expectations, since the data set with only observations has unpredictable *agent behaviour* transitions, necessarily causing higher errors.

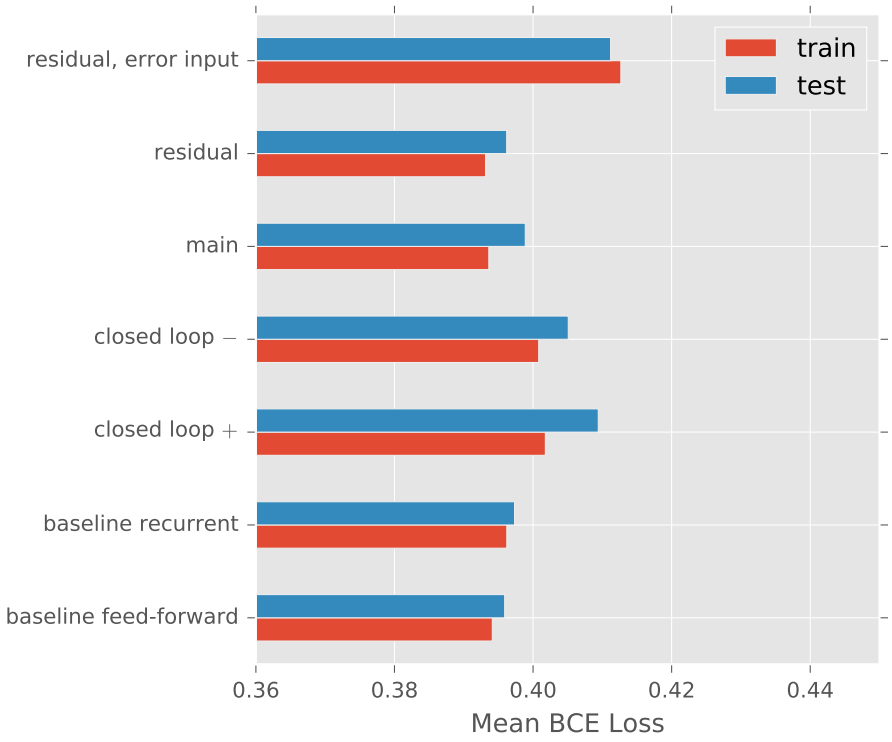


Figure 5.9.: Train and test BCE loss.

In Figure 5.9 a comparison of the losses of all the runs, on both the train and test splits, can be found. As should be expected, test losses are generally a bit higher than train losses. The values of "residual, error input" are comparatively high, contrasting with what might be expected after seeing Figure 5.3. This might just be because the training process was halted randomly, not at an optimal point, so final performance cannot be a focus of this analysis.

6. Conclusion

In this work, I have re-validated a recurrent model structure used in previous work (Oh et al., 2015), (Chiappa et al., 2017), but on a different environment. I have also argued for an specific variant for action-conditioning the model via an action-encoder whose output is concatenated to codes of other observations. This variant performs well in my experiments, although further work is needed on this topic.

Borrowing from (Schmidhuber, 2015), a specific approach to generating actions is proposed, detailing it in accordance to encoding and decoding symmetry constraints. In order to improve credit assignment, in this work it is proposed to allow backpropagation through the generated (and input) action. This closed loop should in principle cause the action-decoder and controller to learn to minimize prediction errors. This should be tested in future work.

The results of the experiments in Figure 5.7 are used to propose a way of fully guaranteeing credit assignment in a correctly predicting model. The final design to be tested in future work can be seen in Figure 6.1.

In model-free approaches, the functions learned need to learn an expectation over future rewards. This model-based approach leaves the task of learning expectations to the model, only requiring that a model of the rewards is also learned. The full credit assignment paths of the final design potentially allow directly reinforcing the Controller and Model through the reward-predicting network. Validating this should also be the focus of future work.

The final proposed model has the following properties: It uses recurrent connections for conditioning predictions on all previous observations and actions. It generates (and learns from) predictions for both observations and rewards. Observations and actions are each encoded by an specific function and the codes are concatenated into an observation-action-code. Only the difference between predicted observations and the true observations becomes an input to the model. This forces the model to use the recurrent connections to carry information across time-steps, which guarantees credit assignment as far backwards in time as the model can be perfectly accurate. Actions are generated via a Neural Network that is symmetric to the action-encoder. This network can be considered as a controller and as an action-decoder. Prediction errors can backpropagate through time through the action encoder and decoder, which should have the effect of biasing the action generation towards actions that minimize prediction errors. Model and controller can be reinforced through gradient ascent via backpropagation through time. Backpropagation through time occurs both through the generated actions and the recurrent connections. This should reinforce positively both sequences of actions and observation expectations, always biasing the model

towards acting and predicting in a way that increases rewards. All this should be tested in future work.

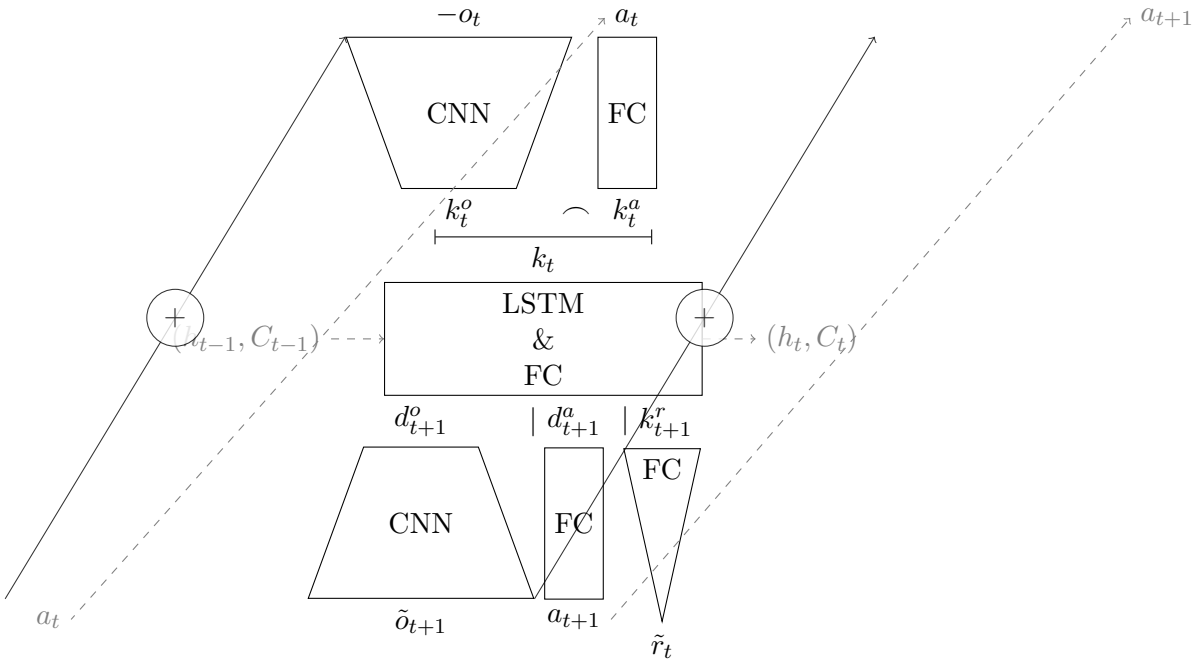


Figure 6.1.: Final proposal of a Closed-Loop Reinforced Controller and Model

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/.

B. Bakker. Reinforcement learning with long short-term memory. In *Advances in neural information processing systems*, 2002.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *CoRR*, 2016. URL http://arxiv.org/abs/1606.01540.

T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv e-prints*, Dec. 2015. URL https://arxiv.org/abs/1512.01274.

S. Chiappa, S. Racanière, D. Wierstra, and S. Mohamed. Recurrent environment simulators. *CoRR*, 2017. URL http://arxiv.org/abs/1704.02254.

C. Finn and S. Levine. Deep visual foresight for planning robot motion. *CoRR*, 2016. URL http://arxiv.org/abs/1610.00696.

C. Finn, I. J. Goodfellow, and S. Levine. Unsupervised learning for physical interaction through video prediction. *CoRR*, 2016. URL http://arxiv.org/abs/1605.07157.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010.

G. Goh. Why momentum really works. *Distill*, 2017. URL http://distill.pub/2017/momentum.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL http://www.deeplearningbook.org.

A. Graves. Generating sequences with recurrent neural networks. *CoRR*, 2013. URL http://arxiv.org/abs/1308.0850.

K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. LSTM: A search space odyssey. *CoRR*, 2015. URL http://arxiv.org/abs/1503.04069.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, 2015a. URL http://arxiv.org/abs/1502.01852.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, 2015b. URL http://arxiv.org/abs/1512.03385.

N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver. Memory-based control with recurrent neural networks. *CoRR*, 2015. URL http://arxiv.org/abs/1512.04455.

N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver. Emergence of Locomotion Behaviours in Rich Environments. *ArXiv e-prints*, July 2017. URL https://arxiv.org/abs/1707.02286.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 1997.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 1991.

S. M. Kakade. A natural policy gradient. In *Advances in neural information processing systems*, 2002.

N. Kalchbrenner, A. van den Oord, K. Simonyan, I. Danihelka, O. Vinyals, A. Graves, and K. Kavukcuoglu. Video pixel networks. *CoRR*, 2016. URL http://arxiv.org/abs/1610.00527.

A. Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. URL http://karpathy.github.io/2015/05/21/rnn-effectiveness/. Date accessed 2017-08-21.

A. Karpathy, J. Johnson, and F. Li. Visualizing and understanding recurrent networks. *CoRR*, 2015. URL http://arxiv.org/abs/1506.02078.

G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. *CoRR*, 2017. URL http://arxiv.org/abs/1706.02515.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 2015. URL http://arxiv.org/abs/1509.02971.

J. R. Medel and A. E. Savakis. Anomaly detection in video using predictive convolutional long short-term memory networks. *CoRR*, 2016. URL http://arxiv.org/abs/1612.00390.

S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. C. Courville, and Y. Bengio. Samplernn: An unconditional end-to-end neural audio generation model. *CoRR*, 2016. URL http://arxiv.org/abs/1612.07837.

M. Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 1961.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, 2016. URL http://arxiv.org/abs/1602.01783.

MXNet. Mxnet architecture - deep learning system design concepts, 2017. URL https://mxnet.apache.org/architecture/index.html. Date accessed 2017-09-03.

N. Neverova, P. Luc, C. Couprie, J. J. Verbeek, and Y. LeCun. Predicting deeper into the future of semantic segmentation. *CoRR*, 2017. URL http://arxiv.org/abs/1703.07684.

A. Odena, V. Dumoulin, and C. Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016. URL http://distill.pub/2016/deconv-checkerboard.

J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. P. Singh. Action-conditional video prediction using deep networks in atari games. *CoRR*, 2015. URL http://arxiv.org/abs/1507.08750.

C. Olah. Understanding lstm networks, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Date accessed 2017-08-29.

OpenAI. Universe, 2017. URL https://github.com/openai/universe.

V. Patraucean, A. Handa, and R. Cipolla. Spatio-temporal video autoencoder with differentiable memory. *CoRR*, 2015. URL http://arxiv.org/abs/1511.06309.

M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. Parameter space noise for exploration. *CoRR*, 2017. URL http://arxiv.org/abs/1706.01905.

pytorch. pytorch. https://github.com/pytorch/pytorch, 2017.

A. Radford, R. Józefowicz, and I. Sutskever. Learning to generate reviews and discovering sentiment. *CoRR*, 2017. URL http://arxiv.org/abs/1704.01444.

M. Riedmiller. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. In *ECML*. Springer, 2005.

J. Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, 2014. URL http://arxiv.org/abs/1404.7828.

J. Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *CoRR*, 2015. URL http://arxiv.org/abs/1511.09249.

J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, 2015. URL http://arxiv.org/abs/1502.05477.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *ArXiv e-prints*, July 2017. URL https://arxiv.org/abs/1707.06347.

M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 1997.

R. Shwartz-Ziv and N. Tishby. Opening the black box of deep neural networks via information. *CoRR*, 2017. URL http://arxiv.org/abs/1703.00810.

R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*. MIT Press Cambridge, 1998. URL http://incompleteideas.net/sutton/book/the-book-1st.html.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press Cambridge, 2017-draft. URL http://incompleteideas.net/sutton/book/the-book-2nd.html.