Development and Evaluation of a Visual Attention Model with Python and Tensorflow

Oleg Yarin

A thesis presented for the degree of
Bachelor of Science

First supervisor: Prof. Dr. Christian Herta
Second supervisor: Dr. Vera Hollink

Applied computer science
HTW Berlin
Germany
31-05-2017

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural network approaches have received much attention in the last several years. It's becoming a popular choice for performing various tasks like speech and image recognition, object detections etc. as these methods have dramatically increased accuracy compared to traditional machine learning approaches. However, achieving high accuracy on recognition tasks is still computationally expensive and needs improvements in performance. This study will be a close resemblance of the recurrent neural network of visual attention which is able to extract necessary information from an image by looking at it in low resolution, and then adaptively select parts that are most relevant for a task [4].

The idea of visual attention was inspired by how human perception works. Humans do not perceive a visual scene as a whole but focus on parts of the scene that gives the most useful information to them. Humans are also capable of combining information from different parts of a picture. They then connect it to build a subjective knowledge of the picture (or sequence of pictures) [6]. Taking into account these properties, researchers from google Deepmind build a model which can be described as follows:

> Instead of processing an entire image or even bounding box at once, at each step, the model selects the next location to attend to; based on past information and the demands of the task.... The model is a recurrent neural network (RNN) which processes inputs sequentially, attending to different locations within the images (or video frames) one at a time, and incrementally combines information from these fixations to build up a dynamic internal representation of the scene or environment.[4]

One of the main advantage of this model, is that the computation required is controlled and is independent of the input image size. Deepmind's researchers evaluated their model on several image classification and dynamic visual control problems which showed a better performance when compared with convolution neural network[6].

The evidence from this study suggests application of this model on large scale object recognition as well as classification of sequence of images, which will be a great fit since the model's performance is not dependent on the size of an input object.

The main aim of this study is to extend the current knowledge of the work mentioned above and build a model which will be able to classify a set of images and develop appropriate prototype system since it can be useful in a variety of areas. However, the experiments in the current work is limited by low-resolution images and mostly will concentrate on classifying a group of objects as this restriction will reduce complexity of the experiments and therefore reach better results on a task of classifying a group of images.

1.1 Motivation

This approach to classify a group of images has a potential to help with automated detection and classification of breast cancer metastases, which is the main concern of camelyon challenge. Camelyon challenge provides the whole-slide images (WSI) of lymph nodes of different patients. Based on this images, competitors solutions should be able to decide about availability of breast cancer metastases in lymph nodes. [7]
Camelyon challenge is an inspiration for this work since pathologist's efforts along with the assistance of automated detection system will reduce significantly not only the workload of pathologists but the human error rate in diagnosis as well.

This work will be the first step in building software that will be capable of classifying whole-slide images of histological lymph node at the patient level. That is, bringing together estimations from multiple lymph node slides into a single outcome.

Digital pathology is a very attractive field for machine learning researchers since whole-slide images have a very high resolution and are typically about

200000 x 100000 pixels. To give you some sense of data, camelyon challenge provides data for 200 patients, where each patient has 5 different slides. It means that in total they release about 1000 slides and that is 55.88gb of uncompressed data [7].

It is quite clear that using CNN for this task is computationally very expensive. Applying model of visual attention promises to solve the issue of high-resolution pictures at a computational level. Therefore making an extensible piece of software, that will allow further improvements is also one of the main concerns of this work.

Chapter 2

Theory

2.1 Artificial Neural Networks

Why Neural Networks? Before going into what actually artificial neural networks are, let's first try to face the question why do we need it in this paper. The problem that we give to our application to solve can be shortly summarized in the following statement: " Given a group of images, find the patterns in them that are more influential on your belief that an image(or a group of images) belongs to a specific class. "

This problem is known as pattern recognition problem or in our case visual pattern recognition problem [8]. The solve this problem it's required to develop ability for a machine to recognize patterns that will help to make a decision about a class. The obstacles that can appear by solving this problem can be more visible if we will try to write a conventional computer program, i.e. bunch of rules to identify these patterns. What seems to be easy for us, is really hard to describe algorithmically. In these system the computational steps are deterministic hence not flexible enough for the whole variety of input data [1].

Solving problem differently Artificial Neural Networks(and machine learning in general) are looking at the problem in a different way. They don't execute programs instructions, they don't have sequential semantic and normally are not deterministic. They acquire their "knowledge" by extracting patterns from raw data, which normally called training data(which normally is a set of tuple $(input, label)$ This approach also know as concept of statistical pattern recognition. [8] Artificial Neural networks have recently shown an excellent performance and accuracy at recognizing objects compared with other machine learning techniques [9].

What is Neural Network? Artificial Neural Network(ANN), often referred just as Neural Network(NN), in simply words is a computational model, which was inspired by how human/animal brain works. Artificial NN is modeled based on the neuronal structure in the brain's cortex. Though the inspiration was from the brain, it's indeed much much simpler than brain in terms of number of neurons that is used in ANN [10]. To understand how neural networks works it is crucial to understand first the way a *perceptron* work.

Perceptron is a simple type of an artificial neuron. Given several binary inputs, perceptron is meant to produce a single binary output.

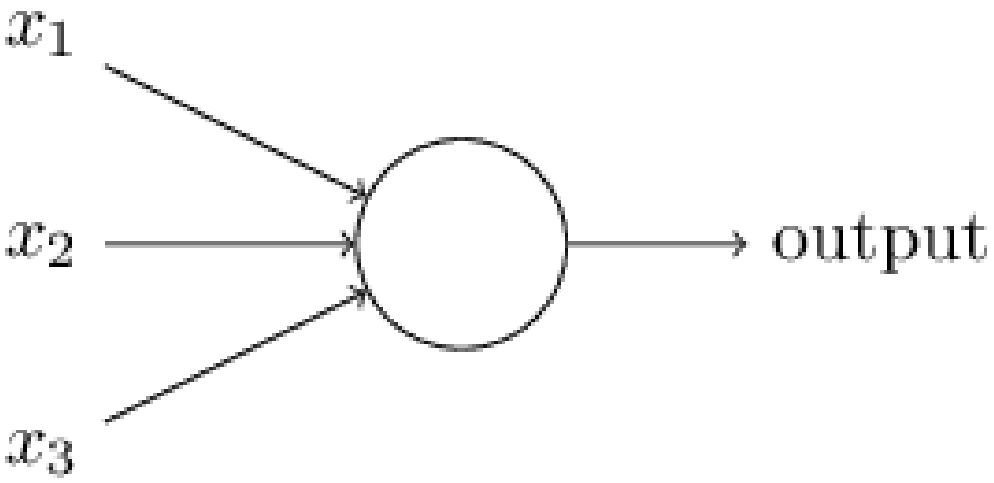

Figure 2.1: A perceptron [1].

In the figure 2.1 the perceptron has three inputs: x_1, x_2, x_3. To produce an output the perceptron posses of *weights*: w_1, w_2, w_3 which represents connection between input and output. Weights determine how important is an input to the output. That said, the perceptron output is determined by whether the weighted sum $\sum_j w_j x_j$ is more or less than some *threshold* value:

$$output = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq threshold \\ 1, & \text{if } \sum_j w_j x_j > threshold \end{cases} \tag{2.1}$$

In shortly, this is a computational model which make a decision by weighting up the evidence(input data) [11].

Of course such a model is not capable of making complicated decisions, but by extending the model to more complex network of perceptrons, we might improve the model.

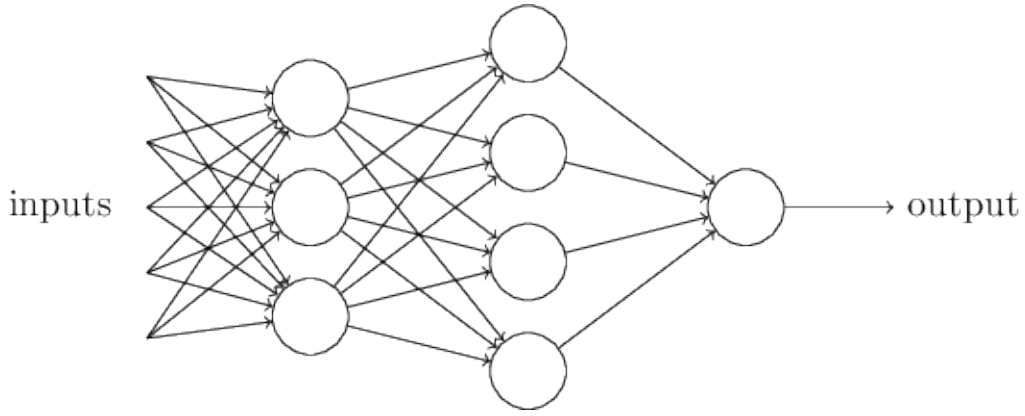

Figure 2.2: Two layer perceptron network[1].

In the network shown on figure 2.2, we can observe two layer of perceptrons. First layer will correspond to the first column of perceptrons and will be responsible for the weighting up input, in contrast to second layer (second column of perceptrons) which determines output by weighting up the result from the first layer. Therefore second layer is located on more abstract level from input data compared and can make more complicated decisions. Further layers might be capable of making even more sophisticated decisions.

2.1.1 The Basics of Neural Networks

Neural Network Now that we know the way perceptrons work, it's fairly easy to understand Neural Network. However we need to change the mathematical notation a bit. For the sake of convenience, let's move *threshold* in equation 2.1 to the left part and replace it with a variable known as the bias : $b = -threshold$. Let's also simplify the sum sign: $\sum_j w_j x_j$ by writing weights and input as vectors and use a dot product to multiply them: $\sum_j w_j x_j = W \cdot x$. Using changes described above we can rewrite the equation 2.1 as following:

$$output = \begin{cases} 0, & \text{if } W \cdot x + b \leq 0 \\ 1, & \text{if } W \cdot x + b > 0 \end{cases} \tag{2.2}$$

Bias is a measure of how influential is a certain neuron on making output 1. Some people also use more biological terms: the bias is a measure of how easy it is to get an neuron to fire. To devise the neural network next improvement over the perceptron network is that network should not be limited to have an input only binary value, but any value. The same applies on the output. Output being only binary value will limit the ability to make sophisticated decision. Therefore we introduce a function known as *an activation function* before actually outputting a value: $output = g(W \cdot x + b)$ [1] .

Activation function $g(\cdot)$ is known as an activation function. Activation function helps to control the output and non-linearity of the network. Activation function also plays a crucial role in multi layer architecture, where it helps to prevent the values of each layer from blowing up. For example, let's take a look at logistic sigmoid activation function which has following form:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

TODO: write about tahn activation function

The use of the sigmoid activation will make a network to produce output to be interpreted as posterior probabilities. Probability interpretation helps to provide more powerful and more useful results [8]. There are a good variety of activation functions but in this work we mainly will use the sigmoid activation function and *rectified function*. Rectified function is fairly simple. It produces 0 when input is less than 0, and it does not change input value if input is more than 0:

$$R(z) = max(0, z) \tag{2.4}$$

Now that we derived a concept of Neural Network, we can talk more about what is called feedforward neural network and the terms related to it.

Feedforward neural network is a network where the output from one layer is used as input to the next layer. In feedforward NN information is always fed forward in contrast to recurrent neural network(RNN) where information can go in a loop. We will take a closer look at RNNs in section section 2.2.

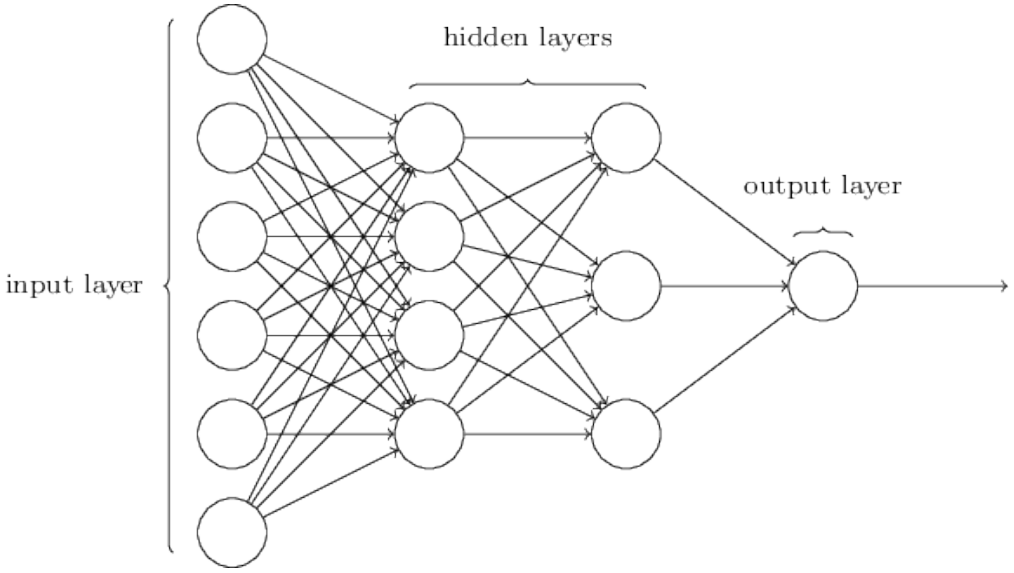

Figure 2.3: The architecture of neural networks[1].

Basically Feedforward neural network is exactly what we described above. Let's name different parts of feedforward NN:

- Input layer - the leftmost layer in the network. The Neurons within input layer, called input neurons.
- Output layer - the rightmost layer in the network. The Neurons within input layer, called output neurons.
- Hidden layers - all the layers excluding input and output layers.

For example, the Neural network in figure 2.3 consist of

- 6 input neurons
- 2 hidden layers
  - first hidden layer consist of 4 neurons
  - second hidden layer consist of 3 neurons
- output layer consist of one single neuron

2.1.2 Training an Artificial Neural Network

As mentioned above Neural Network is capable of solving complicated pattern recognition problems. However designing an neural network is not sufficient for this. It's also requiring to train an network. In this paragraph we will introduce learning procedure. But before going into learning, let's recap how our neural network model looks like:

$$y = g(W \cdot x + b) \tag{2.5}$$

Where g is an activation function, W - weights, b-biases, x - input data. The space of different weights and biases values building a space of solution for cetrain problem. The goal of the training is to find the best parameters for neural network (W, b), that suited our problem.

Training data To solve pattern recognition problem we need to provide a continuous feedback to NN, which NN can use to learn from. This feedback in machine learning called *training data*. Training data consist of the input data samples and appropriated outputs. You can think of training data as of an list of tuples: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(n)}, y^{(n)})$ where x - input, y - output (also known as *ground truth*), n - amount of training examples. Neural network trained on training data should be able to generalise the output on unseen input data. The goal of the learning is to train network on training data and make it to capable of generalizing the output on unseen input data.

Cost In order to teach the model, it's essential to understand what does it mean for an neural network to be good or to be bad. For this purpose it's required first to define a cost function. Cost(also knows as error) is the metric for the NN(or any other function approximation method), which represents how far of the model is from desired outcome. If cost is big , our network does not work well. With the cost function, it's possible to define our training goal more precisely: the smaller our cost, the better our model works, therefore the goal of the learning is to minimize the cost of our model.

Types of cost There are a plenty of ways to define cost of the model. Let's consider the common type of cost function called *mean squared error function*. Mean squared error function has following form:

$$I_{W'} = \frac{1}{2n} \sum_{i=1}^{n} (y_{W'}(x^{(i)}) - y^{(i)})^2 \tag{2.6}$$

- $y_{W'}$ - is our neural network function with the parameters (W', b'),
- $x^{(i)}, y^{(i)}$ - is input and output of a training sample respectively.

Another common type of function to measure the cost of NN known as *cross entropy*. In short, cross entropy gives a way to express how different two distributions are:

$$I_{W'} = -\sum_{i=1}^{n} y_{W'}(x^{(i)}) log(y^{(i)}) \tag{2.7}$$

where:

- $y_{W'}$ - is our neural network function with the parameters (W', b'),
- $x^{(i)}, y^{(i)}$ - is input and output of a training sample respectively.

[1]

Gradient Descent Once we defined our cost function, we need to find a set of parameters W, b which make the cost as small as possible. The most common algorithm used to minimize the cost function called *gradient descent*. Let's explain the algorithm on an example function:

$$f = f(\Theta) \tag{2.8}$$

where $\Theta = \vec{v_1}, \vec{v_2}, ...$ are variables that we want to minimize.
Gradient descent uses partial derivative to iteratively update parameters. Derivative of a function shows how function output will change with respect to very, very small change in input $\Delta\Theta$. For example, partial derivative with respect to variable $\Delta\vec{v_1}$ will tell us, how different the output will be Δf if we change $\vec{v_1}$ on the small amount. This property of derivative is used in gradient descent algorithm. Essentially, the gradient descent performs updates on the variables to be minimized according to partial derivative of the cost function with respect to this variables.
Gradient descent adopt the following procedure. Beginning with an initial guess for value $v = \vec{v_1}, \vec{v_2}, ...$, we update the vector v by moving a small distance in v-space in direction in which our function f raises most rapidly, i.e. in the direction of $-\Delta_\Theta f$. Iterating this process, we can devise the new set of parameters $\Theta^{(new)}$:

$$\vec{v_i^{new}} = \vec{v_i} - \alpha \frac{df(\vec{v_i})}{d\vec{v_i}} \tag{2.9}$$

where α - is a small positive number knows as *learning rate*. Learning rate determines the smoothness of updates and it's very important to choose it appropriately since if learning rate is too small the learning can be too slow, while, if learning rate is too big, algorithm updates can be too big to achieve the minimum (it can overstep the minimum).

Depends on the conditions this will converge to the parameters Θ where the function f is minimized. One important things to notice is that, we can use the gradient descent algorithm only if f in equation 2.8 is differentiable. That means, if we want to use gradient descent algorithm our cost function should be differentiable [8].

Mini-batch Gradient Descent Normally gradient descent algorithm is associated with the update on the loss computed with whole set of training data, while, gradient descent where updates is performed only using loss computed on a small batch of data knows as *mini-batch gradient descent*. Much faster convergence can be achieved in practice using mini-batch gradient descent. [12]

Backpropagation In order to compute gradients in gradient descent algorithm backpropagation algorithm is normally used. Backpropagation is the procedure of computing gradients applying a gradient chain rules and updating the weights accordingly. It performs first an forward update to receive the network's error value. This error value then is back propagated beginning from output layer(neuron) through all neurons till the input in order to associate it with extent of this error(Δ) which an certain neuron is responsible for. Once this extent is calculated, it performs weights update [13].

2.2 Recurrent Neural Network

Why recurrent NN? Motivation for recurrent neural network(RNN) is that in contrast to feedforward neural network, RNNs are capable of having internal memory, i.e. capable of memorizing information, therefore deal better with sequential input data. RNNs are closer to the way human's brain works. We don't start our thinking from scratch, all of us have different background, memories and experience and based on this we're making our own decision and actions.
As already mentioned in chapter 1, in this work the network should be able attend to different locations within an image, i.e. choose a location and process only area with respect to this location. The network then incrementally

combines the information from different location and based on the knowledge(memory) extracted from a location, network chooses a new location to attend. As you might notice since more steps need to be required, we will have sequential data. Hence, RNNs are underlying concept for this work.

What is RNN RNN is a special type of neural network architecture, which can accept a sequence with an arbitrary length without changing weights of a network. RNN are capable of persisting the information by means of recurrences, i.e. including the output of the network into next computational steps and summarizing this information into an object called *state*. [14].
To simplify the understanding you can imagine RNN as an composition of identical feedforward network, one for each step in time, passing the message to a successor. Essentially, it's a computational unit that is repeated over and over again and can be also thought as an for-loop. One neural network in this composition known as *RNN cell*.

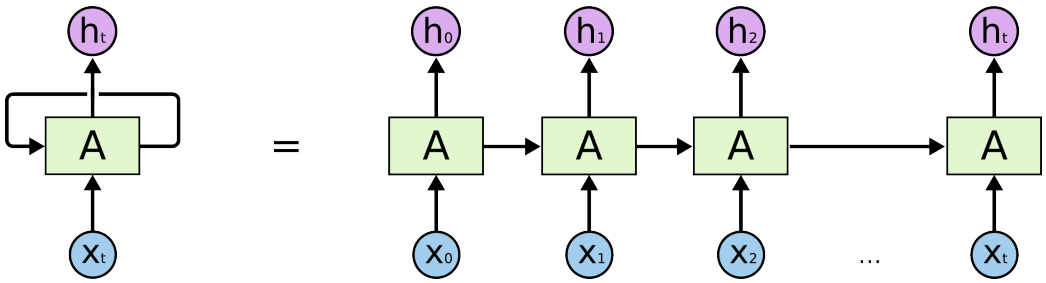

Figure 2.4: Unrolling recurrent neural network(source: [2])

On the right side of the figure 2.4 you can see an unrolled RNN that accepts as input $x_0, x_1, x_2, ..., x_t$ and produces following output: $h_1, h_2, ..., h_t$. One time step represents a layer in terms of forward neural network. The whole concept can be explained in the following equation:

$$\begin{pmatrix} s_t \\ o_t \end{pmatrix} = f \begin{pmatrix} s_{t-1} \\ x_t \end{pmatrix} \tag{2.10}$$

where:

- s_t, s_{t-1} - are states at time step t and $t-1$ respectively,
- o_t - is the output at time step t,
- x_t - is the input at time step t,

- f - is a recurrent function(normally called as RNN cell).

As you might notice, the all calculations responsible for extracting and memorising information performed in f, which provide knowledge about specific *RNN architecture(RNN cell)*. Thus the choice of recurrent function f(RNN cell) is essential for RNNs to work and remember information. There are a lot of variations of RNN cells, but we mostly will consider one of the recent and most widely known architecture called *Long Short-Term Memory (LSTM)*.

2.2.1 Long Short-Term Memory (LSTM)

Long Short Term Memory networks(LSTMs) are a special architecture of RNN cell, capable of learning long-term dependencies. [15] LSTMs have the ability to remember new information and forgetting old, unnecessary information using concept of gates. LSTM cell holds information in the object called *state*(C_t) and only the gates are permitted to manipulate and change this state. Gates are represented as an sigmoid layer and pointwise operation and will be explained below.

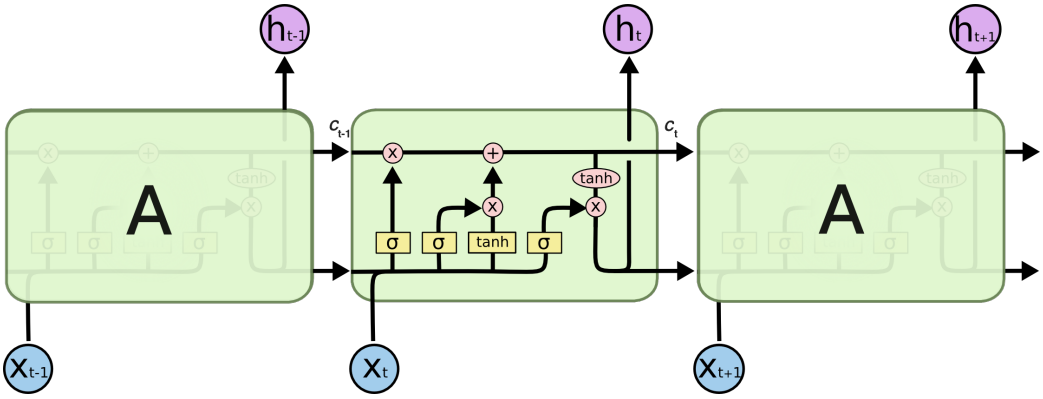

Figure 2.5: Structure of LSTM cell(Source: [2])
An yellow square with σ inside represents a neural network with sigmoid activation function, while yellow square with *tahn* inside represents a neural network with *tahn* activation function.

As you can see from the figure 2.5, LSTM cell has four layers, which build up three gates to interact with the state: *Forget gate*, *Input gate*, *Output gate*.

Forget gate layer The sigmoid layer that you can see on the right side is called "forget gate layer". As you might notice from the name, this gate

is responsible for remembering information(or forgetting). It concatenates output from previous state: h_{t-1} with the input at the timestep t: x_t. Then the result is fed to the neural network with sigmoid activation function which produces an output with values from 0 to 1. Where 0 means to completely forget the information and 1 means to leave the information in the state:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.11}$$

Input gate layer composed of two networks: networks with sigmoid and tahn activation functions. The first sigmoid network decides which values needed to be updated and till what extent, while the network with tahn activation function creates a new candidate state value. Then the outputs from then networks are multiplied with each other to create an update for the LSTM cell's state:

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ C_t^{(candidate)} &= tahn(W_C \cdot [h_{t-1}, x_t] + b_C) \end{aligned} \tag{2.12}$$

Updating the state It's fairly simple now to update the old state C_{t-1} using f_t from equation 2.11 to forget information and using new state candidate values $C_t^{(candidate)}$ and it's extent i_t from equation 2.12:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot C_t^{(candidate)} \tag{2.13}$$

Output gate layer is responsible for the cell's output h_t. This layer is allowed to read information from the state C_t and decides what information should be outputted.

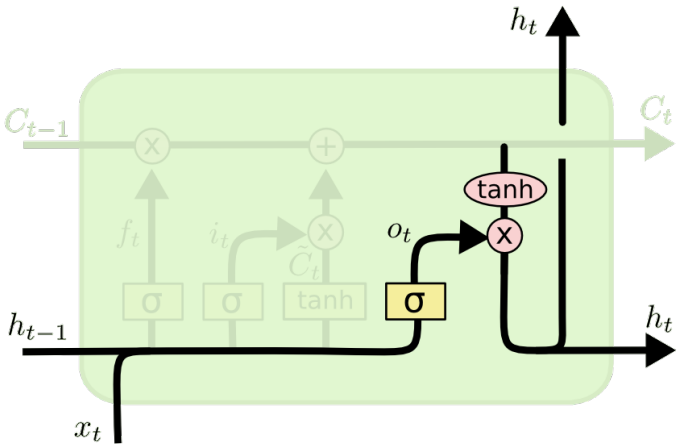

Figure 2.6: LSTM's output gate layer (Source: [2])

As you can see from figure 2.6: firstly, state cell goes through tahn function and then multiplied with output from the neural network of output gate layer:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.14}$$

$$h_t = o_t \cdot tahn(C_t) \tag{2.15}$$

h_t is the output of LSTM cell at time step t as well as the input for RNN cell at time step $t + 1$. The gates stabilize the state and solve the vanishing gradient problem, hence it's very important for an LSTM cell to work.[10]

Backpropagation Through Time Taking into account that RNN nets share all weight between layers(time steps), there is a specific technic for computing gradient when training the network called *Backpropagation Through Time(BPTT)*. It propagates the error all the way back through the time to the time step 0, that's the reason it's called BPTT. [16]

We can think about it as using feedforward neural network's backpropagation but with the constrain that the weights of the layers should be the same. However as RNN might have a hundred of thousands time steps it is common practice to truncate the backpropagation only back to few time steps, instead of backpropagating it to the first time step.

2.3 Reinforcement Learning

Why reinforcement learning? As you might recall from chapter 1, the recurrent visual attention model extracting information from a picture by attending to certain locations of the picture and aggregating the information from these locations. This property will make our network avoid locations with insignificant information, hence ignore locations with clutter. In order to teach the network output next location to attend given previous location, we need to provide training data to the neural network. The problem is here, that we don't know the right answer for this. We can only say whether the network made a right classification decision after the network has already chosen several locations. Consequently, the training of the network parameters will be a very difficult task. As previously mentioned in section 2.1.2, using gradient descent with backpropagation for training NN is possible only with differentiable cost function like mean squared error function or cross entropy function. However, we can't use this functions without the knowing the right answer, therefore defining the cost function would be a complicated task. This sort of tasks is studied by a field of machine learning called *reinforcement learning(RL)*.

What is reinforcement learning? Reinforcement learning concerned with teaching an agent to take actions based on reward signal, even if this signal is delayed. These agents are trained to maximise the total sum of such reward signals. The underlying idea behind RL to represent the nature of learning where agent learning about the the world by interacting with it. By performing this interactions we're observing the changes in the world from which we can learn about the consequences of this interactions, and about what interactions to perform to achieve a goal. Reinforcement learning provides a computational approach to perform goal-directed learning by interacting with environment. The main difference between supervised learning and reinforcement learning is that in RL there is no instructions about the right answer. but a training information or reward signal is used to evaluate the taken actions. That is, instead of providing true label for the system instantaneously, system is receiving a reward signal after each performed action and the goal of RL system is to teach an agent using his own experience to achieve a certain goal.

2.3.1 Components of reinforcement learning

To better understand the main components of RL let's take a look at one of the recent RL systems where the system needed to learn how to play Atari

2600 games.

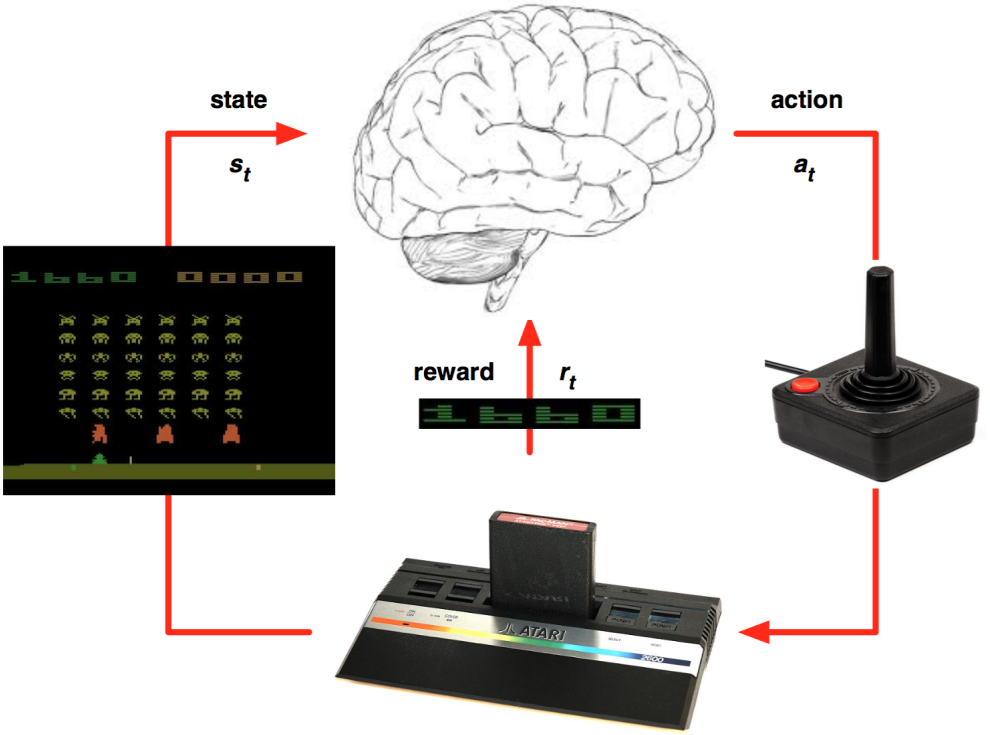

Figure 2.7: RL system to play Atari games (Source: [3])

In the figure 2.7, the brain represents the agent. Agent is our computational system or decision making system. Agent in RL interacts with environment by performing actions. An action would be moving a joystick in the right. By moving a joystick we interact with the environment, which in this case is the true state of the Atari game engine. After the environment receives an action, it gives back to the agent an observation in the form of the video frame shown on the screen and reward signal which reflects the points scored.

State Let's now abstract from our example and describe the flow of RL system more precisely. At each time step t the agent executes an action A_t, receives the observation O_t and receives scalar reward R_t. The environment receives an action A_t, emits observation O_{t+1}, emits scalar reward R_{t+1}. Then t is incremented after the environment's step. In RL instead of working with observations, one works with the *state* or *agent state*. The agent state is the

data the agent uses to pick the next action. State is formally a function of the history(data that agent received so far):

$$H_t = O_1, R_1, A_1, O_2, R_2, A_2, ..., O_t, R_t, A_t \tag{2.16}$$

$$S_t^a = f(H_t) \tag{2.17}$$

where H_t - is the history object at time step t and S_t^a - is the agent's state at time step t

Because history H_t can be very hard to maintain as it grows rapidly over the time, it is very common to talk about *markov state* in RL. Markov state is meant to contain all useful information from the history as well as possess of *markov property*:

$$P[S_{t+1}|S_t^a] = P[S_{t+1}^a|S_1^a, ..., S_t^a] \tag{2.18}$$

where S_t^a - is the agent's state at time step t. It means that the state is only dependent on the present state and not on successors states. Hence, once the state is knows, we can erase the history.

Reward As mentioned before R_t is scalar feedback signal, which indicates how well agent is doing at time step t. The job of an agent to maximise the sum of rewards received after t steps. This sum is also known as cumulative reward.

Additionally to this an agent may possess of following components: a *policy*, a *value function*, and a *model of the environment*.

A Policy is agent's behaviour function. It maps agent's state to actions to be taken by agent, when agent are in those states. Normally, the policy is something that we want to find. Once the best policy is known, we solved RL problem. Policy can be deterministic: $a = \pi(s)$ as well as stochastic: $\pi(a|s) = P[A_t = a|S_t^a = s]$

Value function describes how good is it to be in a particular state. The value of a state is the total amount of reward an agent can expect to to receive when following policy π, starting from that state:

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t^a = s] \tag{2.19}$$

where E_π is the expectation of the cumulative reward from time step t following policy π given the state s, γ - is a discount factor [0, 1]. which will be explained in subsection 2.3.3

While reward determines how well is current action at given time step, value of a state give us more information about long term desirability of a state taking into account the values of all possible states an agent can end up in after leaving this state. It's crucial to understand the difference between reward and value: reward is immediate, while the value giving us insights about the cumulative reward the agent can possibly get from this state on.

Model of the environment represents what the environment will do next. Given the current state s and action a, model defines the probability of an agent to end up in a state s':

$$P^a_{ss'} = P[S_{t+1} = s_0 | S^a_t = s, A_t = a] \tag{2.20}$$

$$R^a_s = E[R_{t+1} | S^a_t = s, A_t = a] \tag{2.21}$$

where P is state transition probability matrix and R is a reward give the probability of next state given current state and action:

2.3.2 Partially Observable Environments

One distinguishes between two type of environments in RL problems. *Fully observable environments* where an agent is capable of directly observing the state of environment: $O_t = S^a_t = S^e_t$ - where S^e_t is environment's state, and partially observable environment. In this work we will concentrate on *partially observable environments.*

Partially observable environments In partially observable environments the agent's state is not equal to environment state, instead the agent is constructing his own representation of environment state from the the external input(observations) that the environment provide. Partially observable environments is a special instance of what is known in RL community as partially observable Markov decision process (POMDP). In our work we are constructing the agent's state by injection the input provided by environment into RNN:

$$S^a_t = \sigma(S^a_{t-1} \cdot W_s + O_t \cdot W_o) \tag{2.22}$$

where S^a_t and S^a_{t-1} are agent state at time step t and $t-1$ respectively, O_t - is external input (in our work that is glimpse), and W_s, W_o - appropriate weights.

2.3.3 Markov Decision Processes(MDP)

The agent is the algorithm that we trying to build, the agent is interacting with environment. *Markov Decision Process(MDP)* describes this environment. Markov Decision Process is an extension of a Markov chain and is one of the core concept that is used in reinforcement learning and almost all problems in RL can be described by using MDP. We have already defined some elements of MDP in section 2.3.1, however MDP's theory gives our a way to solve problems.

Main components of MDP The MDP is defined using following elements:

- *Finite set of state*s $\mathcal{S}$ - a set of Markov states that we described in section 2.3.1
- *Finite set of actions* $\mathcal{A}$ - a set of all possible actions. An action $A_t = a$ $\in$A(S_t), where $\mathcal{A}(S_t)$ - is a set of all possible actions that can be taken in state S_t
- *Reward function* R - is the function which describes the reward based on a state and action: $R_s^a = E[R_{t+1}|S_t = s, A_t = a]$
- *State transition probability matrix* $\mathcal{P}$ give the probability of next state given current state and action: $P_{ss'}^a = P[S_{t+1} = s_0|S_t^a = s, A_t = a]$
- *Discount factor* γ - the discount factor determines the present value of future rewards.

Another important definition that is used in MDP known as return G_t. Return is nothing more than all cumulative reward that an agent can get from time step t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.23}$$

where R_t - is immediate reward at time step t, and γ - is the discount factor.

Policy in MDP is a distribution over actions given a state. It maps a state s to a probability of taking action a, where $a \in \mathcal{A}(S_t = s)$:

$$\pi(a|s) = P[A_t = a|S_t^a = s] \tag{2.24}$$

Policies in MDP are time independent. That is, no matter what time step t it is, we have still the same policy distribution at each time step.

Discount Most rewards in MDP are discounted by discount factor γ. There are several reason for that. Firstly, it is mathematically convenient to use discounts as this will prevent the return value from being exploded to infinity. Secondly, discount give us a way to tune the model to prefer the immediate reward over delayed or vice versa.

Value functions in MDP MDP distinguishes between two types of value functions: *state-value function* and *action-value function*.

State-value function is basically the same value function that we defined in section 2.3.1: it describes the *value* of state S_t if following policy π which is expected return starting from state S_t and then following policy π. We can rewrite the equation 2.19 using our new definition of return G_t:

$$v_\pi(s) = E_\pi[G_t|S_t = s] \tag{2.25}$$

where E_π - is the expected value of the state S_t when following policy π.

Action-value function is defined in very similar way beside the fact that it also takes into consideration the action taken by the agent at time step t. It's denoted $q_\pi(s, a)$ and equal to expected return starting from state s, taking the action a and then following policy π:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] \tag{2.26}$$

Monte Carlo methods This value functions can be estimated using Monte Carlo methods. The idea is to let agent play in the environment by following an

Optimal policy Now that we know how to represent the value of a state, we can concentrate on the problem that we really care about: find a best behaviour for the MDP. That is, the policy that achieves more of reward compared with other policies. MDP literature give us a definition of the what is known as optimal policy:

A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an optimal policy. [17]

There can be more than one optimal policy, nonetheless we denote an optimal policy as π_*. The goal of the any RL task is to find the optimal policy.

2.3.4 Policy-Based Reinforcement Learning

Policy-Based Reinforcement Learning is a field in RL where agent works directly with the policy and does not necessary represents a value function or a model, but still may compute action-value or state-value functions. We'll use policy-based approaches in this work as those methods have better convergent properties more efficient for high-dimensional action space. [4]

Firstly, We need to find a way to represent the policy. With the near continuous action space to store all states will require a lot of memory. Therefore for problems with large state space is recommended using function approximation methods.

Using function approximators such as neural network will allow to estimate policy function:

$$\pi_\theta = P[a|s, \theta] \tag{2.27}$$

where $P[a|s, \theta]$ - is the probability of taking action a given state s.

π_θ is the policy with parameters θ. We can represent our policy with the help of different function approximators. The most common function approximator used in RL is neural network that was introduced in section 2.1. So θ can represent parameters of neural network, although any other function approximator can be used here such as decision tree or nearest neighbour approximator.

Our goal is to find the optimal policy with the parameters θ directly from the agent's experience.

Note: The representation in 2.27 will even allow to generalise the value functions from seen states to unseen. As we will concentrate in this work mostly on policy In a similar way, we can also estimate policy function:

TODO: polcy approximation

Cost Now we have a way to initialise our policy with any parameters θ. However in order to estimate the policy, we also need to find a way to measure it's quality, i.e. define the cost function J. One way to defined our cost function is to assign it assign to the value of the start state:

$$J(\theta) = V^{\pi_\theta}(S_1) = E_{\pi_\theta} \tag{2.28}$$

To remind you, the start value is the return that an agent will get from start state following policy π_θ. This will work only for environment with end state.

Policy Our agent needs to take action while going through the environment. For continuous state space it's common to use Gaussian distribution with fixed variance σ^2 and parametrized mean:

$$\mu = \phi(s)^T\theta \tag{2.29}$$
$$a \sim \mathcal{N}(\mu(s), \sigma^2) \tag{2.30}$$

where $\phi(s)^T$ - is a feature vector of state s.
Feature vector normally represents the availability of certain features in a particular state s.
For example, the first entry in the feature can determine whether an robot is close to a wall. In state s_1 where robot actually has a wall a close to him this first entry should be then equal to 1 or close to 1. In contrast to state s_2 where robot has no wall close by, this entry in the feature vector should be 0 or close to 0. The feature vector might have hundreds of entry features like this.

Training Now we know the objective function, we can the gradient descent algorithm to maximise it.

Policy gradient theorem says that gradient for any differentiable cost policy function is:

$$\Delta_\theta J(\theta) = E_{\pi_\theta}[\Delta_\theta log(\pi_\theta(s,a))Q_{\pi_\theta}(s,a)] \tag{2.31}$$

Reinforce rule Using return v_t as an unbiased sample from $Q_{\pi_\theta}(s,a)$ in 2.31 we can derive the REINFORCE rule:

2.4 Recurrent Models of Visual Attention

As as it was mentioned in chapter 1, Recurrent Models of Visual Attention(also known as the recurrent attention model(RAM)) is a computational approach presented by google Deepmind's researchers to reduce and contronl computational cost while classifying images.

Note: The theory described in this section based on the [4], unless otherwise stated.

Why RAM? Today's state of the art approaches to classify the images is convolutional neural network(CNN). Nevertheless, to train CNN model on the high resolutional images will require days of time. Similar to RNN models, some computations in CNN are shared, however the main cost is laid on applying the convolutional filter on the entire image, hence the bigger image is the more computations is required.[9]

Recurrent model attention is trying to face this problem by controlling amount of computations by adaptively selecting a sequence of locations and only processing these location at high-resolution

The reccurent attention model The attention problem in RAM is considered as sequential decision process of an agent who interact with visual environment. The idea is, that agent never sees the whole input, but only receives the observations via bandwidth-limited sensor. The goal for our agent is to find and select those locations on the images where the agent can extract most of information needed for a classification decision. To summarize it:

- Agent selects a location on the image
- Agent receives an observation
- Agent extracts the information from the observation via bandwidth-limited sensor with respect to the location.
- Based on what he extracted, he selects the next location.
- If current time step is more than some predefined number T, agent is forced to do a classification decision.

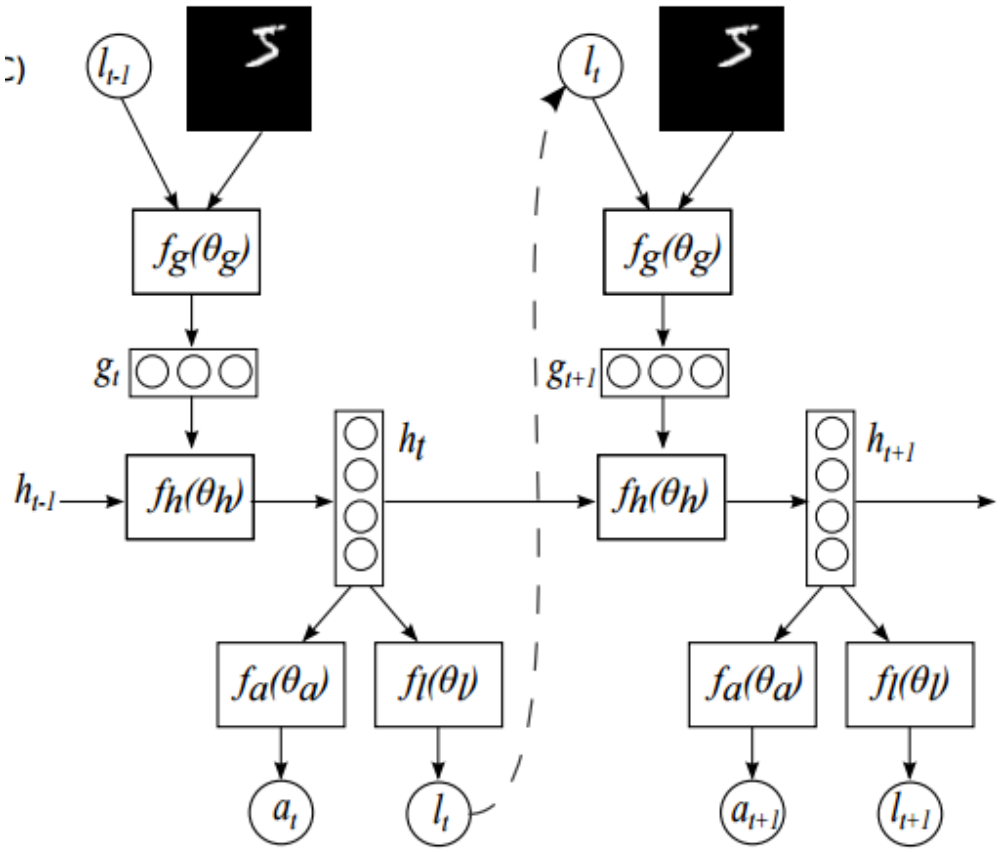

Figure 2.8: A) Glimpse Sensor, B) Glimpse Network, C) Model Architecture, (Source: [4])

In figure 2.8, you can see *glimpse sensor*, *glimpse network*, and the whole architecture of the model.

Glimpse sensor Glimpse sensor is formally the bandwidth-limited sensor which takes image and location as input and produces the bandwidth-limited representation of it. Thus, the main responsibility of the glimpse sensor is to build from an image x_t the image representation $\rho(x_t, l_{t-1})$ with respect to the location l_{t-1}. The sensor select a number of patches from an image x_t centered at the location l_{t-1} as you can see in 2.8. The first patch is processed at high resolution, while for patches further from l_{t-1}, progressively lower resolution is used. This representation contained multiple resolution patches of an image is known as *glimpse* and is denoted as $\rho(x_t, l_{t-1})$. [18] The glimpse

sensor is used within glimpse network to produce glimpse feature vector.

Glimpse network As you can see in figure 2.8 B), glimpse network takes an image x_t and location l_{t-1} as an input and produces the glimpse feature vector $g_t = f_g(x_t, l_{t-1}; \theta_g)$. Glimpse network uses glimpse sensor to create a glimpse $\rho(x_t, l_{t-1})$. Then it feeds this glimpse and the location l_{t-1} into two independent neural networks with parameters θ_g^0 and θ_g^1 respectively followed by another neural network which combines the information from both components. The last NN parametrized by θ_g^2 produces then final feature vector $g_t = f_g(x_t, l_{t-1}; \theta_g)$, where $\theta_g = \{\theta_g^0, \theta_g^1, \theta_g^2\}$.

Model Architecture The model architecture is shown on picture C) in the figure 2.8. The agent is built around the RNN and uses RNN's output as internal state. RNN's output h_t contains information from past glimpses and locations. It's possible because the RNN at each time step t receives g_t as external input and summarizes it by using a LSTM cell : $h_t = f_h(h_{t-1}, g_t; \theta_h)$.

Actions Agent perfumes two types of actions: environment actions and location action. Location action l_t decides where to allocate the next glimpse on the picture. The location network $f_l(h_t; \theta_l)$ is a NN layer that produces the next location l_t parametrized by θ_l. We will consider the location network a bit deeper in the **??**.
Depends on the task, environment actions can vary, therefore we'll concentrate on the environment action that is relevant for this work. In our work, environment actions makes the classification decision of an image. It's formulated using softmax output from environment neural network $f_a(h_t; \theta_a)$, where θ_a - is a set of parameters of neural network f_a.

Rewards As we making a classification decision only after T number of steps, our reward signal will be delayed. Therefore agent is receiving a reward $r_n = 1$ if the classification decision was correct and 0 otherwise.

Training of the agent is performed by REINFORCE rule described in subsection 2.3.4.

Chapter 3

Analysis

The main objective of this work will be to build an extensible prototype, which will be built on the model of visual attention and additionally extended to classify a set of images. The following chapter discusses the main concerns that have to be taken into consideration in order to achieve the objective.

Firstly, let's define the clear problem statement:

Given a dataset where each sample consist of a group of images. A certain label is asserted to every sample in a dataset. The goal is to build a extensible prototype upon RAM that is capable of classifying this samples. The prototype is the resemblance of the RAM model with with ability to accept the above mentioned dataset.

3.1 Quality concerns

This statement make it clear that prototype should be extensible. Extensible means that other people(including author itself) are able to extend the prototype. People should comprehend the code as well as start and configure procedures in order to extend the prototype. Therefore following best practices and conventions, using widely known frameworks, producing clean and readable code to stay consistent with other modern software is an important point of this work.

It's more convenient to talk about conventions once programming language and libraries are chosen for this work.

TensorFlow There is a big variety of frameworks and libraries used in machine learning. Among the famous libraries is library called TensorFlow. TensorFlow is an open source library developed and maintained by engineers from Google [19].

TensorFlow framework has gained a huge popularity among machine learning community as well as in industry compared to another framework [6]. TensorFlow is an open source software library that makes computations more efficient by building a computation graph and deploying them to one or more CPUs or GPUs.

TensorFlow framework gives a wide variety of statistical distributions, wide range of loss functions, and a huge amount of neural network algorithms while not necessarily losing flexibility. In order to make learning process traceable, TensorFlow provides TensorBoard, which is a web interface for graph visualization built directly into TensorFlow. TensorBoard is an important and uniq feature that excels TensorFlow from similar libraries; it dramatically improves debugging experience as well as helps for understanding models developed using TensorFlow. Aforementioned features of the framework as well as it's great API for Python is a great fit for this work.

The current work will use Python as programming language, as it's one of the most popular language among machine learning community.

Code conventions Now that we assigned interfaces, let's define conventions that this work will follow. As TensorFlow recommends using PEP 8 style guide to stay consistent with community [20]. PEP 8 gives the set of rules about how to lay out the code, how to name function, methods and classes and others code style related things

PEP 257 convention is also considered as this will help to avoid errors when generating documentation. PEP 257 convention set rule about organizing docstrings in Python [21]. Overall, the work follows The Hitchhiker's Guide to Python, which advises about the way of testing, organizing and documenting the code [22].

Design patterns Despite that a lot of code from researches are not following and not using design pattern, it's really simplifies understanding and reasoning about the code, especially while extending or adjusting the model. Therefore using design patterns is highly recommended. Below is a list of design pattern recommendations:

- Composed method - it's saying that methods should have one identifiable task as well as be on the same abstraction. This will increase flexibility of the methods and prevent from confusing code and mistakes. [23]
- Extract method - when one method is too long, one should try to extract some part of it into another function taking into account that

this function should be useful on its own. [24]

- Method object - normally applied when one can't use extract method on a function because new function would take a lot of parameters from the original method. The idea is to create a small class that will hold this parameters as properties and then to create small methods within this class without passing the parameters to it but instead access them as class properties. [23]

- Method comment - instead of having a comment close to not self-explanatory statement, one should rather create a function with self-explanatory name that holds this statement. Most comments are redundant, therefore one can simplify the system by introducing small methods. [23]

- Temporary variable - instead of having very long expression, it's recommended to assign part of the expression to a variable. This variable can also have the self-explanatory name, consequently increasing the readability of the code.[23]

- Simple Enumeration Parameter - having a simple name of a parameter when iterating over enumeration: `for each in plugins:` [23]

- Cascade pattern - when a method of a class doesn't return any value, one can return instance of the class in order to be able to call a next method of the instance in cascade way: `self.doThis().doThat()`.[23]

- Single responsibility principle - it forces every class to have only one clearly distinguishable responsibility.[25]

Since ML programming is more method oriented, the most pattern above concentrated on how to keep functions readable, precise and flexible.

We will see examples of those patterns in the **??**.

To recap, the prototype should be not only a good start for making further improvements on large scale objects but also be a piece of software that bears the following properties: extensible, well documented, integrable with other softwares, easy configurable, readable code.

3.2 Analysis of the previous work

As we will build the current model upon RAM described in section 2.4, it makes sense to look over the existing implementations of the RAM model.

The authors of RAM paper haven't provide the implementation, but fortunately there are few implementations built by open source community. It's common to see that people, who working on a model as open source project, are often do that in their free time. As a consequence, it is expected that those model are not high-quality prototypes, therefore in order to rely on the project one needs to be meticulous while choosing one. Following criteria were considering when choosing the model for the basis of this work:

- Structure and readability of the code - at least presence of the basic structure
- Configuration - it should be comprehensible how one can change parameters of the model
- Reproducible results
- Presence of basic documentation
- Framework - as we have already chosen TensorFlow, projects built upon TensorFlow are preferable.

After doing the research and taking into account the above listed criteria, the choice fell on the project from github user 'zhongwen'. We will refer to this implementation as *basic RAM implementation* [1] .

This project is built using TensorFlow framework.

3.2.1 Differences with original RAM paper

Building a machine learning software is not as straight forward as building conventional software. In machine learning, in order to achieve high accuracy one uses different methods which are normally hard to comprehend without knowing the theory behind it. Absence of testing is one another point which makes it harder to understand others code. Considering this circumstances careful analyzing of the approaches used in the basic RAM implementation is desired for the success of this work.

Inference in the current work was performed differently from what was described in [4]. Let's take a look at how the gradient of objective function is calculated in [4]:

$$\Delta_\theta J = \frac{1}{M}\sum_{i=1}^{M}\sum_{t=1}^{T}\Delta_\theta \log \pi_\theta(u_t^i|s_{1:t}^i)(R_t^i - b_t) \tag{3.1}$$

[1]The original project is located on github: https://github.com/zhongwen/RAM .

where J - is a cost function, M - number of episodes, T - is a time step where the agent is forced to make a classification decision, $s^i_{1:t}$ - are state sequences obtained by running the current agent π_θ for $i = 1...M$ episodes, b_t - baseline to reduce variance, u_{ti} - action at time step t of episode t.

As we can see the gradient is calculated based on running M episodes. Firstly, the agent chooses sequentially n location and make a classification decision. Then we run M similar episodes like this. After this, we compute the gradient. The basic RAM implementation used slightly different approach. It does not run multiple episodes to update the parameters, but instead it does update parameters on every episode. However it duplicates the same samples M times and then obtain M different outputs and average them. This practice was introduced in [5]. It was also indicated that running attention model multiple times on each image with the classification predictions being averaged gave the best performance [5].

Adam Opmitimizer It also worth to notice that instead of stochastic gradient descent that was used in RAM paper, the basic RAM implementation using Adam optimizer from [26] with exponentially decaying learning rate.

Gradient clipping In contrast to RAM paper, the basic RAM implementation additionally clipped gradient by global norm of their values to prevent the vanishing and the exploding gradient problems [27].

3.2.2 Limitations

Unfortunately, the basic RAM implementation used old version of TensorFlow, which was unstable at that moment, therefore migration of the project to the latest TensorFlow version is required[2]

Looking at basic RAM implementation purely from software engineering side, one can observe severals flaws. Not following python conventions as well as lack of design patterns and partial presence of incoherent structures in the code are known obstacles when analyzing and extending the code. Therefore code is required to be refactored and cleaned, as well as restructured and documented under the conventions and practices described in section 3.1.

As of functional requirements, lack of one feature was detected in basic RAM implementation. As you can remember from section 2.4, glimpse sensor is taking multiple patches from an image, where each patch is taken at different resolution. The basic RAM implementation is not capable of

[2] Migration was performed according to TensorFlow migration guide: https://www.tensorflow.org/install/migration

taking multiple patches, but only a single one. Without ability of taking multiple patches within a glimpse, accuracy of inference will more likely to decrease. Therefore this functionality should be available in the work in order to achieve better results.

Additionally to this, TensorBoard wasn't configured in basic RAM implementation. Training such massive model as RAM can be complex and sometimes confusing. Tools like TensorBoard make models easier to understand, debug and optimize, therefore having fully configured TensorBoard is another prerequisite for a good prototype.

3.3 Extension

In order to fulfill the objective of this work, it's required to extend the basic RAM implementation in a following way: it should accept a group of images as input and based on that input make a classification decision. Group in this case means more than one image(e.g. 2,3 or 5 images). More formally: *given a group of images, extended model returns a probability distribution over K different outcomes, where K - is amount of classes in the classification problem.* We'll refer to amount of the image in a group as n_i.

This section will discuss several ways of how to achieve that. In the figure 3.1 you can see original architecture of RAM. The potential extension variations listed below will use the notation, which was presented in section 2.4.

3.3.1 Picker network

You might remember the output of LSTM cell h_t in [4]. The external output is used in the action network $f_a(\theta_a)$ as well as in location network $f_l(\theta_b)$ as shown in figure 3.1 to produce a classification decision or next location respectively. In a similar way, we can invent a new network that takes as input the LSTM cell's output h_t, have parameters θ_p and outputs the probability distribution over n_i. We shall call this network *picker network*. The purpose of the picker network is to pick an image in the group to explore appropriate image, hence the name. By manifesting this extension, the model as a whole will have three neural networks consuming the output of LSTM cell h_t:

- Classification network $f_a(h_t; \theta_a)$ - makes a classification decision after T number of time steps.
- Location network $f_l(h_t; \theta_l)$ - decides where to allocate the next glimpse on the picture.

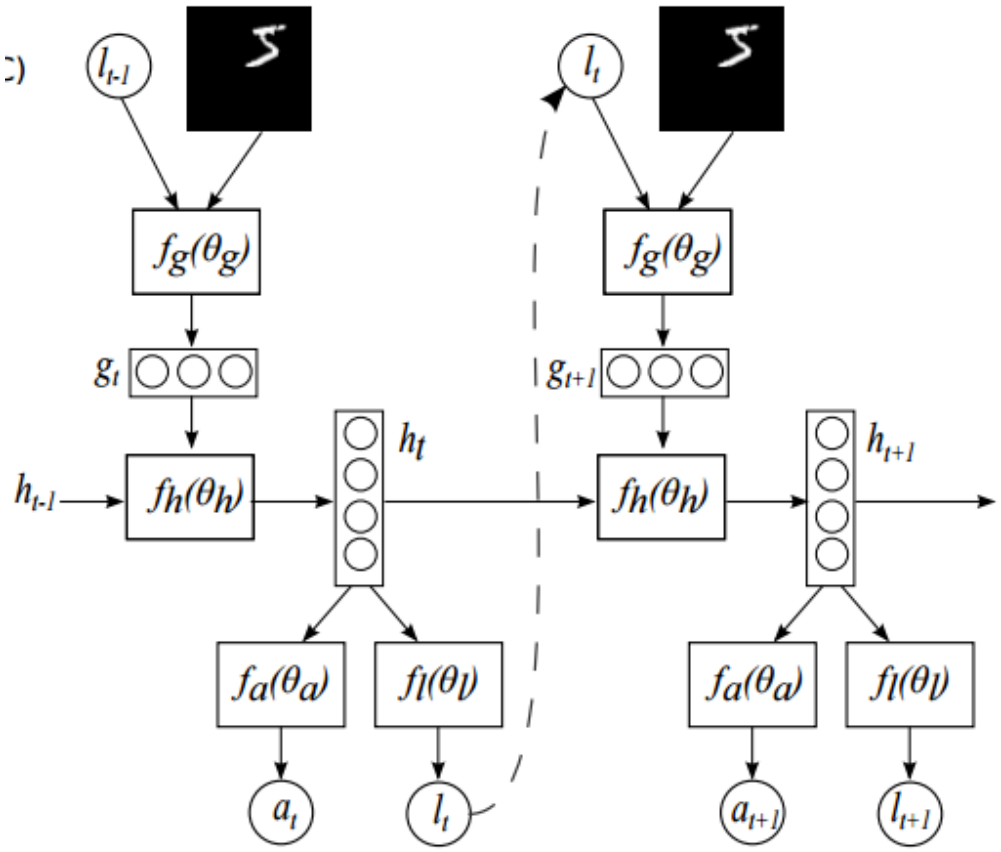

Figure 3.1: Original architecture of RAM.

- Picker network $f_p(h_t; \theta_p)$ - decides what picture to allocate the next glimpse on.

All of networks above have a similar structure:

$$f_a(h_t; \theta_a) = exp(Linear_{\theta_a}(h_t))/Z \tag{3.2}$$

$$f_l(h_t; \theta_l) = Linear_{\theta_l}(h_t) \tag{3.3}$$

$$f_p(h_t; \theta_p) = Softmax(Linear_{\theta_p}(h_t)) \tag{3.4}$$

where Z - is a normalizing constant, $Softmax$ - is softmax activation function $Linear_\theta(x)$ - represents a linear transformation $Linear_\theta(x) = W_\theta x + b_\theta$ with weights W_θ and bias b_θ.

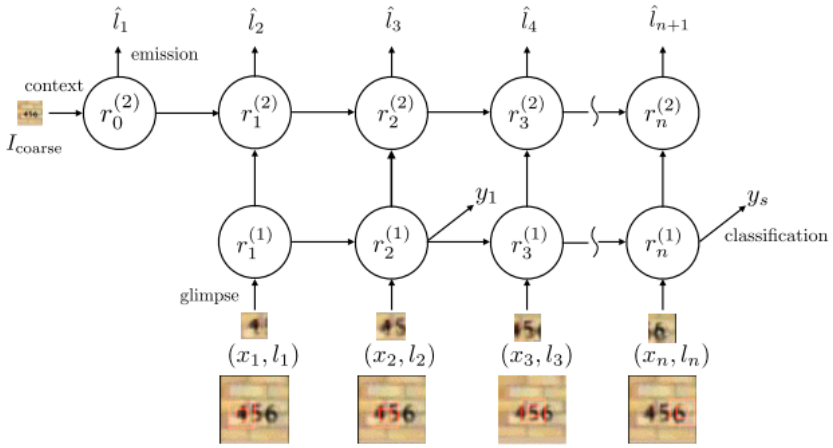

Figure 1: The deep recurrent attention model.

Figure 3.2: Source: [5]

3.3.2 Deep attention model

This idea was inspired by the approach used in [5]. Idea is to completely separate classification network from location network by introducing new RNN layer on top of the current RNN layer.

In figure 3.2 you can observe first RNN denoted $r^{(1)}$ which works the same way as was described in section 2.4. Output of RNN $r^{(1)}$ was consumed by location network and classification network in [4]. This will be different in current approach. We can describe it as following: We introduce new RNN $r^{(2)}$ on top of first RNN:

- The new RNN is initialized by output from convolution network(called Context Network) applied on the down sampled low-resolution version of input image. The feature vector outputted from convolutional layer provides sensible hints on where the potentially interesting regions are. The reason behind it was described in [5]:

  > The existence of the contextual information(feature vector), however, provides a "short cut" solution such that it is much easier for the model to learn from contextual information than by combining information from different glimpses

- The new RNN takes as input the output of RNN $r^{(1)}$: $r_n^{(2)} = f_{r^{(2)}}(r_n^{(1)}, r_{n-1}^{(2)})$, where $r_n^{(1)}$ - is the output of RNN $r^{(1)}$ at time step n, $f_{r^{(2)}}$ - is RNN cell function(e.g. LSTM cell), $r_n^{(2)}$ and $r_{n-1}^{(2)}$ are outputs of RNN $r^{(2)}$ at time step n and $n-1$ respectively.

- The output of new RNN is used to produce next location. The output is fed into fully connected network, which maps the feature vector produced by new RNN into coordinate object $[i, (x, y)]$, wherer i - is a index in the group of images.

The reason for having the second RNN is to implicitly represent decision about location and prevent using location information in classification. Separation is crucial: first RNN network is responsible for classification, while second one for choosing the right location. One state will hold information about classification, while the state of RNN cell on the top will hold information about locations. Advantage of this work is having this separation of concerns between location and classification information which theoretically should improve the model. This approach we will call *Deep attention model*.

3.3.3 Exploration network

In original RAM paper, model is required to execute a number of steps before making classification decision: " The attention network used in the following classification experiments made a classification decision only at the last timestep $t = N$. "[4] It was also suggested as a further improvement to introduce new action which decides when model should stop taking further glimpses:

> Finally, our model can also be augmented with an additional action that decides when it will stop taking glimpses. This could, for example, be used to learn a cost-sensitive classifier by giving the agent a negative reward for each glimpse it takes, forcing it to trade off making correct classifications with the cost of taking more. [4]

We can use alter this suggestion by introducing a new action which instead of indicating when the model should stop taking new glimpse, will indicate when model should take a next image in a group. Once there is no more images we can force the model to make a classification decision.

We introduce new network which take as input the state of RNN cell, and outputs probability distribution over two actions which correspond to

whether the model will stop taking new glimpses or continue to explore image. We shall call this network *the exploration network*.

By manifesting this extension, similar to the approach described in subsection 3.3.1, the model will have three neural networks consuming the output LSTM cell h_t:

- Classification network $f_a(h_t; \theta_a)$ - makes a classification decision after T number of time steps.
- Location network $f_l(h_t; \theta_l)$ - decides where to allocate the next glimpse on the picture.
- Exploration network $f_e(h_t; \theta_e)$ - decides whether the next glimpse should be taken from the i image from the group or from the $i + 1$ image.

New exploration network computes the distribution in the following way

$$f_e(h_t; \theta_e) = Softmax(Linear_{\theta_e}(h_t)) \tag{3.5}$$

The main advantage of this approach is the flexibility since it can easily be combined with deep attention model introduced in subsection 3.3.2. This approach can be also augmented with following actions:

- First Action - go back, that is, take previous image from the group of images.
- Second Action - go forward, that is, take next image from the group of images.
- Third Action - finish, that is, stop taking glimpse and do the classification decision.

3.4 Dataset

Before starting the construction of the model, one requires to think about an appropriate dataset on which the model can be tested. MNIST dataset[3] is recognized as being the simplest dataset among the neural network community, therefore building a dataset upon it would be easier. The simplicity of dataset would help to understand problems occurring while developing a model. Additionally to this, researchers from researchers from google Deepmind used MNIST to evaluate RAM, which indicated that MNIST data

[3]The MNIST data is hosted on [28]

would be good fit for the extension as well. [4] MNIST dataset is dataset consist of images of handwritten digits each labeled from set: $\mathcal{A} = \{0..9\}$, where $\mathcal{A}$ - is a set of available classes in MNSIT dataset. MNSIT dataset consist of 55,000 training samples, 10,000 test samples, 5,000 validation samples. Requirement for the dataset can be formulated as following:
Each sample in the dataset should contain fixed number of images. Amount of images in an sample should be more than 1. The dataset should have at least two classes, which are used to label each of the sample. The dataset should provide enough data to train, validate and evaluate the model.

There is different variations for building the dataset of group of images from MNIST dataset.

Procedure One of the variations could be described as following: having limit of 10 classes in MNIST dataset, we decide about the class(or classes) of images that hold information relevant for classification decision(it can be f.e. classes labeled with 4 and with 3 in MNIST), we'll refer to those images as non-noise images. Subset of classes of non-noise image we denote as: $\mathcal{D} = \{d_1, ..., d_n\}$. We'll have also images which does not contain any information relevant for decision making, we'll refer to this images as noise images. Noise images is generated using images from classes that are mutually exclusive with classes of non-noise images: $\mathcal{T} = \mathcal{A}/\mathcal{D}$, where $\mathcal{T}$ - is a subset of classes $\mathcal{D}$ for labeling noise images. We then decide about the amount the images in a group n. We then choose the number $k < n$ which is going to be amount of noise images per sample in dataset.

We then divide the original MNIST dataset into two:

- First dataset - consisting of noise images, where each sample is a group(array) of noise images with the size of k.
- Second dataset - consisting of non-noise images, where each sample is a group(array) of noise images with the size of $n - k$.

We then create a new dataset and fill the sample that is a concatenation of sample from first dataset with sample of second dataset described above. This way we're getting a new dataset where each sample has a size of n. Using this method we can create two dataset with different k, but the same n and label them as dataset of first class, and dataset of second class.

The conceptional idea behind this dataset, the the model should learn to understand that noise image does not bring any information relevant for the task. Practical solution for classification task of this dataset would be to understand what is non-noise images are, count them, and depends on the results classify the sample . If model would work correctly on this dataset,

it will confirm that the model works correctly for identifying images that are irrelevant for decision making

3.5 Testing

It's well-known that unit testing in ML models is not as trivial as in conventional software, therefore unit testing should be applied only on part of the code where behaviour is fully predictable. One such example for unit testing would to check whether certain components of the system having expected shape of vectors and matrixes. As code for producing the dataset having procedural structure, it should be fully tested in order to avoid unexpected behaviour in the model.

Chapter 4

Design

In this chapter we will discuss the design of the prototype

Bibliography

[1] M. A. Nielsen, "Neural Networks and Deep Learning," 2015.

[2] Colah Christopher, "Understanding LSTM Networks – colah's blog," 2015.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[4] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent models of visual attention," *CoRR*, vol. abs/1406.6247, 2014.

[5] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple object recognition with visual attention," *CoRR*, vol. abs/1412.7755, 2014.

[6] P. Goldsborough, "A Tour of TensorFlow Proseminar Data Mining,"

[7] "CAMELYON17 - Home."

[8] C. M. Bishop, "Neural networks for pattern recognition," *Journal of the American Statistical Association*, vol. 92, p. 482, 1995.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

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

[11] F. Rosenblatt, *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory), Spartan Books, 1962.

[12] Karpathy Andrej, "CS231n Convolutional Neural Networks for Visual Recognition," 2016.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, oct 1986.

[14] D. Kriesel, *A Brief Introduction to Neural Networks*. 2007.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[16] P. Werbos, "Backpropagation through time: what does it do and how to do it," in *Proceedings of IEEE*, vol. 78, pp. 1550–1560, 1990.

[17] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," 2012.

[18] H. Larochelle and G. Hinton, "Learning to combine foveal glimpses with a third-order Boltzmann machine," *Nips-2010*, pp. 1243–1251, 2010.

[19] 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. Software available from tensorflow.org.

[20] "TensorFlow Style Guide — TensorFlow."

[21] D. Goodger and G. van Rossum, "PEP 257 – Docstring Conventions — Python.org," 2001.

[22] K. Reitz, "The Hitchhiker's Guide to Python! — The Hitchhiker's Guide to Python."

[23] K. Beck, *Smalltalk Best Practice Patterns. Volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ, 1997.

[24] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series, Pearson Education, 2003.

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[27] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," nov 2012.

[28] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges," 2010.