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 [**?**].

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) [**?**]. 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.[**?**]

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[**?**].

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. [**?**]
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 [**?**].

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 [**?**]. 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 [**?**].

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. [**?**] Artificial Neural networks have recently shown an excellent performance and accuracy at recognizing objects compared with other machine learning techniques [**?**].

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 [**?**]. 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.

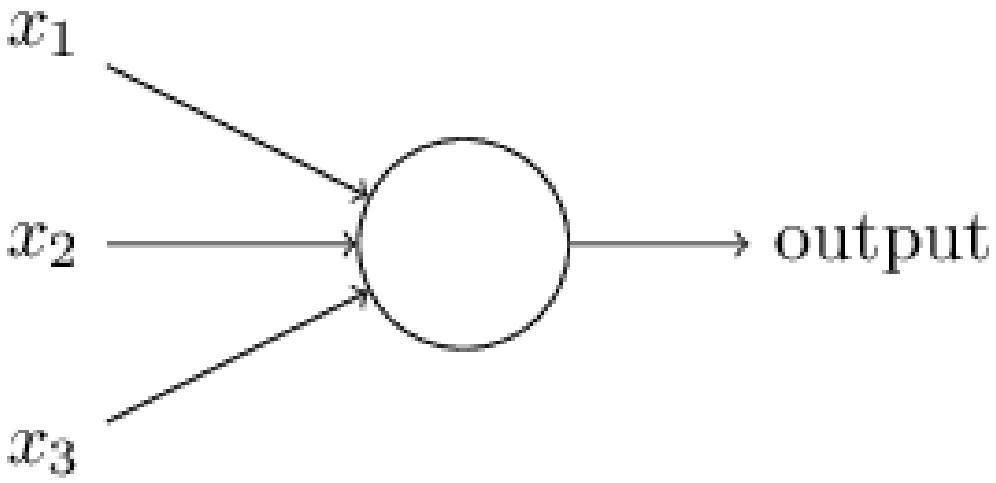

Figure 2.1: A perceptron [**?**].

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) [**?**].

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.

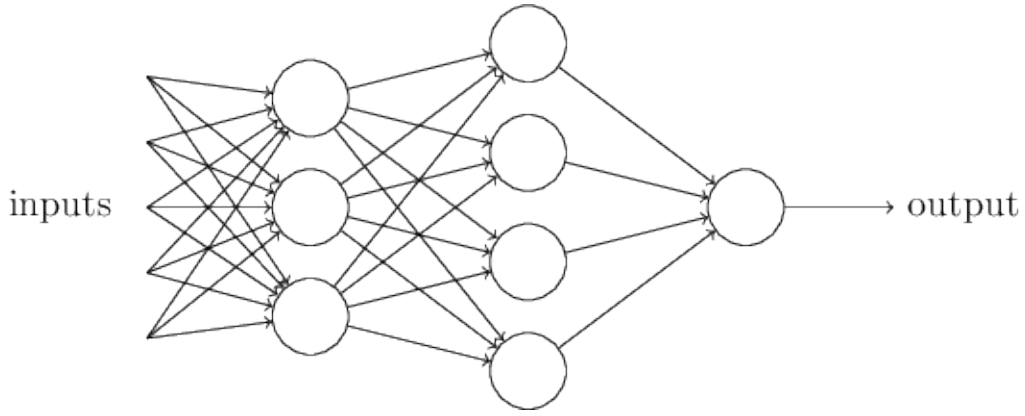

Figure 2.2: Two layer perceptron network[**?**].

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: $output = g(W \cdot x + b)$ [**?**] .

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 [**?**]. 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.

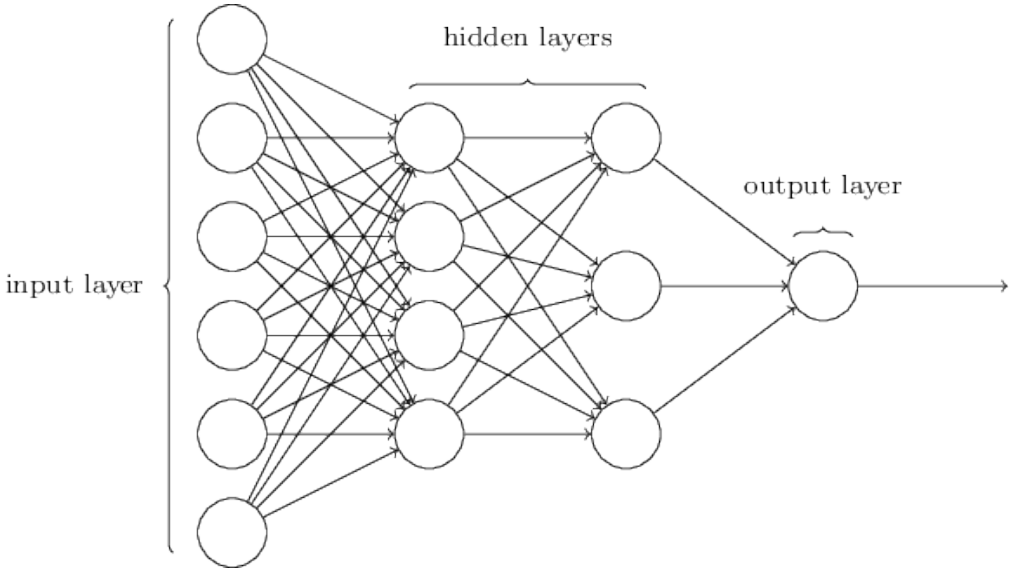

Figure 2.3: The architecture of neural networks[**?**].

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.

[?]

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 [**?**].

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. [**?**]

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 [**?**].

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*. [**?**].
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*.

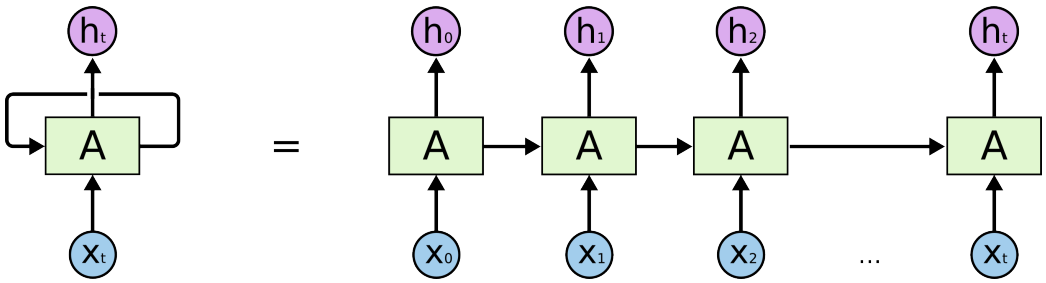

Figure 2.4: Unrolling recurrent neural network(source: [**?**])

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. [**?**] 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.

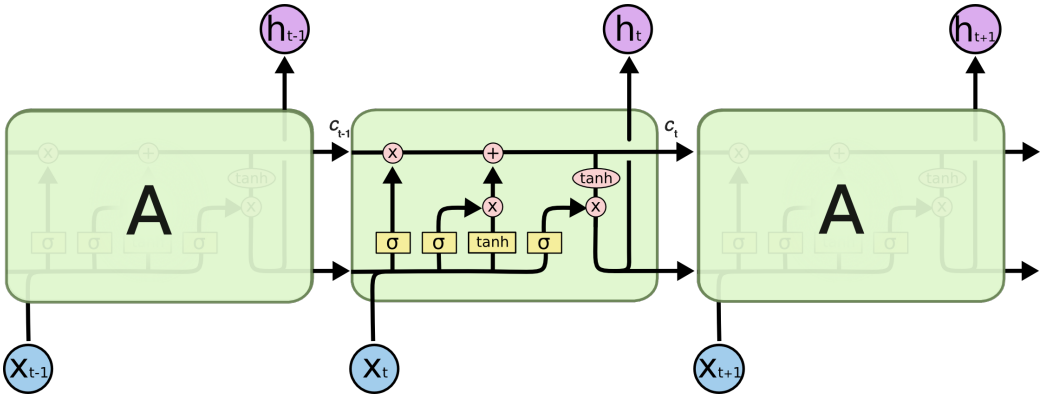

Figure 2.5: Structure of LSTM cell(Source: [**?**])
An yellow square with σ inside represents a neural network with sigmoid activation function, while yellow square with *tahn* inside represents a neural network with *tahn* activation function.

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.

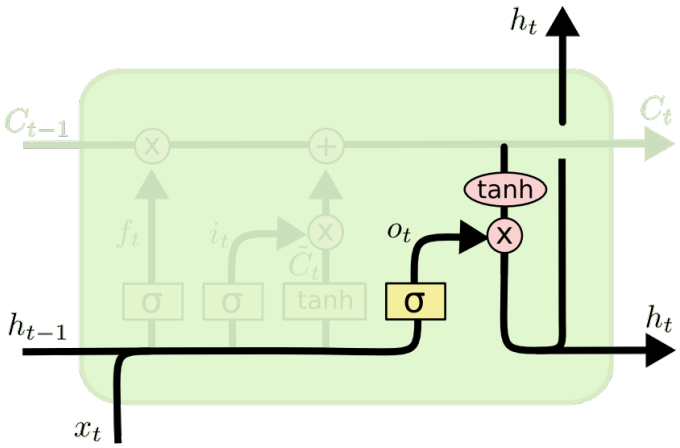

Figure 2.6: LSTM's output gate layer (Source: [**?**])

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.[**?**]

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. [**?**]

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 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.

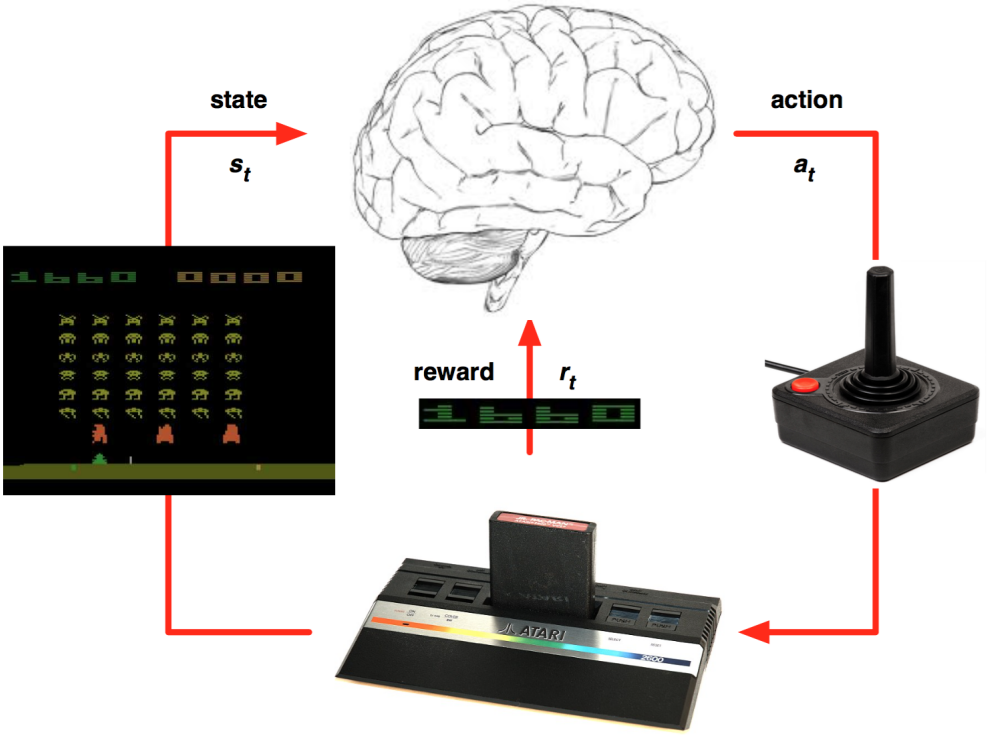

Figure 2.7: RL system to play Atari games (Source: [**?**])

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. We will investigate in POMDP more in **??**.

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 states* $\mathcal{S}$ - 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 $a \in \mathcal{A}(S_t = s)$:

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

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 $\mathrm{E}_\pi - is the expected value of the state \mathrm{S}_t$ 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:

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

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.

Bellman equations The most important reason that MDPs is used in RL problems because it give us the opportunity to use *Bellman equations*. Bellman equations forms the way of computing, optimizing and evaluating value functions $v_\pi(s)$ and $q_\pi(s, a)$.

Bellman expectation equation decomposes the value functions to represent in a recursive way through the value of successor state:

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|S_t = s] \\ &= E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \\ &= E_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s] \\ &= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned}$$

In a similar way, action-value function is decomposed:

$$\begin{aligned} q_\pi(s,a) &= E_\pi[G_t|S_t = s, A_t = a] \\ &= E_\pi[|S_t = s, A_t = a] \\ = E_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s, A_t = a] \\ &= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \end{aligned}$$

If we look beneath the math, these equations formally telling us that the value of a state is equal to the immediate reward that the agent get after leaving current state plus discounted value of the state where the agent will end up after leaving current state.

TODO:

Optimal value function Now that we know how to represent the value of a state, we can concentrate on the problem that we really care about, which is to find a best behaviour for MDP.

Explorations, greedy polciy TODO: Explaint glie monte carlo control

Monte Carlo method, How to assert value to return, i.e. represent value. **TODO:** also stricti definition. LOOK up latex definition.

Intuition behind value function: Lookup table In order to beeing able comprehend the concept of value function let's consider it's representation as lookup table. Imagine environment CartPole which can be described following:

> A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

We can represent state as tuple of every unit of horizontal space and the slope between pole and vertical. As example it can look like following:

Table 2.1: Example of state space represented as look up table

Now that we know how to define a state space we need to represent actions. Actions in this example would R and L, which moves the the card 1 unit to the right and to the left per time step respectively. After we defined the action space, we just calculate the reward according to the state by using monte carlo method. So our lookup can look as following:

Once we are done, can choose the policy by always choosing the state which has the biggest value.

2.3.4 Policy-Based Reinforcement Learning

Now, when we understand how is the environment managed, we can finally look at the concept of policy more accurately. 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.

TODO: the equation about average and etc. choose only one

But how we can evaluate a policy We need somehow to find a way to represent this state as we want to defined the value of each state. We can do it using lookup table but in our case RNN can produce near continuous output and to represent that as look up table 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 value function:

$$\hat{v}(s, W) \approx v_\pi(s) \tag{2.27}$$

$$\hat{q}(s, a, W) \approx q_\pi(s, a) \tag{2.28}$$

Note: The representation in 2.27 will even allow to generalise the value functions from seen states to unseen. In a similar way, we can also estimate policy function: **TODO: polcy approximation**

$$\pi_\theta = P[a|s, \theta] \tag{2.29}$$

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 like decision tree or nearest neighbour approximator.

In our problem, observations will come from glimpses to RNN which we will use as a state for our agent:

$$S_t^a = \sigma(S_{t-1}^a \cdot W_s + O_t \cdot W_o) \qquad \text{(2.22 revisited)}$$

where S_t^a and S_{t-1}^a 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.

Policy gradient theorem it basically says that gradient for any differentiable objective function the policy gradient is:

$$\Delta_\theta J(\theta) = E_{\pi_\theta}[\Delta_\theta log(\pi_\theta(s,a)) Q_{\pi_\theta}(s,a)] \qquad (2.30)$$

Reinforce rule Using return v_t as an unbiased sample from $Q_{\pi_\theta}(s,a)$ in **??**. As suggested by in **??**, which is unbiased sample from value function in .

and then actual sequential of steps from. The whole process can simulated as following:

Initialise θ randomly

do a t of episodes and etc.

2.3.5 POMDP

Bibliography

[1] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent Models of Visual Attention," *Nips-2014*, pp. 1–9, jun 2014.

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

[3] "CAMELYON17 - Home."

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

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

[6] 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.

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

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

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

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

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

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

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

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

[15] 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.