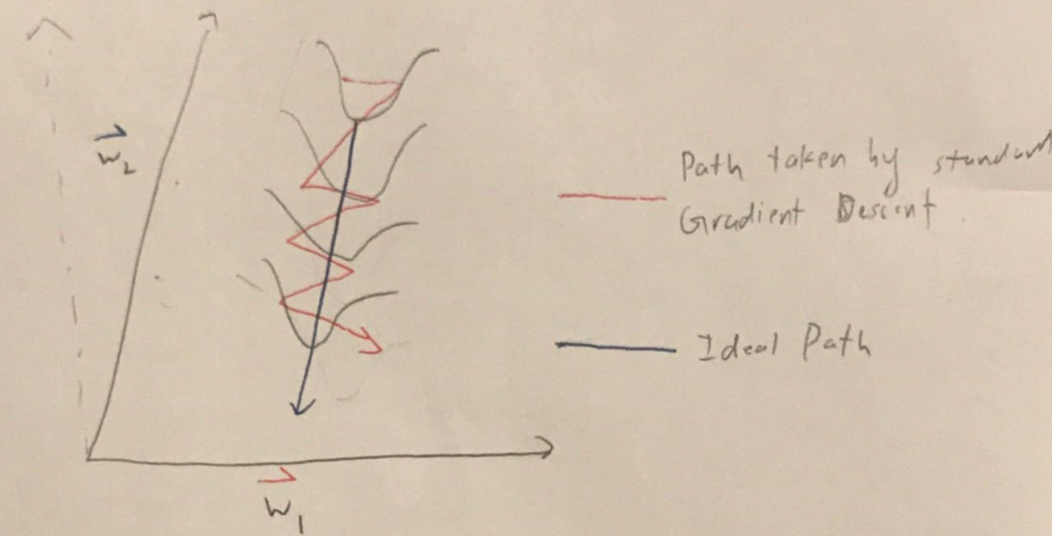
Coursework 2:

1) Rms vs. Adam Optimisation

Normal gradient descent struggles (e.g. batch/SGD) struggle in finding optimal minima given the complexity of loss functions with high dimensionality. These loss functions often have contours such as pathological curvature which is shown in the diagram below:



Above is a ravine-like region that is part of the complete 3-D loss function. As you can see, when GD gets stuck in this region, it will oscillate along the w_1 axis because the gradient along this path is steeper than the gradient along the w_2 axis. Thus, instead of taking the more optimal, ideal path as shown in blue, it takes the indirect, less direct red path. This sort of oscillation becomes an issue when selecting the learning rate for regions with this sort of pathological curvature. A learning rate that is too large will end up diverging along the $\vec{w}_1$ axis. A learning too small will take much too long to converge and can oftentimes lead to observe to falsely believe that a local minima has been reached.

To solve this issue we have rmsprop and adam optimisation algorithms.

A) **RMSprop:**

RMSprop is an algorithm of gradient descent that reduces the oscillation along the $\vec{w}_1$ axis. Along w/ the standard gradient descent formula, it incorporates past gradients with each gradient given exponentially decreasing weights.

— next page —

RMSProp continued

Below are the equations for RMSProp

arbitrary constant

Exponential Average of squares of gradients ⇒ $\boxed{v_t := \rho v_{t-1} + (1-\rho) * g_t^2}$

past exponential average

square of current gradient

step size (learning rate)

Gradient Update ⇒ $\boxed{\Delta w_t = -\frac{\eta}{\sqrt{v_t + \epsilon}} * g_t}$ ← current gradient

exponential average of gradients

very small constant

Weight Update ⇒ $\boxed{w_{t+1} = w_t + \Delta w_t}$

Key observations from above:

1) "Exponential average of gradients" weighs the most recent gradient most heavily

2) "Exponential average of gradients" incorporates past history of gradients to update weight. It then divides the gradient by the square root of the average. Because this is a squared average, the "$\sqrt{v_t + \epsilon}$" denominator will be larger along the $\vec{w}_1$ axis and smaller along the $\vec{w}_2$ axis. Therefore, the gradient update is corrected more heavily along the $\vec{w}_1$ direction than the $\vec{w}_2$ direction. [(* Thus, this allows the user to use a larger learning rate without worrying about divergence on the $\vec{w}_1$ axis by impeding the gradient in that direction. *)]

Thus, RMSprop may not reduce the amount of oscillation but it very well reduces the magnitude of these oscillations.

Therefore, 0.001 would be reasonable for the learning rate since it is neither too big nor small: there will be no divergence along the $\vec{w}_1$ axis. Second, it is large enough to converge to a good local minimum. However, 0.000001 is way too small. RMSprop divides this value by $\sqrt{v_t + \epsilon}$ and this will never converge to a local minimum. It will be stuck in an oscillating state along the pathological curvature. Lastly, 1 is likely too big and would lead to divergence on the $\vec{w}_1$ axis.

Adam Optimizer

Adam optimizer is very similar to RMSProp with a small addition.

same $\rightarrow$ $v_t = \beta v_{t-1} + (1-\beta) * g_t^2$ notice squared

new! $\rightarrow$ $s_t = \beta s_{t-1} + (1-\beta) * g_t$ notice not squared

exponential average of gradients

$$\Delta w_t = -\eta \frac{s_t}{\sqrt{v_t + \epsilon}} * g_t$$ $\rightarrow$ new!

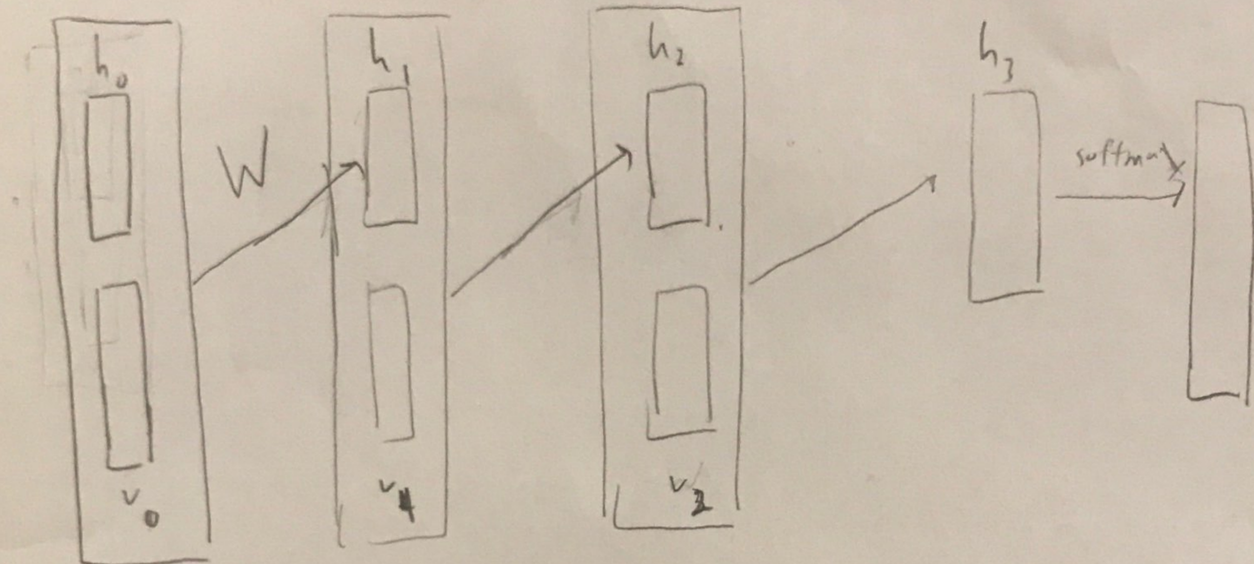
$$w_{t+1} = w_t + \Delta w_t$$

The only difference is that adam incorporates the exponential average of gradients in addition to the exponential average of squared gradients in its weight update.

[The overall effect of this addition is that the algorithm will improve the speed of computation in certain cases by reducing both size and magnitude of oscillation along the $\vec{w_1}$ axis] size of oscillation can be reduced and a more ideal path towards the minima can be taken because the "s_t" term (many of them) will have opposite signs and will cancel each other out. This does not happen in RMSprop because it only accounts for squares of gradients which will all have the same sign

2) Question 2: LSTM vs. recurrent neural network, but mostly LSTM.

Below is an RNN:



(1) Inputs are represented by v_i and are input into RNN sequentially

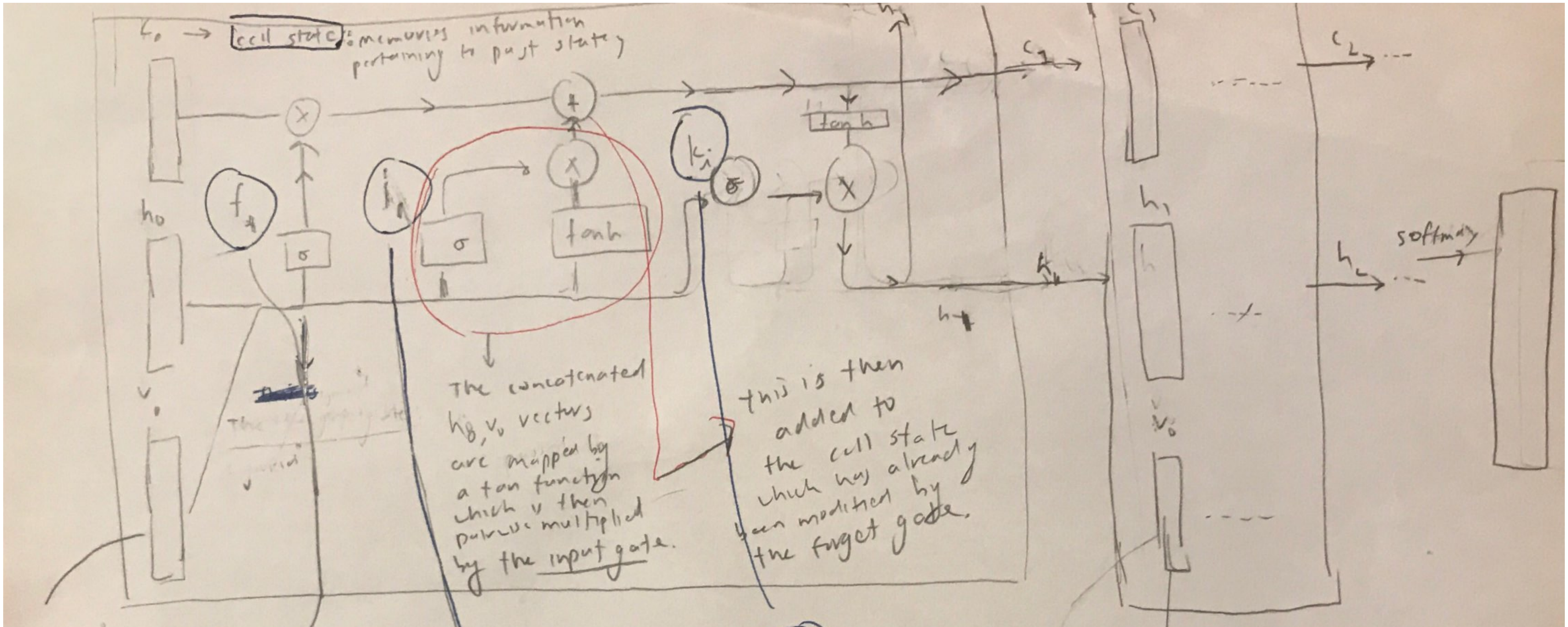
(2) The hidden state at the h_0 is initialized randomly and concatenated with the input v_0 and then mapped by W to the hidden state at h_1.

(3) The mapping W is the same at each step and maps the concatenation of v_i, h_i to h_{i+1}, a new hidden state.

(4) The final hidden state at h_3 is then mapped to probabilities with an arbitrary map

While both RNNs and LSTMs naturally incorporate the sequential aspects of the data in their models, LSTMs are more powerful because it includes an additional "cell state" which retains information from the past and can transmit this information to steps in the future for modelling. This "cell state" is referred to the "memory" of the LSTM.

— More on next page —



Input 1

f_1 **The forget Gate**

- The sigmoid function is applied on the concatenated hidden state and input vectors to obtain a new vector between 0 & 1. values close to 0 perform "forget" operations and close to 1 perform "remember" operations. This vector is then multiplied pairwise by the cell state to tell it what to keep and what to forget.

i_1 **The input Gate**

- Here, the sigmoid function is used to determine which new inputs are important to keep and which ones should be forgotten. similarly, values close to 0 are forgotten & close to 1 are kept.

k_1 This last sigmoid function takes concatenated h_0, v_0 and maps it to another vector of 0 & 1's. this is used to tell the cell state which values should be kept and discarded to form the new hidden state.
(cell state is 1st passed through tanh map).

Input 2

Essentially, because the LSTM needs a W_1 matrix for the forget gate, a W_2 for the input gate, W_3 matrix before the 1st tanh activation, and a W_4 matrix for the new hidden state output, it is much more powerful than an RNN but at the same time, much more complex.

An RNN will continuously incorporate past historical information even if it is not useful but an LSTM has the complexity to select what information to keep and what information to remove.

$\boxed{\text{Most importantly}}$ the LSTM can recall and reweight information from increasingly long time steps in the past because of the "cell state" while in a RNN, information from increasingly long time stamps is weighted less heavily.

(3) Carry out value-iteration by hand: $n = 4$, $p = 0.3$

From observation, it is clear that if our capital is $n = 0$, then the Q values will all be 0

have n ↓ , bet X →

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Iteration 1:

have $n = 1$, bet $X = 2$

$= 0.3V^*(2) + 0.7V^*(0)$ → reward

$= 0 + 0.7 \max[0,]$ based on current Q table.

$= 0 + 0$

$n = 2, \; x = 1$

$= 0.3V^*(3) + 0.7V^*(1)$

$= 0 + 0$

$= 0$

$n = 2, \; x = 2$

$= 0.3V^*(4) + 0.7V^*(0)$ ← optimal value function

$= 0.3(4)$

$= 1.2$

updated Q table

$n = 3, \; x = 1$

$= 0.3V^*(4) + 0.7V^*(2)$

$= 0.3(4) + 0.7(1.2)$

$= 2.40$

Have ↓, Bet →

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	1.2	0
3	0	2.04	1.5	1.8

update values

$n = 3, \; x = 2$

$= 0.3V^*(5) + 0.7V^*(1)$

$= 0.3(5) + 0.7 \max[1,]$

$= 1.5 + 0 = 1.5$

update

$n = 3, \; x = 3$

$= 0.3(6) + 0.7(0)$

$= 1.8$

Iteration 2:

Bet $x \rightarrow$

Have $n \downarrow$

	0	1	2	3
0	0	0	0	0
1	0	0.36	0	0
2	0	0.864	1.2	0
3	0	2.04	~~1.5~~ 1.752	1.8

$n=1, x=1$

$= 0.3V^*(2) + 0.7V^*(0)$

$= 0.3 \max[2,1] + 0$

$= 0.3 * 1.2 = 0.36$

$n=2, x=1$

$= 0.3V^*(3) + 0.7V^*(1)$

$= 0.3 \max V^*(3) + 0.7 \max V^*(1)$

$= 0.3(2.04) + 0.7(0.36)$

$= 0.864$

$n=2, x=2$

$= 0.3V^*(4) + 0.7V^*(0)$

$= 1.2$

no update

$n=3, x=1$

$= 0.3V^*(4) + 0.7V^*(2)$

$= 1.2 + 0.7(1.2)$

$= 2.04$

no update

$n=3, x=2$

$= 0.3V^*(5) + 0.7V^*(1)$ $\max[1,]$

$= 1.5 + 0.7(0.36)$

$= 1.752$

$n=3, x=3$

$= 0.3V^*(6) + 0.7V^*(0)$

$= 1.8$

no update

So, after the 3rd iteration values stay the same so we have now obtained our Q table with optimal values for each state action. Based on Q table above, optimal policy would be:

- Bet 1 chip when n=1 ($V^* = 0.36$)
- Bet 2 chips when n=2 ($V^* = 1.2$)
- Bet 1 chip when n=3 ($V^* = 2.04$)

(4)

(a) The fundamental problem for reinforcement learning is that networks are tough to train. While in supervised learning the inputs and outputs are constant, the inputs/outputs for (deep Q Learning) Reinforcement Learning are always changing. To combat this problem, a few methods including, but not limited to are used:

1) Target Network:

This network uses 2 deep networks in order to fix the Q-value targets temporarily so that they don't move. One deep network is used only for retrieving the Q-values. The other network computes all updates to parameters in training. This way, parameter changes occuring in the 2nd network will not impact the first network immediately. After updating for a significant number of epochs, the networks are then synced.

2) Experience Replay:

This is where mini-batches are sample. Basically, Q-learning updates are applied on batches of data obtained from past agent experience. The agent, essentially, will store its data at each time step and then this data will be pooled over multiple episodes into a "replay memory". To train the agent, data is randomly sampled from this "pooled memory" entity.

Lastly, during this learning stage, the performance of the network is assessed (as stated by atari article) is by collecting a fixed set of states and then running a random policy before training starts and tracking the average of the maximum predicted Q for these states.

Another metric used to evaluate performance was to periodically compute the total reward the agent collects during an episode and then averaging the reward over a number of episodes.

Scanned with CamScanner

(4)

b) In response to the blog, it is correct in the sense that a partial solution can indeed be collected. Even if the state space is large, by obtaining all of the Q-values pertaining to that state space, as long as the agent is in that state space, the agent will know what to do and a solution not necessarily optimal can be found.

However, the blog is misleading because it makes it sound like

a) a solution will be found. This is not true as the agent will not know what to do if an agent is put in another state space without calculated Q-values.

b) The solution will not necessarily be optimal. Q-values in other state spaces still have an effect on the Q-values of the state space in question. This is the nature of value-iteration! updating one state space Q value will have cascade effects on the calculation of Q-values in other state spaces. ?
thus, the Q-values calculated in the restricted state space of interest are not necessarily accurate and an optimal solution will not necessarily be found.