Optimization Algorithms

↳ So, we started with Gradient Descent Algorithm for single Sigmoid Neuron & we saw how to extend to network of neurons with back propagation.

Goal

↳ Finding a better way to traversing the Error Surface so that we can reach the minimum value quickly without resorting to brute force search.

↳ (Betterway of navigating to Error Surface)

Gradient Descent Rule

So the better way we could arrive it from Principled Solution from the Taylor series.

⇒ So, we move in a direction opposite to Gradient

Parameter Update Equations

$$w_{t+1} = w_t - \eta \nabla w_t$$

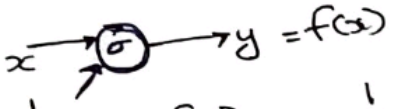
$$b_{t+1} = b_t - \eta \nabla b_t$$

Where,

$$\nabla w_t = \frac{\partial \mathcal{L}(w,b)}{\partial w}$$

$$\nabla b_t = \frac{\partial \mathcal{L}(w,b)}{\partial b}$$

* So, now we have a principled way of moving in the w-b plane than our guess work Algorithm.



$$f(x) = \frac{1}{1+e^{-(wx+b)}}$$

↓

So, we were just trying to adjust these weights & biases by hand

↓

And we realized it's clearly not good and but we still try to do a guesswork, where we were driven by this loss function. (It tells whether the current guess is better than the previous guess or not).

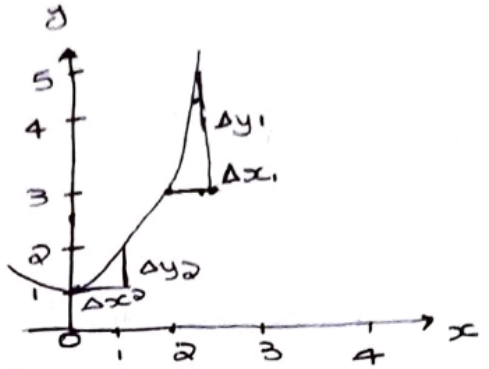
↓

So, what we are actually doing is there is this Error Surface Exists, which can be plotted for all w, b values. So, we are finding a path over the surface so that we enter to the better region.

So When we run this Algorithm it was intially slow then Suddenly Picked up & then it again become slow.

⇓

This is because,



⇒ When the curve is steep the gradient $\left(\frac{\Delta y_1}{x_1}\right)$ is large } ⇒ Larger in terms of Magnitude.

⇒ when the curve is gentle the gradient } ⇒ Smaller in terms of Magnitude.

$\left(\frac{\Delta y_2}{x_2}\right)$ is small

Algorithm

```
t ← 0
max_iterations ← 1000
While t < max_iteration do
    | w_{t+1} ← w_t - η∇w_t
    | b_{t+1} ← b_t - η∇b_t
end
```

Momentum based Gradient Descent

Observation in Gradient Descent

↳ ① It takes a lot of time to navigate regions having a small gentle slope

② This is because gradient in these regions are very small.

So if we had set this to max_iteration Equal to 1000, & the intilization happens to be such that you stuck in large flat region, Then those 1000 iterations just keep moving around the flat region.

* we will never enter into a valley. Valleys are what we are interested that's because where we could find the minimal function.

Intution

↳ If I am repeatedly being asked to move in the same direction then I should probably gain some confidence and start taking bigger steps in that direction.

Update rule

$$Update_t = \gamma \cdot update_{t-1} + n\nabla w_t$$

$$w_{t+1} = w_t - update_t$$

$$Update_0 = 0$$

$$Update_1 = \gamma \cdot update_0 + n\nabla w_1 = n\nabla w_1$$

$$Update_2 = \gamma \cdot update_1 + n\nabla w_2 = \gamma \cdot n\nabla w_1 + n\nabla w_2$$

$$\vdots$$

$$Update_t = \gamma \cdot update_{t-1} + n\nabla w_t = \gamma^{t-1} \cdot n\nabla w_1 + \gamma^{t-2} \cdot n\nabla w_1 + \cdots + n\nabla w_t$$

* Is moving fast good? would there be a situation where momentum would cause us to run passour goal?

Ex:

So we have a very peculiar Error Surface. Two Pleateaus at top and a very sharp valley. So the Error is high on the either side of the minima valley.

∴ Momentum based gradient descent oscilates in and out of the minima valley as the momentum carries it out of the valley.

→ Takes a lot of u turns before finally converging. Despite these u turns it still converges faster than Vanilla gradient descent.

Nesterov Accelerated Gradient Descent (NAG)

So to reduce the oscillations we use

Intution

→ Look before you leap

→ Recall the $update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$

→ So, we know that we are going to move by at least by $\gamma \cdot update_{t-1}$ & then a bit more by $\eta \nabla w_t$

→ Why not calculate the gradient (∇w_{look_ahead}) at this Partially updated value of w ($w_{lookahead} = w_t - \gamma \cdot update_{t-1}$) insetad of calculating it using the current value w_t.

Update Rule

$$w_{lookahead} = w_t - \gamma \cdot update_{t-1}$$

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_{lookahead}$$

$$w_{t+1} = w_t - update_t$$

→ Looking ahead helps NAG in correcting its course quicker than momentum based gradient descent.

→ Hence, the oscillations are smaller and the chances of escaping the minima valley also smaller.

Stochastic & Mini Batch Gradient Descent

↳ In Gradient Descent Algorithm it goes over the entire data once before updating the parameters.

So if we have million of point, we will go over all these million points & make this one update.

⇓

Now imagine the consequence, when you are in plateau region. So imagine how much time it will take to converge. (one step update). ⇒ Very slow.

⇓

To make it better ?? "Stochastic Gradient Descent."

⇓

So this Algorithm updates the parameters for Every single data point. So, here we have million data Points we update million times.

⇓

This is not the True Gradient. True Gradient is the Summation over all the points. Now this is no longer a true gradient this is just a point estimator, this is just a approximation of gradient.

⇓

We see many oscillations Why??

↳ Because we are making greedy decisions.

⇓

Because we are looking at one point, the point says to decrease the loss with respect to move in this direction & we blindly move in that direction. So all these points are actually trying to just make things better for themselves. They are not thinking about what is happening to all other Points in my data right.

→ So some decision which I took with respect to where to move which was locally favourable for one of these Point not good for another point.

⇓

Can we reduce the oscillations by improving our stochastic estimates of the gradient?

⇒ Yes, lets look at mini Batch gradient Descent.

***

So looking at one data point is bad because it is very noisy, looking at the Entire data is bad because it is Very time Consuming. So you need to do something in b/w which is mini batch gradient descent.

Algorithm	# of steps in 1 Epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini Batch Gradient Descent	N/B

Tips for Adjusting Learning Rate & Momentum

↳ One could argue that we could have solved the pblm of navigating gentle slopes by setting the learning rate high?!

↳ So it is not that we choose high 'η' and get away with it we actually want a adaptive 'η' that somehow figureout that it is on gentle slope move ~~slowly~~ fast and now I am on fast slope move slow.

⇒ Tune learning rate [Try diff values on log scale 0.001, 0.01, 0.1, 1, 10]

⇒ Run few Epoch with each of these figure out a learing rate which works best

⇒ Now do a finer search around this value around 0.05, 0.08, 0.3 etc.

⇒ Tips for annealing learning rate

① Step decay: Halve the learning rate after Every 5 Epochs.

↳ after 5 epochs we expect closer to the solution

(or)

⇒ Halve the Learning rate after an Epoch if the validation error is more than what it was at the End of the Epoch.

② Exponential decay: $\eta = \eta_0^{-kt}$ where η_0 & k are hyperparmeters and t is the step number.

↳ Where with each time step we just keep decreasing the learning rate.

③ 1/t Decay: $\eta = \frac{\eta_0}{1+kt}$ where η_0 & k are hyperparmeters & t is the step number.

⇒ Tips for Momentum

$$\mu_t = \min\left(1 - 2^{-1-\log_2[t/250]+1}, \mu_{max}\right)$$

Where μ_{max} Choosen from {0.999, 0.995, 0.99, 0.9}

↳ So as the Time step increasing gamma is increasing.

Gradient Descent with Adaptive Learning Rate

↳ Refer PPt & Video (5.9)

Explanation for why we need bias correction in adam

↳ Proof

} Refer it once.

*

Sequence Learning Problems

↳ So far we have dealed with feedforword and Convolutional neural network and both these networks the input was always of a fixed size. Further, Each input to the network was independent of the previous or future inputs

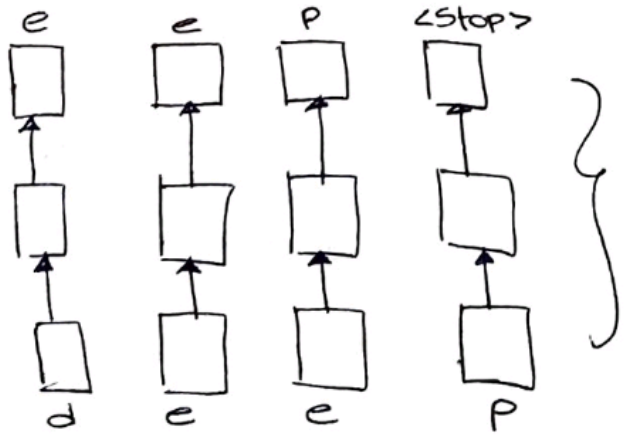
→ But in many applications input is not of a fixed size. Further succesive input may not be independent of each other.

Ex: Consider the task of auto Completion

① Successive inputs are no longer independent

② the length of the input is not fixed

③ Each network performing the same task.



} Each of this is a fully connected layer.

So these are known as Sequence learning problems. Where we have a sequence of inputs and then you need to produce some outputs and each input actually corresponds to one ti

→ Soln the task of Predicting the Parts of Speech for each word in a sentence. Once we see an adjective we are almost sure that next word is noun.

↳ Thus the current output depends on the current input as well as the previous input.

↳ Unlike the case of CNN where we feeded an apple it is no dependence on whether the previous Input that I pass to the network was an apple (or) car (or) what not.

⇒ Here we are intrested in producing the output at each time step.

⇒ Sometimes we may not be intrested in producing the output at each stage.

↳ Ex: task of predicting the polarity of a movie review. (Sentiment analysis)

⇓

So after reading the entire review model should give the prediction otherwise it would be incomplete.

⇒ Even here the network is performing the same task but the only thing is it don't care about the intermediate outputs.

→ Sequence could be composed of any things For Example, a video could be treated as a sequence of images.

→ We may want to look at the Entire Sequence & then detect the activity being performed.

Ex: Surya Namaskar.

Recurrent Neural Networks (RNN)

↳ So we have seen Sequence learning problems. So the Question is how to model this?

The Model will Come up with :

① Dependence b/w the inputs. (Output actually depend on multiple inputs)

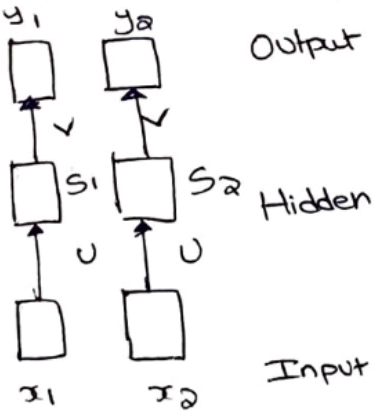
② Variable no of inputs.

③ It should also make sure the function Executed at each time step is the same.

(Because at Every time step we are doing the same activity).

④

⇒ What is the function being Executed in each time step?



$$S_i = \sigma\ (Ux_i + b)$$

$$y_i = O\ (Vs_i + c)$$

i = time step

⇒ Since, we are simply going to compute the same function at each time step, the no of time steps doesn't matter

↳ Since we want same function to be Executed at each time step we would share the same network (Same parameters at each time step).

→ we just make a multiple copies of the network & Execute them at each time step.

⇒ How do we account for the case that the output actually depend on multiple inputs not on the current input?

Infeasible way of doing this

↳ At each time step we will feed all the previous inputs to the network.

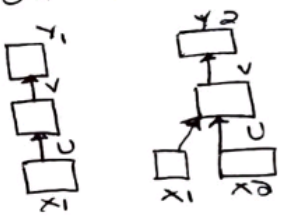
⇓

The problem with this is it violates the rules of the Conditions

→ First, the function being executed at each time step now is different.

$$y_1 = f_1(x_1)$$

$$y_2 = f_2(x_1, x_2)$$

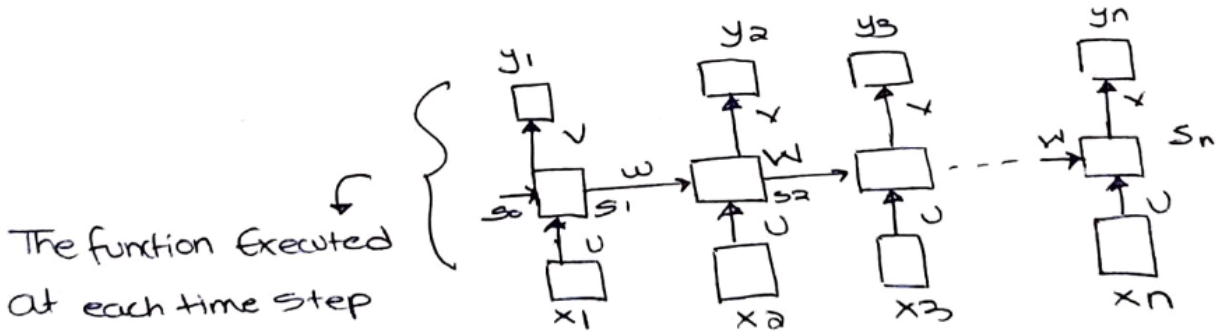


↳ So network is now sensetive to the length of the network.

⇓

Solution

↳ Is to add a reccurent connection to the network.



The function executed at each time step for y_2 because s_0 is the boundary.

$$s_i = \sigma(Ux + Ws_{i-1} + b)$$

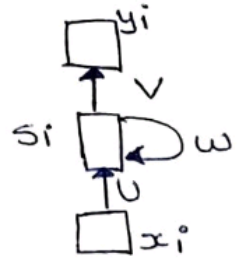
$$y_i = O(Vs_i + c)$$

(OR)

$$y_i = f(x_i, s_i, W, U, V)$$

Representing in more compact way is

↳



⇒ s_i is the state of network at time step i

Back propogation through time

Dimensions of the parameters :

$x_i \in R^n$ (n-dimensional input)

$s_i \in R^d$ (d-dimensional state)

$y_i \in R^K$ (say K classes)

$U \in R^{n \times d}$

$V \in R^{d \times K}$

$W \in R^{d \times d}$

⇒ We train this network by using Back propogation.

Why not standard back propogation Algorithm ??

↳ Ex: Task of Autcompleteion

For simplicity we assume that there are only 4 characters in vocabulary (d, e, p, <stop>)

→ At each time step we need to predict one of these 4 characters

So, suitable output function is Softmax and Suitable loss function is Cross Entropy.

→ Softmax is usually paired with Cross Entropy Loss Function. This Combination simplifies gradients during back propogation & makes the training more efficent.

↳ Suppose we intialize the U, V, W randomly & the network Predicts the probabilities and the true probabilites which we have.

So the two Questions??

① Total loss made by the model??

② How do we back propogate this loss & update the Parameters ($\theta = \{U, V, W, b, c\}$) of the network??

(1)

↳ so the total loss is simply the sum of loss over all time steps.

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} \mathcal{L}_t(\theta)$$

$$\mathcal{L}_t(\theta) = -\log(y_{tc})$$

$y_{tc} \Rightarrow$ predicted probability of true character at time step t

(2)

For Back propagation we need to compute the Gradients w.r.t W, U, V

↳ (i) The loss function w.r.t to derivative $V \Rightarrow \frac{\partial \mathcal{L}(\theta)}{\partial V}$

$$\frac{\partial \mathcal{L}(\theta)}{\partial V} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t(\theta)}{\partial V}$$

(ii) The loss function w.r.t to derivative $W \Rightarrow \frac{\partial \mathcal{L}(\theta)}{\partial W}$

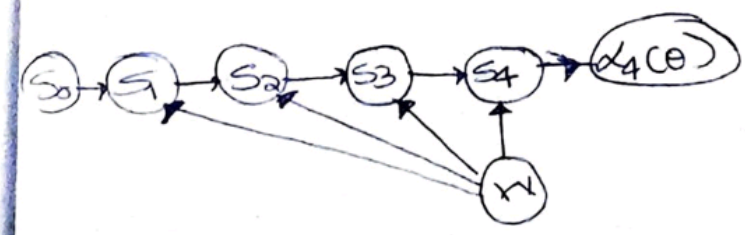
$$\frac{\partial \mathcal{L}(\theta)}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t(\theta)}{\partial W}$$

→ By the Chain rule of derivatives we know that $\frac{\partial \mathcal{L}_t(\theta)}{\partial W}$ is obtained by summing gradients along all paths from $\mathcal{L}_t(\theta)$ to W

⇓

What are the paths connecting $\mathcal{L}_t(\theta)$ to W??

↳ consider $\mathcal{L}_4(\theta)$



⇒ So we have an ordered network. In this each state is computed one at a time. (first S_1 then S_2 & so on)

$$\frac{\partial L_4(\theta)}{\partial w} = \frac{\partial L_4(\theta)}{\partial s_4} \frac{\partial s_4}{\partial w}$$

↓ This is direct ↓ Lets compute this

$$s_4 = \sigma(W s_3 + b)$$

$$\frac{\partial s_4}{\partial w} = s_3$$

↳ It is again depend on W.

→ So, thats the pobln with an Ordered network. We cannot Compute the Gradient of s_4 w.r.t w assuming s_3 is a Constant. s_3 is not a constant again its a function of W & W is the parameter w.r.t computing derivative

↳ In such network the total derivative $\frac{\partial s_4}{\partial w}$ has two parts.

Explicit : $\frac{\partial^+ s_4}{\partial w}$ treating all other inputs as constant

Implicit : Summing over all indirect paths from s_4 to w

Derivation

$$\frac{\partial s_4}{\partial w} = \underbrace{\frac{\partial^+ s_4}{\partial w}}_{\text{Explicit}} + \underbrace{\frac{\partial s_4}{\partial s_3} \frac{\partial s_3}{\partial w}}_{\text{Implicit}}$$

$$= \frac{\partial^+ s_4}{\partial w} + \frac{\partial s_4}{\partial s_3}\left[\frac{\partial^+ s_3}{\partial w} + \frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial w}\right]$$

$$= \frac{\partial^+ s_4}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial^+ s_3}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\left[\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial w}\right]$$

$$= \frac{\partial^+ s_4}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial^+ s_3}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\left[\frac{\partial^+ s_2}{\partial w} + \frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial w}\right]$$

$$= \frac{\partial^+ s_4}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial^+ s_3}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial^+ s_2}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\left[\frac{\partial^+ s_1}{\partial w}\right]$$

For simplicity,

$$\frac{\partial s_4}{\partial w} = \frac{\partial s_4}{\partial s_4}\frac{\partial^+ s_4}{\partial w} + \frac{\partial s_4}{\partial s_3}\frac{\partial^+ s_3}{\partial w} + \frac{\partial s_4}{\partial s_2}\frac{\partial^+ s_2}{\partial w} + \frac{\partial s_4}{\partial s_1}\frac{\partial^+ s_1}{w}$$

$$= \sum_{k=1}^{4} \frac{\partial s_4}{\partial s_k}\frac{\partial^+ s_k}{\partial w} \quad //$$

Finally we have,

$$\frac{\partial L_4(\theta)}{\partial w} = \frac{\partial L_4 \theta}{\partial s_4}\frac{\partial s_4}{\partial w}$$

$$\frac{\partial s_4}{\partial w} = \sum_{k=1}^{4} \frac{\partial s_4}{\partial s_k}\frac{\partial^+ s_k}{\partial w}$$

In General, $\star$

$$\boxed{\frac{\partial L_t(\theta)}{\partial w} = \frac{\partial L_t(\theta)}{\partial s_t}\sum_{k=1}^{t}\frac{\partial s_t}{\partial s_k}\frac{\partial^+ s_k}{\partial w}}$$

↳ This Algorithm is called Back propagation through Time (BPTT) as we back propagate over all these ~~steps~~ previous time steps.

The problem of Exploding & Vanishing

$$\frac{\partial s_t}{\partial k} = \frac{\partial s_t}{\partial s_{t-1}}\frac{\partial s_{t-1}}{\partial s_{t-2}} \cdots \frac{\partial s_{k+1}}{\partial s_k}$$

$$= \prod_{j=k}^{t-1} \frac{\partial s_{j+1}}{\partial s_j} \quad (\text{or}) \quad \frac{\partial s_j}{\partial s_{j-1}}$$

↳ we are intested in this term.

$a_j = W s_{j-1} + b$ ⇒ Preactivation

$s_j = \sigma(a_j)$ ⇒ Hidden representation

$$\frac{\partial s_j}{\partial s_{j-1}} = \frac{\partial s_j}{\partial a_j} \frac{\partial a_j}{\partial s_{j-1}}$$

$$a_j = [a_{j1}, a_{j2}, \cdots a_{jd}]^d$$

$$s_j = [\sigma(a_{j1}), \sigma(a_{j2}) \cdots \sigma(a_{jd})]$$

$$\frac{\partial s_j}{\partial a_j} = \begin{bmatrix} \frac{\partial s_{j1}}{\partial a_{j1}} & \frac{\partial s_{j2}}{\partial a_{j1}} & \cdots \\ \vdots & \ddots & \\ & & \end{bmatrix}$$

$\rightarrow = 0$ because it doesn't depend on that

$$= \begin{bmatrix} \sigma'(a_{j1}) & 0 & 0 & 0 \\ 0 & \sigma'(a_{j2}) & 0 & 0 \\ 0 & 0 & & \\ 0 & 0 & \cdots & \sigma'(a_{jd}) \end{bmatrix}$$

$$= \text{diag}(\sigma'(a_j))$$

$$\frac{\partial s_j}{\partial s_{j-1}} = \frac{\partial s_j}{\partial a_j} \frac{\partial a_j}{\partial s_{j-1}}$$

$$= \text{diag}(\sigma'(a_j)) W$$

For some reason we are intreseted in the Magnitude.

$$\left\| \frac{\partial s_j}{\partial s_{j-1}} \right\| = \left\| \text{diag}(\sigma'(a_j)) W \right\|$$

$$\leq \left\| \text{diag}(\sigma'(a_j)) \right\| \left\| W \right\|$$

$\because$ $\sigma(a_j)$ is bounded function because we are using sigmoid, tanh, $\sigma'(a_j)$ is bounded.

If σ is logistic function then the bound is,

$$\sigma'(a_j) \leq \frac{1}{4} = \gamma \quad [\text{If } \sigma \text{ is logistic}]$$

↓

Max derivative we can get from the curve. [] $\frac{1}{4}$

For Tanh Function, $\leq 1 = \gamma$ [If σ is tanh]

$$\leq \|\text{diag}(\sigma'(a_j))\| \|W\|$$

$\|\text{diag}(\sigma'(a_j))\|$ → So this quantity is bounded and we call it as 'r'

$\|W\|$ → Weight matrix is also bounded because they are the real weights.

$$\leq \gamma\lambda$$

$$\left\| \frac{\partial s_t}{\partial s_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial s_j}{\partial s_{j-1}} \right\|$$

$$\leq \prod_{j=k+1}^{t} \gamma\lambda$$

$$\leq (\gamma\lambda)^{t-k}$$

→ If $\gamma\lambda < 1$ the gradient will vanish

→ If $\gamma\lambda > 1$ the gradient could Explode

$$\therefore \frac{\partial L_t(\theta)}{\partial W} = \frac{\partial L_t(\theta)}{\partial s_t} \sum_{k=1}^{t} \frac{\partial s_t}{\partial s_k} \frac{\partial^+ s_k}{\partial W}$$

This vanishes If this vanishes the entire gradient could vanish.

→ And if Gradients are vanished we have no updates we stuck where we are.

→ If the gradient Explodes. If we have a very large gradient its gone way far away from where you are in the WB plane.

Because your update is $W = W - \eta \nabla W$

this value is large so we are moving somewhere far.

↑ to avoid this remember the (t-k) terms and the problem appears when (t-k) is rather t is close to T & k is closed to 1.

Solution: Don't backpropogate through all the time steps Use an Approximation that if you are at time step n we just look at (n-k) time steps and not all the way back.

↳ Exploding & Vanishing

→ Other thing to avoid Exploding

⇓

In Gradient Descent we are always intrested in the direction. So, we can just normalize it.

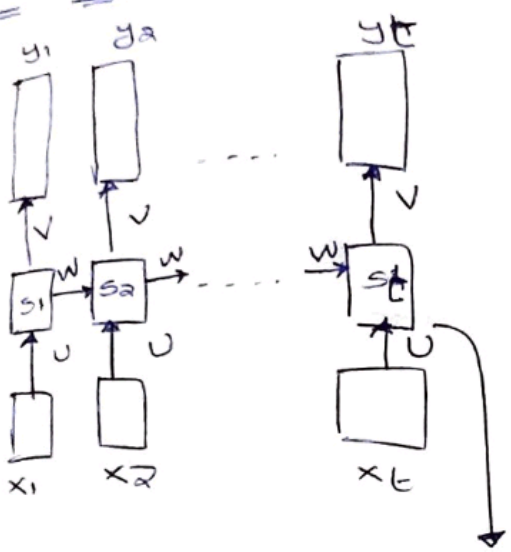
$$\frac{\nabla W}{\|\nabla W\|} < k$$ } clip the Gradients

↳ Just a intution for the Magnitude.

Some Crazy Details (13.5) ⇒ Refer once.

LSTM & GRU

Selective Read, Selective Write, Selective Forget - The White Board Analogy



⇒ The state (s_i) of a RNN records information from all previous time steps

⇒ At each new time step the old information is accumulated by the current input.

But now the issue is that this state is going to be finite size dimension. Now as we keep writing information to that cell we are morphing the information that you had written in the earlier time steps.

⇓

So, the pblm is we want the information from all previous steps but the other hand we just have a finite amount of memory to deal with so, it is bound to read over ridden & the information get morphed so much that completely impossible to say what was the original contribution at time step 1.

↳ So this is the pblm with the ~~RE~~ RNN (Recurrent Neural Network).

↳ Similar pblm occurs when the information flows backwards (back propogation).

↳ It is very hard to assign the responsibility of the error caused at the time step t to arbitary time steps before it very far away time steps it is very hard so this is the vanishing gradient pblm.

⇒ So both during the forward propogation information Variables
Even during the backward propogation information Variables

Analogy

Fixed Size White Board

↳ we follow the following strategy at each time step

① Selectively write on Board
② Selectively read the already written content
③ Selectively forget some content.

Selective write

↳ There may be many steps in the derivation but we may just skip few which are not very imp.

Ex: a=1 c=5

ac = ? ⇒ ac = 5 ✓

We skip a=1 c=5 ac = 1×5 = 5 } These steps

⇓

Because we dont write Everything for the reason we run out of the memory.

Selective read

↳ While writing one step we typically read some of the previous steps we have already written & then what to write next.

Ex: Step ② we computed bd

step ③ bd + a

↳ information of step ② is impo

Vanishes Selective Forget

on Vanishes

↳ Once the board is full we need to delete some obsolete information.

⇓

we typically delete the last useful information.

* Concrete Example Where RNN also need to selectively read Write & forget??

↳ Consider the task of predecting the sentiment of a Review.

⇓

By the time we reach the end of the document the information Obtained from first few words is completely lost.

Ideally we want to

↳ forget the information added by stopwords (a, the, etc)

↳ selectively Read the information added by Previous sentiment bearing words.

↳ selectively write new information from the current wood to the state.

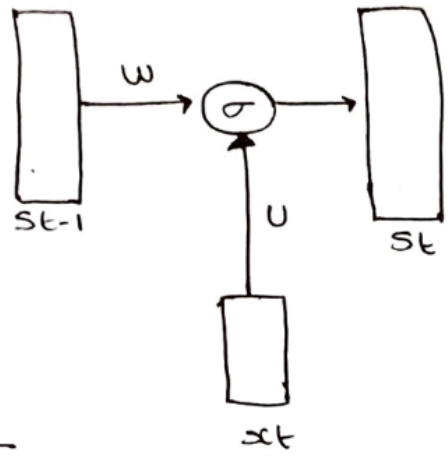
*

So we have a computed state from the RNN S_{t-1}. Now I want to compute a new state of the RNN. And now we have the new information (x_t) & compute a new state (s_t)

⇓

While doing this We want to make sure that we use selective read, write, forget.

Selective Write



$$[S_t = \sigma(WS_{t-1} + Ux_t)]$$

⇒ So now instead of passing S_{t-1} as it is to S_t we want to pass only some portion of it to next state.

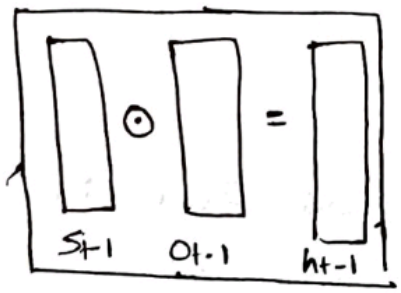
⇒ So whenever we compute S_t we don't want to write the whole of S_{t-1} just want to use selective portions.

⇒ So what we do is we introduce something known as gate. (O_{t-1})

↓

which decides what fraction of each Element of S_{t-1} should be passed to the next state.

⇒ So we take orginal State S_{t-1} and doan Element wise product with gate O_{t-1} (Outputgate) and then write the product to new vector which is h_{t-1}.



⇒ O_{t-1} is restricted b/w 0 & 1

⇒ But how do we compute O_{t-1}? How does the RNN know what fraction of the state to pass on??

⇓

If we want to Learn Something we always introduce a parametric form of that quantity and then learn the parameters of that function.

→ RNN has to learn O_{t-1} along with other parameters (W, U, V)

↳ we compute O_{t-1} and h_{t-1} as

$$O_{t-1} = \sigma\left(W_o \underbrace{h_{t-2}}_{\text{Output at } t-2} + U_o \underbrace{x_{t-1}}_{\text{Input at timestep } t-1} + b_o\right)$$

↳ so we use sigmoids, because we want the fraction to be b/w 0 to 1

$$h_{t-1} = O_{t-1} \odot \sigma(S_{t-1})$$

O_t is called the output gate as it decides how much to pass (write) to the next time step.

Selective Read

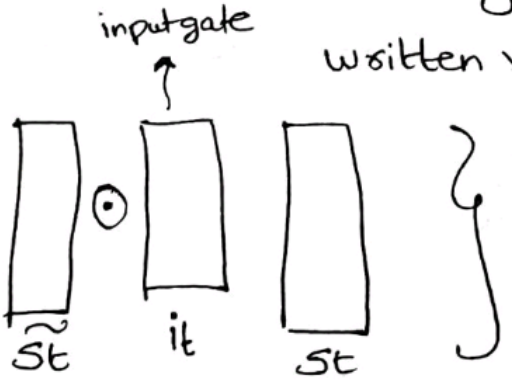
↳ we will now use h_{t-1} to compute the new state at the next time step

↳ we will also use x_t which is the new input at time step t.

$$\bar{S}_t = \sigma(W h_{t-1} + U x_t + b)$$

↓

This Eq form similar to RNN instead S_{t-1} we are using h_{t-1} because h_{t-1} have only selective written values.



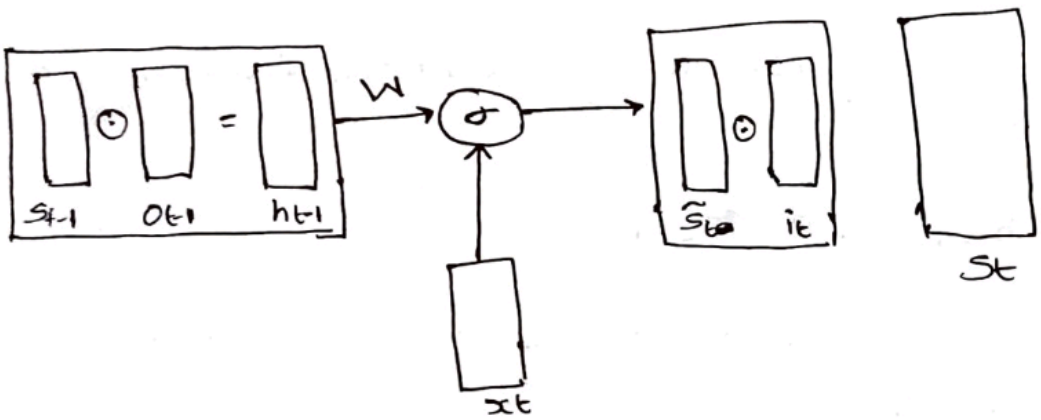
↳ we again need to selectively pass it or to. so again we introduce the gate it is called the Read gate.

↳ You don't want to pass it on as it is to S_t.

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$

$i_t \odot \tilde{S}_t \Rightarrow$ Selective read state information.

So far we have,



Previous state : S_{t-1}

Output gate : $O_{t-1} = \sigma(W_o h_{t-2} + U_o x_{t-1} + b_o)$

Selective Write : $h_{t-1} = O_{t-1} \odot \sigma(S_{t-1})$

Current / Temporary state : $\tilde{S}_t = \sigma(W h_{t-1} + U x_t + b)$

Input gate : $i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$

Selective read : $i_t \odot \tilde{S}_t$

Selective Forget

↳ How do we combine S_{t-1} & $\tilde{S}_t$ to get new state.

Simple Way

↳ $S_t = S_{t-1} + i_t \odot \tilde{S}_t$

↓

We should forget something do not use directly

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

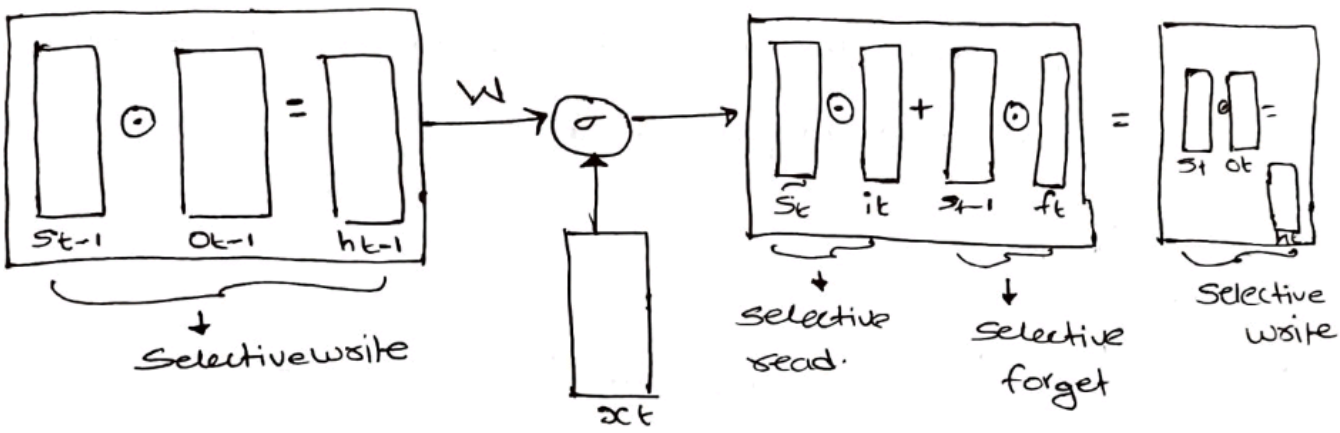
$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$$

Gates

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

States

$$\tilde{s}_t = \sigma(W h_{t-1} + U x_t + b)$$
$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$$
$$h_t = o_t \odot \sigma(s_t)$$



$s_t \rightarrow C_t$ } current state

→ LSTM has many variants which include different no of gates and also different arrangement of gates.

→ This is the most popular variant of LSTM.

Another popular variant in LSTM is Gated Recurrent Units (GRU).

Gates :

$$o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$$

States : $\tilde{s}_t = \sigma(W(o_t \odot s_{t-1}) + U x_t + b)$

$$s_t = (1 - i_t) \odot s_{t-1} + i_t \odot \tilde{s}_t$$

→ No Explicit forget gate (Input gates are tied)

→ The gates depend directly on S_{t-1} & not intermediate h_{t-1}

How LSTM avoid the pblm of Vanishing Gradients??

↳ In Rnn due to the Recurrent network structure there is the Vanishing Gradient pblm.

↳ Even in LSTM we have Recurrent Structure. But Why it doesn't vanishing Gradient??

↓

Intuition

↳ During forward propogation the gates control the flow of information.

↳ They prevent any irrelevant information from being written to the state.

↳ Similarly during back propogation they control the flow of gradients

$$S_t = f_t \odot S_t + i_t \odot S_t$$

↑

If this vanishes in the worst case. We still have the Gradient f_t

↓

This is good Why is this Ok Because f_t decides how much flew in the forward direction.

→ Refer the Video (14.3)

↓↓

Part-1

⇒ The key difference from Vanilla RNN is that flow of information and gradients is controlled by the gates which ensures that the gradient vanish only when they should.

Illustrative proof ⇒ Refer the video (14.3) ⇒ part 2