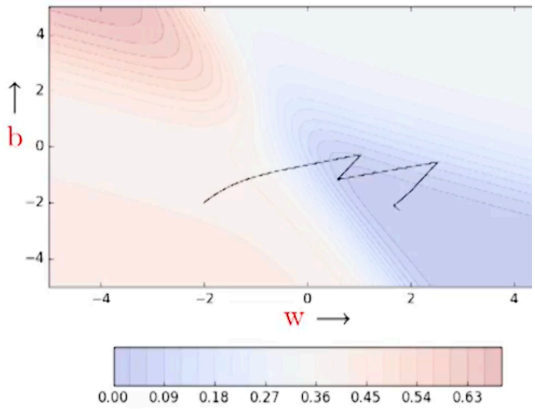
DL Basics2

Wednesday, 20 March 2024 2:02 AM

ADJUSTING LEARNING RATE AND MOMENTUM:

- One could argue that we could have solved the problem of navigating gentle slopes by setting the learning rate high (i.e., blow up the small gradient by multiplying it with a large η)
- Let us see what happens if we set the learning rate to 10
- On the regions which have a steep slope, the already large gradient blows up further
- It would be good to have a learning rate which could adjust to the gradient ... we will see a few such algorithms soon



Tips for initial learning rate ?

- Tune learning rate [Try different values on a log scale: 0.0001, 0.001, 0.01, 0.1. 1.0]
- Run a few epochs with each of these and figure out a learning rate which works best
- Now do a finer search around this value [for example, if the best learning rate was 0.1 then now try some values around it: 0.05, 0.2, 0.3]
- Disclaimer: these are just heuristics ... no clear winner strategy

Tips for annealing learning rate

- **Step Decay:**
  - Halve the learning rate after every 5 epochs or
  - Halve the learning rate after an epoch if the validation error is more than what it was at the end of the previous epoch
- **Exponential Decay:** $\eta = \eta_0^{-kt}$ where η_0 and k are hyperparameters and t is the step number
- **1/t Decay:** $\eta = \frac{\eta_0}{1+kt}$ where η_0 and k are hyperparameters and t is the step number

number

Tips for momentum

- The following schedule was suggested by Sutskever *et. al.*, 2013

$$\mu_t = min(1 - 2^{-1-log_2(\lfloor t/250 \rfloor + 1)}, \mu_{max})$$

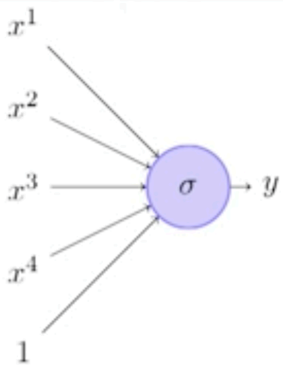
where, μ_{max} was chosen from $\{0.999, 0.995, 0.99, 0.9, 0\}$

Line search :

- In practice, often a line search is done to find a relatively better value of η
- Update w using different values of η
- Now retain that updated value of w which gives the lowest loss
- Esentially at each step we are trying to use the best η value from the available choices
- What's the flipside? We are doing many more computations in each step
- We will come back to this when we talk about second order optimization methods

```
def do_line_search_gradient_descent():
    w, b, etas = init_w, init_b, [0.1, 0.5, 1.0, 5.0, 10.
    for i in range(max_epochs):
        dw, db = 0, 0
        for x,y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        min_error = 10000 #some large value
        best_w, best_b = w, b
        for eta in etas:
            tmp_w = w - eta * dw
            tmp_b = b - eta * db
            if error(tmp_w, tmp_b) < min_error:
                best_w = tmp_w
                best_b = tmp_b
                min_error = error(tmp_w, tmp_b)
        w, b = best_w, best_b
```

GD WITH ADAPTIVE LR: ADA GRAD



$$y = f(x) = \frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}}$$

$$\mathbf{x} = \{x^1, x^2, x^3, x^4\}$$

$$\mathbf{w} = \{w^1, w^2, w^3, w^4\}$$

- Given this network, it should be easy to see that given a single point $(\mathbf{x}, y)$...
- $\nabla w^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$
- $\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2$ ... so on
- If there are n points, we can just sum the gradients over all the n points to get the total gradient
- What happens if the feature x^2 is very sparse? (*i.e.*, if its value is 0 for most inputs)
- ∇w^2 will be 0 for most inputs (see formula) and hence w^2 will not get enough updates
- If x^2 happens to be sparse as well as important we would want to take the updates to w^2 more seriously
- Can we have a different learning rate for each parameter which takes care of the frequency of features ?

Intuition

- Decay the learning rate for parameters in proportion to their update history (more updates means more decay)

Update rule for Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

... and a similar set of equations for b_t

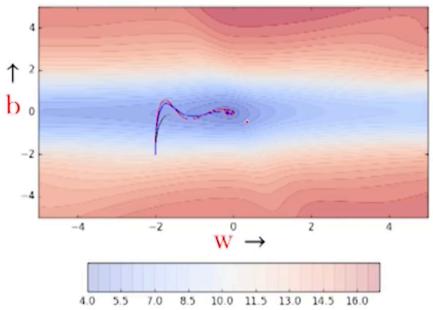
- To see this in action we need to first create some data where one of the features is sparse
- How would we do this in our toy network ? Take some time to think about it
- Well, our network has just two parameters (w and b). Of these, the input/feature corresponding to b is always on (so can't really make it sparse)
- The only option is to make x sparse
- **Solution:** We created 100 random (x, y) pairs and then for roughly 80% of these pairs we set x to 0

```
def do_adagrad():
    w, b, eta = init_w, init_b, 0.1
    v_w, v_b, eps = 0, 0, 1e-8
    for i in range(max_epochs):
        dw, db = 0, 0
        for x,y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        v_w = v_w + dw**2
        v_b = v_b + db**2

        w = w - (eta / np.sqrt(v_w + eps)) * dw
        b = b - (eta / np.sqrt(v_b + eps)) * db
```

- GD (black), momentum (red) and NAG (blue)
- There is something interesting that these 3 algorithms are doing for this dataset. Can you spot it?
- Initially, all three algorithms are moving mainly along the vertical (b) axis and there is very little movement along the horizontal (w) axis
- Why? Because in our data, the feature corresponding to w is sparse and hence w undergoes very few updates ...on the other hand b is very dense and undergoes many updates
- Such sparsity is very common in large neural networks containing 1000s of input features and hence we need to address it

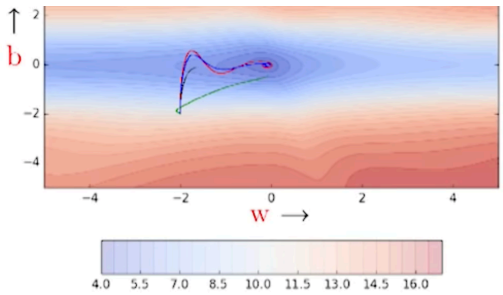


- Let's see what Adagrad does....

- By using a parameter specific learning rate it ensures that despite sparsity w gets a higher

learning rate and hence larger updates

- Further, it also ensures that if b undergoes a lot of updates its effective learning rate decreases because of the growing denominator
- In practice, this does not work so well if we remove the square root from the denominator (something to ponder about)
- What's the flipside? over time the effective learning rate for b will decay to an extent that there will be no further updates to b
- Can we avoid this?



Intuition

- Adagrad decays the learning rate very aggressively (as the denominator grows)
- As a result after a while the frequent parameters will start receiving very small updates because of the decayed learning rate
- To avoid this why not decay the denominator and prevent its rapid growth

Update rule for RMSProp

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

$\beta \geq 0.95$

$V_1 = 0.05 \nabla w_1^2$

$V_2 = 0.95\,0.05 \nabla w_1^2 + 0.05 \nabla w_2^2$

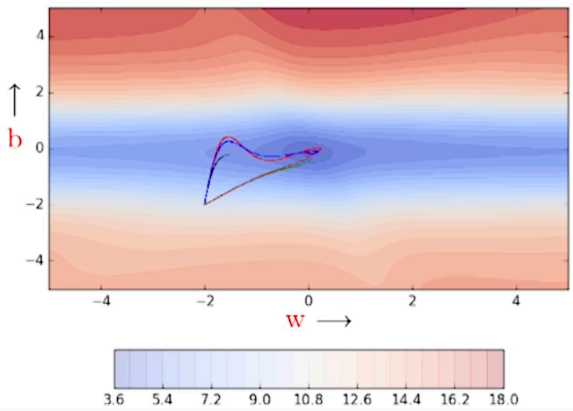
$V_t = \nabla w_0 \quad \nabla w_1$

... and a similar set of equations for b_t

```
def do_rmsprop() :
    w, b, eta = init_w, init_b, 0.1
    v_w, b_updates, eps, beta1 = 0, 0, 1e-8, 0.9
    for i in range(max_epochs) :
        dw, db  = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        v_w = beta1 * v_w + (1 - beta1) dw**2
        v_b = beta1 * v_b + (1 - beta1) db**2

        w = w - (eta / np.sqrt(v_w + eps)) * dw
        b = b - (eta / np.sqrt(v_b + eps)) * db
```



- Adagrad got stuck when it was close to convergence (it was no longer able to move in the vertical (b) direction because of the decayed learning rate)
- RMSProp overcomes this problem by being less aggressive on the decay

5)

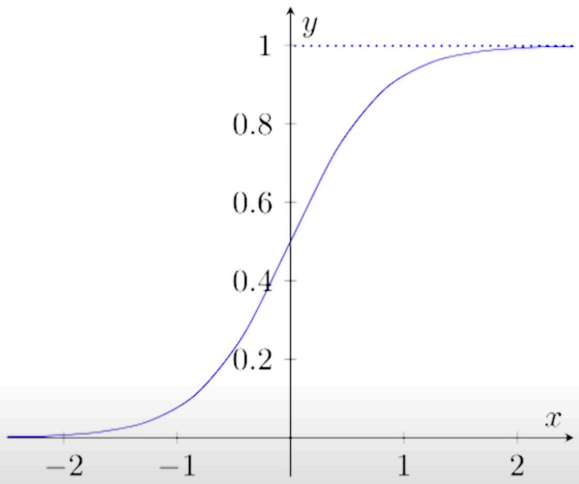
Things to remember

- Training Neural Networks is a ***Game of Gradients*** (played using any of the existing gradient based approaches that we discussed)
- The gradient tells us the responsibility of a parameter towards the loss
- The gradient w.r.t. a parameter is proportional to the input to the parameters (recall the "..... $* x$" term or the "..... $* h_i$" term in the formula for ∇w_i)

Things to Remember

- Sigmoids are bad
- ReLU is more or less the standard unit for Convolutional Neural Networks
- Can explore Leaky ReLU/Maxout/ELU
- tanh sigmoids are still used in LSTMs/RNNs (we will see more on this later)

6)ACTIVATION FUNCTIONS:

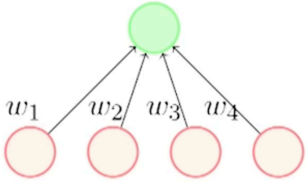


- A sigmoid neuron is said to have saturated when $\sigma(x) = 1$ or $\sigma(x) = 0$
- What would the gradient be at saturation?
- Well it would be 0 (you can see it from the plot or from the formula that we derived)

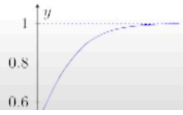
Saturated neurons thus cause the gradient to vanish.

Subtitles/closed captions (c)

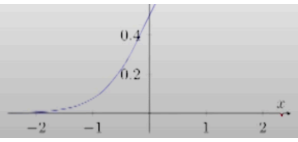
Saturated neurons thus cause the gradient to vanish.



$\sigma(\sum_{i=1}^{4} w_i x_i)$



- But why would the neurons saturate ?
- Consider what would happen if we use sigmoid neurons and initialize the weights to very high values ?
- The neurons will saturate very quickly
- The gradients will vanish and the training will stall (more on this later)



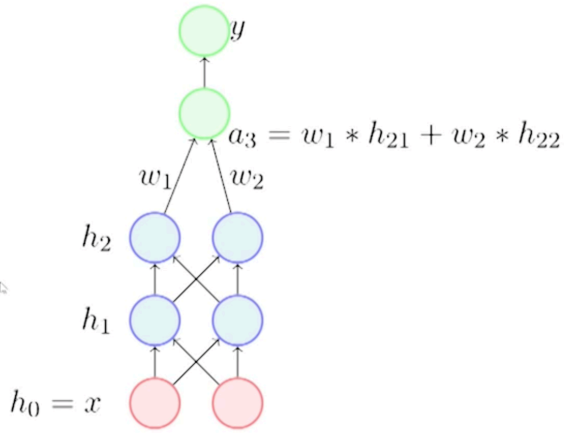
- Saturated neurons cause the gradient to vanish
- Sigmoids are not zero centered
- Consider the gradient w.r.t. w_1 and w_2

$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; h_{21}$$

$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; h_{22}$$

- Note that h_{21} and h_{22} are between $[0, 1]$ (*i.e.*, they are both positive)
- So if the first common term (in red) is positive (negative) then both ∇w_1 and ∇w_2 are positive (negative)

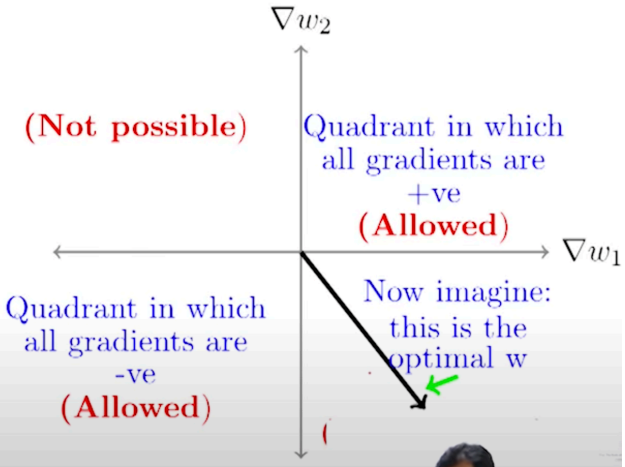
- Why is this a problem??



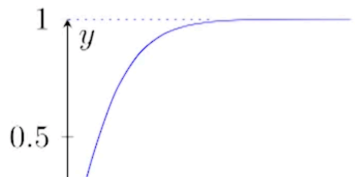
- Essentially, either all the gradients at a layer are positive or all the gradients at a layer are negative

- Saturated neurons cause the gradient to vanish
- Sigmoids are not zero centered

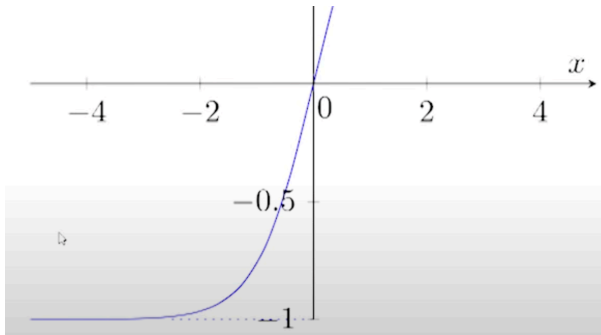
- This restricts the possible update directions



tanh(x)



- Compresses all its inputs to the range [-1,1]
- Zero centered
- What is the derivative of this function?



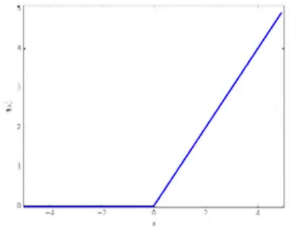
$$\frac{\partial tanh(x)}{\partial x} = (1 - tanh^2(x))$$

- The gradient still vanishes at saturation
- Also computationally expensive

f(x) = tanh(x)

Relu:

ReLU



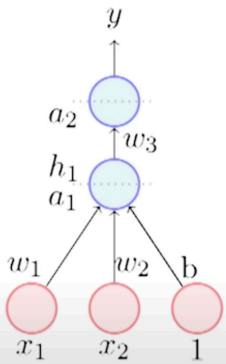
Advantages of ReLU

- Does not saturate in the positive region
- Computationally efficient
- In practice converges much faster than $sigmoid/tanh$[1]

$$f(x) = max(0, x)$$

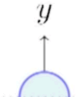
- In practice there is a caveat
- Let's see what is the derivative of ReLU(x)

$$\frac{\partial ReLU(x)}{\partial x} = 0 \quad if \quad x < 0$$
$$= 1 \quad if \quad x > 0$$

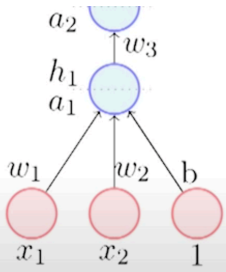


- Now consider the given network
- What would happen if at some point a large gradient causes the bias b to be updated to a large negative value?

$$w_1x_1 + w_2x_2 + b < 0 \quad [if \quad b << 0]$$



- The neuron would output 0 [dead neuron]
- Not only would the output be 0 but during



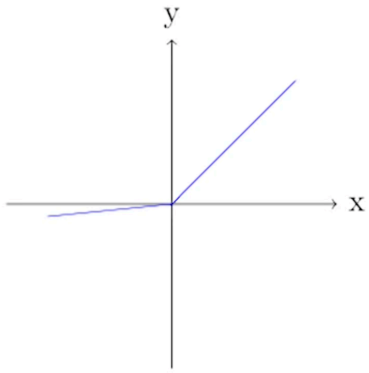
backpropagation even the gradient $\frac{\partial h_1}{\partial a_1}$ would be zero

- The weights w_1, w_2 and b will not get updated [$\because$ there will be a zero term in the chain rule]

$$\nabla w_1 = \frac{\partial \mathscr{L}(\theta)}{\partial y} . \frac{\partial y}{\partial a_2} . \frac{\partial a_2}{\partial h_1} . \frac{\partial h_1}{\partial a_1} . \frac{\partial a_1}{\partial w_1}$$

- The neuron will now stay dead forever!!

Leaky ReLU



f(x) = max(0.01x,x)

- No saturation
- Will not die ($0.01x$ ensures that at least a small gradient will flow through)
- Computationally efficient
- Close to zero centered ouputs

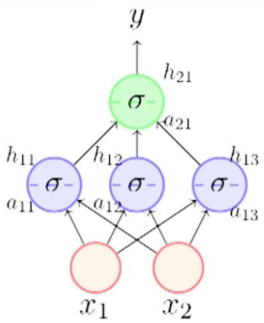
Parametric ReLU

$f(x) = \max(\alpha x, x)$

α *is a parameter of the model*

α *will get updated durir*

7)WEIGHT INITIALIZATION STRATEGY:

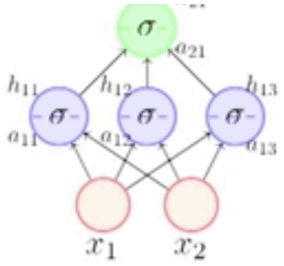


$$a_{11} = w_{11}x_1 + w_{12}x_2$$
$$a_{12} = w_{21}x_1 + w_{22}x_2$$
$$\therefore a_{11} = a_{12} = 0$$
$$\therefore h_{11} = h_{12}$$

- Now what will happen during back propagation?

$$\nabla w_{11} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} . \frac{\partial y}{\partial h_{11}} . \frac{\partial h_{11}}{\partial a_{11}} . x_1$$
$$\nabla w_{21} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} . \frac{\partial y}{\partial h_{12}} . \frac{\partial h_{12}}{\partial a_{12}} . x_1$$
$$but \quad h_{11} = h_{12}$$
$$and \quad a_{12} = a_{12}$$
$$\therefore \nabla w_{11} = \nabla w_{21}$$

- Hence both the weights will get the

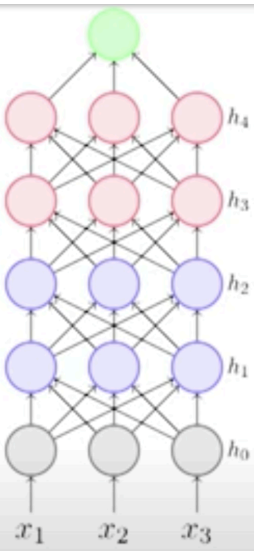


$$a_{11} = w_{11}x_1 + w_{12}x_2$$
$$a_{12} = w_{21}x_1 + w_{22}x_2$$
$$\therefore a_{11} = a_{12} = 0$$
$$\therefore h_{11} = h_{12}$$

same update and remain equal

- Infact this symmetry will never break during training
- The same is true for w_{12} and w_{22}
- And for all weights in layer 2 (infact, work out the math and convince yourself that all the weights in this layer will remain equal)
- This is known as the **symmetry breaking problem**
- This will happen if all the weights in a network are initialized to the **same value**

BATCH NORMALIZATION::



- To understand the intuition behind Batch Normalization let us consider a deep network
- Let us focus on the learning process for the weights between these two layers
- Typically we use mini-batch algorithms
- What would happen if there is a constant change in the distribution of h_3
- In other words what would happen if across mini-batches the distribution of h_3 keeps changing
- Would the learning process be easy or hard?