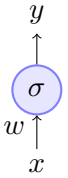
CS7015 (Deep Learning) : Lecture 9

Greedy Layerwise Pre-training, Better activation functions, Better weight initialization methods, Batch Normalization

Mitesh M. Khapra

Department of Computer Science and Engineering
Indian Institute of Technology Madras

Module 9.1 : A quick recap of training deep neural networks

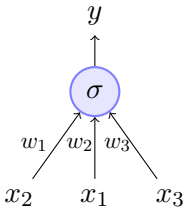


- We already saw how to train this network

$$w = w - \eta \nabla w \quad where,$$

$$\nabla w = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial w}$$

$$= (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x$$



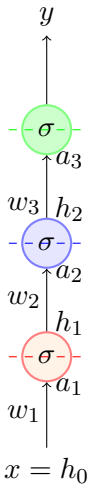
- What about a wider network with more inputs:

$$w_1 = w_1 - \eta \nabla w_1$$

$$w_2 = w_2 - \eta \nabla w_2$$

$$w_3 = w_3 - \eta \nabla w_3$$

$$where, \nabla w_i = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x_i$$



$a_i = w_i h_{i-1}; h_i = \sigma(a_i)$

$a_1 = w_1 * x = w_1 * h_0$

- What if we have a deeper network ?
- We can now calculate ∇w_1 using chain rule:

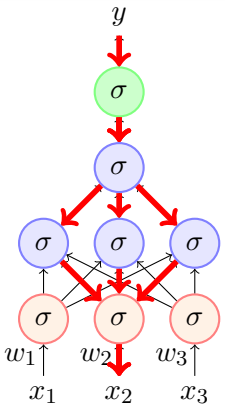
$$\frac{\partial \mathscr{L}(\mathbf{w})}{\partial w_1} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} . \frac{\partial y}{\partial a_3} . \frac{\partial a_3}{\partial h_2} . \frac{\partial h_2}{\partial a_2} . \frac{\partial a_2}{\partial h_1} . \frac{\partial h_1}{\partial a_1} . \frac{\partial a_1}{\partial w_1}$$

$$= \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} * ..... * h_0$$

- In general,

$$\nabla w_i = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} * ..... * h_{i-1}$$

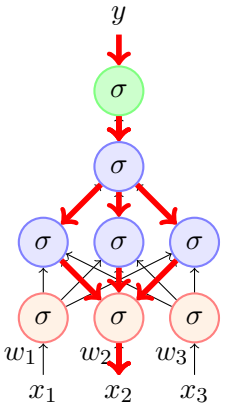
- Notice that ∇w_i is proportional to the corresponding input h_{i-1} (we will use this fact later)



- What happens if we have a network which is deep and wide?
- How do you calculate $\nabla w_2 =$?
- It will be given by chain rule applied across multiple paths (We saw this in detail when we studied **back propagation**)

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)



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

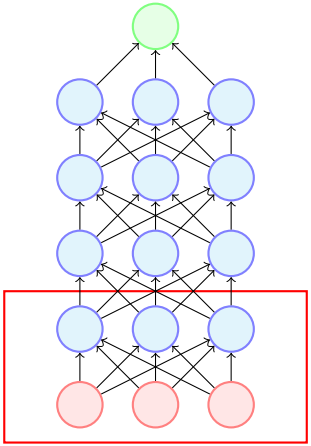
- Backpropagation was made popular by Rumelhart et.al in 1986
- However when used for really deep networks it was not very successful
- In fact, till 2006 it was very hard to train very deep networks
- Typically, even after a large number of epochs the training did not converge

Module 9.2 : Unsupervised pre-training

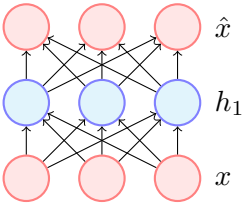
- What has changed now? How did Deep Learning become so popular despite this problem with training large networks?
- Well, until 2006 it wasn't so popular
- The field got revived after the seminal work of Hinton and Salakhutdinov in 2006

[1] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. Science, 313(5786):504–507, July 2006.

Let's look at the idea of unsupervised pre-training introduced in this paper ...
(note that in this paper they introduced the idea in the context of RBMs but we will discuss it in the context of Autoencoders)

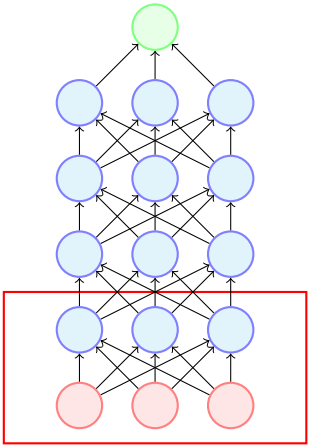


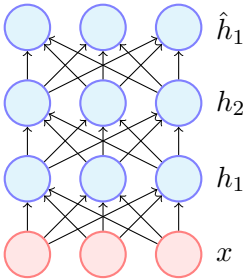
reconstruct x



$$\min \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} (\hat{x}_{ij} - x_{ij})^2$$

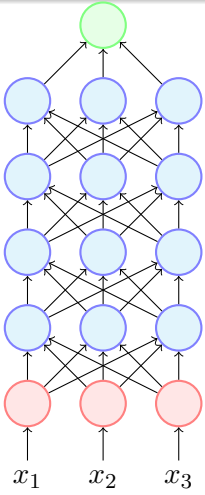
- Consider the deep neural network shown in this figure
- Let us focus on the first two layers of the network (x and h_1)
- We will first train the weights between these two layers using an **unsupervised objective**
- Note that we are trying to reconstruct the input (x) from the hidden representation (h_1)
- We refer to this as an unsupervised objective because it does not involve the output label (y) and only uses the input data (x)





$$\min \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} (\hat{h}_{1_{ij}} - h_{1_{ij}})^2$$

- At the end of this step, the weights in layer 1 are trained such that h_1 captures an abstract representation of the input x
- We now fix the weights in layer 1 and repeat the same process with layer 2
- At the end of this step, the weights in layer 2 are trained such that h_2 captures an abstract representation of h_1
- We continue this process till the last hidden layer (*i.e.,* the layer before the output layer) so that each successive layer captures an abstract representation of the previous layer



$$\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} (y_i - f(x_i))^2$$

- After this layerwise pre-training, we add the output layer and train the whole network using the task specific objective
- Note that, in effect we have initialized the weights of the network using the greedy unsupervised objective and are now fine tuning these weights using the supervised objective

Why does this work better?

- Is it because of better optimization?
- Is it because of better regularization?

Let's see what these two questions mean and try to answer them based on some (among many) existing studies[1,2]

[1]The difficulty of training deep architectures and effect of unsupervised pre-training - Erhan et al,2009

[2]Exploring Strategies for Training Deep Neural Networks, Larocelle et al,2009

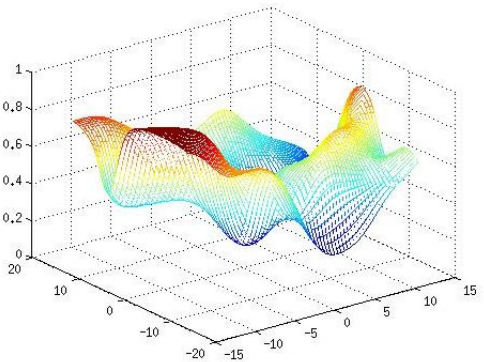
Why does this work better?

- Is it because of better optimization?
- Is it because of better regularization?

- What is the optimization problem that we are trying to solve?

$$\text{minimize} \quad \mathscr{L}(\theta) = \frac{1}{m}\sum_{i=1}^{m}(y_i - f(x_i))^2$$

- Is it the case that in the absence of unsupervised pre-training we are not able to drive $\mathscr{L}(\theta)$ to 0 even for the training data (hence poor optimization) ?
- Let us see this in more detail ...

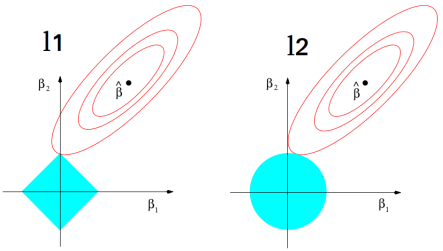


- The error surface of the supervised objective of a Deep Neural Network is highly non-convex
- With many hills and plateaus and valleys
- Given that large capacity of DNNs it is still easy to land in one of these 0 error regions
- Indeed Larochelle et.al.[1] show that if the last layer has large capacity then $\mathscr{L}(\theta)$ goes to 0 even without pre-training
- However, if the capacity of the network is small, unsupervised pre-training helps

[1] Exploring Strategies for Training Deep Neural Networks, Larocelle et al,2009

Why does this work better?

- Is it because of better optimization?
- Is it because of better regularization?



- What does regularization do? It constrains the weights to certain regions of the parameter space
- L-1 regularization: constrains most weights to be 0
- L-2 regularization: prevents most weights from taking large values

[1]Image Source:The Elements of Statistical Learning-T. Hastie, R. Tibshirani, and J. Friedman, Pg 71

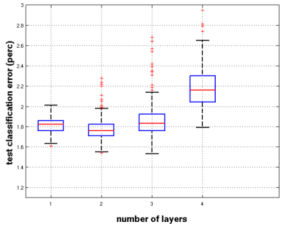
- **Unsupervised objective:**

$$\Omega(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} (x_{ij} - \hat{x}_{ij})^2$$

- We can think of this unsupervised objective as an additional constraint on the optimization problem
- **Supervised objective:**

$$\mathscr{L}(\theta) = \frac{1}{m} \sum_{i=1}^{m} (y_i - f(x_i))^2$$

- Indeed, pre-training constrains the weights to lie in only certain regions of the parameter space
- Specifically, it constrains the weights to lie in regions where the characteristics of the data are captured well (as governed by the unsupervised objective)
- This unsupervised objective ensures that that the learning is not greedy w.r.t. the supervised objective (and also satisfies the unsupervised objective)



- Some other experiments have also shown that pre-training is more robust to random initializations
- One accepted hypothesis is that pre-training leads to better weight initializations (so that the layers capture the internal characteristics of the data)

[1]The difficulty of training deep architectures and effect of unsupervised pre-training - Erhan et al,2009

So what has happened since 2006-2009?

Deep Learning has evolved

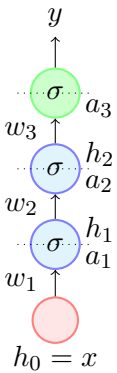
- Better optimization algorithms
- Better regularization methods
- Better activation functions
- Better weight initialization strategies

Module 9.3 : Better activation functions

Deep Learning has evolved

- Better optimization algorithms
- Better regularization methods
- Better activation functions
- Better weight initialization strategies

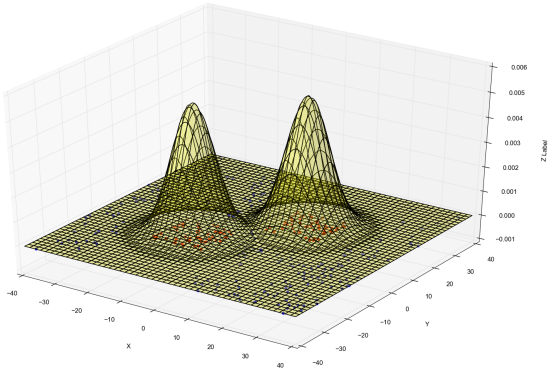
- Before we look at activation functions, let's try to answer the following question: "What makes Deep Neural Networks powerful ?"



- Consider this deep neural network
- Imagine if we replace the sigmoid in each layer by a simple linear transformation

$$y = (w_4 * (w_3 * (w_2 * (w_1 x))))$$

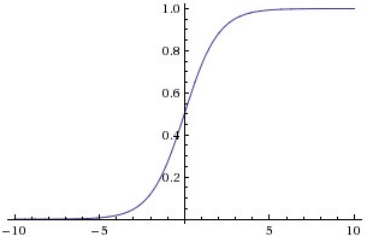
- Then we will just learn y as a linear transformation of x
- In other words we will be constrained to learning linear decision boundaries
- We cannot learn arbitrary decision boundaries



- In particular, a deep linear neural network cannot learn such boundaries
- But a deep non linear neural network can indeed learn such boundaries (recall Universal Approximation Theorem)

- Now let's look at some non-linear activation functions that are typically used in deep neural networks (Much of this material is taken from Andrej Karpathy's lecture notes [1])

[1] http://cs231n.github.io



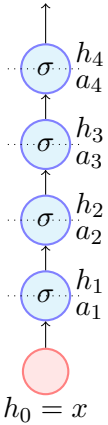
Sigmoid

- $\sigma(x) = \frac{1}{1+e^{-x}}$
- As is obvious, the sigmoid function compresses all its inputs to the range [0,1]
- Since we are always interested in gradients, let us find the gradient of this function

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

(you can easily derive it)

- Let us see what happens if we use sigmoid in a deep network

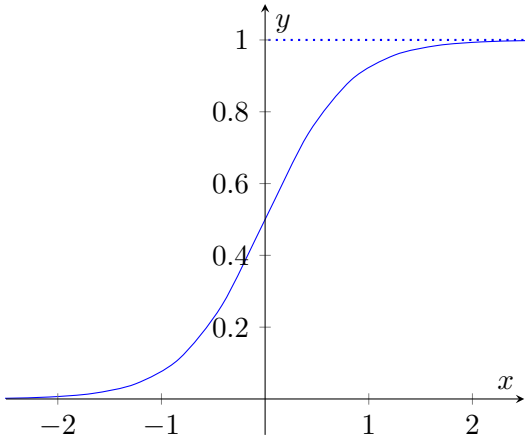


$$a_3 = w_2 h_2$$
$$h_3 = \sigma(a_3)$$

- While calculating ∇w_2 at some point in the chain rule we will encounter

$$\frac{\partial h_3}{\partial a_3} = \frac{\partial \sigma(a_3)}{\partial a_3} = \sigma(a_3)(1 - \sigma(a_3))$$

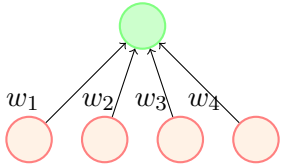
- What is the consequence of this ?
- To answer this question let us first understand the concept of saturated neuron ?



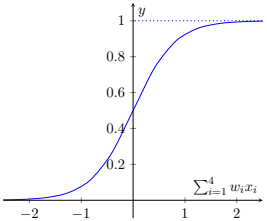
- A sigmoid neuron is said to have saturated when $\sigma(x) = 1$ or $\sigma(x) = 0$
- What would the gradient be at saturation?
- Well it would be 0 (you can see it from the plot or from the formula that we derived)

Saturated neurons thus cause the gradient to vanish.

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)

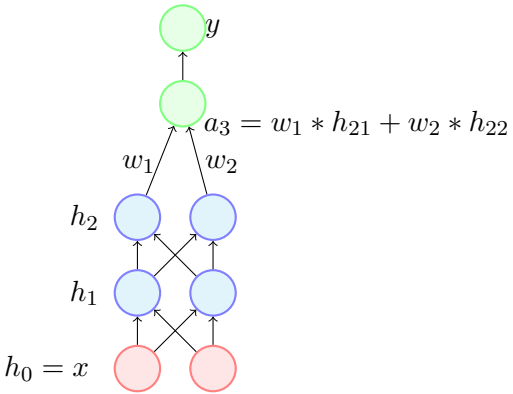
- Saturated neurons cause the gradient to vanish
- Sigmoids are not zero centered
- Consider the gradient w.r.t. w_1 and w_2

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

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

- 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)

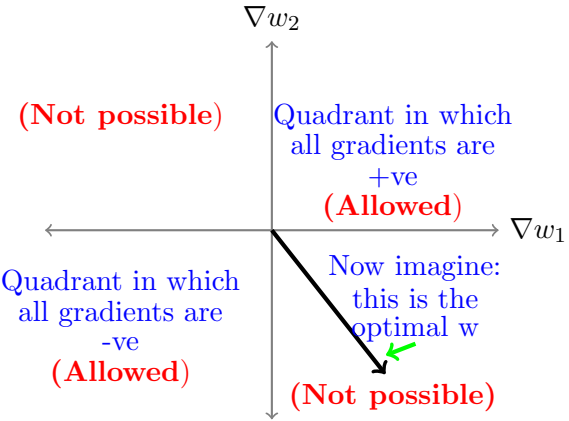
- Why is this a problem??



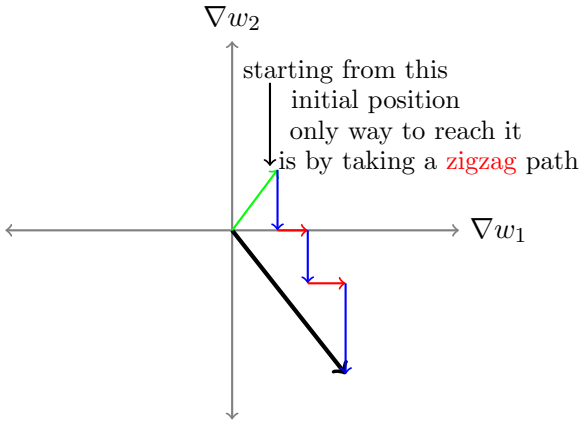
- Essentially, either all the gradients at a layer are positive or all the gradients at a layer are negative

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

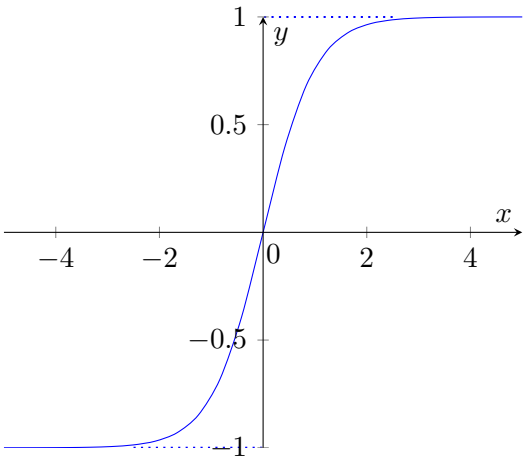
- This restricts the possible update directions



- Saturated neurons cause the gradient to vanish
- Sigmoids are not zero centered
- And lastly, sigmoids are computationally expensive (because of $\exp(x)$)



tanh(x)



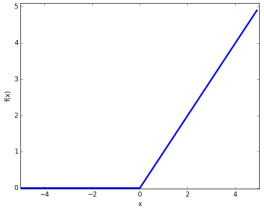
f(x) = tanh(x)

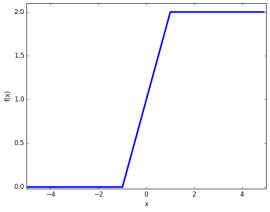
- Compresses all its inputs to the range [-1,1]
- Zero centered
- What is the derivative of this function?

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

- The gradient still vanishes at saturation
- Also computationally expensive

ReLU

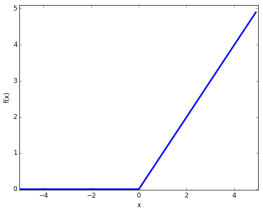




- Is this a non-linear function?
- Indeed it is!
- In fact we can combine two ReLU units to recover a piecewise linear approximation of the sigmoid function

$$f(x) = max(0, x)$$
$$f(x) = max(0, x + 1) - max(0, x - 1)$$

ReLU

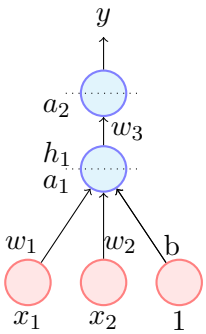


$$f(x) = max(0, x)$$

Advantages of ReLU

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

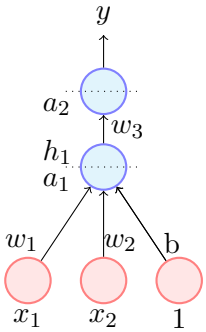
[1] ImageNet Classification with Deep Convolutional Neural Networks- Alex Krizhevsky Ilya Sutskever, Geoffrey E. Hinton, 2012



- 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?

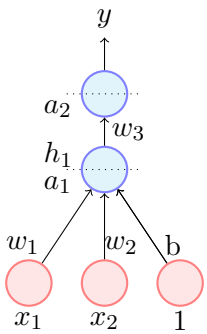


$$w_1 x_1 + w_2 x_2 + b < 0 \quad [if \quad b << 0]$$

- 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]

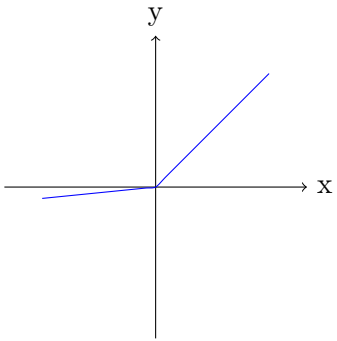
$$\nabla w_1 = \frac{\partial \mathscr{L}(\theta)}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$

- The neuron will now stay dead forever!!



- In practice a large fraction of ReLU units can die if the learning rate is set too high
- It is advised to initialize the bias to a positive value (0.01)
- Use other variants of ReLU (as we will soon see)

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

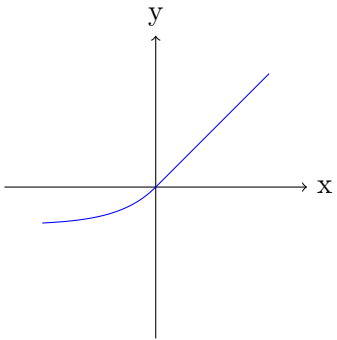
Parametric ReLU

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

α *is a parameter of the model*

α *will get updated during backpropagation*

Exponential Linear Unit



- All benefits of ReLU
- $ae^x - 1$ ensures that at least a small gradient will flow through
- Close to zero centered outputs
- Expensive (requires computation of exp(x))

$$f(x) = x \quad if \quad x > 0$$
$$= ae^x - 1 \quad if \quad x \leq 0$$

Maxout Neuron

$$max(w_1^T x + b_1, w_2^T x + b_2)$$

- Generalizes ReLU and Leaky ReLU
- No saturation! No death!
- Doubles the number of parameters

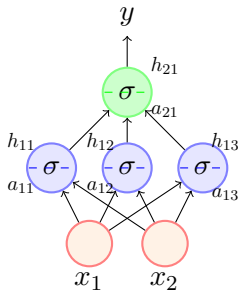
Things to Remember

- 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)

Module 9.4 : Better initialization strategies

Deep Learning has evolved

- Better optimization algorithms
- Better regularization methods
- Better activation functions
- Better weight initialization strategies



$$a_{11} = w_{11}x_1 + w_{12}x_2$$

$$a_{12} = w_{21}x_1 + w_{22}x_2$$

$$\therefore a_{11} = a_{12} = 0$$

$$\therefore h_{11} = h_{12}$$

- What happens if we initialize all weights to 0?
- All neurons in layer 1 will get the same activation
- Now what will happen during back propagation?

$$\nabla w_{11} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} . \frac{\partial y}{\partial h_{11}} . \frac{\partial h_{11}}{\partial a_{11}} . x_1$$

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

$$but \quad h_{11} = h_{12}$$

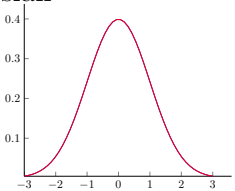
$$and \quad a_{12} = a_{12}$$

$$\therefore \nabla w_{11} = \nabla w_{21}$$

```
D = np.random.randn(1000,500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)

act = {'relu':lambda x: np.maximum(0, x), 'tanh': lambda x: np.tanh(x),
       'sigmoid':lambda x: 1/ (1 + np.exp(-x))}
Hs = {}

for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1]
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) *  0.01

    H = np.dot(X, W)
    H = act[nonlinearities[i]](H)
    Hs[i] = H
```

We will now consider a feedforward network with:

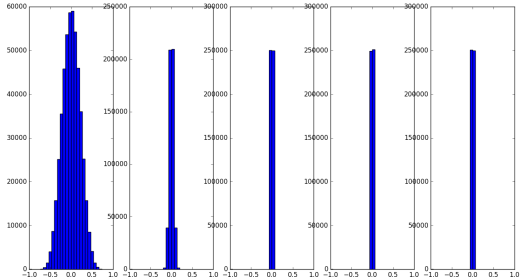
- input: 1000 points, each $\in R^{500}$
- input data is drawn from unit Gaussian



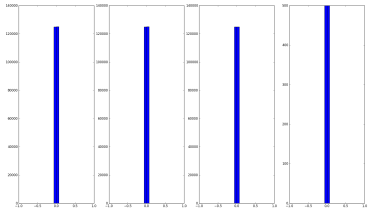
- the network has 5 layers
- each layer has 500 neurons
- we will run forward propagation on this network with different weight initializations

```
W = np.random.randn(fan_in, fan_out) * 0.01
```

- Let's try to initialize the weights to small random numbers
- We will see what happens to the activation across different layers

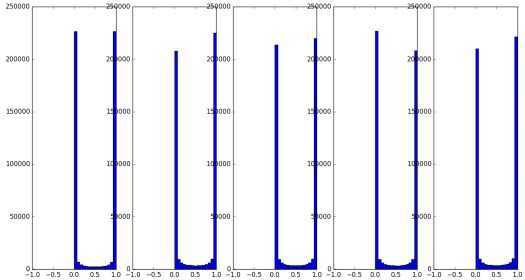


tanh activation functions



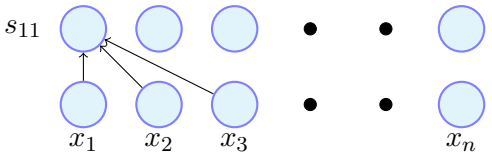
- What will happen during back propagation?
- Recall that ∇w_1 is proportional to the activation passing through it
- If all the activations in a layer are very close to 0, what will happen to the gradient of the weights connecting this layer to the next layer?
- They will all be close to 0 (vanishing gradient problem)

```
W = np.random.randn(fan_in, fan_out)
```



sigmoid activations with large weights

- Let us try to initialize the weights to large random numbers
- Most activations have saturated
- What happens to the gradients at saturation?
- They will all be close to 0 (vanishing gradient problem)



- Let us try to arrive at a more principled way of initializing weights

$$s_{11} = \sum_{i=1}^{n} w_{1i}x_i$$

$$Var(s_{11}) = Var(\sum_{i=1}^{n} w_{1i}x_i) = \sum_{i=1}^{n} Var(w_{1i}x_i)$$

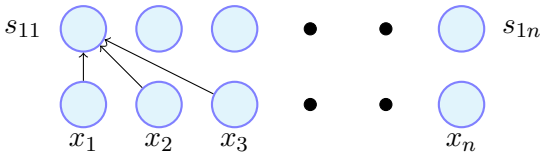
$$= \sum_{i=1}^{n} \big[(E[w_{1i}])^2 Var(x_i)$$

$$+ (E[x_i])^2 Var(w_{1i}) + Var(x_i)Var(w_{1i})\big]$$

$$= \sum_{i=1}^{n} Var(x_i)Var(w_{1i})$$

$$= (nVar(w))(Var(x))$$

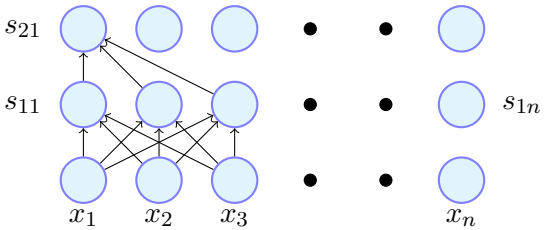
- [Assuming 0 Mean inputs and weights]
- [Assuming $Var(x_i) = Var(x) \forall i$]
- [Assuming $Var(w_{1i}) = Var(w) \forall i$]



- In general,

$$Var(S_{1i}) = (nVar(w))(Var(x))$$

- What would happen if $nVar(w) \gg 1$?
- The variance of S_{1i} will be large
- What would happen if $nVar(w) \to 0$?
- The variance of S_{1i} will be small



$$\boxed{Var(S_{i1}) = nVar(w_1)Var(x)}$$

- Let us see what happens if we add one more layer
- Using the same procedure as above we will arrive at

$$Var(s_{21}) = \sum_{i=1}^{n} Var(s_{1i})Var(w_{2i})$$

$$= nVar(s_{1i})Var(w_2)$$

$$Var(s_{21}) \propto [nVar(w_2)][nVar(w_1)]Var(x)$$

$$\propto [nVar(w)]^2 Var(x)$$

Assuming weights across all layers have the same variance

$$\boxed{Var(az) = a^2(Var(z))}$$

- In general,

$$Var(s_{ki}) = [nVar(w)]^k Var(x)$$

- To ensure that variance in the output of any layer does not blow up or shrink we want:

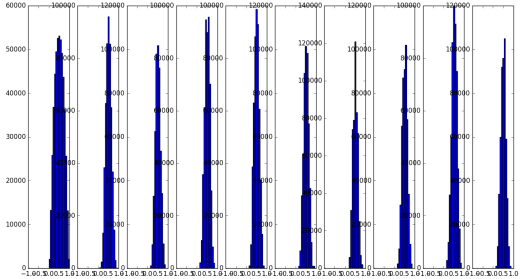
$$n\,Var(w) = 1$$

- If we draw the weights from a unit Gaussian and scale them by $\frac{1}{\sqrt{n}}$ then, we have :

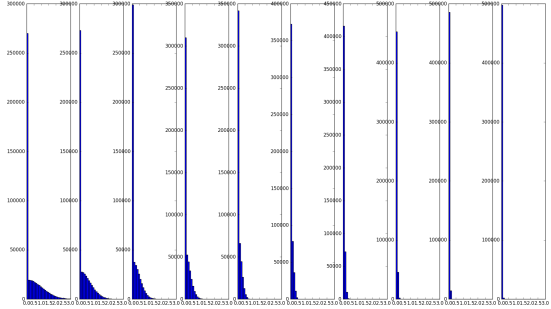
$$\begin{aligned} nVar(w) &= n\,Var(\frac{z}{\sqrt{n}}) \\ &= n\ *\frac{1}{n}Var(z) = 1 \leftarrow (Unit\,Gaussian) \end{aligned}$$

```
W = np.random.randn(fan_in, fan_out) / sqrt(fan_in)
```

- Let's see what happens if we use this initialization



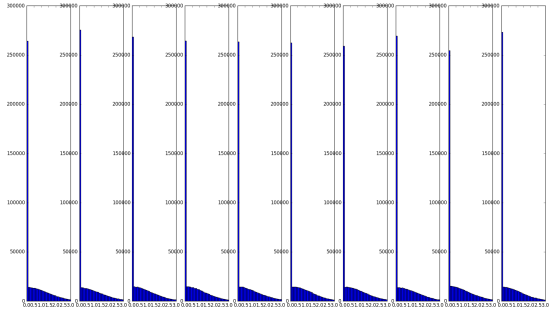
sigmoid activations



- However this does not work for ReLU neurons
- Why ?
- Intuition: *He et.al.* argue that a factor of 2 is needed when dealing with ReLU Neurons
- Intuitively this happens because the range of ReLU neurons is restricted only to the positive half of the space

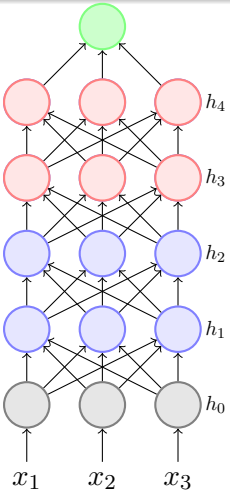
```
W = np.random.randn(fan_in, fan_out) / sqrt(fan_in/2)
```

- Indeed when we account for this factor of 2 we see better performance

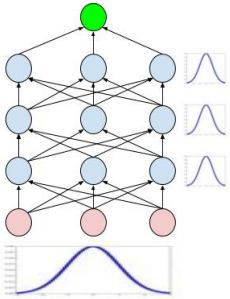


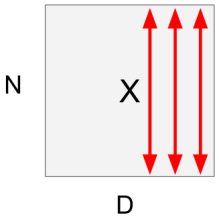
Module 9.5 : Batch Normalization

We will now see a method called batch normalization which allows us to be less careful about initialization



- 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?

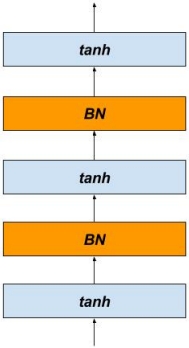




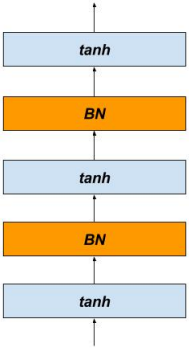
- It would help if the pre-activations at each layer were unit gaussians
- Why not explicitly ensure this by standardizing the pre-activation ?

$$\hat{s_{ik}} = \frac{s_{ik} - E[s_{ik}]}{\sqrt{var(s_{ik})}}$$

- But how do we compute E$[s_{ik}]$ and Var$[s_{ik}]$?
- We compute it from a mini-batch
- Thus we are explicitly ensuring that the distribution of the inputs at different layers does not change across batches



- This is what the deep network will look like with Batch Normalization
- Is this legal ?
- Yes, it is because just as the *tanh* layer is differentiable, the Batch Normalization layer is also differentiable
- Hence we can backpropagate through this layer



γ^k and β^k are additional parameters of the network.

- Catch: Do we necessarily want to force a unit gaussian input to the *tanh* layer?
- Why not let the network learn what is best for it?
- After the Batch Normalization step add the following step:

$$y^{(k)} = \gamma^k \hat{s_{ik}} + \beta^{(k)}$$

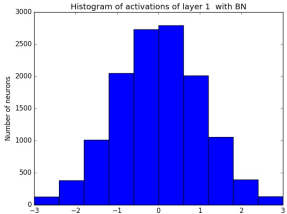
- What happens if the network learns:

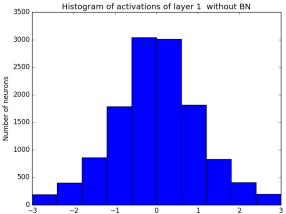
$$\gamma^k = \sqrt{var(x^k)}$$

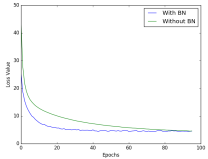
$$\beta^k = E[x^k]$$

- We will recover s_{ik}
- In other words by adjusting these additional parameters the network can learn to recover s_{ik} if that is more favourable

We will now compare the performance with and without batch normalization on MNIST data using 2 layers....


Histogram of activations of layer 1 with BN
Number of neurons
3000
2500
2000
1500
1000
500
0
−3
−2
−1
0
1
2
3


Histogram of activations of layer 1 without BN
Number of neurons
3500
3000
2500
2000
1500
1000
500
0
−3
−2
−1
0
1
2
3


With BN
Without BN
Loss Value
Epochs
50
40
30
20
10
0
0
20
40
60
80
100

2016-17: Still exciting times

- **Eve**n better optimization methods
- Data driven initialization methods
- Beyond batch normalization