CSE 673
COMPUTATIONAL VISION

VENU GOVINDARAJU
DEEN DAYAL MOHAN

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Covid-19 Guidelines

- Effective Aug. 3, the University at Buffalo will require all students, employees and visitors – regardless of their vaccination status – to wear face coverings while inside campus buildings. This includes classrooms, hallways, libraries and other common spaces, as well as UB buses and shuttles.

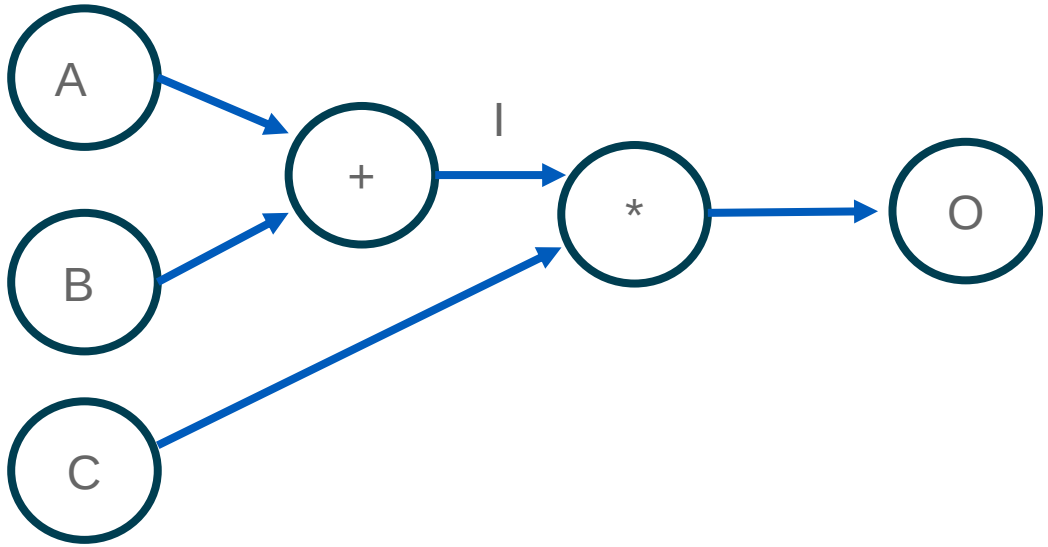
- Students are expected to wear mask in class during lectures (unless you have a UB approved exception)

- Public Health Behavior Expectations https://www.buffalo.edu/studentlife/who-we-are/departments/conduct/coronavirus-student-compliance-policy.html

Computational Graph

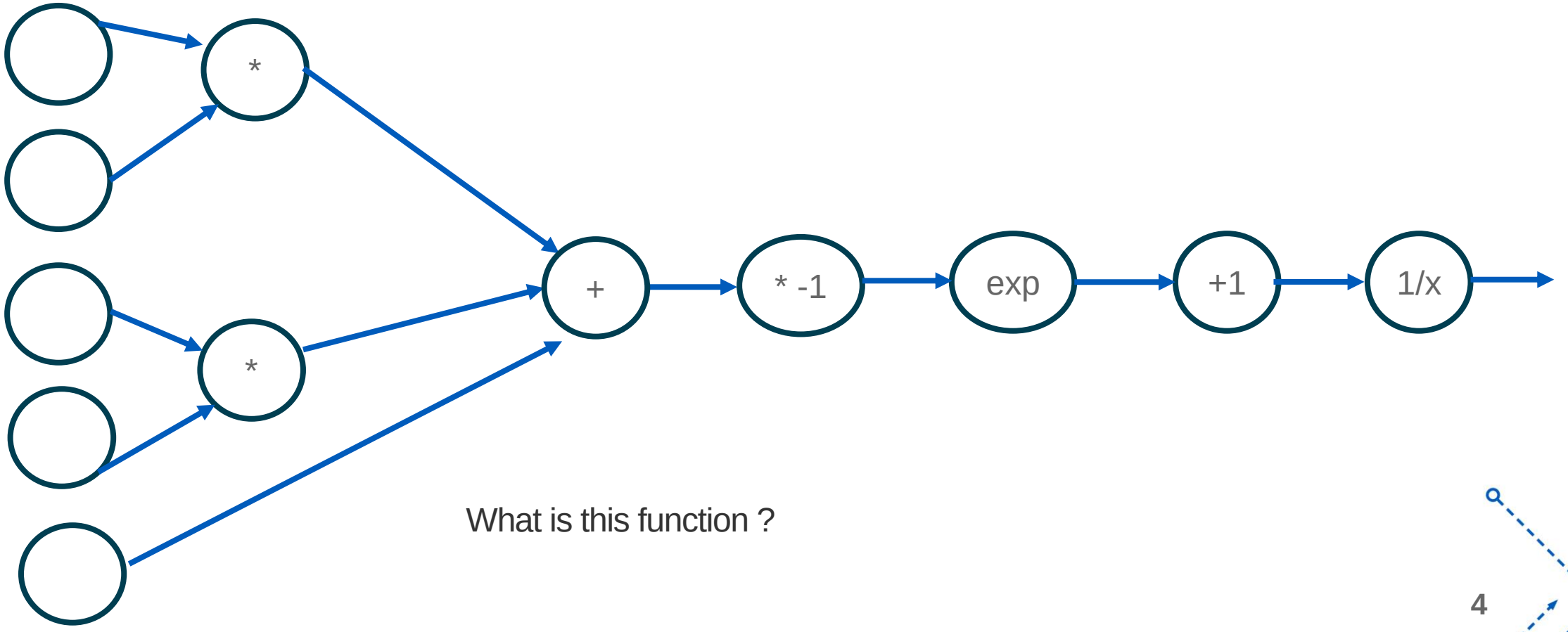
- Computational graphs are a nice way to think about mathematical expressions.

- For example consider O = (A + B) * C. The computational graph for the function would look like this:



Computational Graph

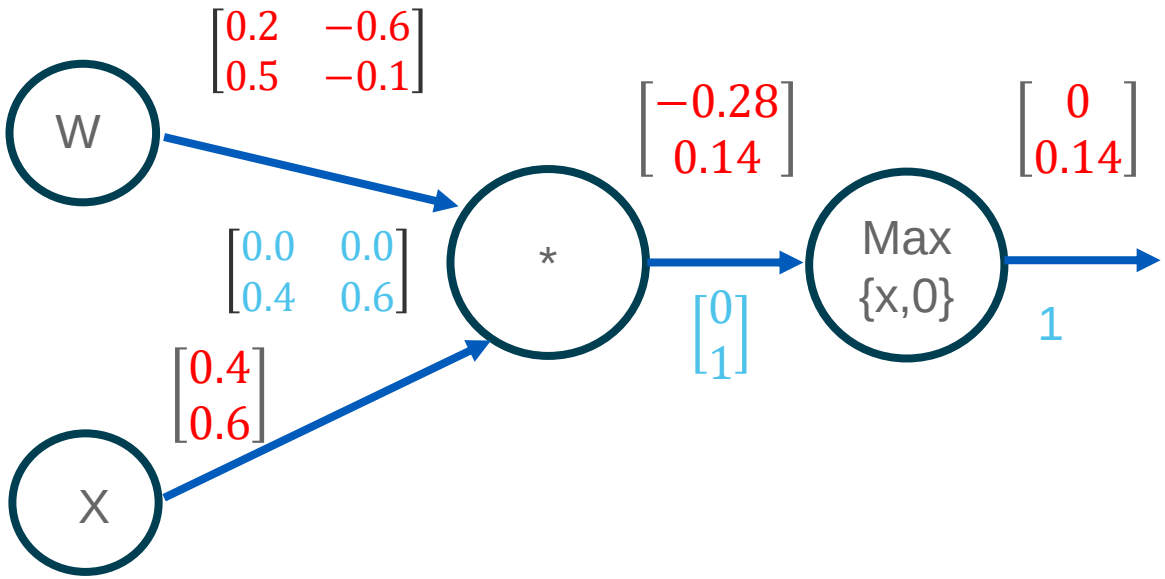
- Let us look at another example



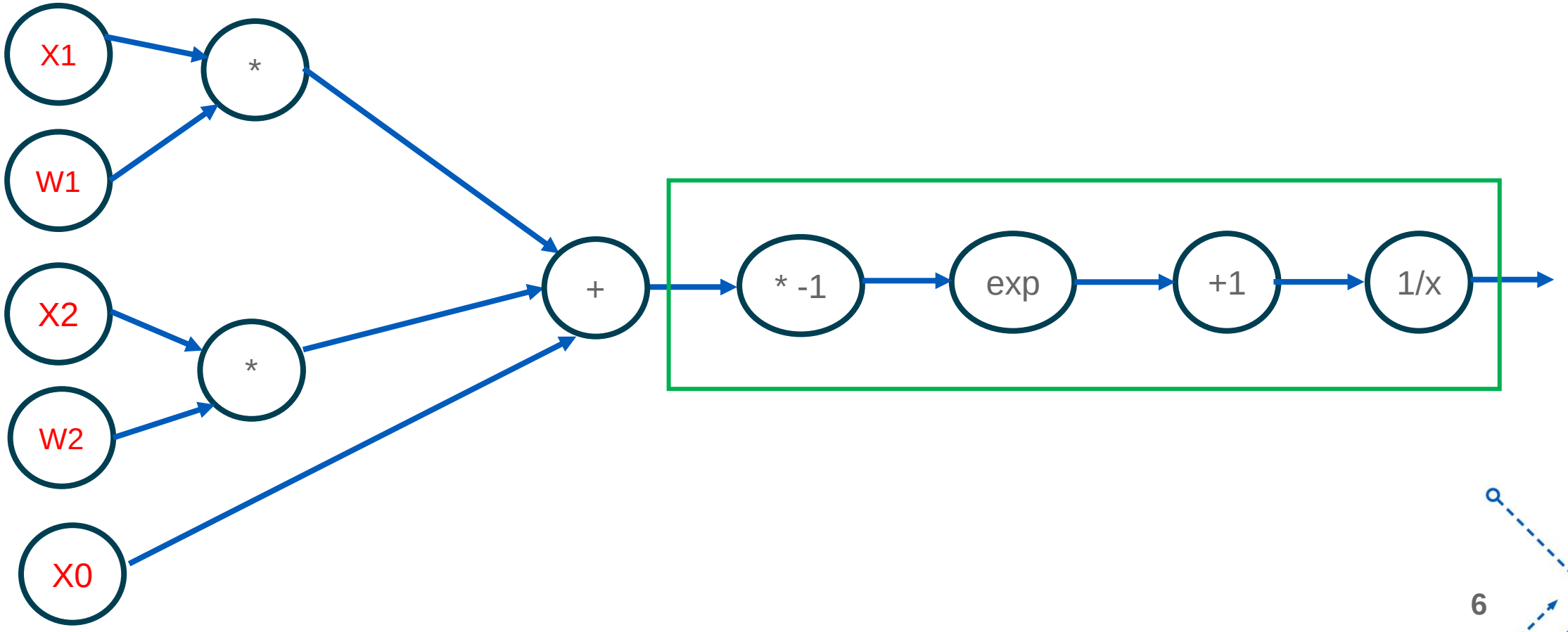
What is this function ?

Computational Graph

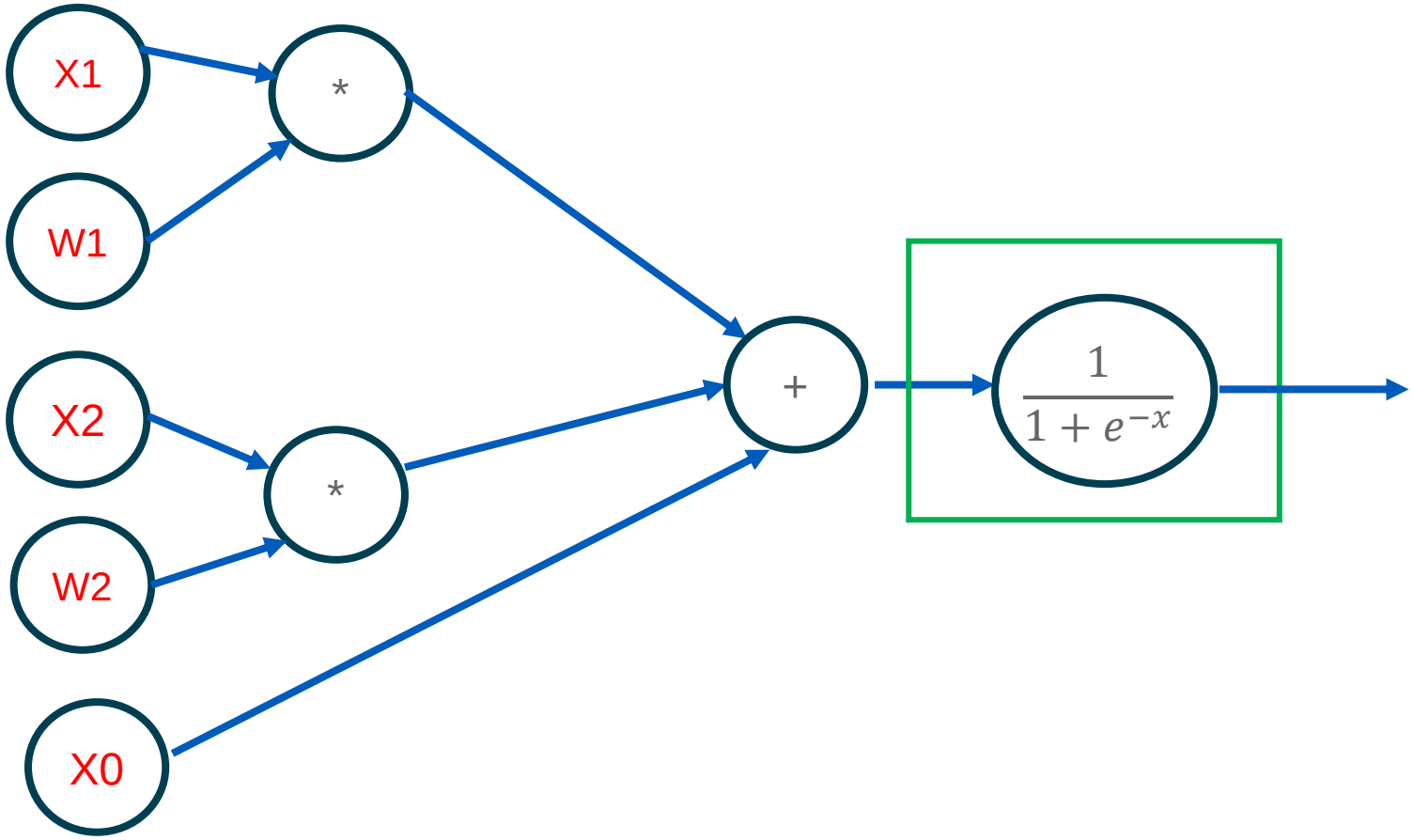
- Let us look at an example with matrices



Computational Graph



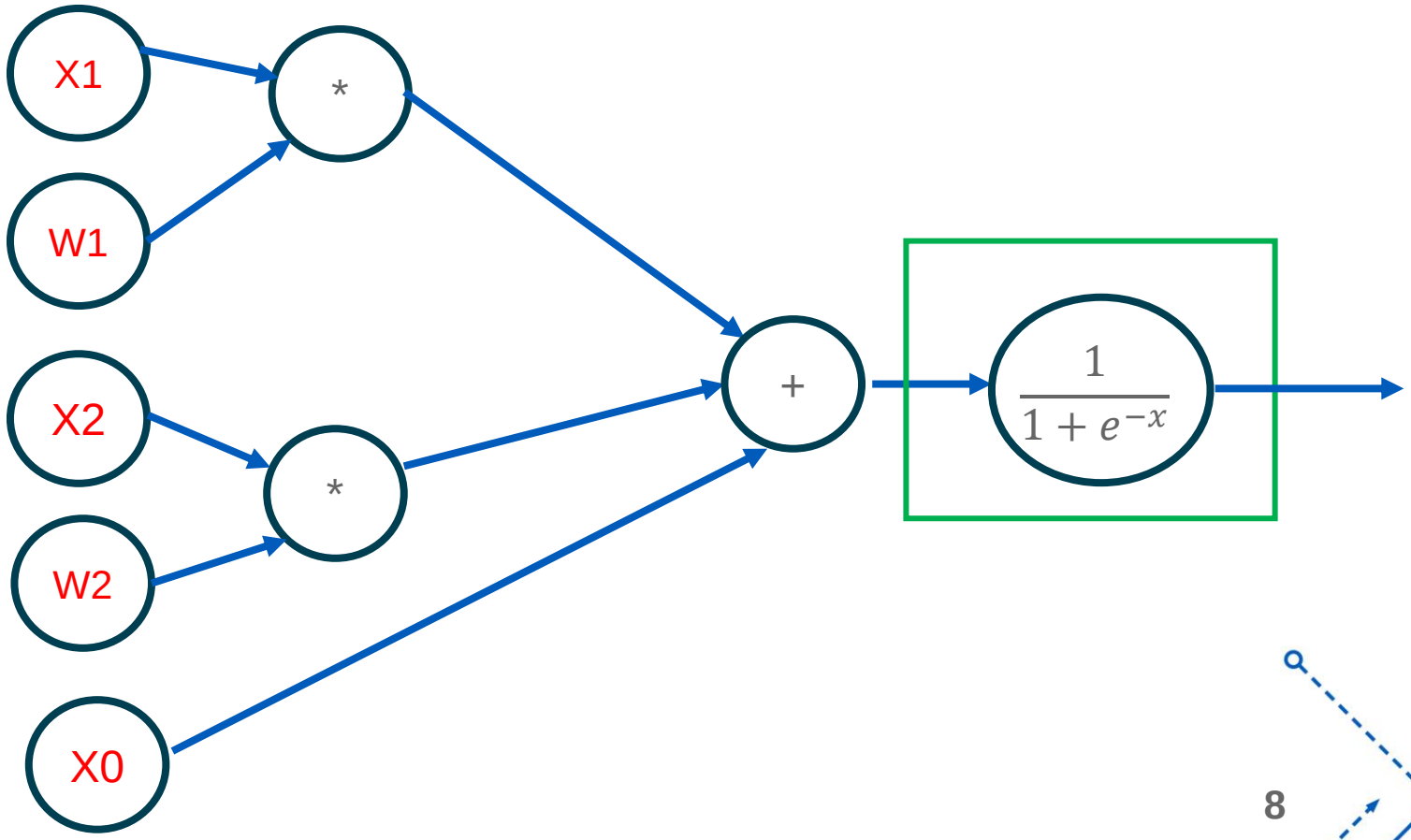
Computational Graph



Computational Graph

- The granularity of the nodes can be custom
- Need to find the derivative of the custom function
- For example, the derivative of $\sigma(x)$ is

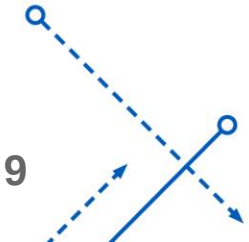
$$= \sigma(x)\big(1 - \sigma(x)\big)$$



Artificial Neural Networks

- But now instead of the nodes we have layers

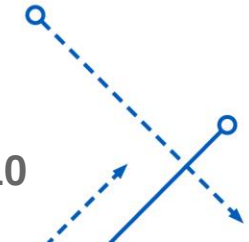
```
class BaseLayer:

    def __init__(self,):

        ...
        ...  # initialize some parameters common to all the layers
        ...

    def forward():
        pass

    def backward():
        pass
```



Artificial Neural Networks

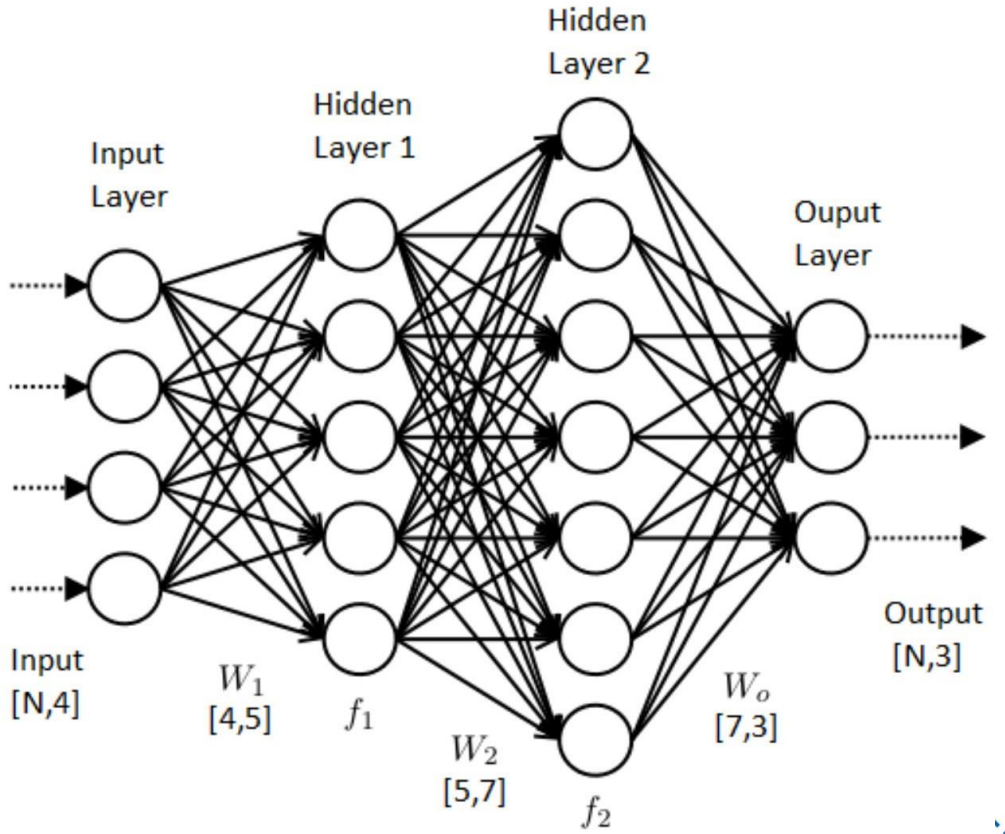
- Then inherit the base class for implementing a layer with some specific functionality

```
class DenseLayer(BaseLayer):

    def __init__(self,input_param):

        ...
        ... # initialize the base class constructor
        ... # initialize layer specific parameters

    def forward(input):

        ...
        ... # perform foward propagation
        ...
        return layerOutput

    def backward(dz):

        ...
        ... # perform backprop
        ...
        return gradient
```



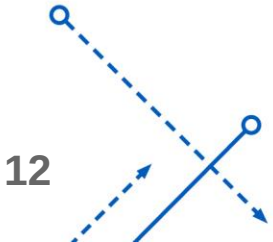
Artificial Neural Networks

- Once the gradients with respect to the necessary parameters are computed, update step is performed
- $w_{new} = w_{Old} - \eta \frac{\partial J}{\partial w_{Old}}$
- Here η is the learning rate
- All the parameters are updated after the gradients are computed
- We will look at other methods of parameter update in a later lecture

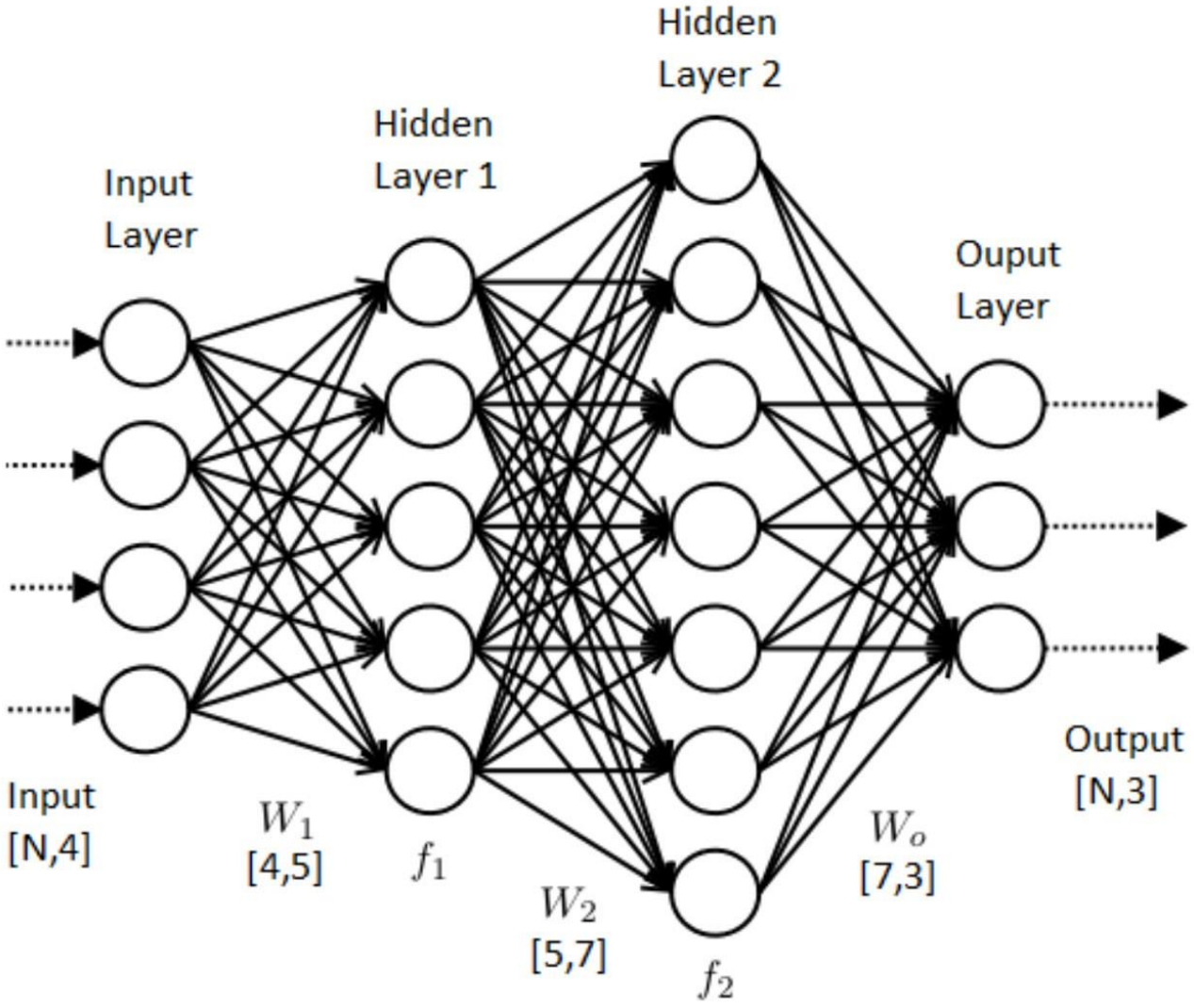


Agenda

- Optimizers
- Activations
- Initializations
- Batch Normalization and Dropout



Optimizers


Hidden Layer 2
Hidden Layer 1
Input Layer
Ouput Layer
Input [N,4]
W_1 [4,5]
f_1
W_2 [5,7]
f_2
W_o [7,3]
Output [N,3]

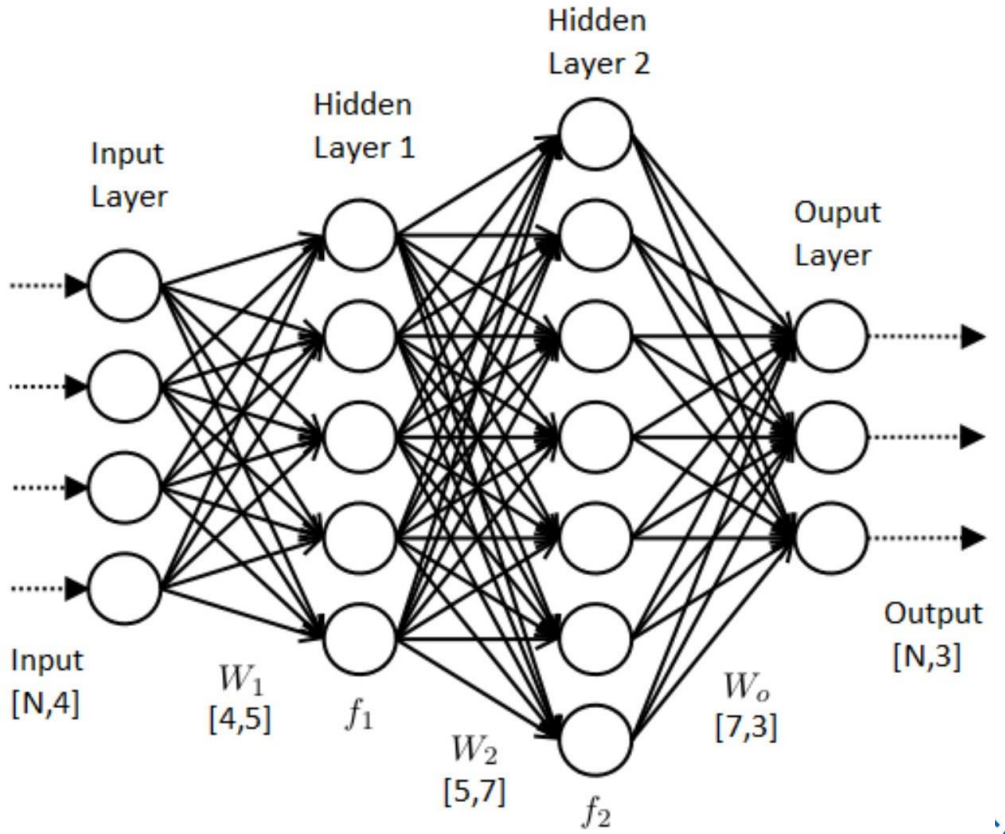
Optimizers

- $w_{new} = w_{Old} - \eta \frac{\partial J}{\partial w_{Old}}$

- This method of updating is called stochastic gradient descent

- In practice a mini batch of samples are passed through , the gradient is computed and then the parameters are updated based on this gradient

- Also referred as mini batch gradient descent



Optimizers

- $w_{new} = w_{Old} - \eta \frac{\partial J}{\partial w_{Old}}$

```
## mini batch stochastic Gradient Descent

for each in epoch:
    for batch in batchs:
        gradient_of_parameters = backward_through_graph(loss_function,batch,parameters)
        parameters = parameters - learning_rate * gradient_of_parameters
```

- The learning rate of all the parameters are the same

Momentum

- Vanilla SGD algorithm tend to suffer when the loss landscape is elongated in one direction
- SGD tends to jitter a lot when going towards the minimum
- Such landscapes are generally present near local minimum
- This is why we rarely use Vanilla SGD for training a Neural Network

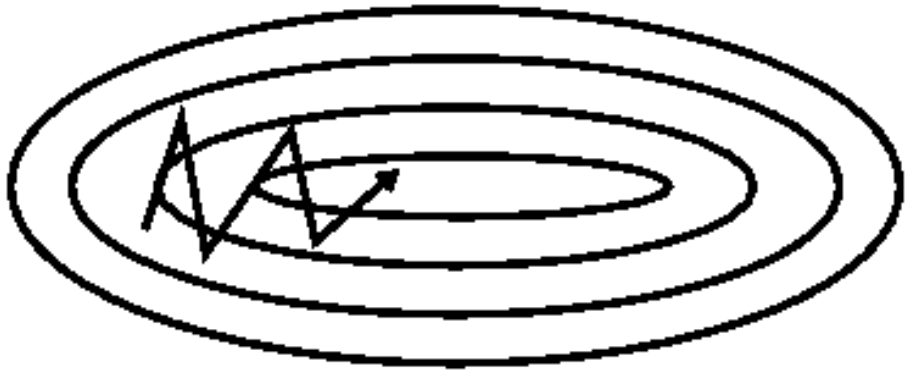
Momentum

- In order to avoid this we introduce momentum into the gradient update

- The idea of momentum is accumulate the power of gradient direction which does not change and dampen the direction which changes a lot

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

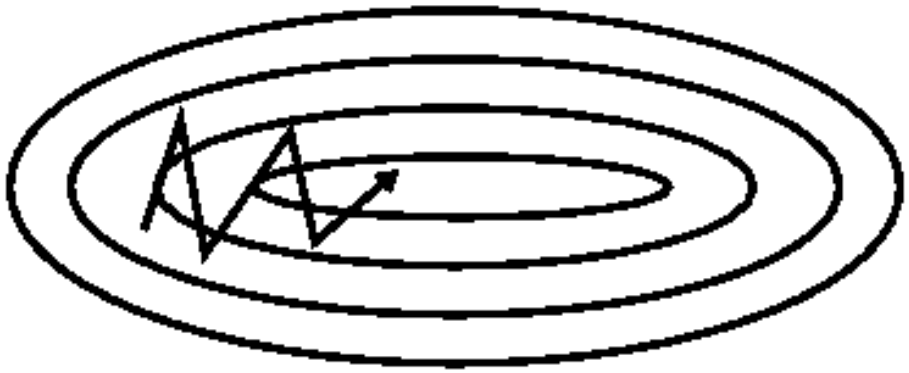
- Here ‘v’ the used as a term to accumulate the gradient across time steps and this term is used to update the parameters



Momentum

- γ is the momentum term, set between 0-1. (typically 0.9)

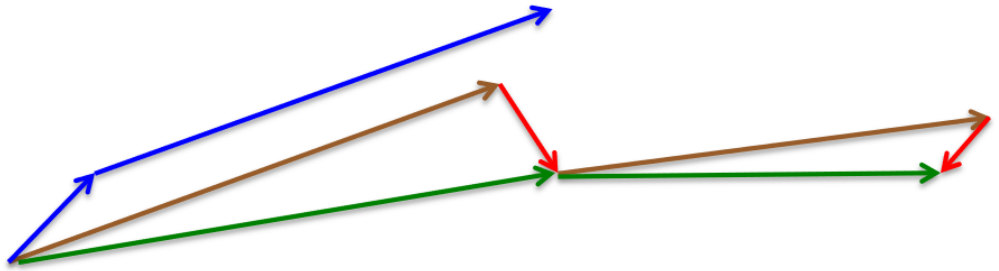
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$



```
for each in epoch:
    for batch in batchs:
        gradient_of_parameters = backward_through_graph(loss_function,batch,parameters)
        v = lamda*v + learning_rate * gradient_of_parameters
        parameters = parameters - v
```

Momentum

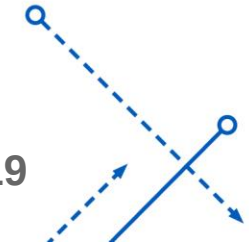
- An improvement to the vanilla momentum is to use Nesterov Momentum.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

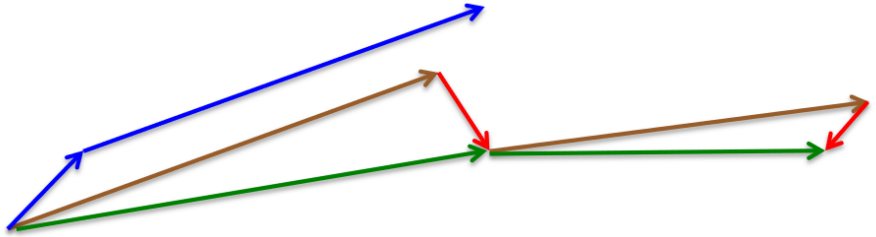
- The idea of Nesterov Momentum is to compute the gradient at the approximate next position in the loss landscape



Momentum

- Mathematically it can be written as

$$v_{t+1} = \mu v_t - \eta \nabla l(\theta + \mu v_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$



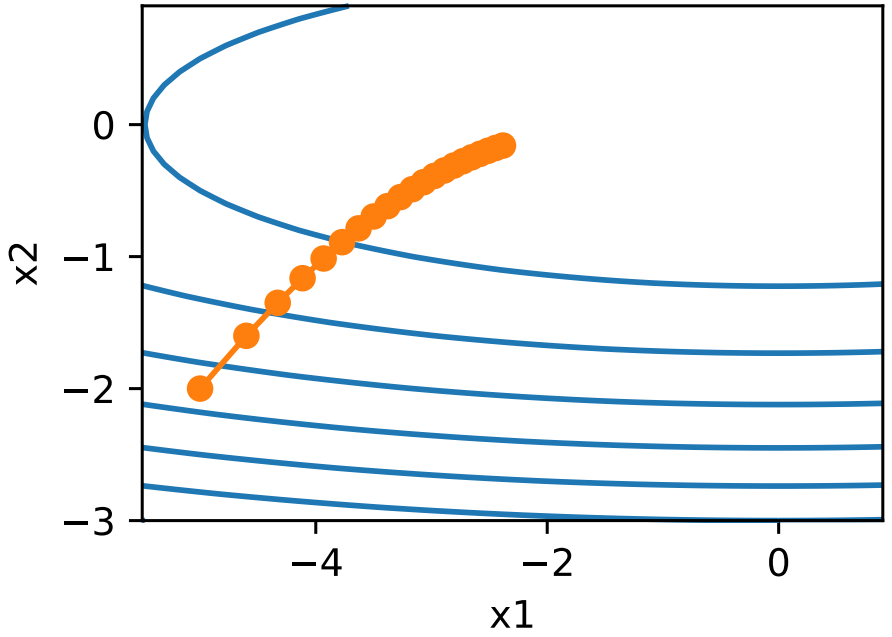
brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

- Here μ is the momentum parameter.
- The equation to simplified using some math tricks. (Look at reference 1 if you are interested)
- Mostly used as the first choice algorithm when using momentum

AdaGrad

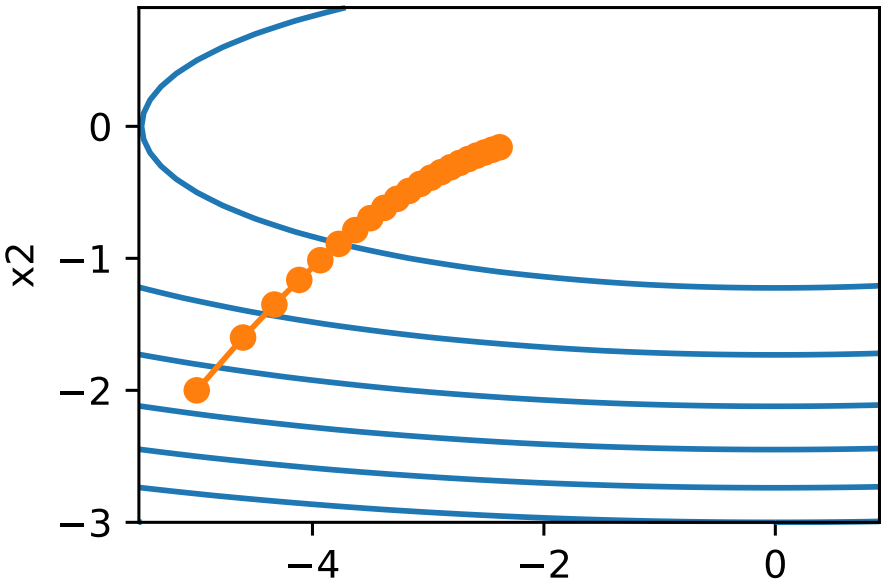
- All the optimizers that we saw so far has a common learning rate for all the parameters
- Adaptive optimizers are able to have per-parameter learning rate. This is helpful as some parameters might need faster or slower update than the other
- The way this is achieved is by accumulating the square of the gradients and dividing the current gradient by the square root of the accumulated term
- Convergence is much faster than SGD based methods



AdaGrad

- Mathematically it is written as

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$




```
for each in epoch:
    for batch in batchs:
        gradient_of_parameters = backward_through_graph(loss_function,batch,parameters)
        G = G + (gradient_of_parameters)**2
        parameters = parameters - learning_rate * gradient_of_parameters /(sqrt(G+1e-10))
```

RmsProp

- What is the disadvantage of AdaGrad?

- RmsProp was developed by Hinton in order to fix the issue with AdaGrad.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

```
for each in epoch:
    for batch in batchs:
        gradient_of_parameters = backward_through_graph(loss_function,batch,parameters)
        G =  G*lamda + (1-lamda)*(gradient_of_parameters)**2
        parameters = parameters - learning_rate * gradient_of_parameters /(sqrt(G+1e-10))
```

Adam

- So can we combine best of both worlds ?

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

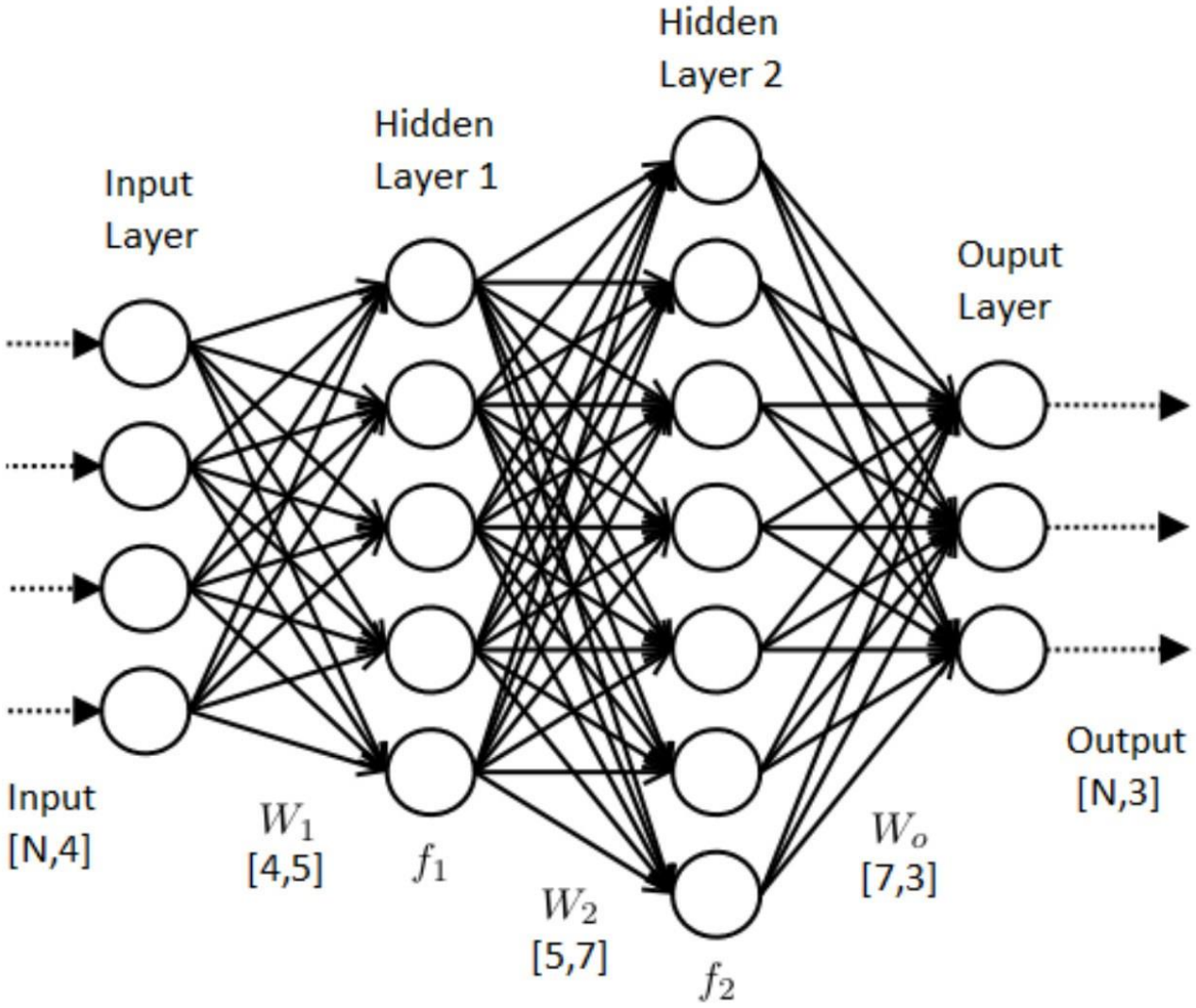
- Combine the momentum based updates of the gradients and also scale the update down by second order moments to give a per-parameter learning rate

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

```
for each in epoch:
    for batch in batchs:
        gradient_of_parameters = backward_through_graph(loss_function,batch,parameters)
        M = Beta1 * M + (1-Beta1)* gradient_of_parameters
        G = Beta2 * G + (1-Beta2)*(gradient_of_parameters)**2
        parameters = parameters - learning_rate * M /(sqrt(G+1e-10))
```

- There is a Bias correction step missing. Figure it out as HW

Activation Functions



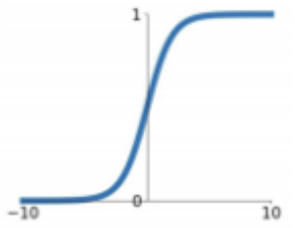
Activation Functions

- So let us take a closer look at some of the activation functions

- Sigmoid activation is one activation function we looked at. It also called the logistic function.

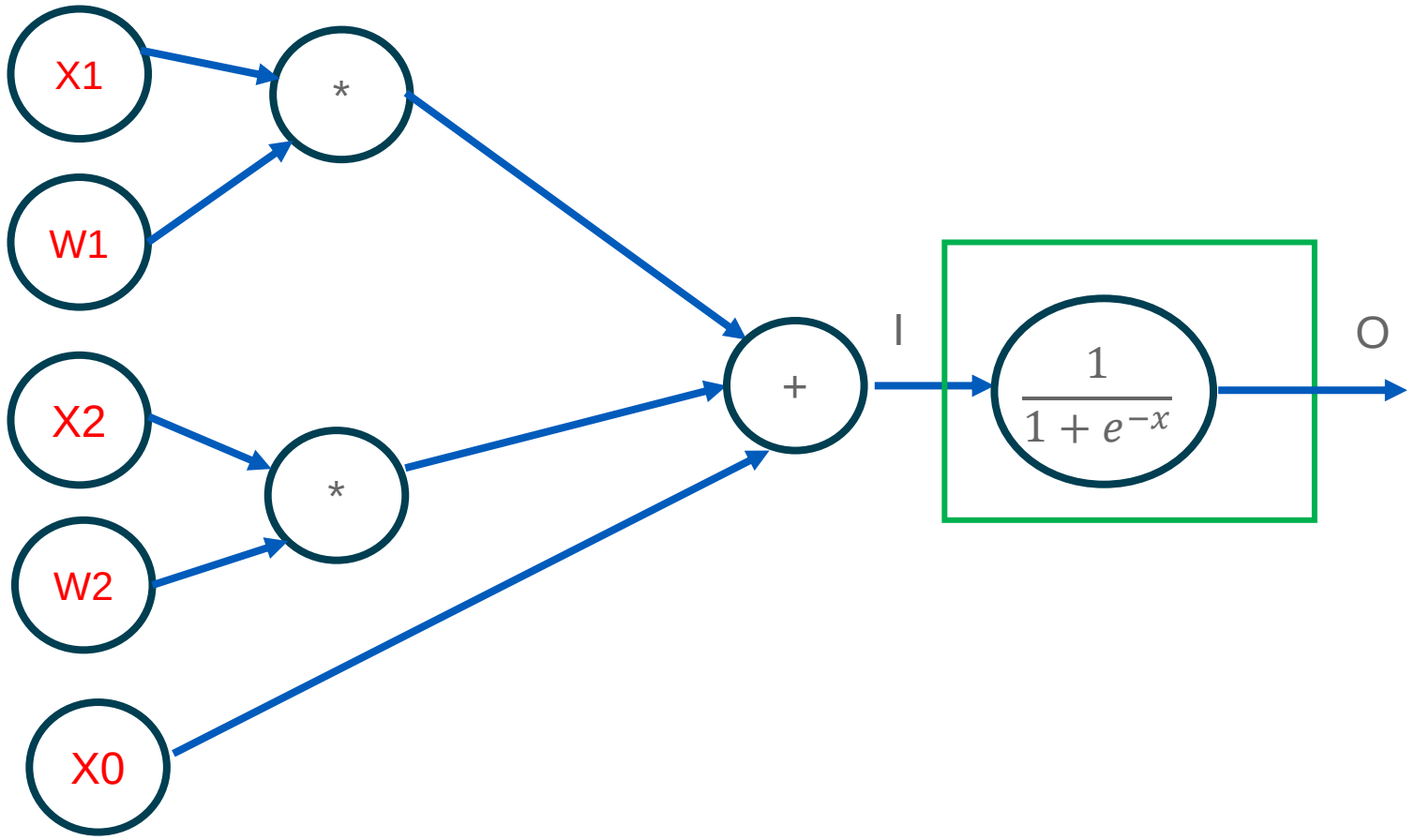
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



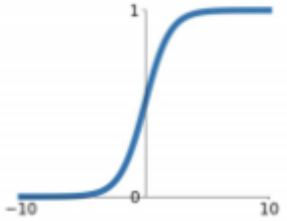
- Why is sigmoid activation not used much in Deep neural networks?

Activation Functions



Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



What happens when the value of I is 10 or above?

What happens if I was between 0-1?

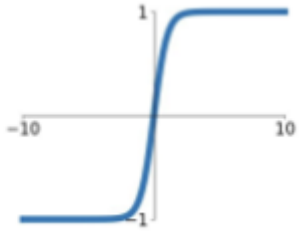
What happens if I is less than -10?

Activation Functions

- Tanh activation is an activation similar to sigmoid

tanh

$\tanh(x)$



What happens when the value of Input is 10 or above?

What happens if input is between 0-1?

What happens if Input is less than -10?

- It has a nice property of outputs being zero centered

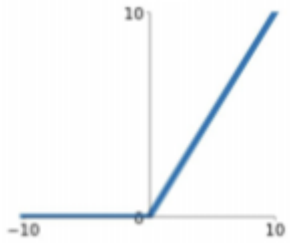
- Still has the effect of killing gradients

Activation Functions

- Rectified linear activation (ReLU)

ReLU

$$\max(0, x)$$



- Does not kill gradients in the positive regions
- No ‘exp’ operation, so computationally faster
- Converges much faster than sigmoid or tanh

What happens when the value of Input is 10 or above?

What happens if input is between 0-1?

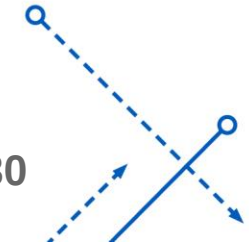
What happens if Input is less than -10?

Activation Functions

- What is Dead ReLU problem?
- The update of the neural network parameters follow this equation

$$w_{new} = w_{Old} - \eta \frac{\partial J}{\partial w_{Old}}$$

- Some times due to bad initialization or high learning rate, the value of weights might go negative and cause ReLU activation to produce an output of 0
- This leads to some neurons not becoming active at all in the network

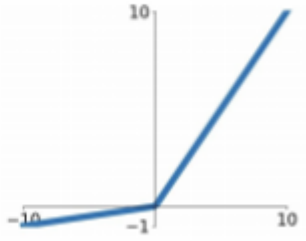


Activation Functions

- In order to address this problem, Leaky ReLU was designed

Leaky ReLU

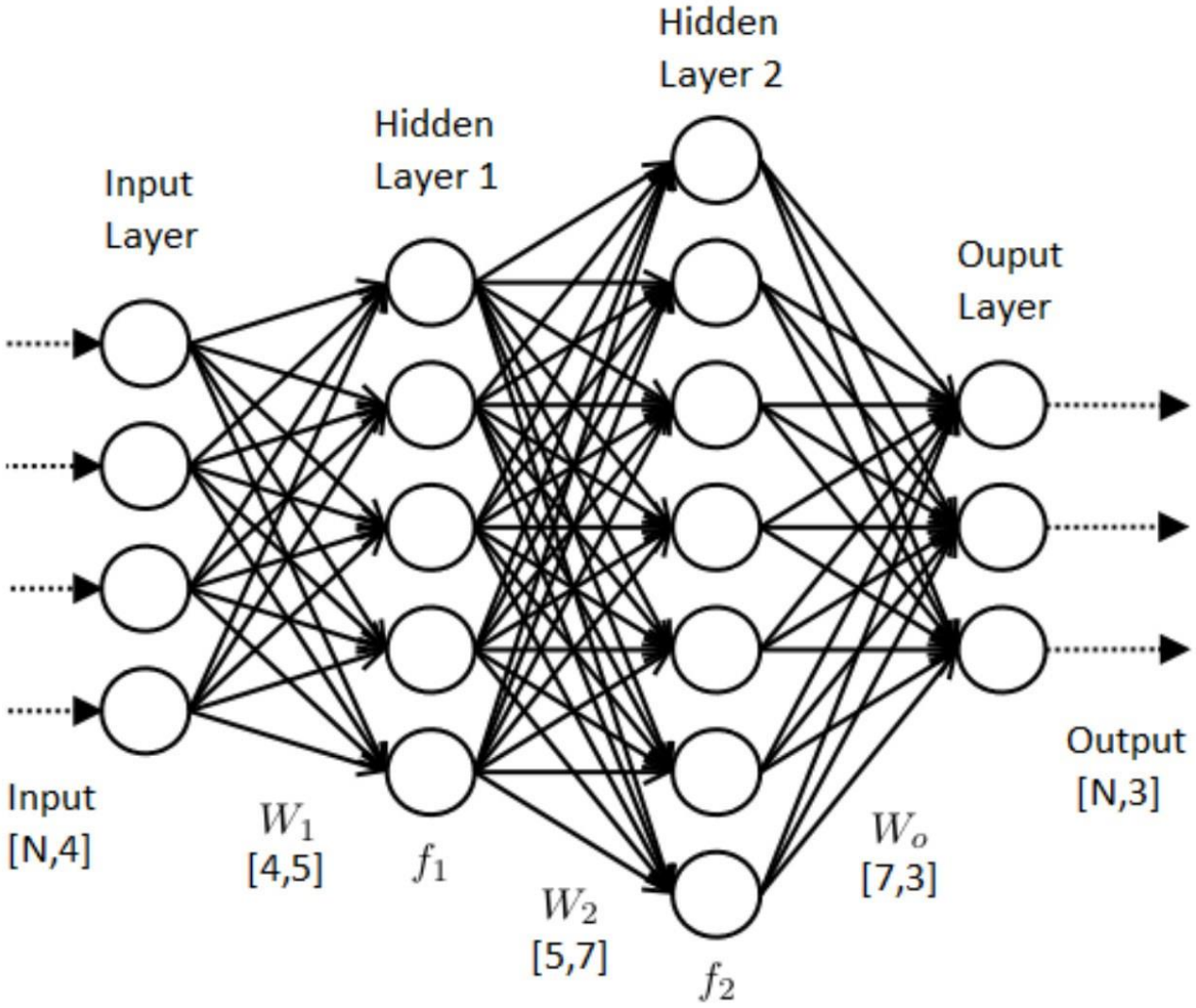
$$\max(0.1x, x)$$



- This functions has all the good properties of ReLU, and allows some gradient flow is the negative region

- This activation is shown to overcome the problem of dead neurons to some extend.

Initializations



Initializations

- There are multiple ways to initialize the parameters of the network, before starting to train
- Network initializations plays an important role in their convergence
- What happens if I initialize all the weights of the network to 0 and start training?
- What happens when random initialization is used?
- Let us look at an example when we initialize with small weights

.

Initializations

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

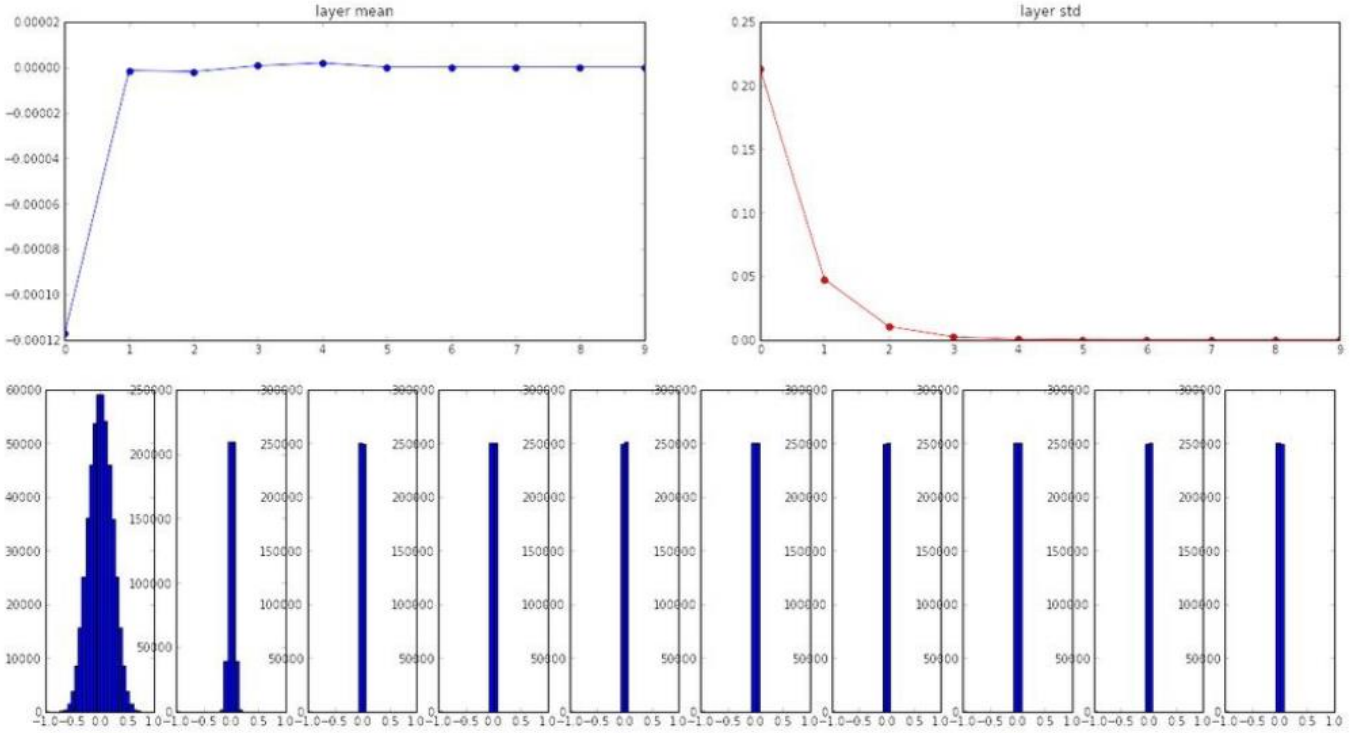
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Ref: http://cs231n.stanford.edu/slides/2016/winter1516_lecture5.pdf

Initializations

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



Ref: http://cs231n.stanford.edu/slides/2016/winter1516_lecture5.pdf

The magnitude of weights are very small in the last layers

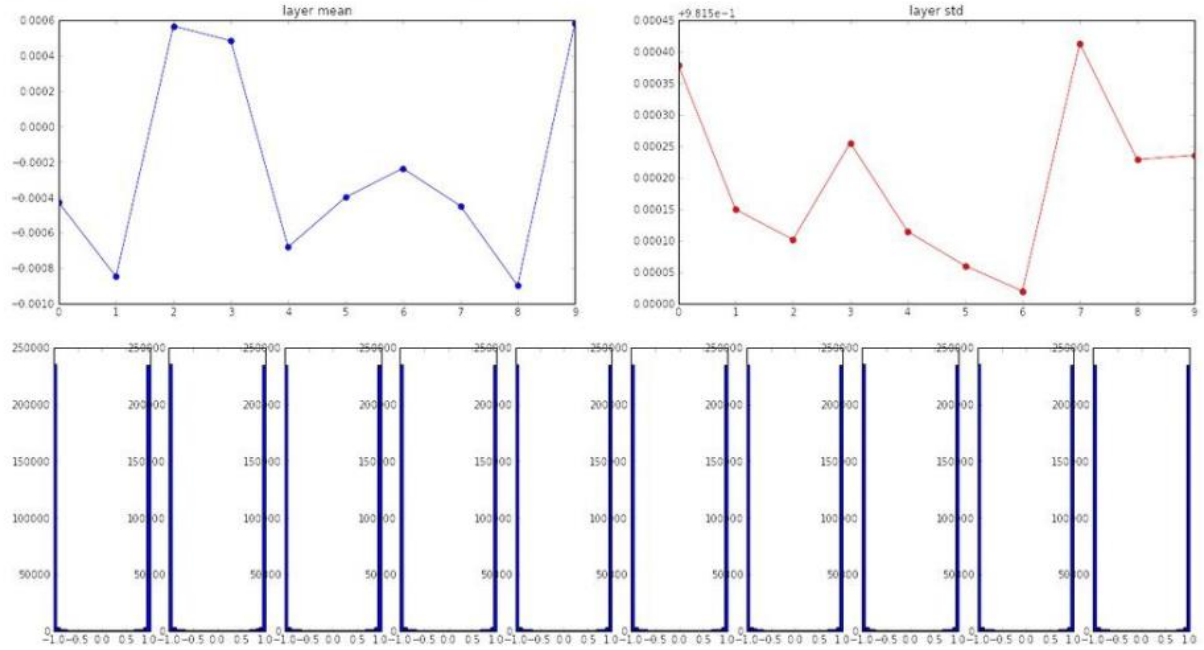
Why is this bad ?

Initializations

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01



All the neurons become saturated.

Why is this bad ?

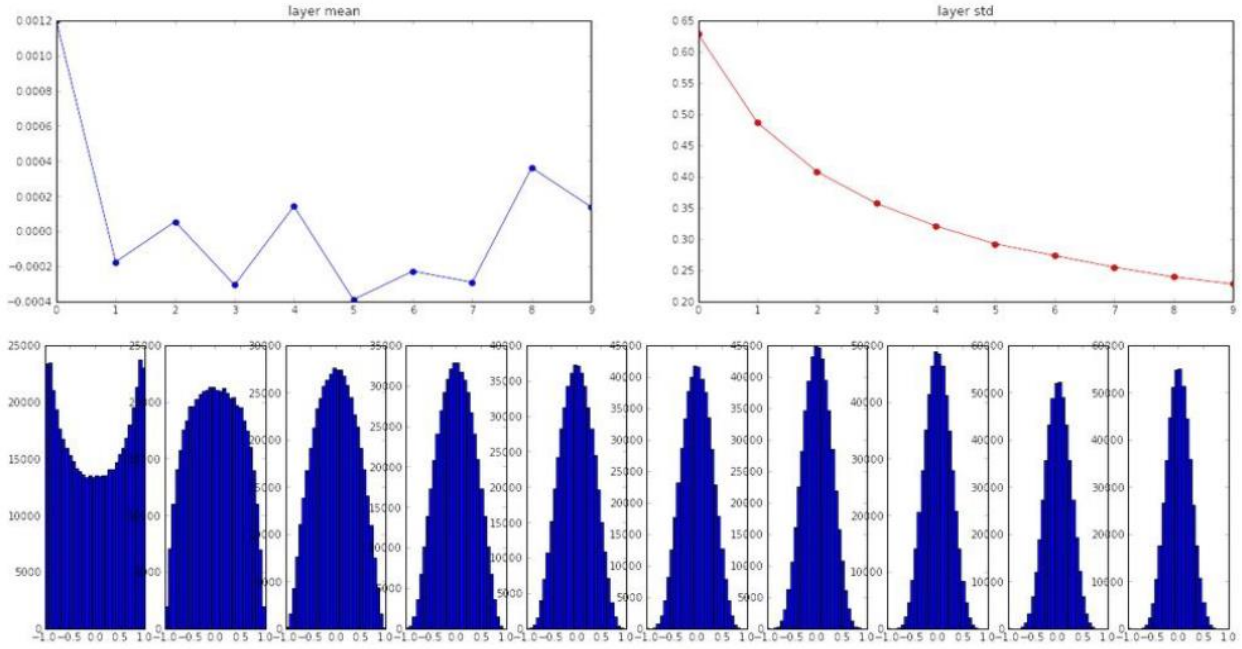
Ref: http://cs231n.stanford.edu/slides/2016/winter1516_lecture5.pdf

Initializations

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

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

"Xavier initialization"
[Glorot et al., 2010]



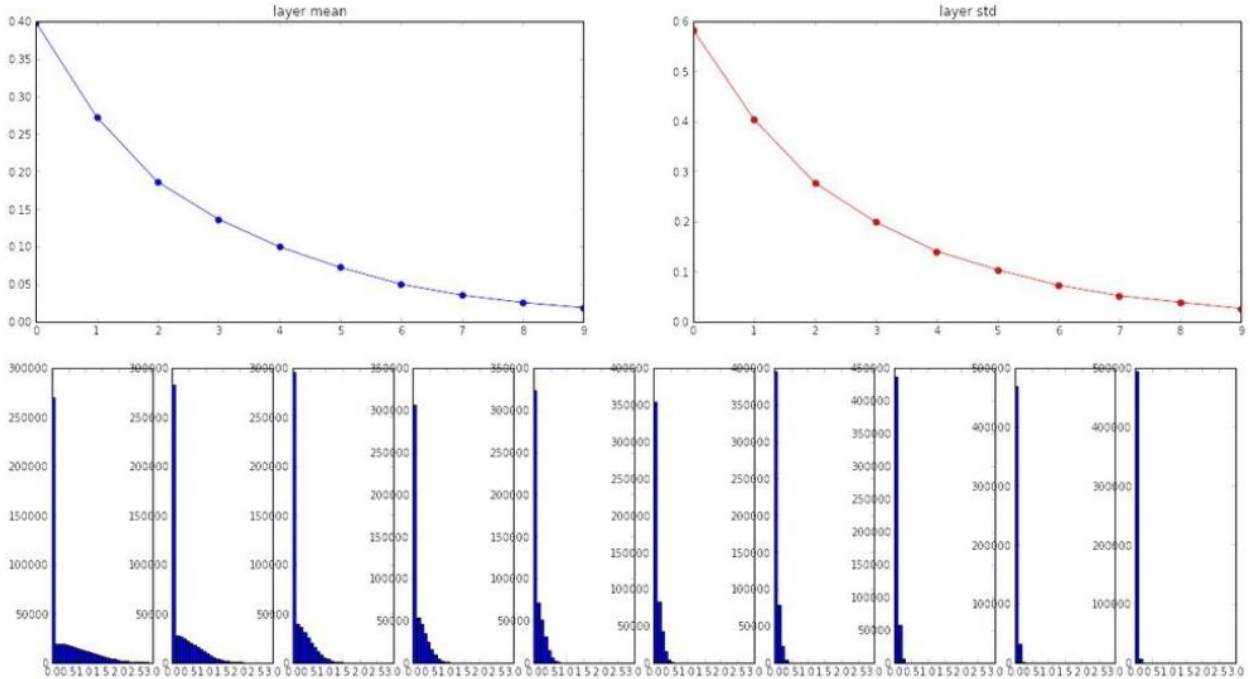
Ref: http://cs231n.stanford.edu/slides/2016/winter1516_lecture5.pdf

Initializations

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

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

but when using the ReLU nonlinearity it breaks.



Ref: http://cs231n.stanford.edu/slides/2016/winter1516_lecture5.pdf

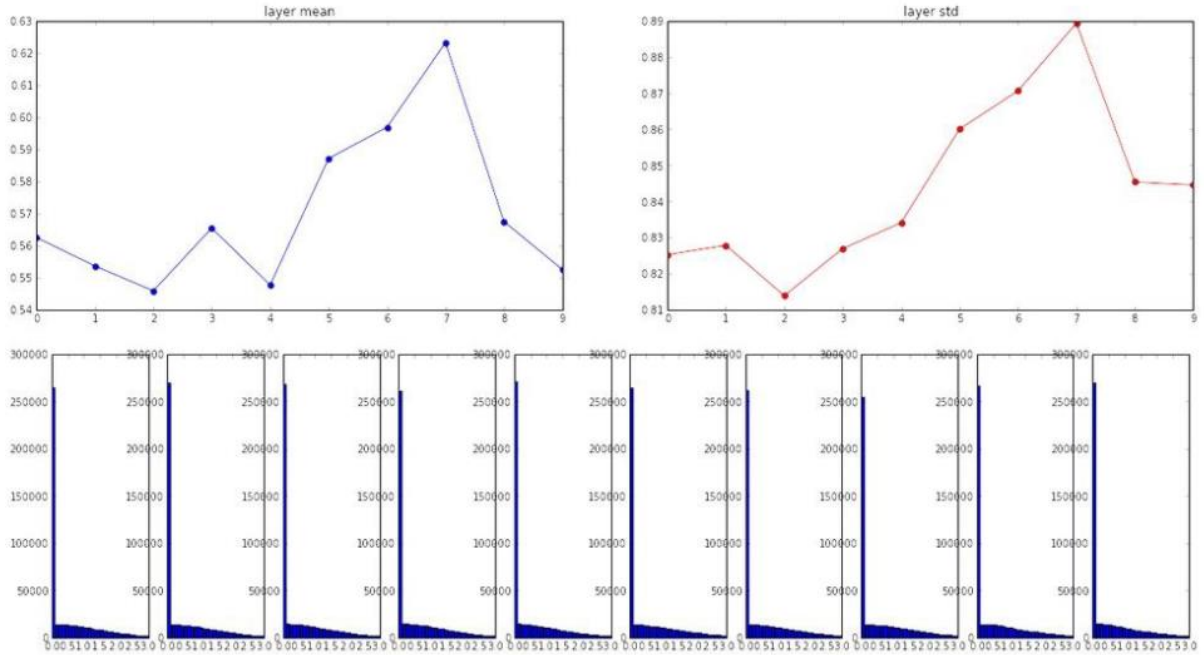
Initializations

.

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

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

He et al., 2015
(note additional /2)



Ref: http://cs231n.stanford.edu/slides/2016/winter1516_lecture5.pdf

Batch Normalization

- The data passed on to the input layer of the network is normalized
- In order to improve the performance, we like to maintain the normalization across layers
- But this is not guaranteed when using different layers
- In order to address this problem Batch Normalization is introduced

.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

- The final algorithm for the batch norm layer would look like this .

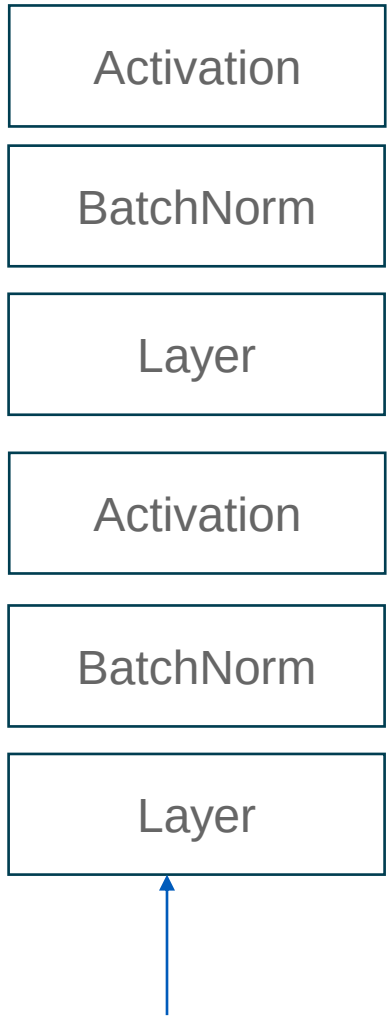
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



Batch Normalization

- Has the ability to adjust the parameters to help the training process.
- The idea of using Batch Norm often is said to reduce the internal covariance shift
- What happens at test time?
- If interested check out NormProp.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

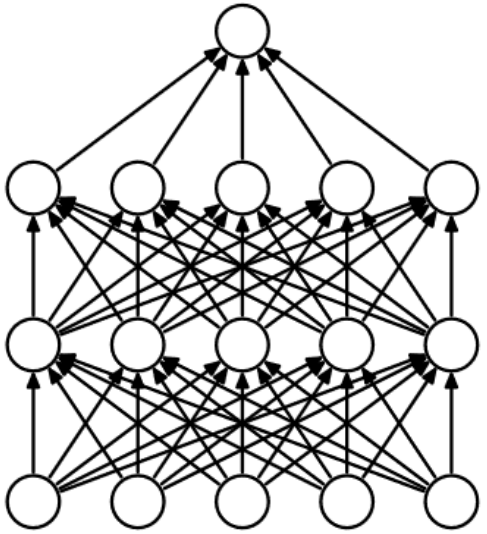
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

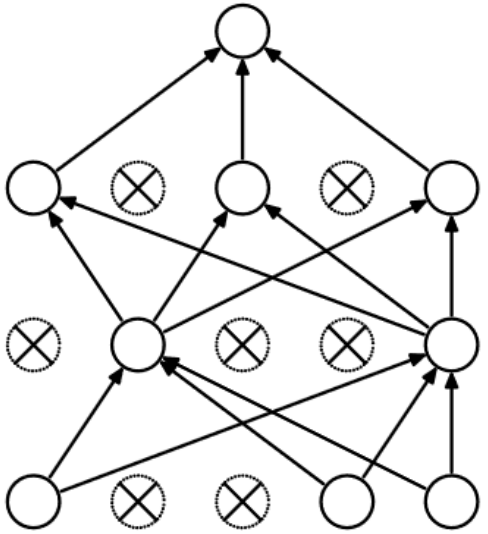
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Dropout

- The process of removing activations from a percentage of neurons
- Random neurons selected from the output of a layer and the value is set to 0.
- The neuron which were selected should be maintained to adjust in the backprop
- Why does this work?
- Not relying on just one neuron. Also ensemble intuition



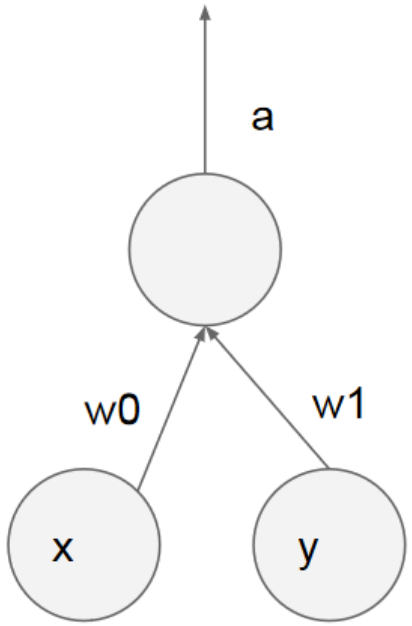
(a) Standard Neural Net



(b) After applying dropout.

Dropout

- What happens at test time



during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)

= ¼ * (2 w0*x + 2 w1*y)

= ½ * **(w0*x + w1*y)**

Ref: http://cs231n.stanford.edu/slides/2016/winter1516_lecture5.pdf


ANY
QUESTIONS
?

References

- http://proceedings.mlr.press/v28/sutskever13.html
- This lecture is inspired from cse 231n https://www.youtube.com/watch?v=i94OvYb6noo&t=2051
- http://neuralnetworksanddeeplearning.com/chap5.html
- https://ruder.io/optimizing-gradient-descent/
- http://cs231n.stanford.edu/
- https://www.google.com/imgres?imgurl=https%3A%2F%2Fmiro.medium.com%2Fmax%2F1400%2F1*Di4V69e4gC16ooF6PZPt-A.png&imgrefurl=https%3A%2F%2Ftowardsdatascience.com%2Feverything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a&tbnid=PFKzBNejYXM4hM&vet=12ahUKEwism4aItO3yAhVrqnIEHSHUCNkQMyhDegQIARBf..i&docid=OXeL--Z4fRwo6M&w=1250&h=1057&q=neural%20networks%20with%20math&hl=en&client=firefox-b-1-d&ved=2ahUKEwism4aItO3yAhVrqnIEHSHUCNkQMyhDegQIARBf