

# DEEP LEARNING

## LECTURE 1 – INTRODUCTION

# OVERVIEW OF TODAY

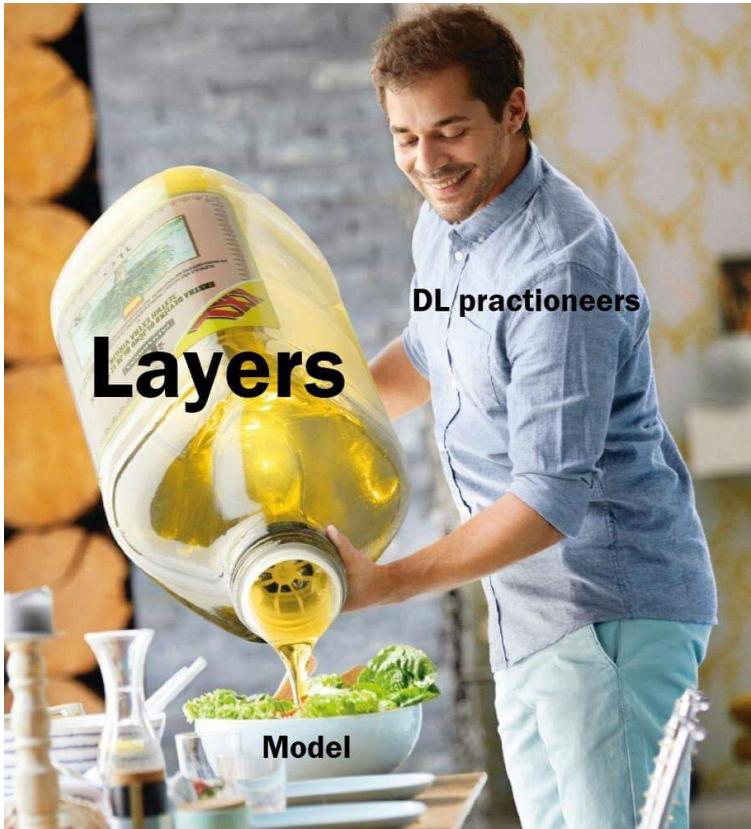
Schedule:

- > Review of last week
- > A bit more maths (Finish last week)
- > Optimisation algorithms
- > Regularisation methods
- > Software & Hardware

Lab:

- > Introduction to Tensorflow & **Keras**

# HOW IS AT1 GOING?



**When you add 1 hidden unit to your network and the error rate doesn't improve**



# OVERVIEW OF TODAY

Schedule:

- > Review of last week
- > A bit more (maths) from last week
- > Optimisation algorithms
- > Regularisation methods
- > Software & Hardware

Lab:

- > Introduction to Tensorflow & Keras

To be perfectly blunt.....

*“All the impressive  
achievements of deep  
learning amount to just  
curve fitting”*

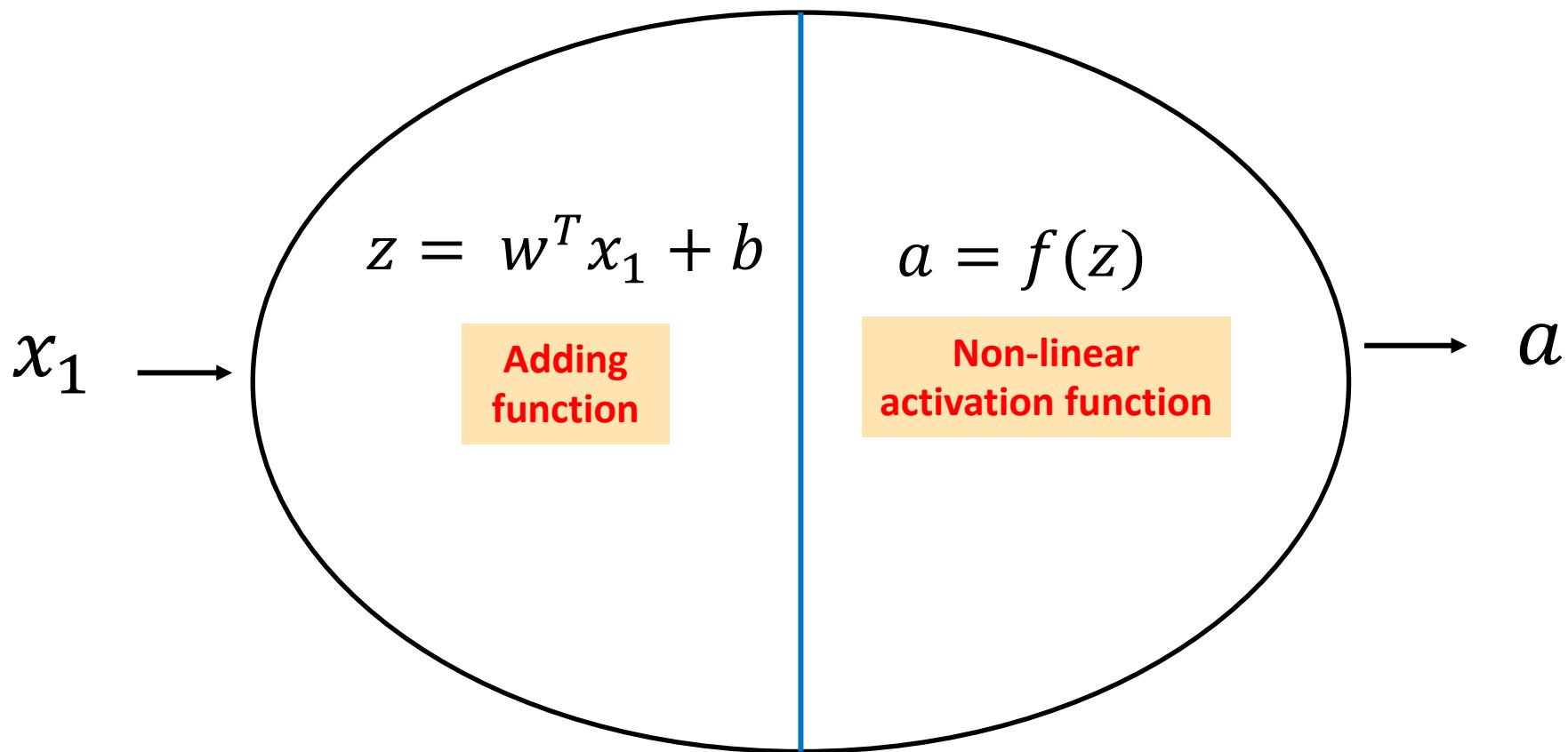
*– Judea Pearl*



# Be careful...



# Inside a Neuron



**Why transpose W?**

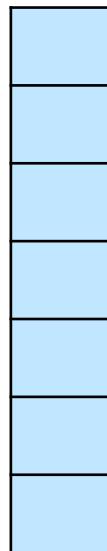
# Why transpose W?

- Recall: ' $w$ ' is our matrix of weights that we will apply to each of our input data points.

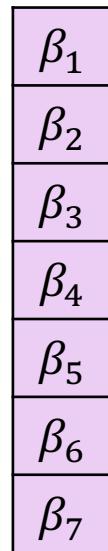
$w$  will therefore have a weight per variable

- We have  $w \in \mathbb{R}^{n_x}$  where  $n_x$  is the number of features of  $x$

Consider 1  
training example  
of 10 features



$x_1$



$w_1$

Can I multiply  
these together  
currently?

No!

We must have form

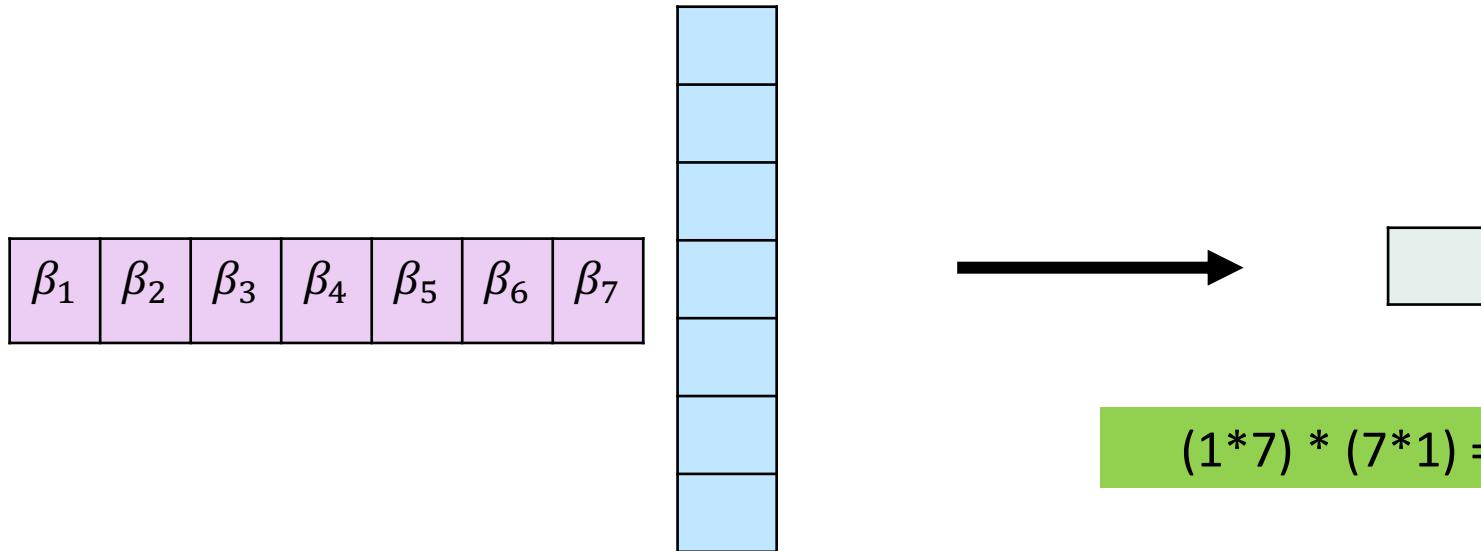
$$\mathbb{R}^{n_x * m} * \mathbb{R}^{m * a}$$

$$\text{Or } (m * n) * (n * p) = m * p$$

Where  $m$  = rows,  $n$  = columns

# Why transpose W?

- To allow us to undertake matrix calculus we must transpose something



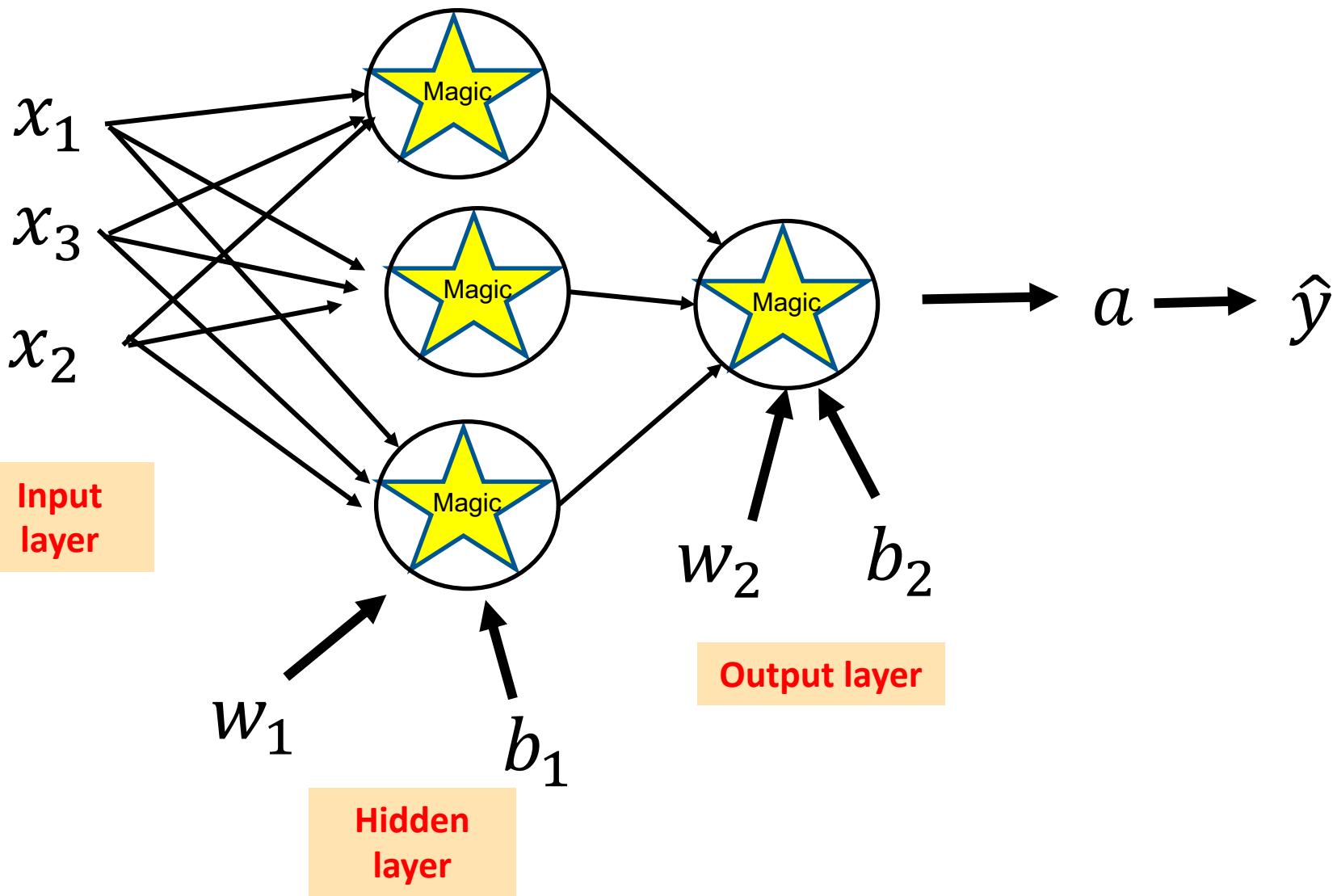
Can I multiply  
these together  
currently?

$$(1*7) * (7*1) = (1*1)$$

Yes!

Columns(x) = rows(w)

# Neuron to ‘network’

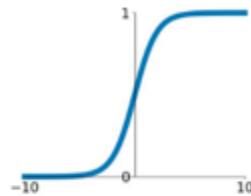


# Other Activation Functions

## Activation Functions

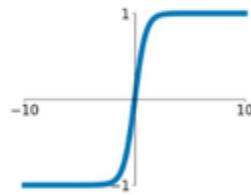
### Sigmoid

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



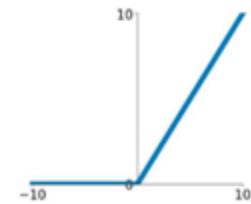
### tanh

$$\tanh(x)$$



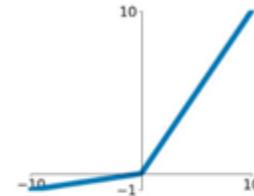
### ReLU

$$\max(0, x)$$



### Leaky ReLU

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

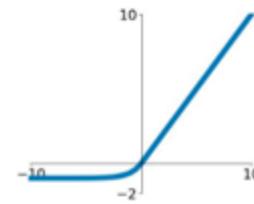


### Maxout

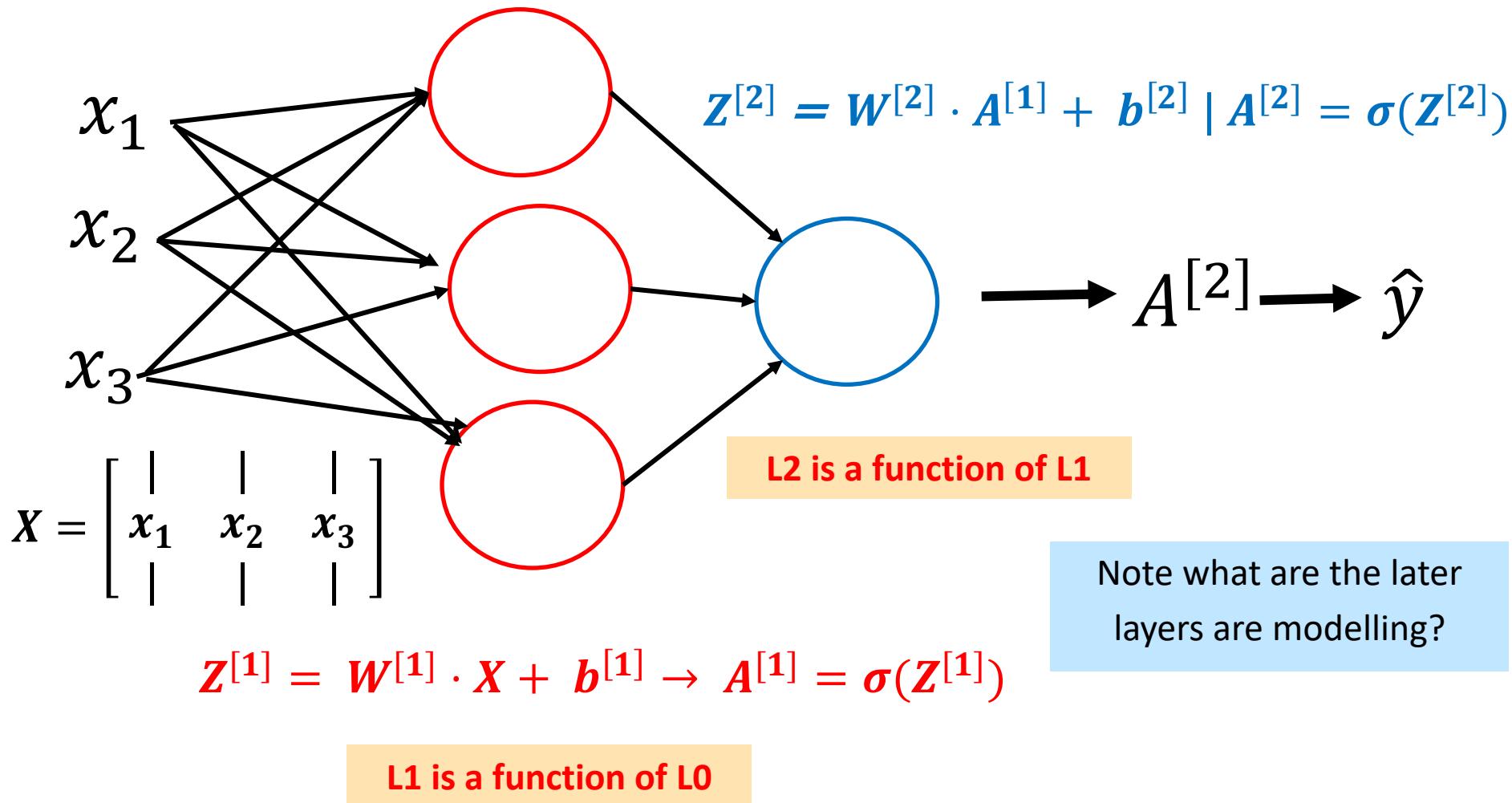
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

### ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# The matrix algebra through our network



# The size of our matrices





$$A^{[0]} = (3,5)$$

$$W^{[1]} = (4,3)$$

$$Z^{[1]} = (4,5)$$

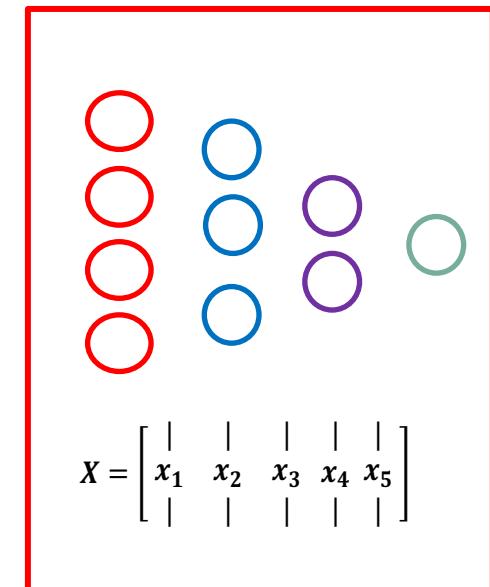
$$b^{[1]} = (4,1)$$

- Let's calculate the sizes:

- $W^{[l]} = (n^{[l]}, n^{[l-1]}) = \mathbf{d}W^{[l]}$

- $Z^{[l]} = A^{[l]} = (n^{[l]}, m) = \mathbf{d}Z^{[l]} = \mathbf{d}A^{[l]}$

- $b^{[l]} = (n^{[l]}, 1) = \mathbf{d}b^{[l]}$



# Our matrices (formulas)


=





+


$$Z^{[1]} = (4,5)$$

$$W^{[1]} = (4,3)$$

$$A^{[0]} = (3,5)$$

$$b^{[1]} = (4,1)$$

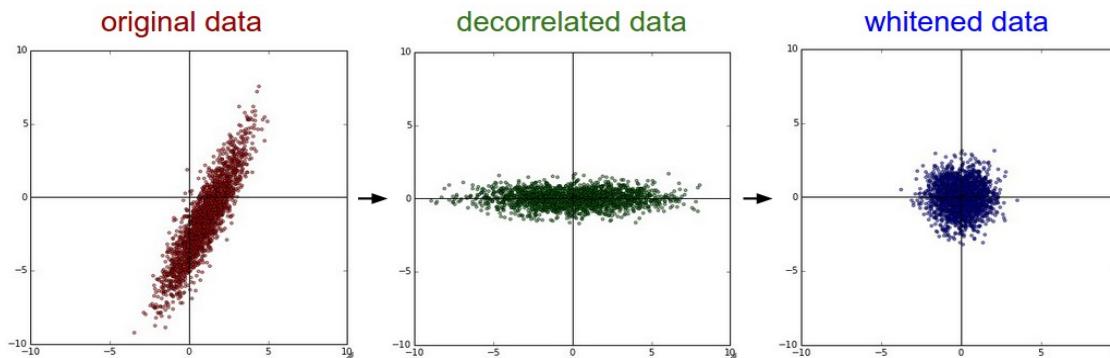
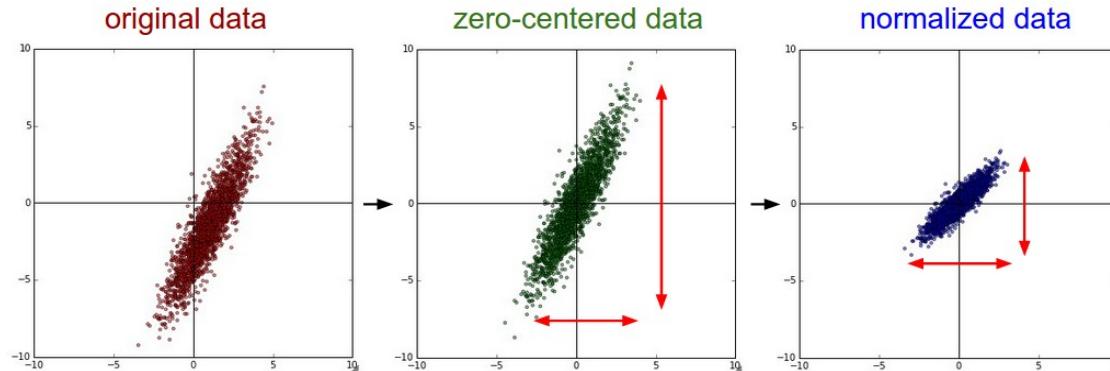
$$Z^{[L]} = W^{[L]} \cdot A^{[L-1]} + b^{[L]}$$

# How to ‘A.I’ in (6) simple steps

1. **Initialize your network**
  - **Select number of hidden layers, number of hidden units per layer, activation functions**
2. **‘Feed Forward’**
  - **We have covered this essentially!**
3. Calculate a loss
  - How bad did our model do?
4. ‘Back propagate’
  - This will allow us to see how our weights affect our loss
5. Update weights
6. Repeat
7. (Optional – regularize)

# Data Preparation:

- **zero-center/ zero-mean** the data
- **normalize** the data.
- Whitening/sphering
  - zCA whitening



# Weight initialization

- Proposition 1: Simply initialize the weights to zero?

**Bad Idea – Why?**

- The weights would all do the same thing, output same values, receive same gradient etc.
- Proposition 2: Add a small non-zero value

**Ok – but has some issues  
(See AT1)**

- Proposition 3: Other, advanced methods

**(See AT1)**

# Weight initialization Part 2

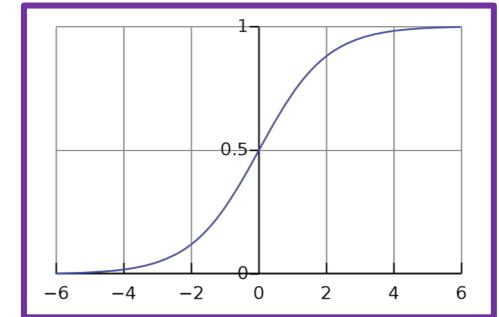
- **Xavier initialisation (Good for Tanh activations)**
  - ([Source](#), Glorot & Bengio et al, 2010)
  - Initialize weights from distribution with zero mean and variance.
  - Multiply by  $\sqrt{\frac{1}{n_i+n_o}}$  (number input, output units)
- **He initialisation (Good for Relu activations)**
  - ([Source](#) He at al., 2015)
  - Similar to above but multiply by  $\sqrt{\frac{2}{n_i}}$

# Creating a loss function (cross-entropy)

- Let us consider a binary outcome, then we can use a sigmoid loss function

$$L(\hat{Y}, y) = -((y \log \hat{Y}) + (1 - y) \log(1 - \hat{Y}))$$

- Consider if  $Y=1$ 
  - $(1 - y) \log(1 - \hat{Y})$  becomes 0
  - We are left with  $-(1) \log(\hat{Y})$
  - What would be  $\hat{Y}$  when we are really sure it IS? (0.999?)
    - $\log(0.990) = \text{Zero!}$
- Consider if  $Y=0$ 
  - $-(y \log \hat{Y})$  becomes 0
  - We are left with  $-(1) \log(1 - \hat{Y})$
  - What would be  $\hat{Y}$  when we are really sure it ISN'T? (0.001?)
    - $\log(0.999) = \text{Zero!}$

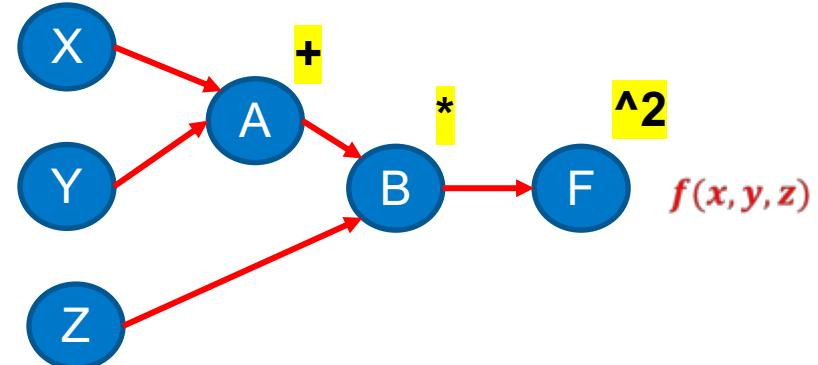


# BACK-PROPAGATION

- Once we have decided on a neural network architecture, the model boils down to a set of weights
- We cannot use optimisation methods from linear models to determine all these weights. Why?
- We use the back-propagation algorithm instead:
  1. Find the error at the output layer (i.e. classification error, MSE)
  2. Use this error to adjust the weight between the last hidden layers and the output layer → we *back-propagate* the error at the output layer to the hidden layer
  3. Repeat 1 and 2 until all hidden layers are updated

# BACK-PROPAGATION

- How does A affect  $f(x, y, z)$ ?  
 $\frac{df}{dA}$  ??
- Hint: What we have are intermediary functions:  
 $A = x + y$



Luckily calculus has a neat trick called the 'chain rule'

$$\frac{df}{dA} = \frac{\partial f}{\partial B} * \frac{\partial B}{\partial A}$$

So what is  $\frac{\partial f}{\partial B}$ ?

$$\frac{\partial f}{\partial B} = 2B \text{ and } \frac{\partial B}{\partial A} = Z$$

$$\text{so } \frac{df}{dA} = 2BZ$$

# BACK-PROPAGATION for Matrices

- What if we don't have scalars but have matrices?

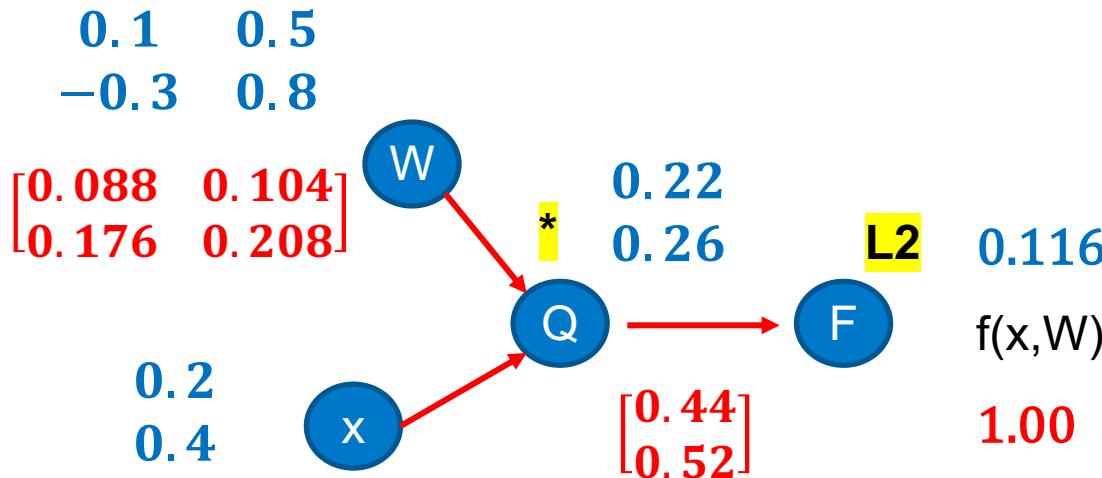
Everything holds true, except we now get a **jacobian matrix** (first order partial derivatives of the weight matrices with respect to the loss function)

- An example computational graph:
  - **Matrix multiplication** (then)
  - **L2 transformation** (sum of the square of each element) v

$$f(x, W) = \|\mathbf{W} \cdot \mathbf{x}\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$$

# Jacobian walkthrough (backwards)

$$f(x, W) = \|\mathbf{W} \cdot \mathbf{x}\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$$



Key takeaway:

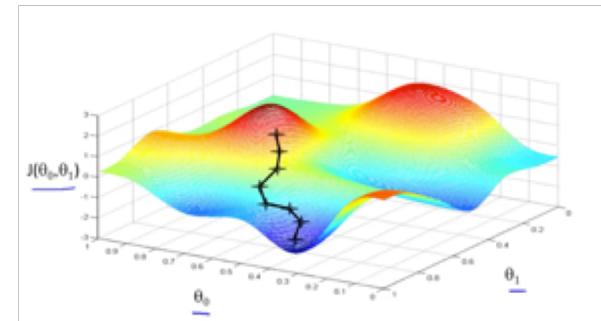
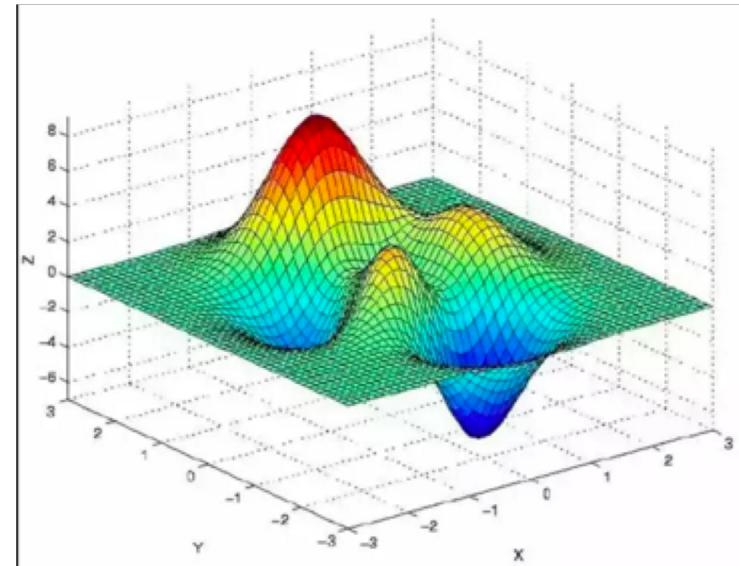
The fundamental principle of the ‘Chain Rule’ can be applied to matrices in order to determine the influence of weight matrix elements on final output (loss)

# UPDATING THE WEIGHTS - GRADIENT DESCENT

- In multiple dimensions, the **gradient** is the vector of **partial derivatives** along each dimension.
- Think of it like a person standing somewhere in a valley of lots of ups and downs. Your goal is to find the lowest point (and efficiently!)
  - The height is the loss function, X-Y-Z are your parameters

(do we have only 3 parameters? How many do we have?)

1. Start somewhere
2. Take a bunch of little steps around us and see where is the steepest slope (negative derivative)
3. Then take a small step in that direction
4. Repeat



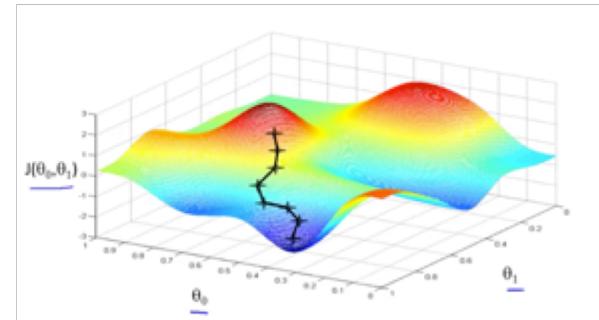
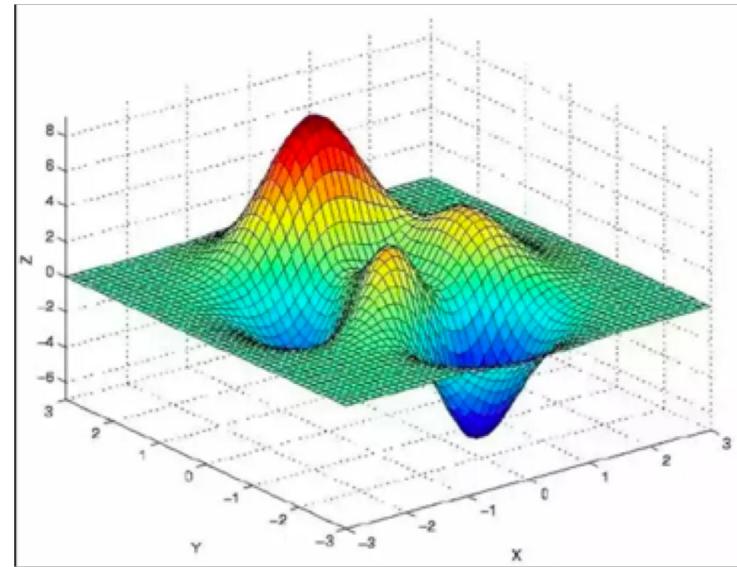
# UPDATING THE WEIGHTS - GRADIENT DESCENT

The small step is termed the learning rate:

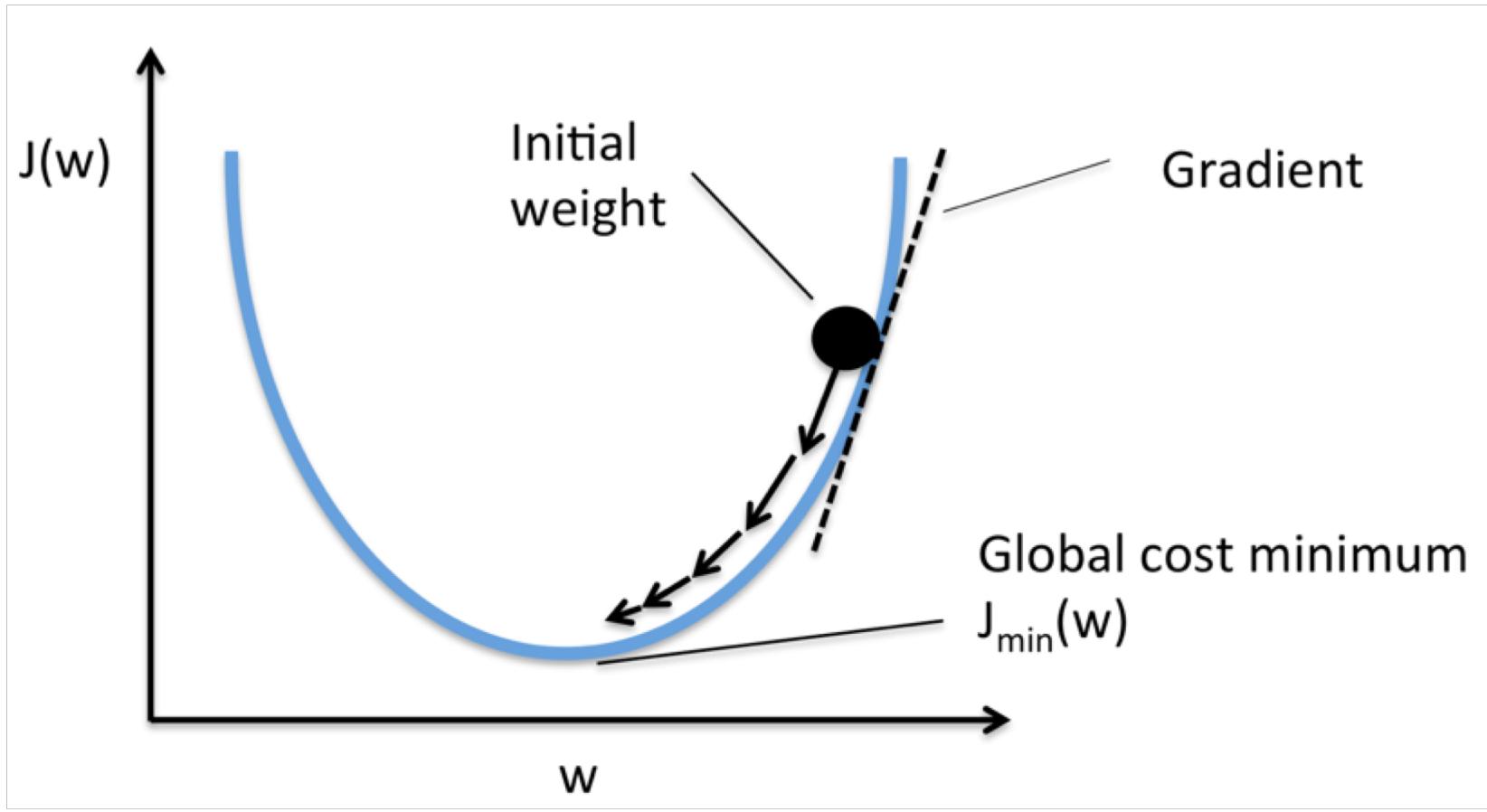
- If  $\alpha$  is too small, gradient descent will take a long time
- If  $\alpha$  is too large, you can fail to converge or overshoot the minimum (diverge)

[https://www.youtube.com/watch?  
v=WnqQrPNYz5Q](https://www.youtube.com/watch?v=WnqQrPNYz5Q)

[https://www.youtube.com/watch?v=  
IHZwWFHWa-w](https://www.youtube.com/watch?v=IHZwWFHWa-w)



# GRADIENT DESCENT



# OVERVIEW OF TODAY

Schedule:

- > Review of last week
- > A bit more (maths) from last week
- > Optimisation algorithms
- > Regularisation methods
- > Software & Hardware

Lab:

- > Introduction to Tensorflow & Keras

# FORWARD PROP (SUMMARY)

- For the first and second layers:

- $Z^{[1]} = W^{[1]} \cdot A^{[0]} + b^{[1]}$ 
  - $A^{[1]} = g^{[1]}(Z^{[1]})$
- $Z^{[2]} = W^{[1]} \cdot A^{[1]} + b^{[1]}$ 
  - $A^{[2]} = g^{[2]}(Z^{[2]})$

- For some layer l:

- $Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$ 
  - $A^{[l]} = g^{[l]}(Z^{[l]})$

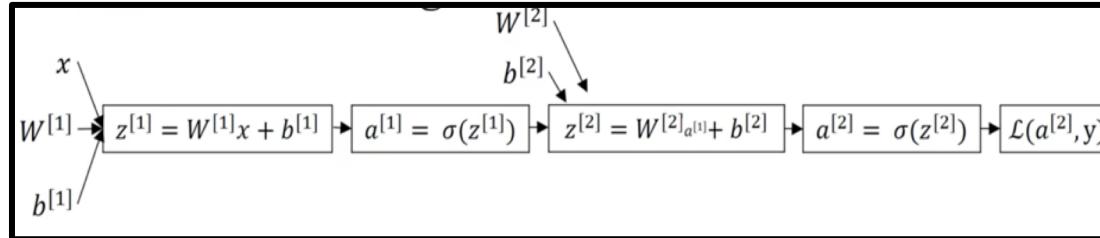
## Matrix Sizes

- $W^{[l]} = (n^{[l]}, n^{[l-1]})$ 
  - $= dW^{[l]}$
- $Z^{[l]} = A^{[l]} = (n^{[l]}, m)$ 
  - $= dZ^{[l]} = dA^{[l]}$
- $b^{[l]} = (n^{[l]}, 1)$ 
  - $= db^{[l]}$

# BACK PROP MATHS

$$L(a^{[2]}, y) = -((y \log a^{[2]}) + (1 - y) \log(1 - a^{[2]}))$$

(1/m) \*



Let's go backwards on this computational graph

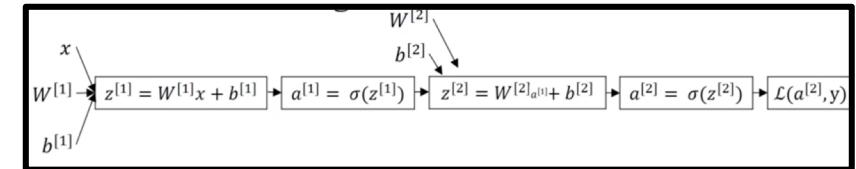
- Differentiate loss w.r.t  $a^{[2]}$

$$da^{[2]} = \frac{dl}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \quad \left( \text{Note: } \frac{d \ln(x)}{dx} = \frac{1}{x} \right)$$

- Differentiate loss w.r.t  $z^{[2]}$

$$dz^{[2]} = \frac{\partial l}{\partial z^{[2]}} = \frac{\partial l}{\partial a^{[2]}} * \frac{da^{[2]}}{dz}$$

# BACK PROP MATHS



- Differentiate loss w.r.t  $a^{[2]}$

$$da^{[2]} = \frac{dl}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \quad \left( \text{Note: } \frac{d \ln(x)}{dx} = \frac{1}{x} \right)$$

- Differentiate loss w.r.t  $z^{[2]}$

$$dz^{[2]} = \frac{\partial l}{\partial z^{[2]}} = \frac{\partial l}{\partial a^{[2]}} * \frac{da^{[2]}}{dz}$$

$$\frac{\partial l}{\partial a^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \quad \& \quad \frac{da^{[2]}}{dz} = a^{[2]}(1-a^{[2]}) \quad \left( \text{Note: } \frac{d \sigma(x)}{dx} = \sigma(x) * (1 - \sigma(x)) \right)$$

$$\therefore \frac{\partial l}{\partial z^{[2]}} = a^{[2]} - y$$

- Differentiate loss w.r.t  $w^{[2]}$

$$dw^{[2]} = \frac{\partial l}{\partial w^{[2]}} = \frac{\partial l}{\partial z^{[2]}} * \frac{dz^{[2]}}{dw^{[2]}} = \mathbf{a}^{[2]} - \mathbf{y} * a^{[1]T}$$

$$\therefore dz^{[2]} a^{[1]T}$$

+ Bias

$$db^{[2]} = dz^{[2]}$$

# DIFFERENTIATE SIGMOID

<https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e>

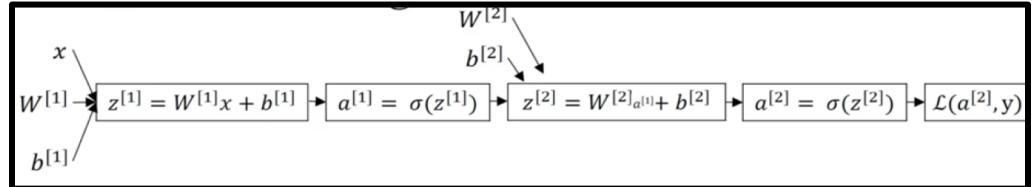
Let's say we want to find the derivative of  $y = \sigma(x) = (1 + \exp(-x))^{-1}$ . So we have:

$$\begin{aligned}\frac{dy}{dx} &= (-1)(1 + \exp(-x))^{-2} \frac{d}{dx}(1 + \exp(-x)) \\&= (-1)(1 + \exp(-x))^{-2}(0 + \frac{d}{dx}\exp(-x)) \\&= (-1)(1 + \exp(-x))^{-2}(\exp(-x))\frac{d}{dx}(-x) \\&= (-1)(1 + \exp(-x))^{-2}(\exp(-x))(-1) \\&= \frac{\exp(-x)}{(1 + \exp(-x))^2} \\&= \frac{1 + \exp(-x) - 1}{(1 + \exp(-x))^2} \\&= \frac{1 + \exp(-x)}{(1 + \exp(-x))^2} - \frac{1}{(1 + \exp(-x))^2} \\&= \sigma(x) - (\sigma(x))^2 \\&= \sigma(x) \cdot (1 - \sigma(x))\end{aligned}$$

<https://math.stackexchange.com/questions/78575/derivative-of-sigmoid-function-sigma-x-frac11e-x>

# CONT.....

- Back to our data:



- Third step back:

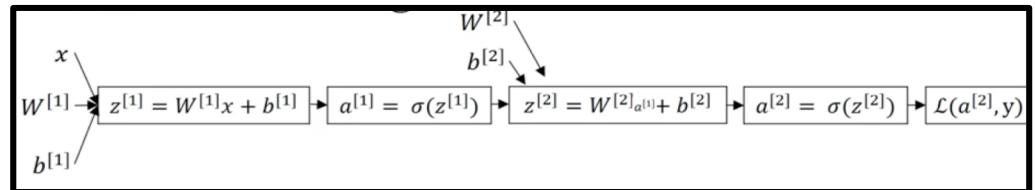
- $$da^{[1]} = \frac{dl}{da^{[1]}} = \frac{\partial l}{\partial z^{[2]}} * \frac{dz^{[2]}}{da^{[1]}} = dz^{[2]}W^{[2]}$$

- Fourth step back:

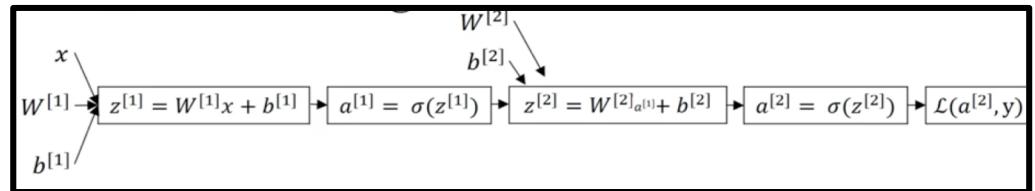
- $$dz^{[1]} = \frac{\partial l}{\partial z^{[1]}} = \frac{\partial l}{\partial a^{[1]}} * \frac{da^{[1]}}{dz^{[1]}}$$
  - $(dz^{[2]}W^{[2]T}) a^{[1]}(1 - a^{[1]})$ 
    - $\frac{da^{[1]}}{dz^{[1]}}$  is  $g^{[1]}'(z^{[1]})$  as we won't always activate with  $\sigma$
- $$dw^{[1]} = \frac{\partial l}{\partial w^{[1]}} = \frac{\partial l}{\partial z^{[1]}} * \frac{dz^{[1]}}{dw^{[1]}}$$
  - $dz^{[1]} a^{[0]T}$  recalling  $a^{[0]} = x$
- $db^{[1]} = dz^{[1]}$

# LET'S VECTORISE!

- $dZ^{[2]} = A^{[2]} - Y$
- $dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
- $db^{[2]} = \frac{1}{m} * \sum dZ^{[2]}$ 
  - $dZ^{[1]} = (W^{[2]T} dZ^{[2]}) * g^{[1]'}(Z^{[1]})$
  - $dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$  noting ( $A^{[0]} = X$ )
  - $db^{[1]} = \frac{1}{m} * \sum dZ^{[1]}$



# LET'S GENERALISE!



- $dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$
- $db^{[l]} = \frac{1}{m} * \sum dZ^{[l]}$
- $dZ^{[l]} = (W^{[l+1]T} dZ^{[l+1]}) * g^{[l]'}(Z^{[l]})$ 
  - OR >>>  $(dA^{[l]}) * g^{[l]'}(Z^{[l]})$
- $dA^{[l-1]} = W^{[l]T} dZ^{[l]}$

# Loss functions – Softmax

- What if we have multiple classes? Introducing Softmax function

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

- Probability Y is some class (given the network) is equal to e-to-the-power of the ‘score’ for that class divided by the sum of each other ‘score’ to the other classes, raised by e
- We are essentially computing a probability distribution over our classes
- Therefore the loss can be seen as the ***negative log likelihood of the correct class***

In Math:

$$L_i = -\log P(Y = y_i | X = x_i)$$

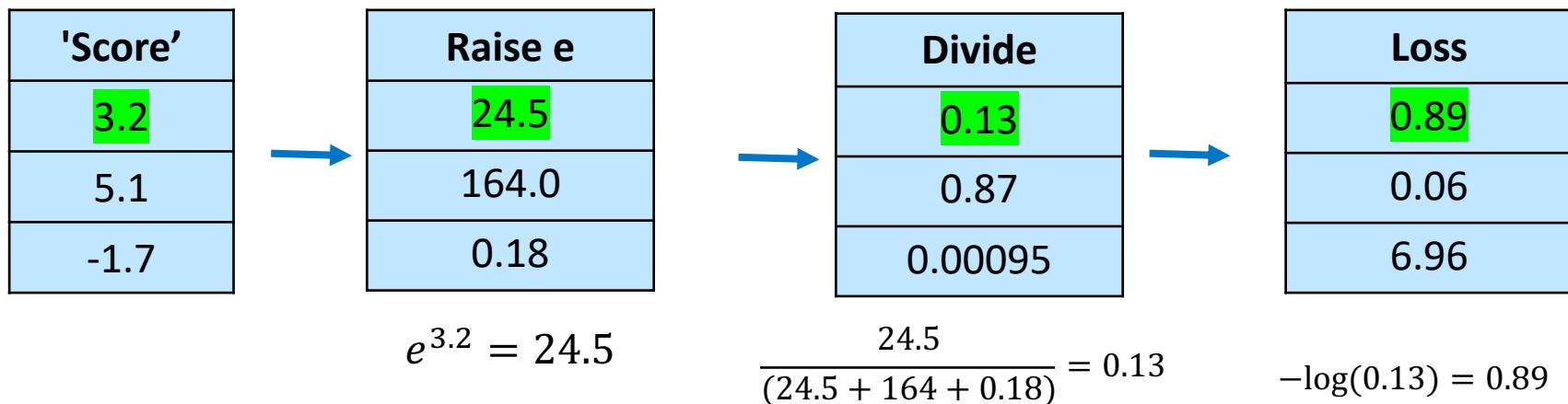
$$L_i = -\log \left( \frac{e^{y_i}}{\sum_j e^{s_j}} \right)$$

Loss = -log(**Probability of true class**)  
(Consider  $-\log(1)$ )

# Loss functions – Softmax

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{Thus } L_i = -\log \left( \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

- Probability Y is some class (given the network) is equal to e-to-the-power of the ‘score’ for that class divided by the sum of each other ‘score’ to the other classes, raised by e

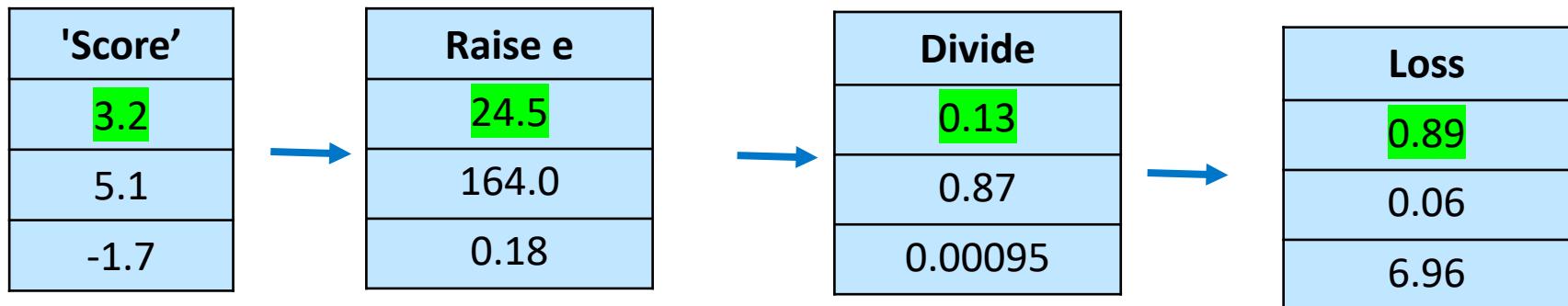


# What do these numbers mean?

- Consider asking the network (you) to place a 'bet' on which class it will be. It can bet any amount and then the loss is how much you lose from that bet if it turns out that that corresponding row is the correct class.

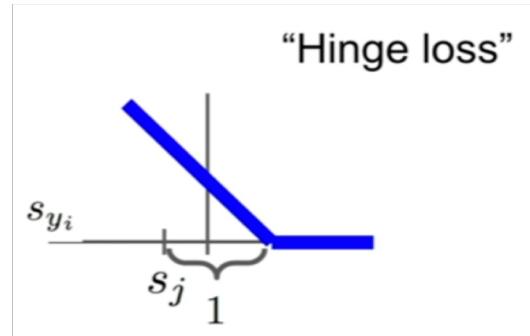
Obviously you want to lose less. Now on the slide consider;

- IF the top row was the correct class. You lose 0.89 because you actually bet more on the middle row.
- IF the middle class was the correct class, you only lose 0.06 On the middle row you bet the most on that row.
- IF the bottom row was the correct class, you made a negative bet so you were really really sure it was NOT the correct class when in fact it was. So you get a huge loss.



# Loss functions – Other

- Hinge loss (Multi-class SVM)



- Regression (MSE, MAE, RMSE)

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (Predicted_i - Actual_i)^2}{N}}$$

Others

[https://towardsdatascience.com/comm  
on-loss-functions-in-machine-learning-  
46af0ffc4d23](https://towardsdatascience.com/comm on-loss-functions-in-machine-learning-46af0ffc4d23)

# OVERVIEW OF TODAY

Schedule:

- > Review of last week
- > A bit more (maths) from last week
- > Optimisation algorithms
- > Regularisation methods
- > Software & Hardware

Lab:

- > Introduction to Tensorflow & Keras

# OPTIMISATION: (MINI)BATCH, STOCHASTIC GD

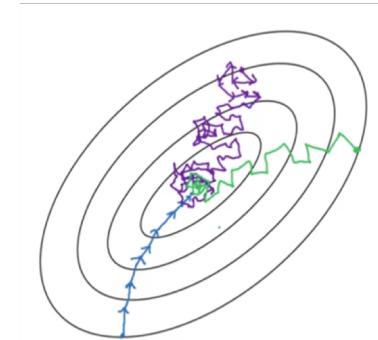
- The problem: If we have many training examples, our input matrix becomes huge!
  - Consider  $X$  where we have 5 or 10 or 15 million examples....

$$X = \begin{bmatrix} | & | & | & | & | \\ x_1 & x_2 & x_3 & x_4 & \dots & x_n \\ | & | & | & | & | \end{bmatrix}$$

- A solution is to undertake **minibatch** gradient descent, so splitting our dataset into size of 256 samples, our cost function becomes:
  - $J^{\{t\}}(w, b) = \frac{1}{256} \sum_{i=1}^{256} L(\widehat{Y^{(i)}}, y^{(i)})$ 
    - **Where  $\{t\}$**  is our minibatch
    - Simply loop through all minibatches, forward-backprop as normal
    - One round (minibatch) is considered an **epoch**

# OPTIMISATION: (MINI)BATCH, STOCHASTIC GD

- If you have:
  - minibatch size is **m** then you have **batch** gradient descent. This can take huge processing power, though will step and converge nicely.
  - Minibatch size is **1** then you have **stochastic** gradient descent. You lose the vectorisation speedup and is quite noisy.
  - If minibatch size is  $>1 < m$  then you have **minibatch** gradient descent
- Common minibatch sizes are 64, 128, 256, 512
  - Technically also a hyperparameter (though not a primary one)



# OPTIMISATION ALGORITHMS: MOMENTUM

- Consider the formula for exponentially weighted moving averages:
  - $V_t = \beta V_{t-1} + (1 - \beta)x_t$  where  $\beta \in [0,1]$  and  $x_t$  is some value at the current time.
    - We can approximate that  $V_t$  is averaging over  $\frac{1}{1-\beta}$  days
  - We are computing an average that takes into account **n** number of trailing days. Consider **0.9** for  $\beta$
  - $V_{100} = 0.9V_{99} + (1 - 0.9)x_{100}$ 
    - $V_{99}$  itself = **0.9** $V_{98}$  and  $(1 - 0.9)x_{99}$  etc
  - Therefore  $V_{100}$  is a small weight (0.1)  $x_{100}$  of the current value and a **sum of all previous temperatures**
    - multiplied by an **exponentially decaying function**
      - So  $0.1x_{100} + 0.1 * 0.9x_{99} + 0.1 * (0.9)^2x_{98}$ ...etc

# OPTIMISATION ALGORITHMS: MOMENTUM

- (From [source](#))
- Adjust  $\beta$  to be lower, you get a more noisy graph as you let local effects be more powerful.
- Larger  $\beta$  smooths more but will respond slower.



# OPTIMISATION ALGORITHMS: MOMENTUM

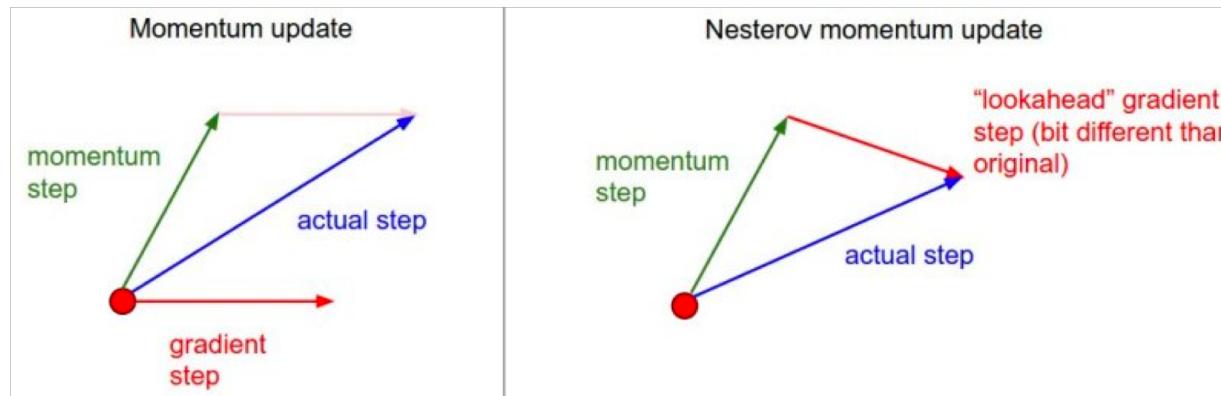
- How does this work for us?
  - Initialise  $V_{dW}, V_{db} = 0$  where  $V_{dW}$  is the same dimensions as  $W$  and  $V_{db}$  has same dimensions as  $b$
- On iteration t:
  - Compute  $dW, db$  as usual on the current mini-batch
  - $V_{dW} = \beta V_{dW} + (1 - \beta)dW$
  - $V_{db} = \beta V_{db} + (1 - \beta)db$
  - Update:
    - $W := W - \alpha * V_{dW}$
    - $b := b - \alpha * V_{db}$
- An extension to this (nesterov momentum) does a ‘step forward backwards’ alteration (often oversteps)

Best  $\beta$ ?

Commonly 0.9 is used  
(‘Averaging over the last 10  
gradients’  $\beta$ ) but perhaps try  
others?

# OPTIMISATION ALGORITHMS: NESTEROV

- An extension to momentum does a ‘step forward backwards’ alteration (often oversteps) (source in notes)
- Rather than taking a weighted average of Velocity and Gradient (at current point) to make our actual step, we instead:
  - Step in direction of velocity
  - Take gradient at that point
  - Go back to original point and mix together these two for the actual step.

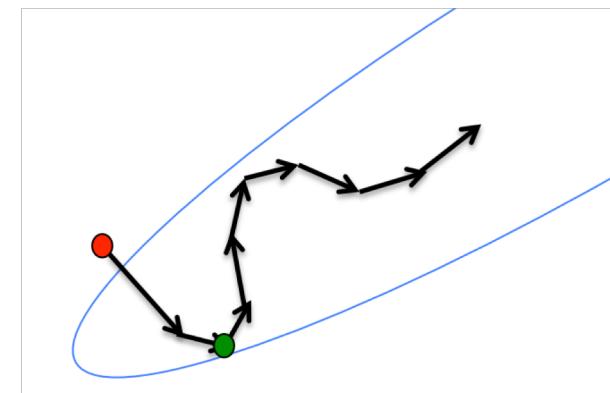


# OPTIMISATION ALGORITHMS: RMSPROP

- [Source](#) (Hinton on coursera...)
- Stands for ‘Root Mean Squared’ Prop
  - Initialise  $S_{dW}, S_{db} = 0$
  - On iteration t:
    - Compute  $dW, db$  as usual on the current mini-batch
    - $S_{dW} = \beta S_{dW} + (1 - \beta)dW^2$
    - $S_{db} = \beta S_{db} + (1 - \beta)db^2$
    - Update:
      - $W := W - \alpha * \frac{dW}{\sqrt{S_{dW} + \varepsilon}}$
      - $b := b - \alpha * \frac{db}{\sqrt{S_{db} + \varepsilon}}$

## Why?

This keeps a moving average of the squared gradient of each weight. This should ‘squash’ high oscillating training run.



# OPTIMISATION ALGORITHMS: ADAM

- ‘Adaptive Moment Estimation’ (Momentum + RMSProp)
  - $\beta_m$  (momentum  $\beta$ ) as first ‘moment’ and  $\beta_r$  (rmsprop  $\beta$ ) as second ‘moment’
  - Initialise  $S_{dW}, S_{db}, V_{dW}, V_{db} = 0$
  - On iteration t:
    - Compute  $dW, db$  as usual on the current mini-batch
    - $V_{dW} = \beta_m V_{dW} + (1 - \beta_m)dW$  &  $V_{db} = \beta_m V_{db} + (1 - \beta_m)db$ 
      - Add a bias correction  $V_{dW}^c = \frac{V_{dW}}{1-\beta_m^t}$  &  $V_{db}^c = \frac{V_{db}}{1-\beta_m^t}$
    - $S_{dW} = \beta_r S_{dW} + (1 - \beta_r)dW^2$  &  $S_{db} = \beta_r S_{db} + (1 - \beta_r)db^2$ 
      - Add a bias correction  $S_{dW}^c = \frac{S_{dW}}{1-\beta_r^t}$  &  $S_{db}^c = \frac{S_{db}}{1-\beta_r^t}$
    - Update:
      - $W := W - \alpha * \frac{V_{dW}^c}{\sqrt{S_{dW}^c + \epsilon}}$
      - $b := b - \alpha * \frac{V_{db}^c}{\sqrt{S_{db}^c + \epsilon}}$

Commonly:

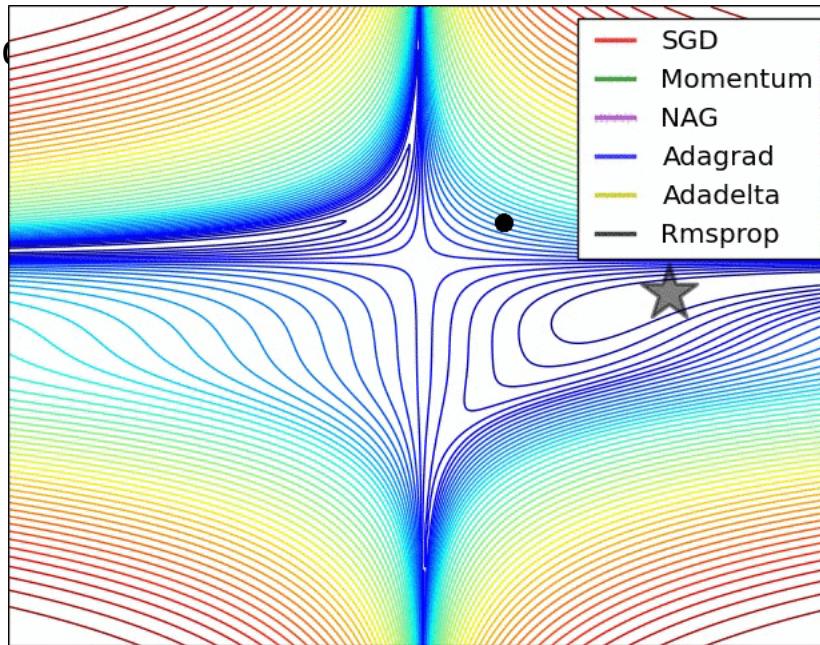
- $\beta_m = 0.9$
- $\beta_r = 0.999$
- $\alpha$  (start) 1e-3 or 5e-4

## Why bias correct?

Moments are biased towards zero during initial time steps  
(especially with small  $\beta$ )

# OPTIMISATION ALGORITHMS: MORE??

- ‘Nesterov-accelerated Adaptive Moment Estimation’ (Nesterov momentum + Adam) ([source](#)) (NADAM)
- Even more????
  - <https://arxiv.org/pdf/1609.04747.pdf>
  - Adagrad, Adadelta, Rmsprop



# OVERVIEW OF TODAY

Schedule:

- > Review of last week
- > A bit more (maths) from last week
- > Optimisation algorithms
- > Regularisation methods
- > Software & Hardware

Lab:

- > Introduction to Tensorflow & Keras

# REGULARISATION: L2 (WEIGHT DECAY)

- L2 regularization can be used in neural networks as well. We add a small penalty term to our loss function which becomes a hyperparameter to tune.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m L\left(\widehat{Y^{(i)}}, y^{(i)}\right) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|_F^2$$

$$\text{Where } \|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij})^2$$

- That is, for each weight matrix in the NN we simply add the square of each element in those matrices.
- Note the F rather than 2 since this is called the Frobenius Norm rather than L2 norm for matrices.
- This ‘zeros out’ some hidden units with smaller weights

# REGULARISATION: L2 & BACK PROP

- We will need to adjust our backprop formulas slightly:

$$dw^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} + \frac{\lambda}{m} w^{[l]}$$

Therefore our update step can remain the same as:

$$w^{[l]} := w^{[l]} - \alpha * dw$$

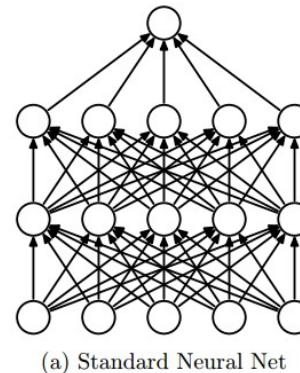
Or expanding is:

- $w^{[l]} := w^{[l]} - \alpha * \left( \frac{1}{m} dZ^{[l]} A^{[l-1]T} \right) - \alpha * \left( \frac{\lambda}{m} w^{[l]} \right)$
- Rearranging:  $w^{[l]} := w^{[l]} - \alpha * \left( \frac{\lambda}{m} w^{[l]} \right) - \alpha * \left( \frac{1}{m} dZ^{[l]} A^{[l-1]T} \right)$
- Rearranging the first part :  $w^{[l]} \left( 1 - \frac{\alpha \lambda}{m} \right)$

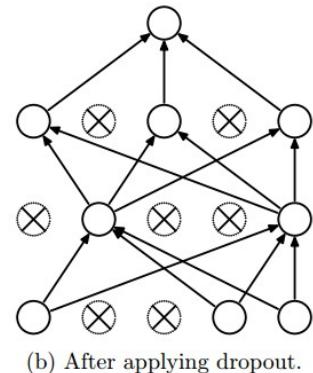
See how we are ‘decaying’ the weight matrix?

# REGULARISATION: DROPOUT

- ([Source](#), Srivastava et al, 2014) Or ([Source](#) Hinton, Srivastava et all 2012) Or ([Source](#) Farley, Goodfellow et al 2014)
- We will need to adjust our backprop formulas slightly:
- On each forward pass (Training example), randomly set some neurons to zero So remove all the ingoing and outgoing links to that node
- The probability of dropping is a hyperparameter,  $p$ 
  - You could also set this differently depending on the layer. So perhaps your last 1 or 2 layers don't need dropout if they are ( $n=2$ ) or ( $n=3$ ) but for more dense layers in the middle you set a higher  $p$ .



(a) Standard Neural Net



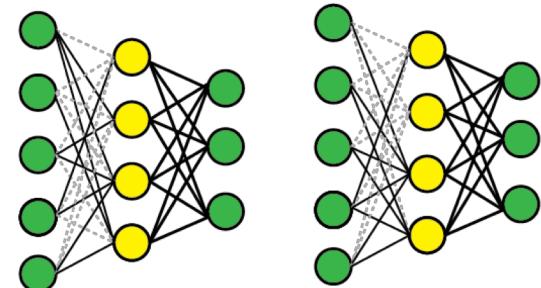
(b) After applying dropout.

# REGULARISATION: DROPOUT



# REGULARISATION: DROPOUT

- Why does it work?
  - Forces the network to have *redundant representation*
  - Can theoretically show it is a version of L2
- A **Problem**:
  - Our output is random and may take longer to train.
- A **Solution** ('Inverted Dropout'):
  - At training time, divide by P so that at test time there is no difference.
  - For example, if we have a 50 unit layer and drop (0.2 = 10 units), we scale the output (A) by ( $A/0.8$ )
  - Some nice code [here](#)
- There is also '[DropConnect](#)' (a generalization of dropout where you zero out some of the weight matrix)



## EXTRA: LEARNING RATE DECAY

- This technique can speed up training and prevent the ‘bounce’ around as the model converges towards an optimum.
- We can thus implement **standard learning rate decay** as follows:

$$\alpha = \frac{1}{1 + d * n} * \alpha_0$$

- Where d is the decay rate and n is the epoch number
- Other variations:
  - Exponential decay:  $\alpha = 0.95^n * \alpha_0$
  - K decay:  $\alpha = \frac{k}{\sqrt{n}} * \alpha_0$  OR  $\alpha = \frac{k}{t} * \alpha_0$  where t is minibatch number
  - Discrete ('staircase'/ 'half-life') decay = half  $\alpha$  at certain t's
  - Manual

# OVERVIEW OF TODAY

Schedule:

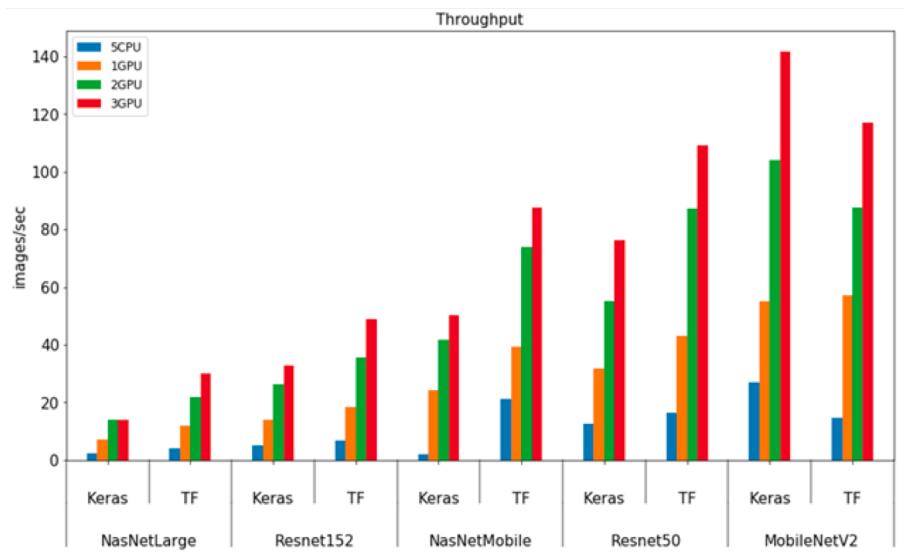
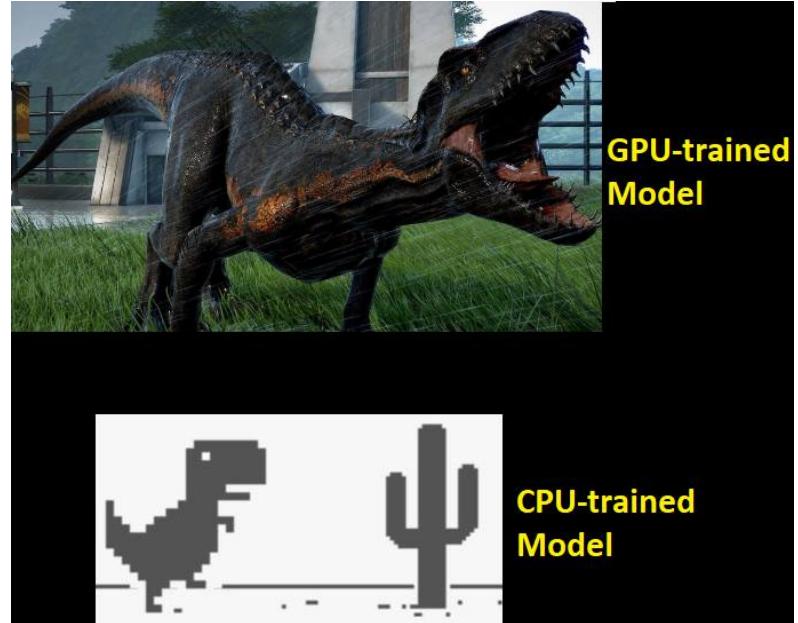
- > Review of last week
- > A bit more (maths) from last week
- > Optimisation algorithms
- > Regularisation methods
- > Software & Hardware

Lab:

- > Introduction to Tensorflow & Keras

# GPU - CPU – TPU?

- CPUs are a few very complex cores (General processing core)
- GPUs are hundreds/thousands of simpler cores (can run thousands of concurrent threads)
  - Uses parallel processing of the matrix work in NN
  - (NVIDIA GPU required for CUDA)
- TPU is a special custom build by Google specifically for deep learning.



# GPU - CPU – TPU?

## CPU vs GPU

	# Cores	Clock Speed	Memory	Price
<b>CPU</b> (Intel Core i7-7700k)	4 (8 threads with hyperthreading )	4.4 GHz	Shared with system	\$339
<b>CPU</b> (Intel Core i7-6950X)	10 (20 threads with hyperthreading )	3.5 GHz	Shared with system	\$1723
<b>GPU</b> (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	\$1200
<b>GPU</b> (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	\$399

**CPU:** Fewer cores, but each core is much faster and much more capable; great at sequential tasks

**GPU:** More cores, but each core is much slower and “dumber”; great for parallel tasks

[http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture8.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf)

# WHAT'S IN A FRAMEWORK?

- Tensorflow the most popular and has all the bells and whistles
- Keras is 'syntactic sugar' (a wrapper) for tensorflow.
- Caffe2 not used as much (from facebook)
- Pytorch recently becoming more popular. Very fast.
- Others (not as popular):
  - CNTK (Microsoft), MXNet (Amazon)



<https://www.youtube.com/watch?v=SJldOOs4vB8>

# LAB – TENSORFLOW & KERAS

