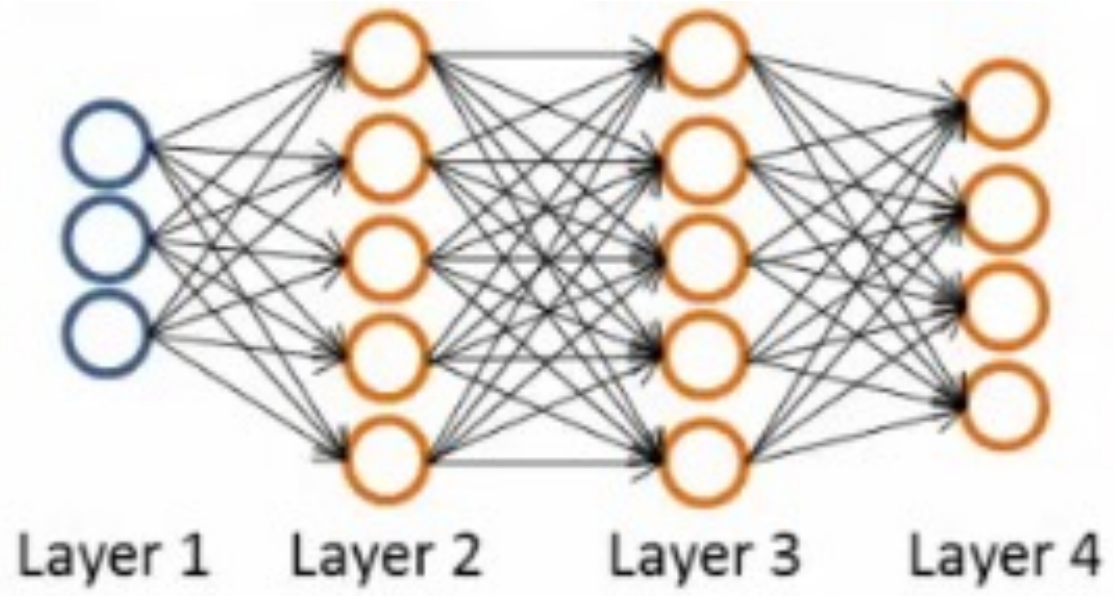
Training Deep Neural Networks

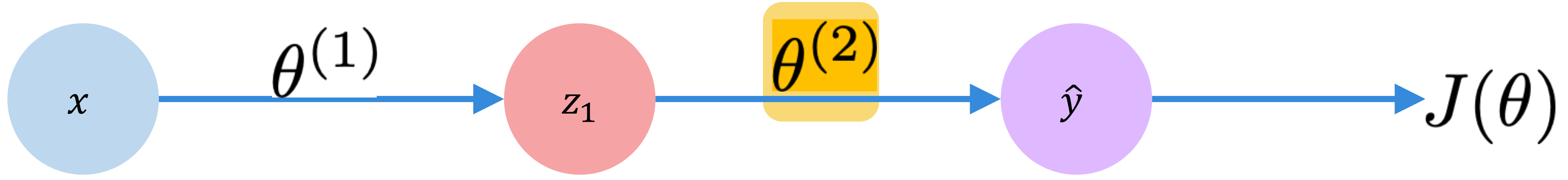
What we have learnt so far?

- Training set is $\{(x^1, y^1), (x^2, y^2), (x^3, y^3) \ldots (x^m, y^m)\}$
- L = number of layers in the network
- s_l = number of units in layer l



Gradient descent steps

How much does a small change in weights affect the empirical loss ?



Algorithm

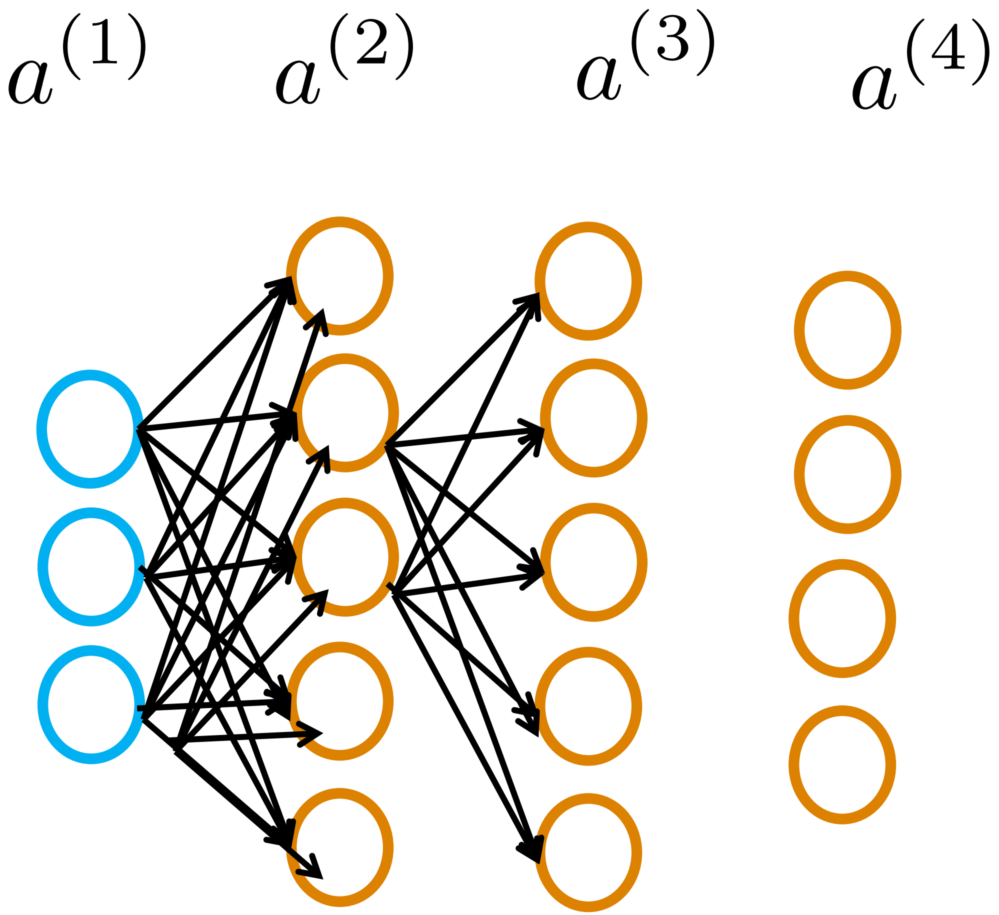
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

$$\frac{\partial J(\theta)}{\partial \theta^{(2)}} = \frac{\partial J(\theta)}{\partial y} * \frac{\partial z_1}{\partial \theta^{(2)}}$$

$$\frac{\partial J(\theta)}{\partial \theta^{(1)}} = \frac{\partial J(\theta)}{\partial y} * \frac{\partial y}{\partial z_1} * \frac{\partial z_1}{\partial \theta^{(1)}}$$

What have we learnt so far?

Forward pass



$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g\left(z^{(2)}\right) \left(\text{add}\, a_0^{(2)}\right)$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g\left(z^{(3)}\right) \left(\text{add}\, a_0^{(3)}\right)$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g\left(z^{(4)}\right)$$

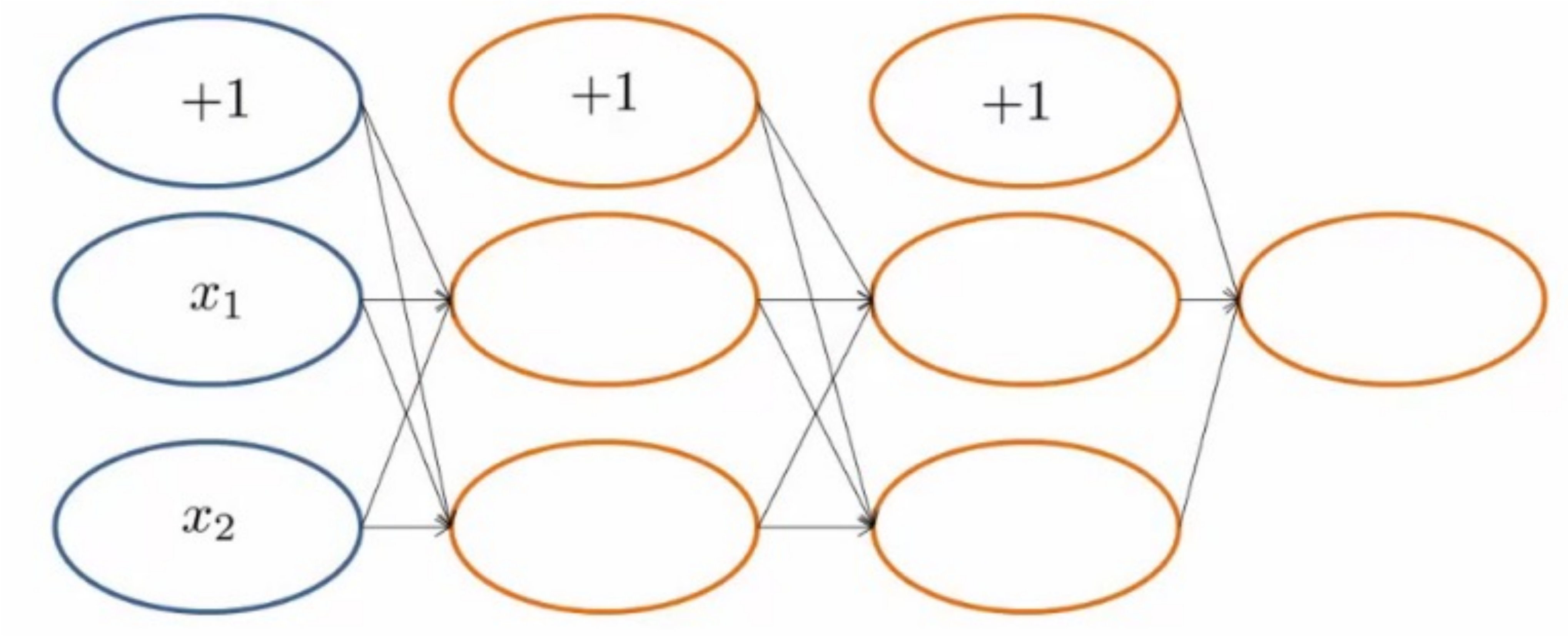
What have we learnt so far? Backpropagation

- Compute $\delta^4 = a^4 - y$, then compute

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

- Θ^3 is the vector of parameters for the layer 3
- $g'(z^3)$ is the first derivative of the activation function g
- $g'(z^3) = a^3 .* (1 - a^3)$ (deriviative of sigmoid)
- **$\delta^3 = (\Theta^3)^T \delta^4 .*(a^3 .* (1 - a^3))$**
- **$\delta^2 = (\Theta^2)^T \delta^3 .*(a^2 .* (1 - a^2))$**

Back propagation intuition

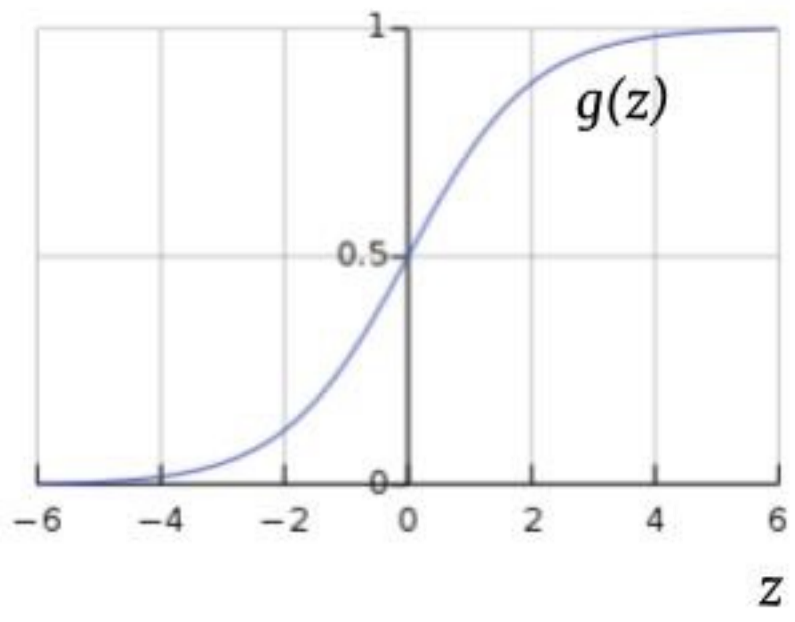


Activation Functions

- What is an activation function?
  - an activation function is used to turn ***z*** which is a linear value, into ***a***, a non-linear value.
- What happens without non-linear activation functions?
- Each layer can use different activation functions (in theory)

Sigmoid Activation Function

- ▶Sigmoid is normally used for final layer
- ▶Intermediate layers can have other activation functions

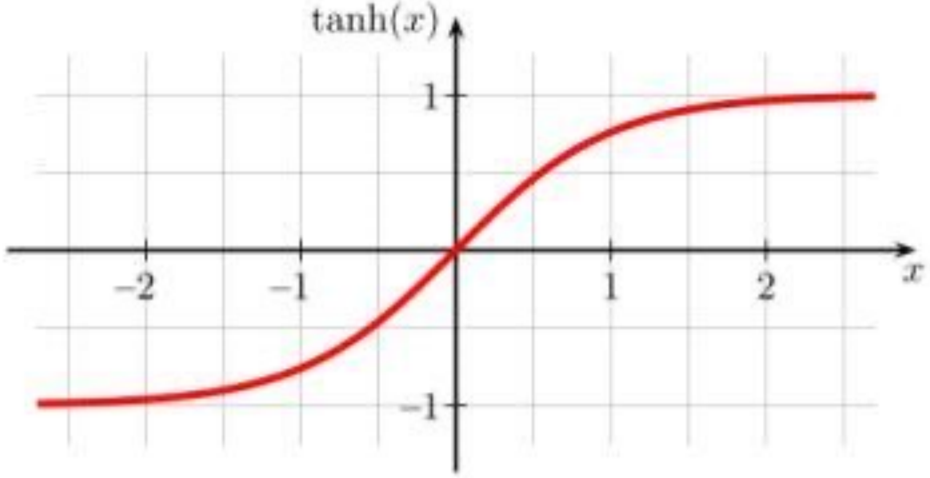


$$a = g(z) = \frac{1}{1+e^{-z}}$$

$$g'(z) = g(z)(1 - g(z)) = a(1 - a)$$

tanh

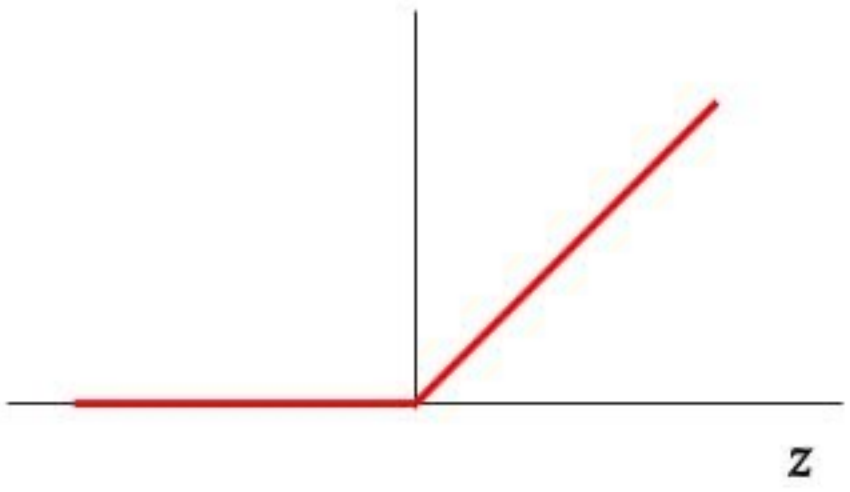
- The tanh function is very similar to the sigmoid function. It is actually just a scaled version of the sigmoid function.
- Tanh works similar to the sigmoid function but is symmetric over the origin. it ranges from -1 to 1.



$$a = g(z) = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$
$$g'(z) = 1 - (tanh(z))^2 = 1 - a^2$$

ReLU (Rectified Linear Unit)

- sigmoid and tanh functions gradient becomes very small if the value of Z is positively large or negatively small, making the descent slow.
- The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

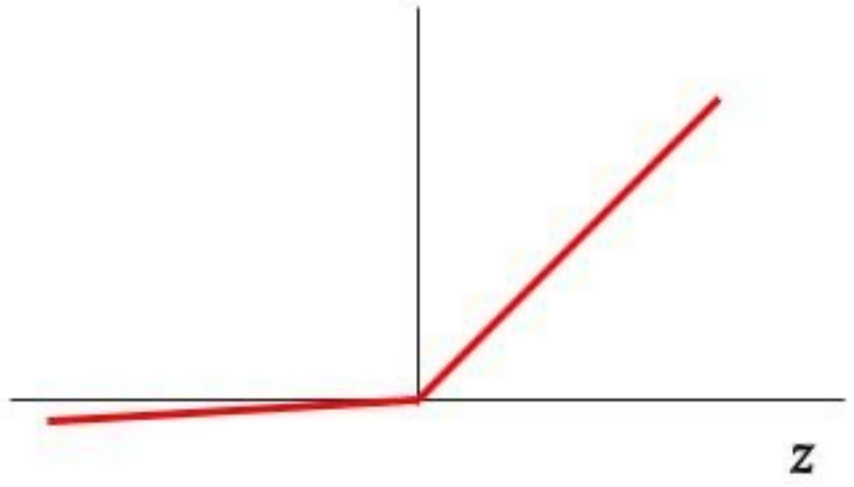


$$g(z) = max(0, z)$$

$$g'(z) = \begin{cases} 0 & \text{if z} < 0 \\ 1 & \text{if z} \geq 0 \end{cases}$$

Leaky ReLU

- But ReLU also falls a prey to the gradients moving towards zero.
- This can create dead neurons which never get activated
- Leaky ReLU is defined to address this problem.



$$g(z) = max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01 & \text{if z} < 0 \\ 1 & \text{if z} \geq 0 \end{cases}$$

Softmax function

► The softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

Which activations functions to use when?

- Sigmoid functions and their combinations generally work better in the case of classifiers
- Sigmoids and tanh functions are avoided due to the vanishing gradient problem
- ReLU function is a general activation function and is used mostly
- If you encounter dead neurons in our networks the leaky ReLU function is the best choice
- ReLU function should only be used in the hidden layers
- As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn't provide with optimal results

Gradient Checking

- Gradient checking is used to double-check that our gradient calculation is correct.
- Use it sparingly as it is very slow.
- For each feature i, calculate the approximate gradient
- Then calculate the difference with the actual gradient calculated during gradient descent:

$$d\theta_{approx} = \frac{J(\theta_i+\epsilon)-J(\theta_i-\epsilon)}{2\epsilon}$$

$$diff = \frac{\|d\theta_{approx}-d\theta\|_2}{\|d\theta_{approx}\|_2+\|d\theta\|_2}$$

More Optimization Algorithms

Gradient Descent with Momentum

- With this method, we update $\boldsymbol{\theta}$ with the exponentially weighted running average of $\boldsymbol{d\theta}$
- This will work faster than the standard gradient descent.
- The intuition is that by averaging the values, it smoothens the oscillation in the gradient descent
- Here $\boldsymbol{V_{d\theta}}$ is an exponentially weighted moving average of $\boldsymbol{d\theta}$. It's initialized to 0 and updated on every iteration:

$$V_{d\theta} = \beta V_{d\theta} + (1-\beta)d\theta \quad \theta = \theta - \alpha V_{d\theta}$$

RMSprop (Root Mean Square Prop)

- The intuition is to dampen large oscillation in some particular directions by penalizing movements that are large (dividing it by square root of itself). The formulas are:

$$S_{d\theta} = \beta S_{d\theta} + (1-\beta)(d\theta)^2$$
$$S_{db} = \beta S_{db} + (1-\beta)(db)^2$$
$$\theta = \theta - \alpha \frac{d\theta}{\sqrt{S_{d\theta}+\epsilon}}$$
$$b = b - \alpha \frac{db}{\sqrt{S_{db}+\epsilon}}$$

- where β is typically 0.999 and epsilon ϵ is very small (e.g. 10^{-8}) and is used to avoid division by zero.

Adam

- Adam (adaptive moment estimation) optimization combines momentum and RMSprop optimizations above.
- This is the momentum part with bias correction:

$$V_{d\theta} = \beta_1 V_{d\theta} + (1 - \beta_1) d\theta \quad S_{d\theta} = \beta_2 S_{d\theta} + (1 - \beta_2)(d\theta)^2$$

$$V_{d\theta}^{corr} = \frac{V_{d\theta}}{1 - \beta_1^t} \quad S_{d\theta}^{corr} = \frac{S_{d\theta}}{1 - \beta_2^{\prime}}$$

$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \quad S_{db} = \beta_2 S_{db} + (1 - \beta_2)(db)^2$$

$$V_{db}^{corr} = \frac{V_{db}}{1 - \beta_1^t} \quad S_{db}^{corr} = \frac{S_{db}}{1 - \beta_2^2}$$

$$\theta = \theta - \alpha \frac{V_{dV}^{\mathrm{corr}}}{\sqrt{S_{d\theta}^{\mathrm{corr}}} + \epsilon} \qquad b = b - \alpha \frac{V_{bcr}}{\sqrt{s_{db}^{ab} + \epsilon}}$$

- The typical hyperparameter values are β1 = 0.9, β2 = 0.999, learning rate α needs to be tuned, and epsilon ϵ is *10*$^{-8}$.

Which one to choose?

- https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

Learning Rate Decay

- ▶Learning rate decay can be used to speed up learning
- ▶Start with higher learning rate and reduce proportional to epoch number
- ▶ it can be calculated as follows, with α0=0.2 and *decay_rate*=1:

$$\alpha = \frac{1}{1+decay\_rate*epoch\_num}\,\alpha_0$$

Batch Normalization

- The idea is to normalize the value of Z in any hidden layers
- This makes the values not fluctuate much between iterations
- This makes the subsequent layers better

Bias and Variance

- When our training set and test set have different distribution, we may have the following results:
  - training error: 1%
  - test error: 10%
  - Is this bias or variance?
  - How to fix it?
- When both training and testing error both are high
  - Is this bias or variance?
  - How it fix it?

Regularization

- How is bias and variance effected by regularization?

Linear regression with regularization

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2$$

Large λ
High bias (underfit)
$\lambda = 10000.\ \theta_1 \approx 0, \theta_2 \approx 0, \ldots$
$h_\theta(x) \approx \theta_0$

Intermediate λ
"Just right"

Small λ
High variance (overfit)
$\lambda \approx 0$

How to choose λ?

- Have a set or range of values to use, for e.g, increment by factors of 2 so 0.01, 0.02, 0.04..., 10
- For each Minimize λ the cost function using gradient descent
- So now we have a set of parameter vectors corresponding to models with different λ value
- Check the cost function value for cross validation set and choose the λ that minimizes cross validation error

Learning curves

- Useful to plot for algorithmic sanity checking or improving performance
- What is a learning curve?

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$