Deep Learning - 2019

Introduction to Neural Networks

Prof. Avishek Anand

Credits: some images taken from http://introtodeeplearning.com

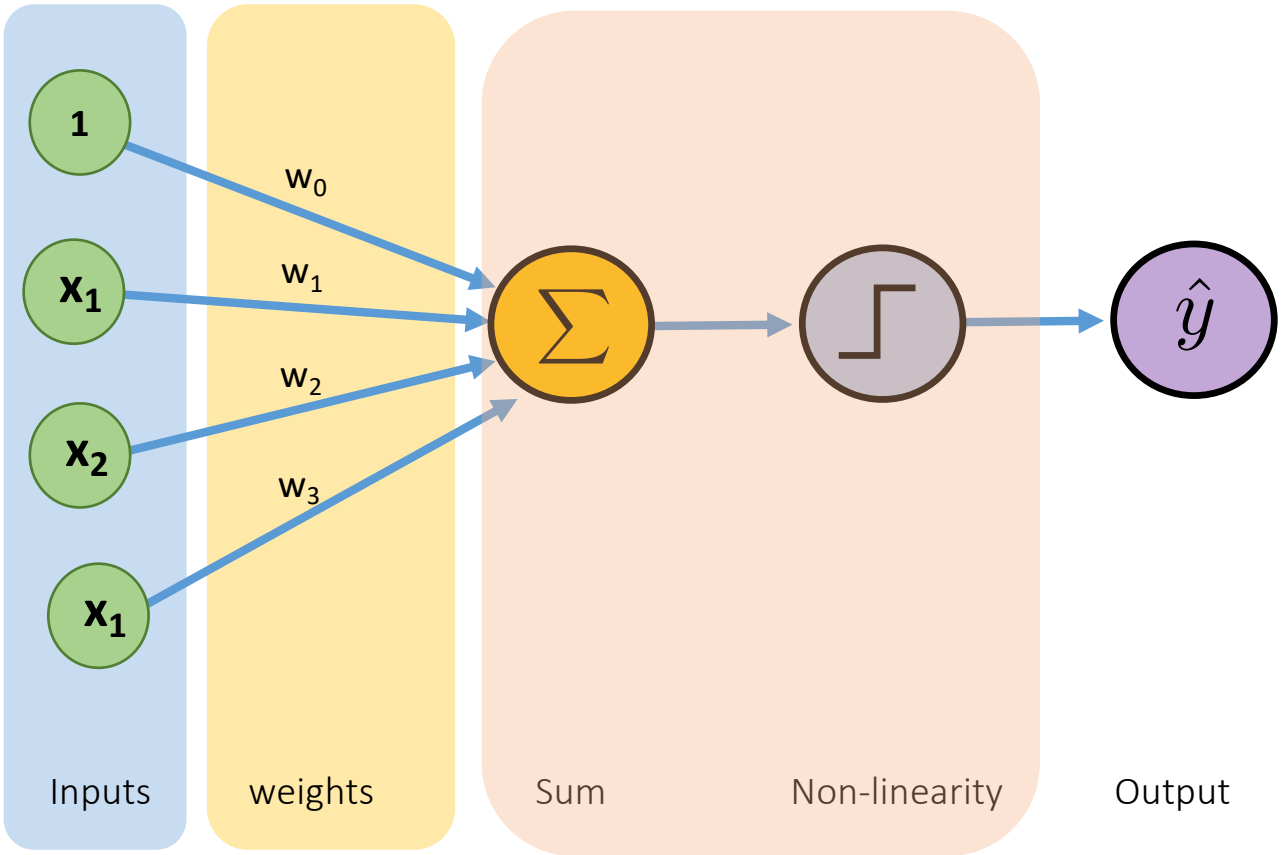
What have we learnt so far…

- Machine Learning is to come up with models that generalize well to unseen data
- We check generalization performance by holding out a test set
- Optimization techniques help reduce training error
  - Design of loss functions is crucial
- Regularization techniques reduce test error

- We looked at linear models for regression and classification

- Today we start with Neural Models for doing the same tasks

Perceptron



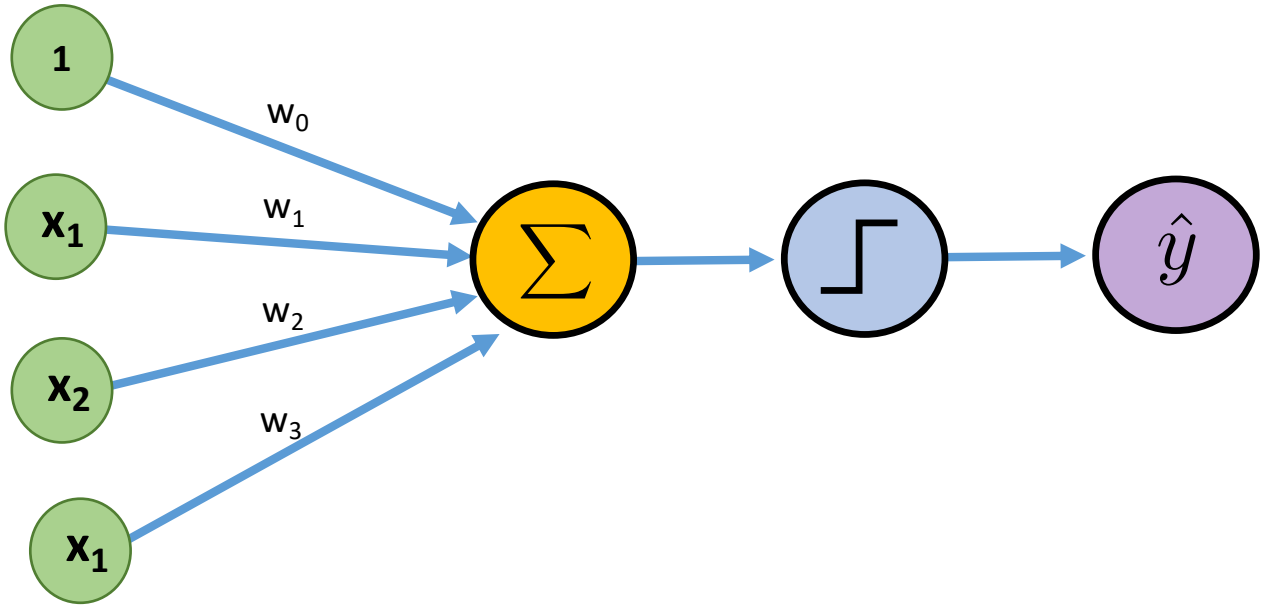
Linear combination

output

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} w_i x_i\right)$$

Non-linearity

Perceptron – Matrix Form



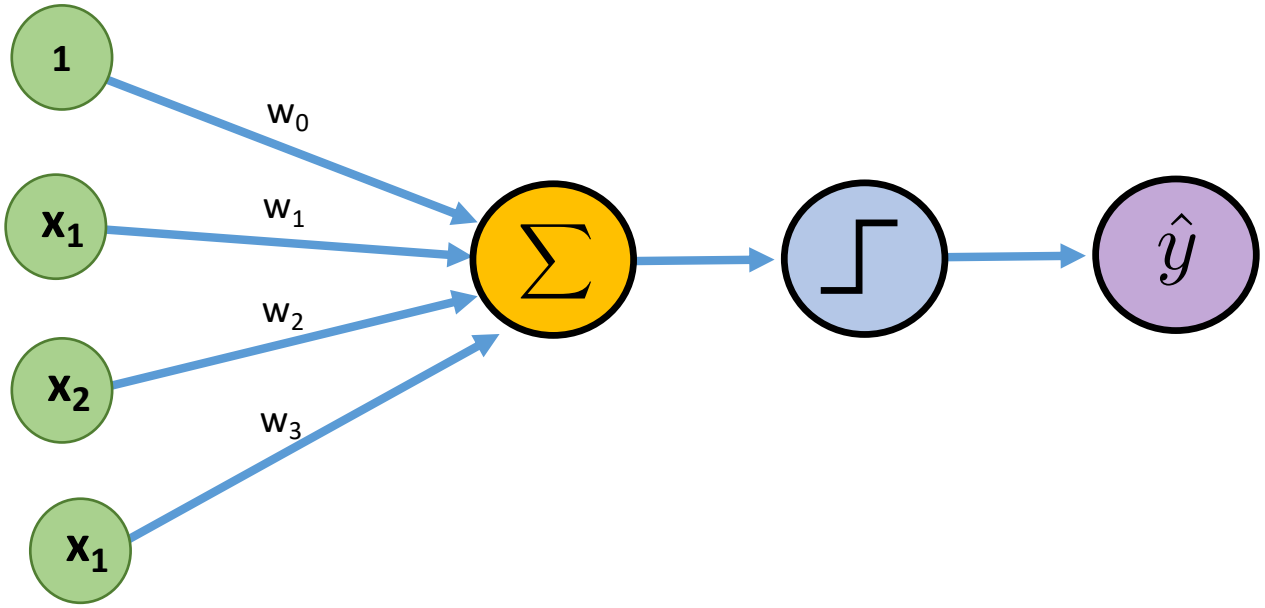
Inputs weights Sum Non-linearity Output

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} w_i x_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{W}^T \boldsymbol{x}\right)$$

$$\text{where: } \boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Perceptron – Activation Function



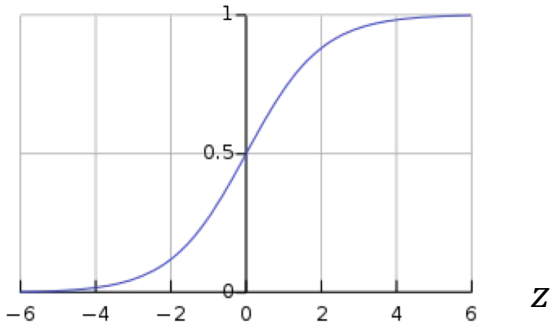
Inputs weights Sum Non-linearity Output

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} w_i x_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{W}^T \boldsymbol{x}\right)$$

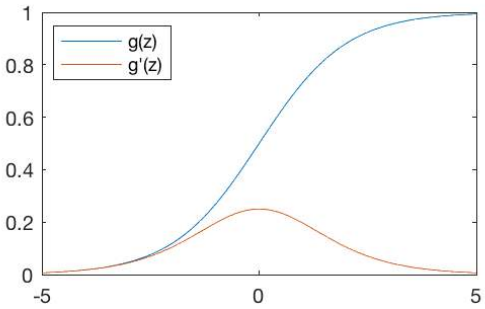
Example: Sigmoid Function

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



Activation Functions

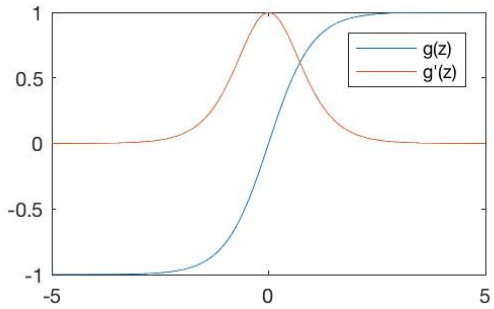
Sigmoid Function



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

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

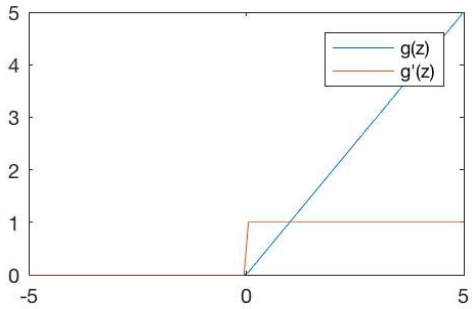
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

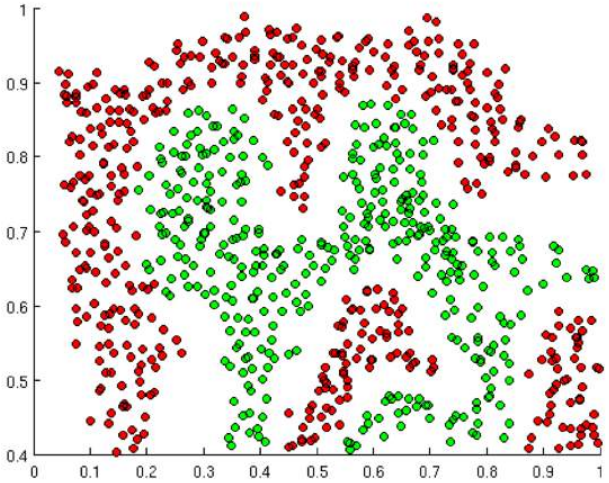


$$g(z) = \max(0, z)$$

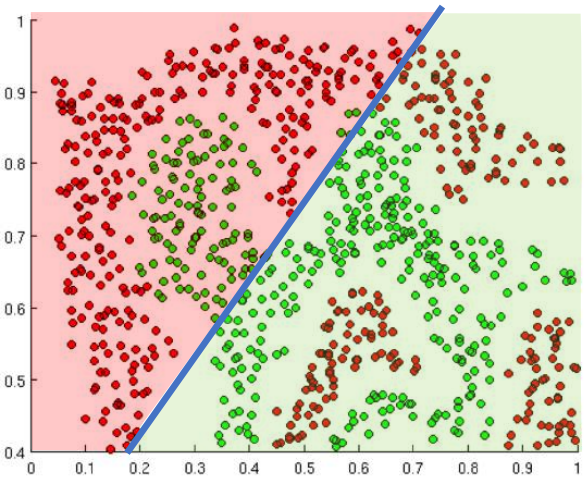
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

More about them later in the course…

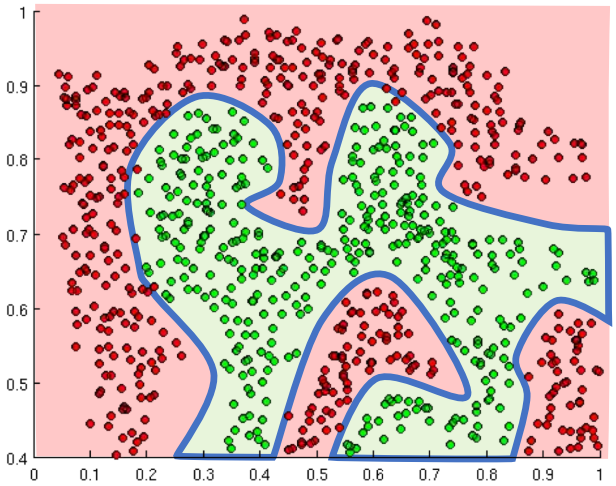
Why Activations



If you want to separate green and red points



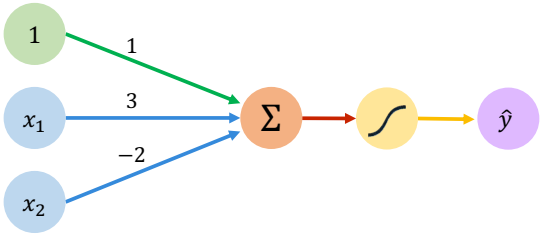
Linear activations produce linear decisions irrespective of the network size



Non linear activations approximate complex decision boundaries

*Activations are key to introducing **non-linearity** into the representation space*

Perceptron – Example



$w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

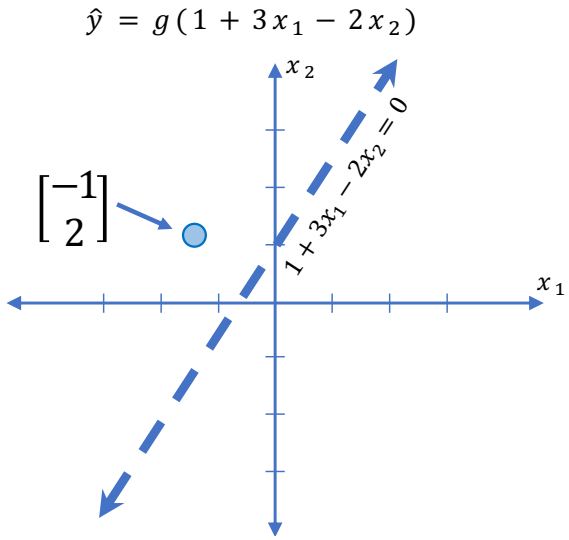
$$\hat{y} = g\left(w_0 + \boldsymbol{W}^T \boldsymbol{x}\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{W}^T \boldsymbol{x}\right)$$

$$= g\left(1 + \begin{bmatrix} 3 \\ -2 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right)$$

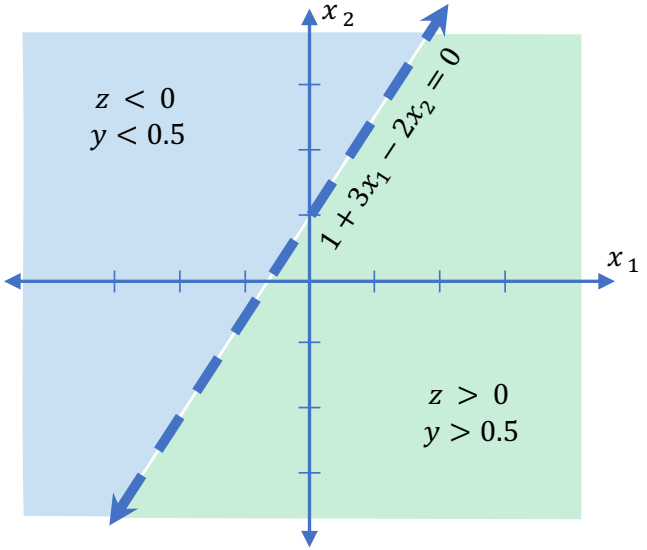
$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$

This is just a line in 2D

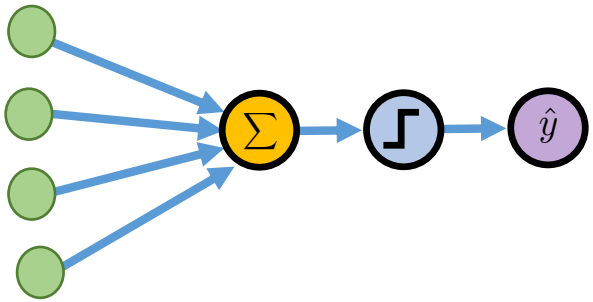


Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

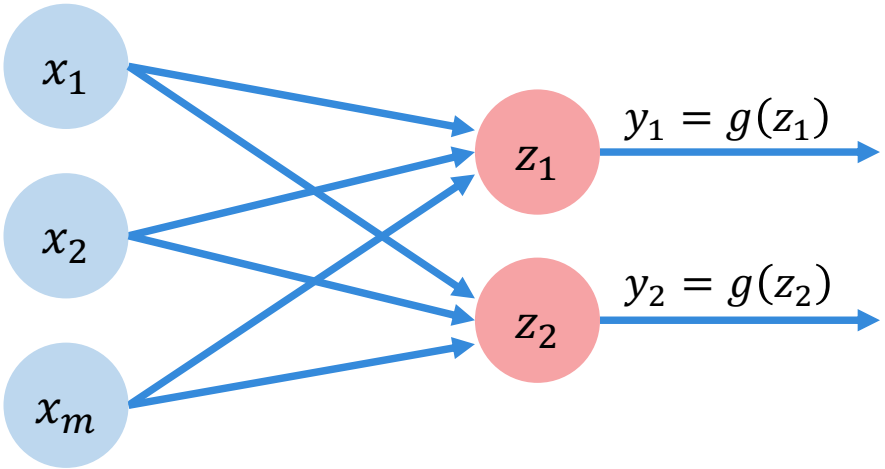
$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$



Multiple Perceptrons

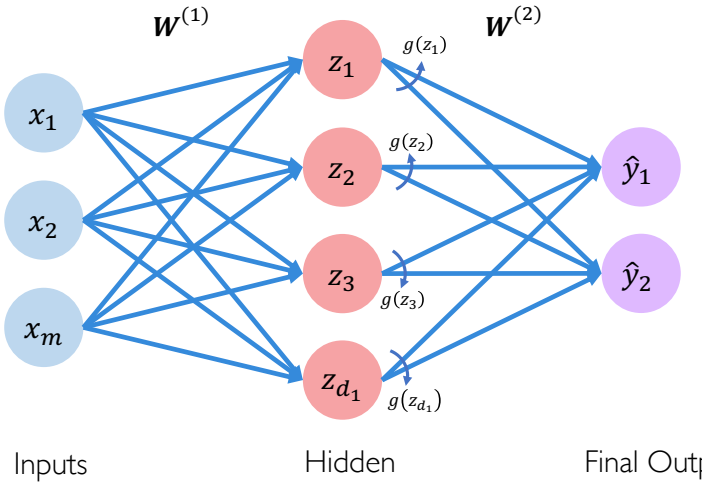


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} w_i x_i\right)$$



$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

Single Layer Neural Network

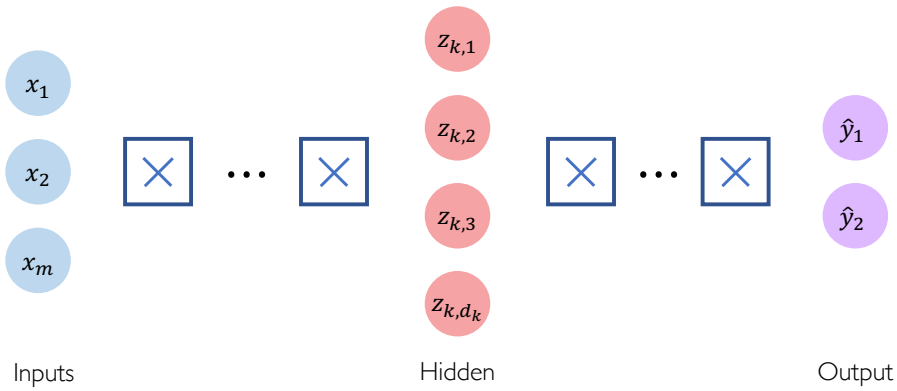


Example for z_2

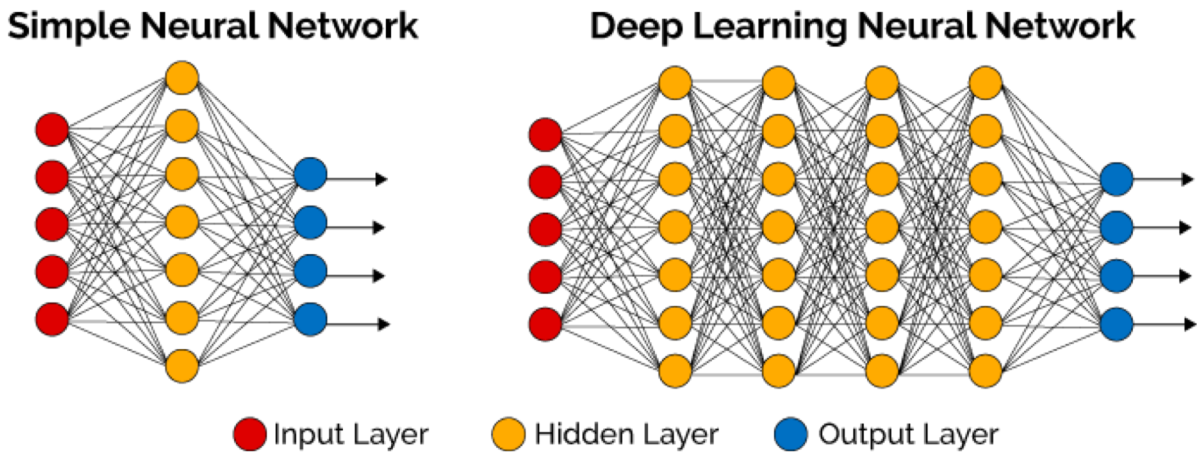
$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} w_{j,2}^{(1)} x_j$$
$$= w_{0,2}^{(1)} + w_{1,2}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + w_{m,2}^{(1)} x_m$$

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} w_{j,i}^{(1)} x_j \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} w_{j,i}^{(2)} z_j \right)$$

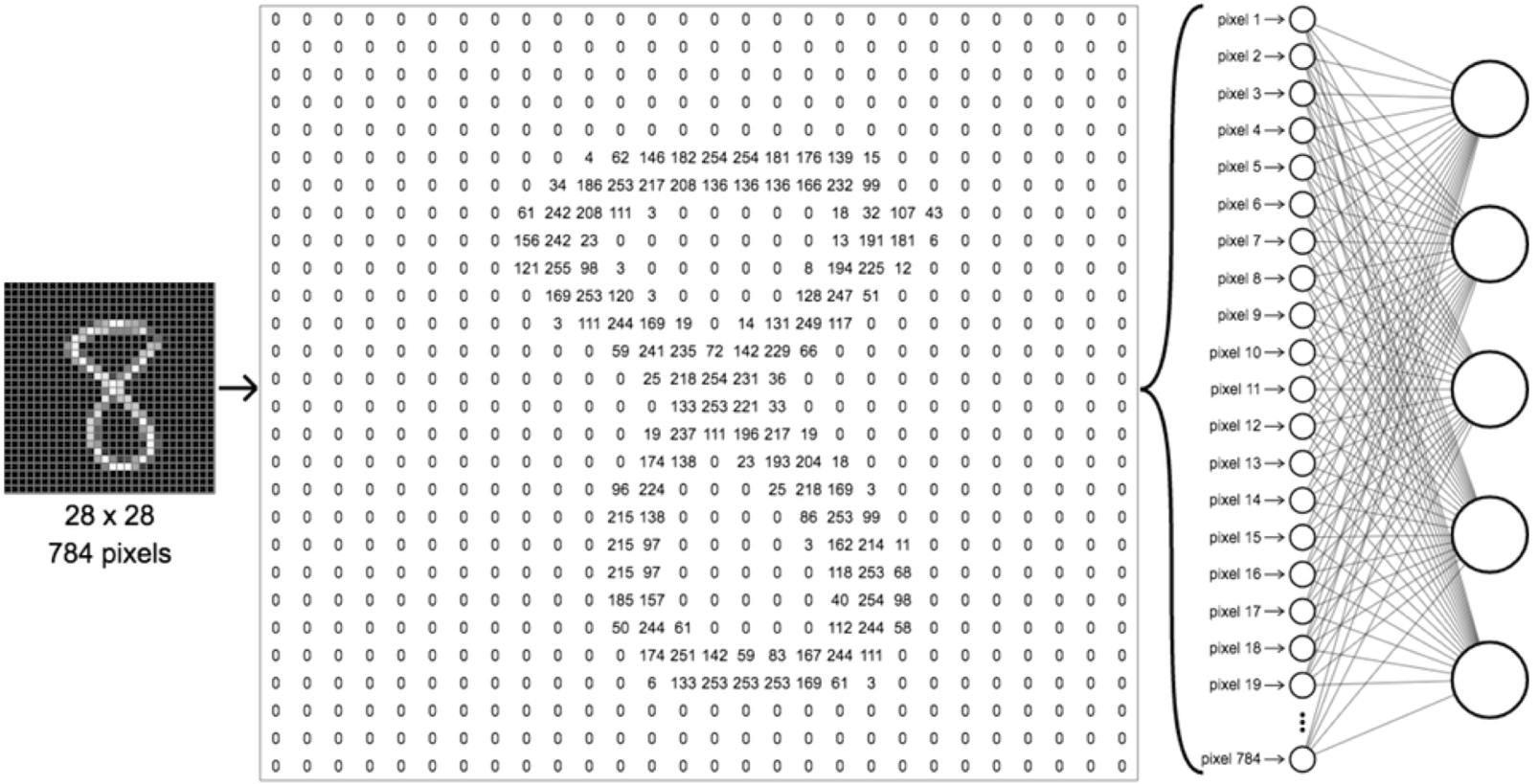
Deep Neural Network



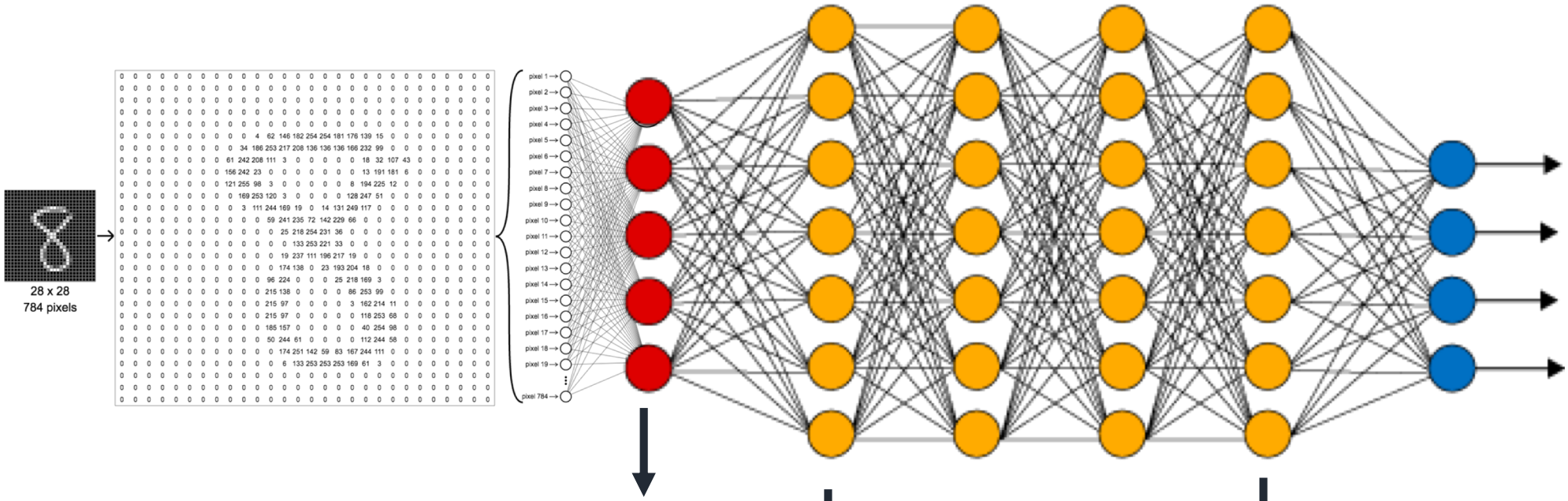
$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g\left(w_{j,i}^{(k)} z_{k-1,j}\right)$$



Linearized input



Neural Network as composition of functions

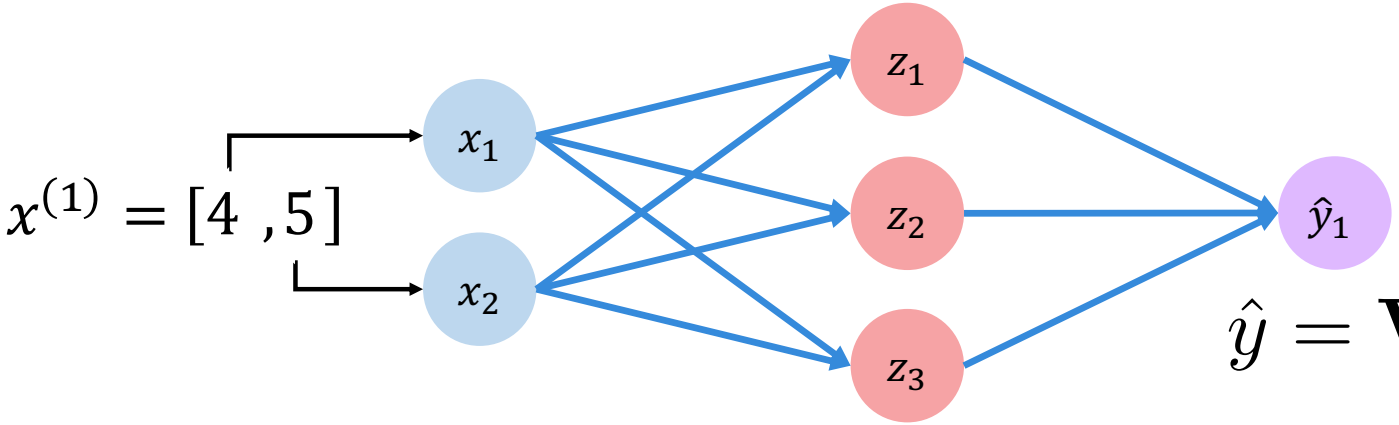


$$\mathbf{z}_1 = g\left(\mathbf{W}_1^T \mathbf{x}\right)$$

$$\mathbf{z}_2 = g\left(\mathbf{W}_2^T \mathbf{z}_1\right)$$

$$\mathbf{z}_k = g\left(\mathbf{W}_k^T \mathbf{z}_{k-1}\right)$$

Applying a Neural Network – Forward Pass



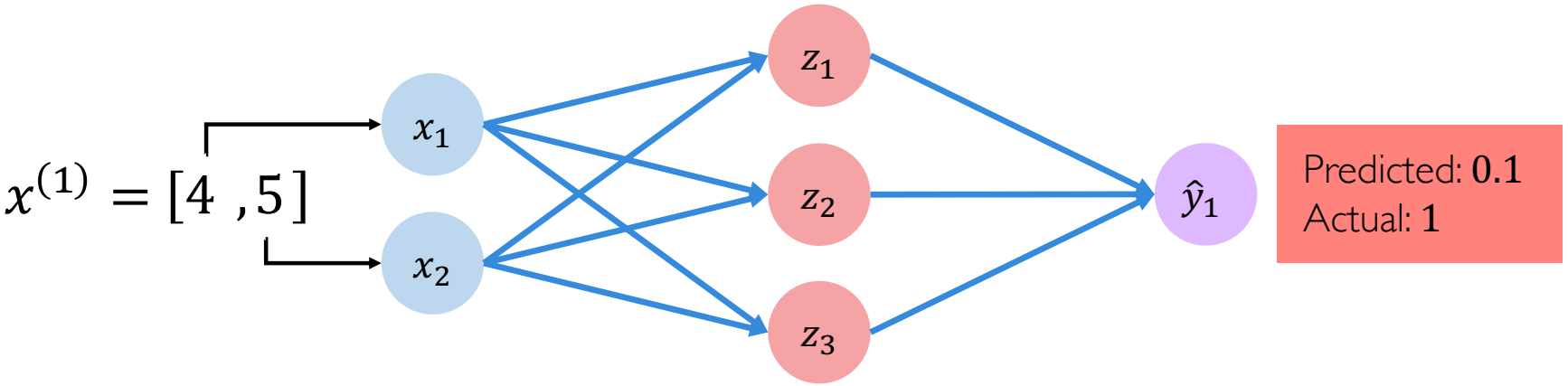
Let us write the entire neural network with weights $W = \{W_1, W_2\}$ *as*

$$f\left(x^{(i)}; \boldsymbol{W}\right)$$

How do we quantify the goodness or error in the predicted value ?

Credits: images taken from http://introtodeeplearning.com

Quantifying loss



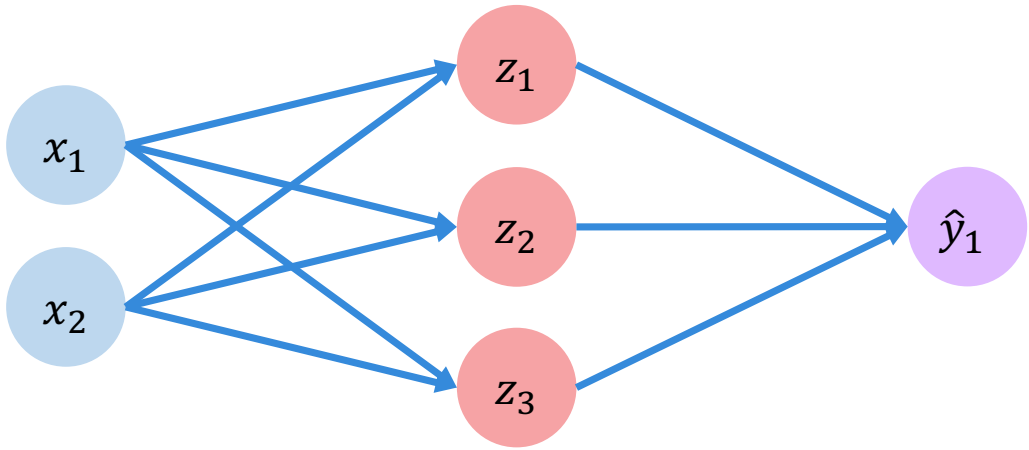
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

predicted actual

Empirical Loss

*The **empirical loss** measures the total loss over our entire dataset*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; W\right), y^{(i)}\right)$$

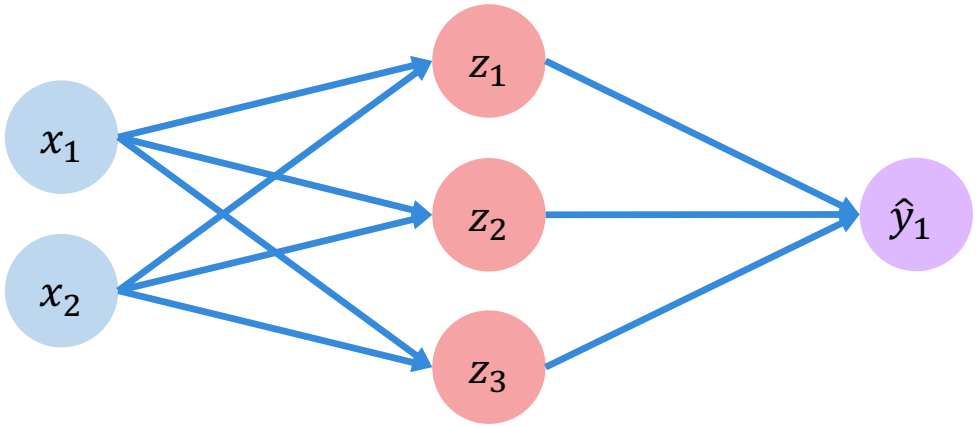
Objective Function

predicted

Actual

Loss Functions– Squared Loss

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



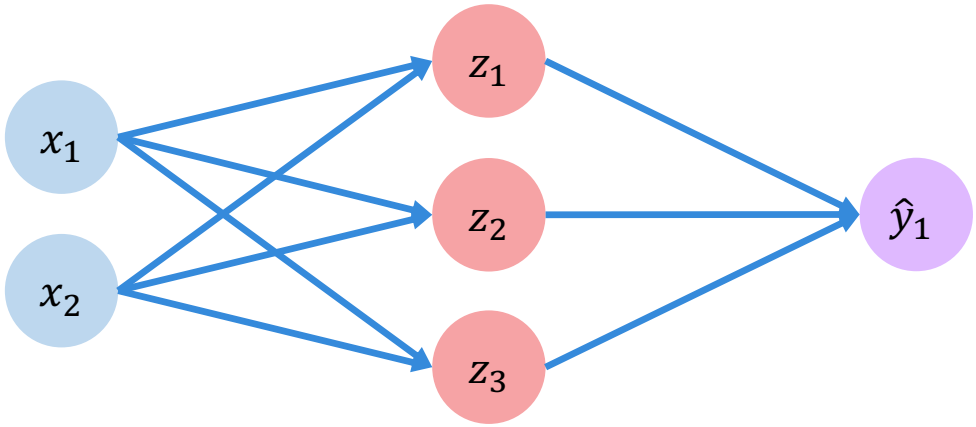
$$f(x) \quad y$$

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\left(x^{(i)};\boldsymbol{W}\right)}_{\text{predicted}}\right)^2$$

Loss Functions – Cross Entropy Loss

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f\left(x^{(i)};\boldsymbol{W}\right)\right) + \left(1 - y^{(i)}\right)\log\left(1 - f\left(x^{(i)};\boldsymbol{W}\right)\right)$$

Actual — predicted — Actual — predicted

Used when the output is [0,1] – example probability values

Training Neural Networks

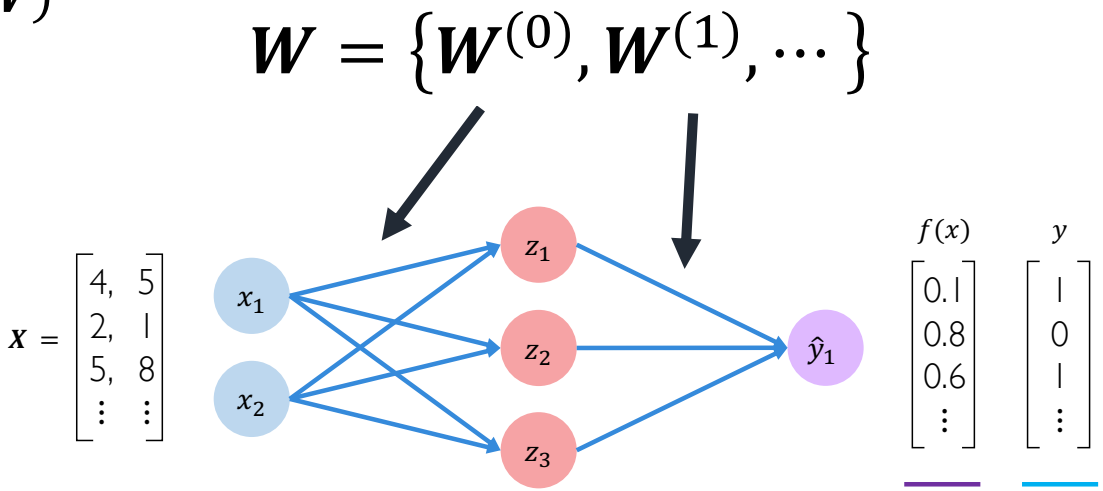
Optimizing Loss

- We want to find W such that the empirical loss is as low as possible

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

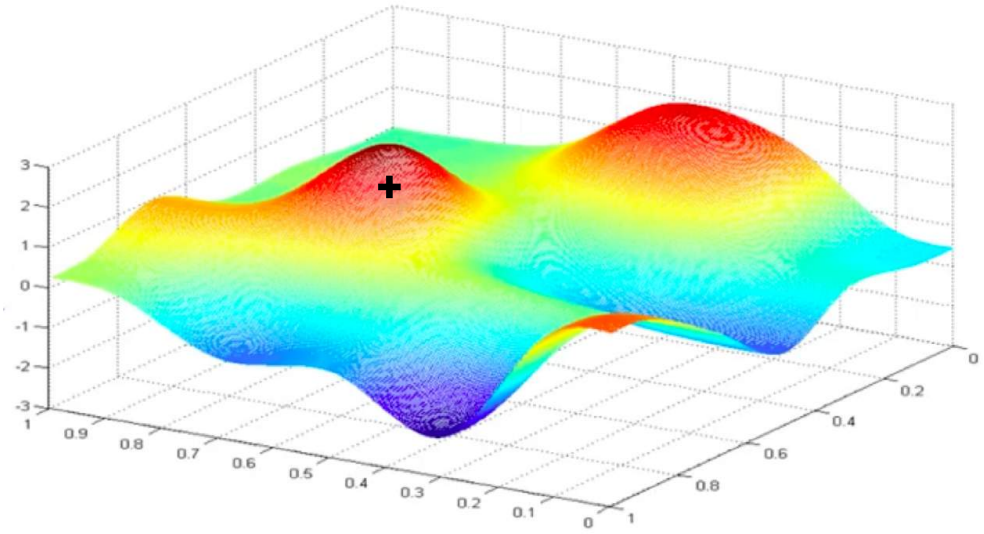
How many parameters do we have ?



Loss Surface

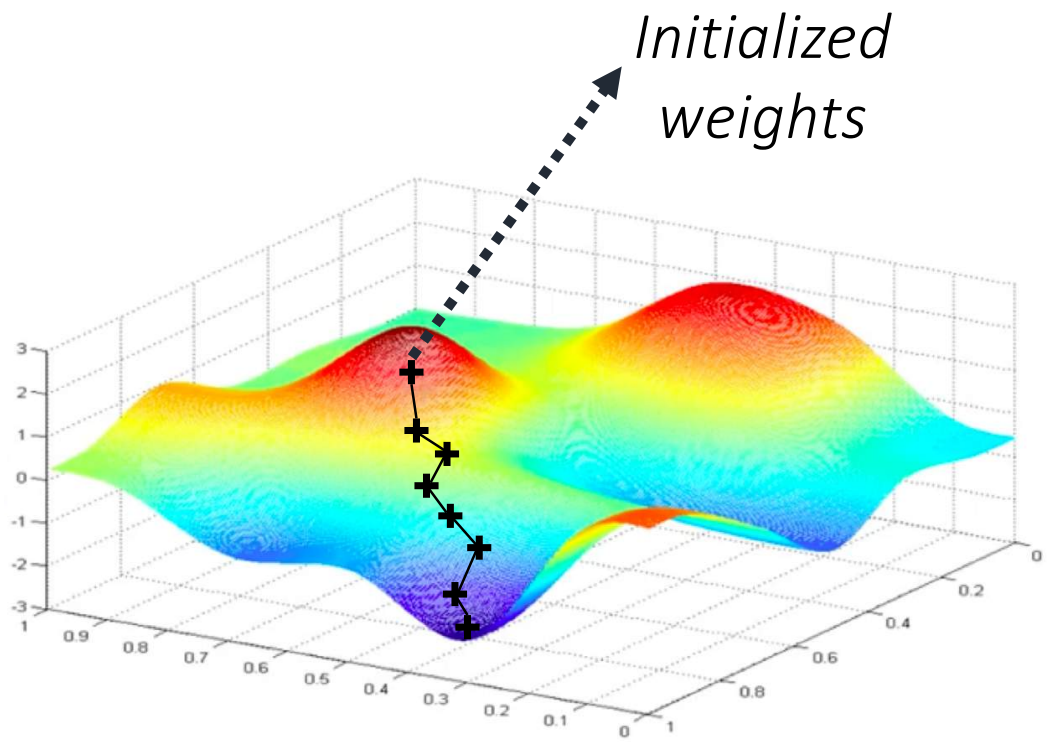
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(f\big(x^{(i)}; \boldsymbol{W}\big), y^{(i)}\big)$$

Predicted Actual



Minimizing Loss = finding the minima in this loss surface

Gradient Descent



Gradient Descent Algorithm

Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

while *training error has not converged* **do**

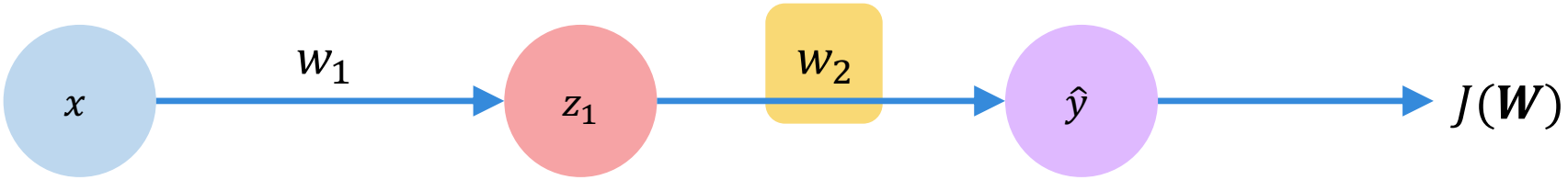
  Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Computing gradients -- Backpropagation

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



Gradient Descent Algorithm

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

while *not converged* **do**

  Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

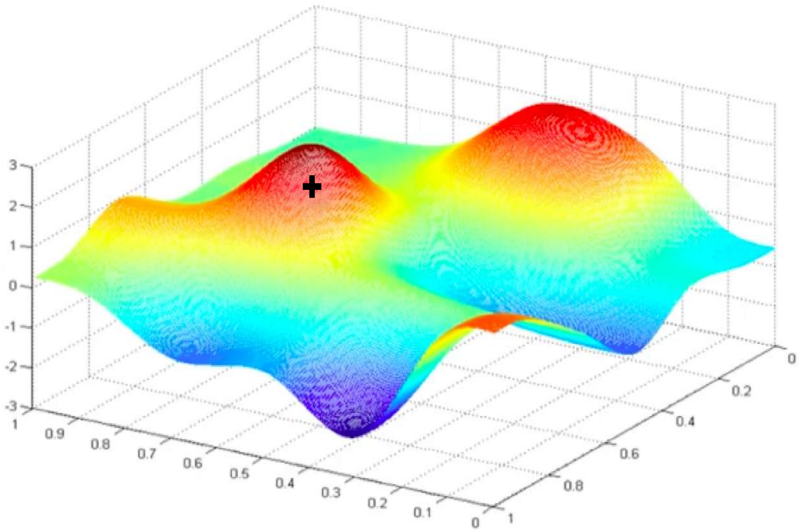
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

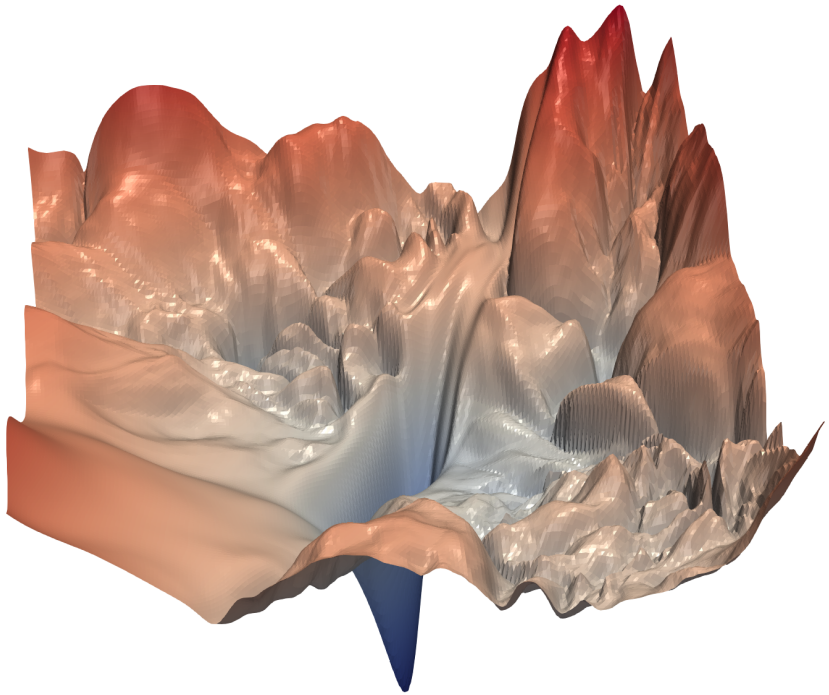
Calculus of back propagation

Video Time

Training in Practice

Loss surface in reality





$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

Local minimas, saddle points

[Visualizing the loss landscape of neural nets, Li et al 2018]

Learning rate

- Small learning rates have slow convergence and get stuck in local minimas
- Large learning rates overshoot and might diverge
- Stable learning rates converge smoothly and avoid local minima

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

- Fix learning rates based on
  - The size of the gradient
  - Rate of learning
  - Size of particular weights
  - … <More on this in upcoming lectures>

Momentum
Adagrad
Adadelta
Adam
RMSProp

Mini-batches

Gradient Descent Algorithm

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

while *not converged* **do**

  Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Stochastic Gradient Descent

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

while *not converged* **do**

  Pick single data point i

  Compute gradient $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Computationally expensive

Batched Gradient Descent

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

while *not converged* **do**

  Pick batch B of data points

  Compute gradient $\frac{\partial J(W)}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$

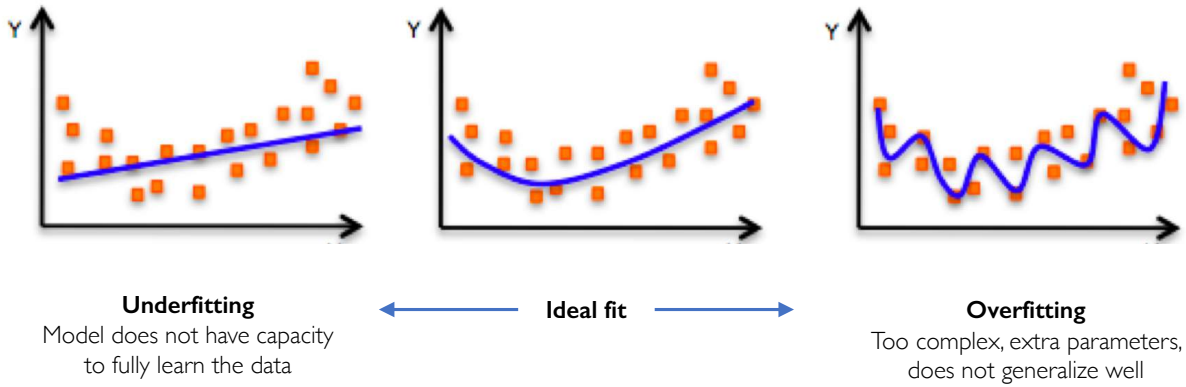
  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Fast to compute and much better estimate of the true gradient

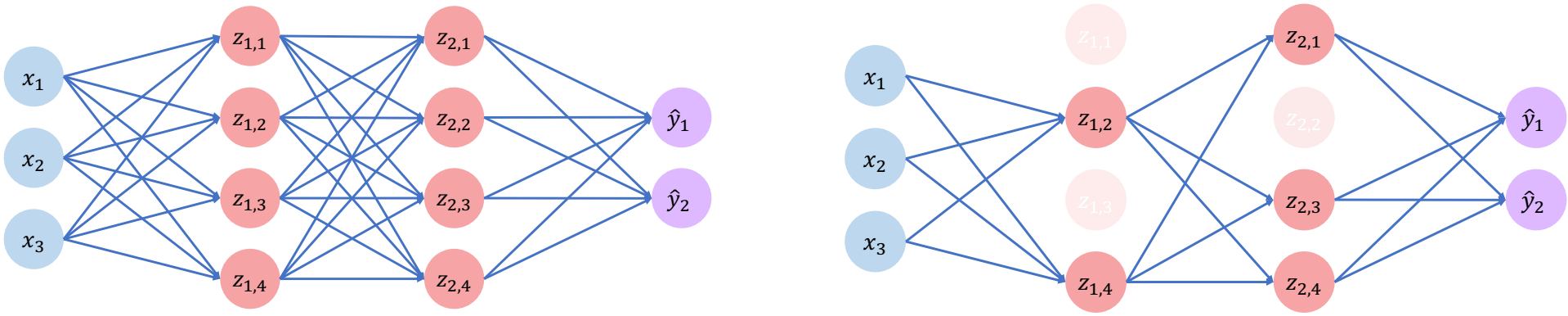
- Mini-batches : Computed the gradients for a batch of points rather than each
- Smoother convergence, allows for larger learning rates
- Can parallelize computation, fully use GPU potential

Overfitting and regularization



- Deep Neural Networks have a large capacity (leads to more complex models) and hence tend to overfit
- Regularization: Forces the model to learn simpler model
- How do we regularize deep networks ?

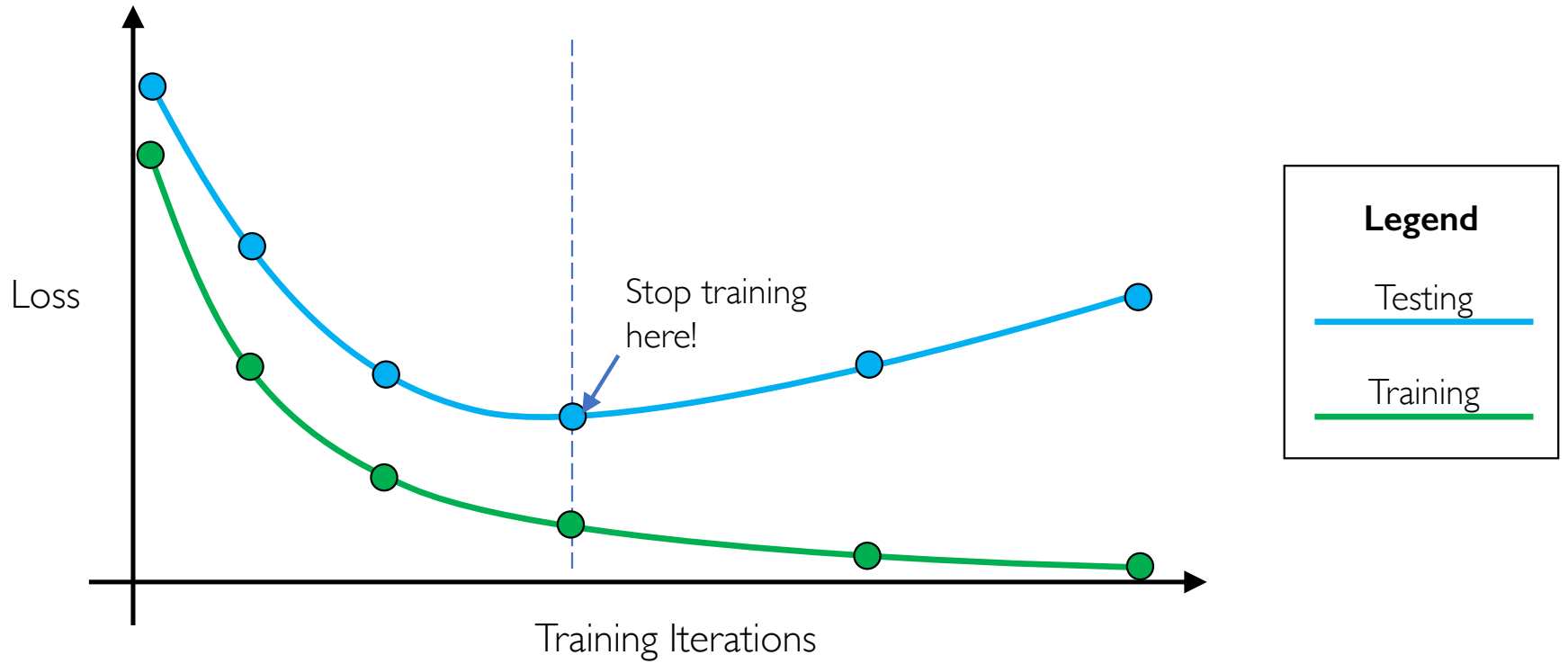
Dropout



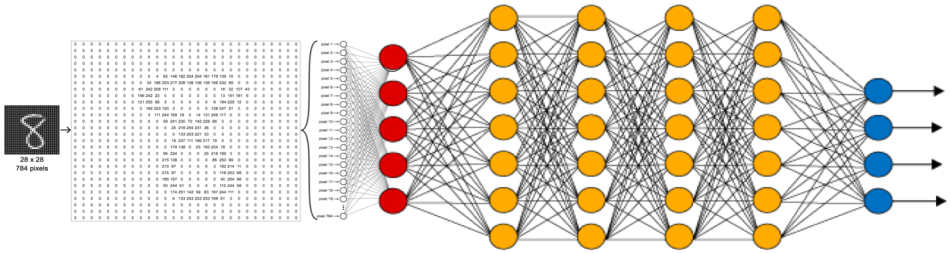
- Drop some of the neurons (setting activation = 0)
- Forces the network to learn using a smaller capacity --> robust to overfitting

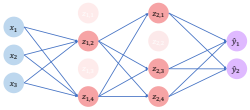
Early Stopping (optional)

- Stop before there is a chance to overfit



Conclusions





- Deep neural networks are compositions of perceptrons
- Activations provide non-linearity
- Back Propagation for parameter learning

- In practice
  - pay attention to learning rates
  - Use mini-batches
  - regularization using drop-outs, early stopping

References

- Tricks for gradient descent: https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/tricks-2012.pdf
  - Use stochastic gradient descent when training time is the bottleneck.
  - Randomly shuffle the training examples.
  - Use preconditioning techniques (when the hessian is ill-conditioned)
  - Monitor both the training cost and the validation error.
  - Check the gradients using finite differences.
  - Experiment with the learning rates γ using a small sample of the training set.
  - Leverage the sparsity of the training examples

- How the Backpropagation Algorithm works: http://neuralnetworksanddeeplearning.com/chap2.html
- https://colah.github.io/posts/2015-08-Backprop/

- Backprop. Elaborate calculations: http://cs231n.stanford.edu/handouts/derivatives.pdf