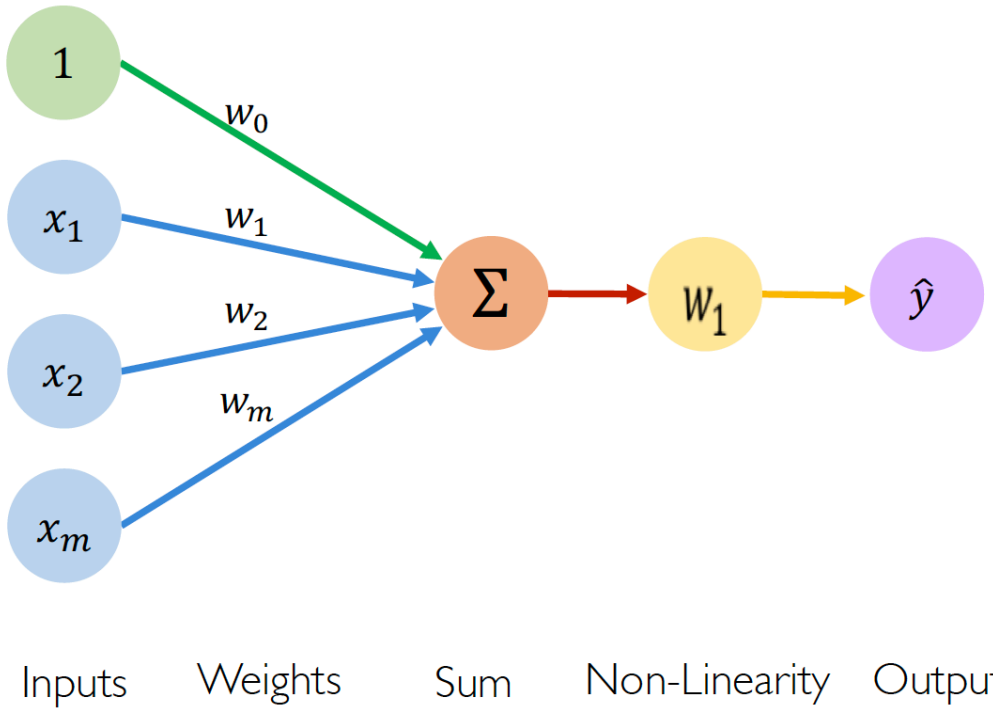
Lecture 24: Deep Neural Network

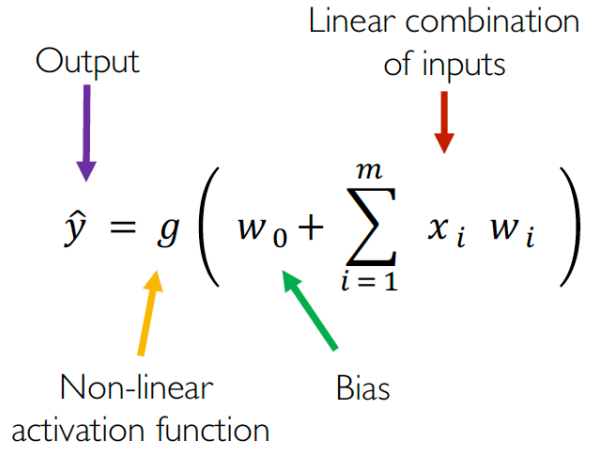
BIOS635

04/16/2020

Some slides taken/modified from other people's lectures online.

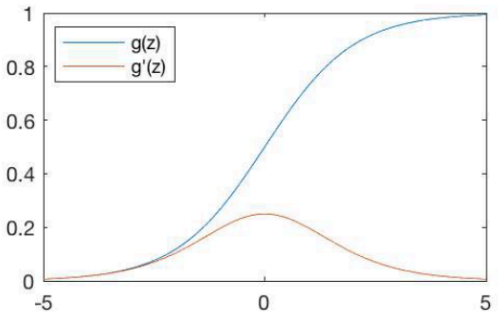
The perceptron: the forward propagation





Common activation function

Sigmoid Function

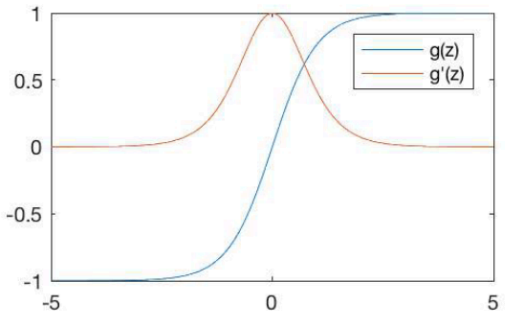


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

tf.nn.sigmoid(z)

Hyperbolic Tangent

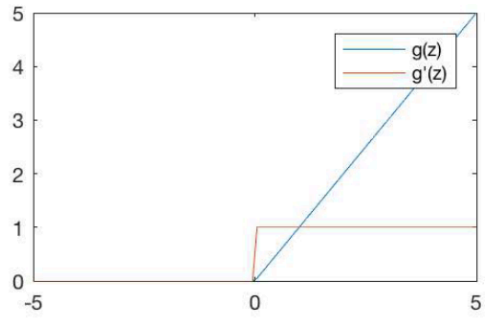


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

tf.nn.tanh(z)

Rectified Linear Unit (ReLU)



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

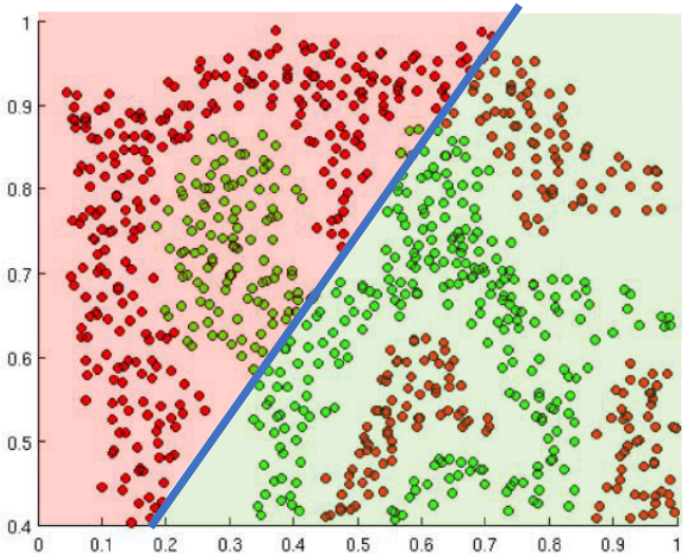
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

tf.nn.relu(z)

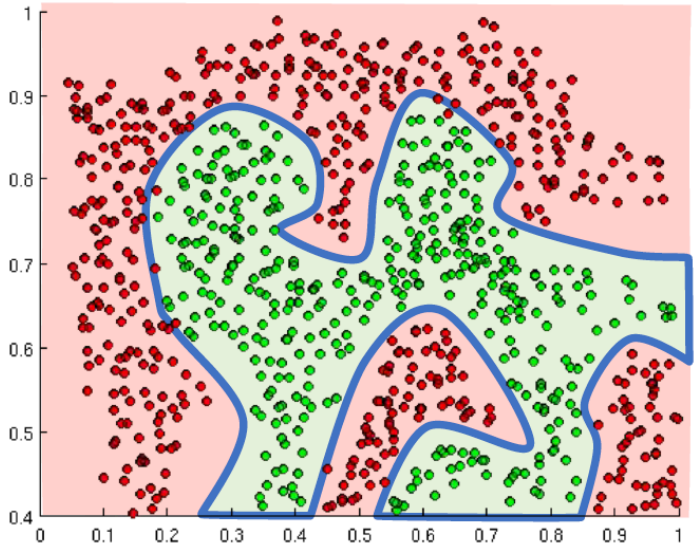
NOTE: All activation functions are non-linear

Importance of activation functions

The purpose of activation functions is to ***introduce non-linearities*** *into the network*

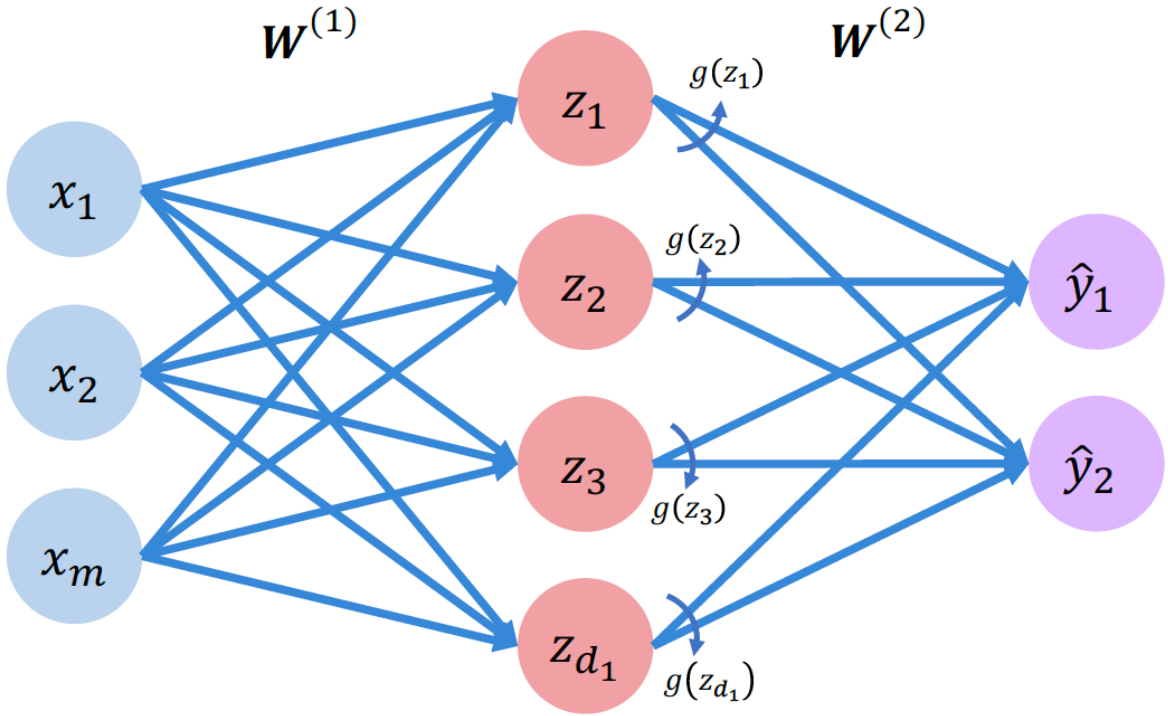


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

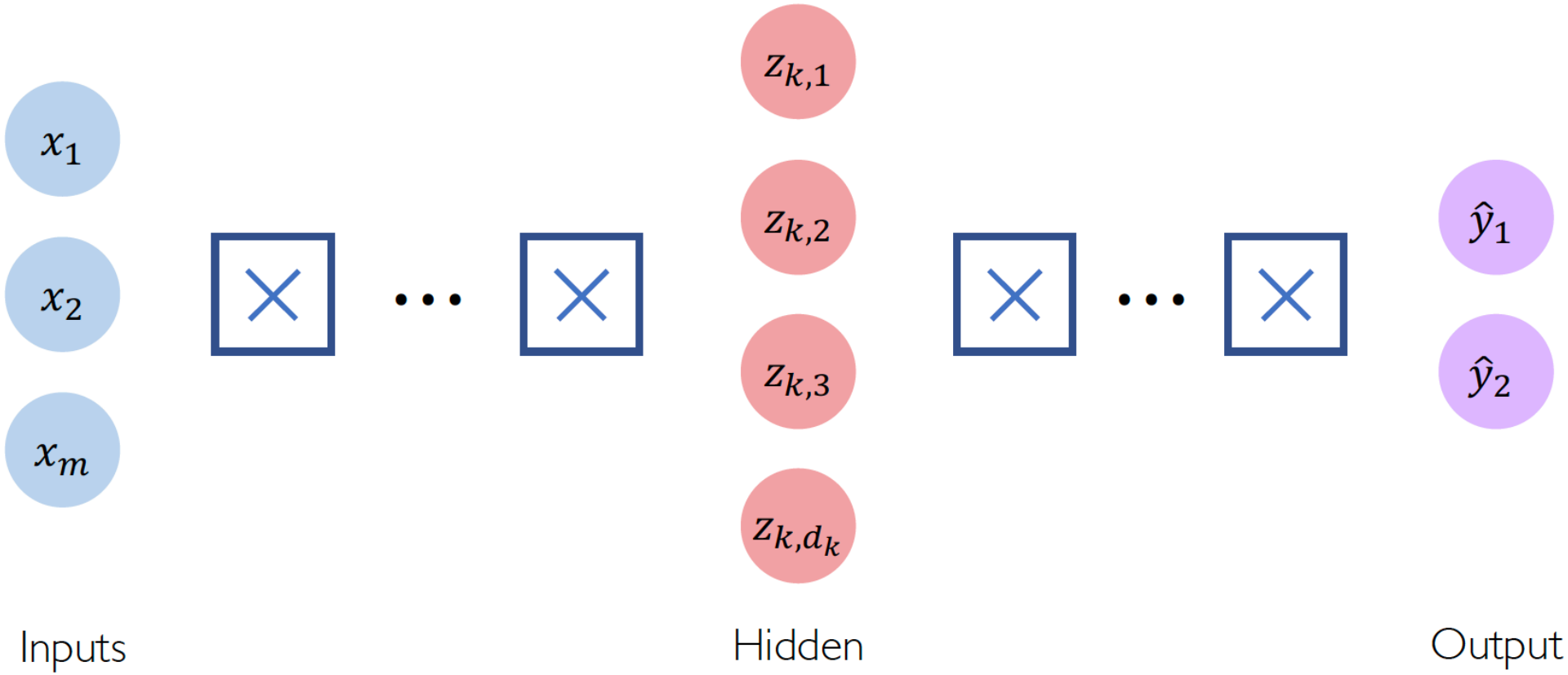
Perceptron: one hidden layer neural network



Inputs Hidden Final Output

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

Deep neural network

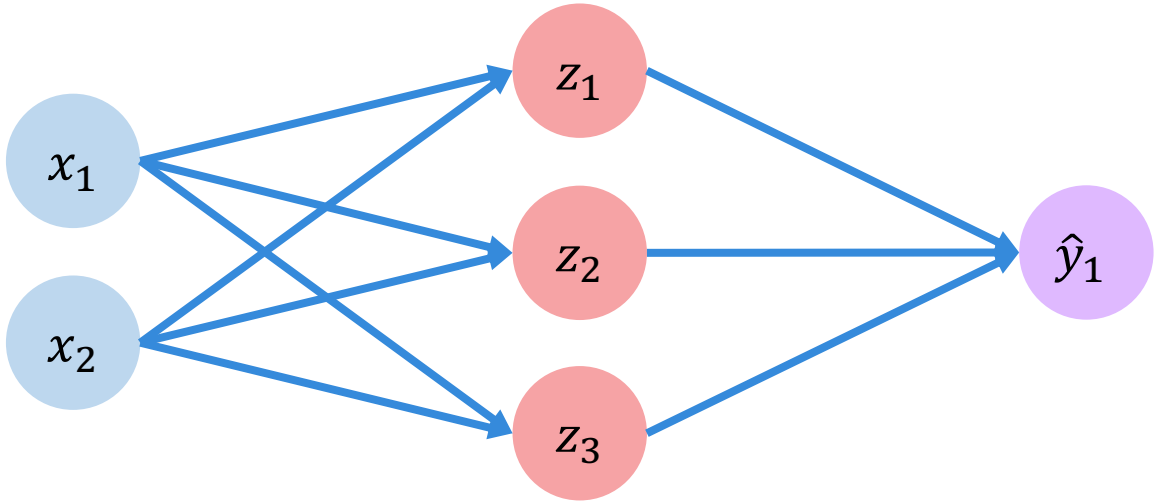


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Empirical loss

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

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



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

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

Predicted Actual

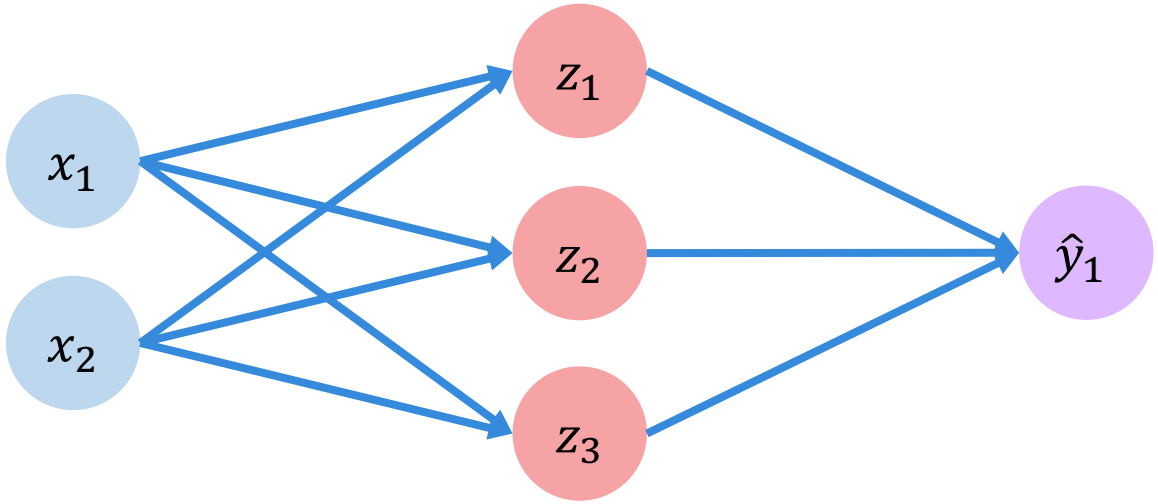
Also known as:

- Objective function
- Cost function
- Empirical Risk

Binary cross entropy loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*

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



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

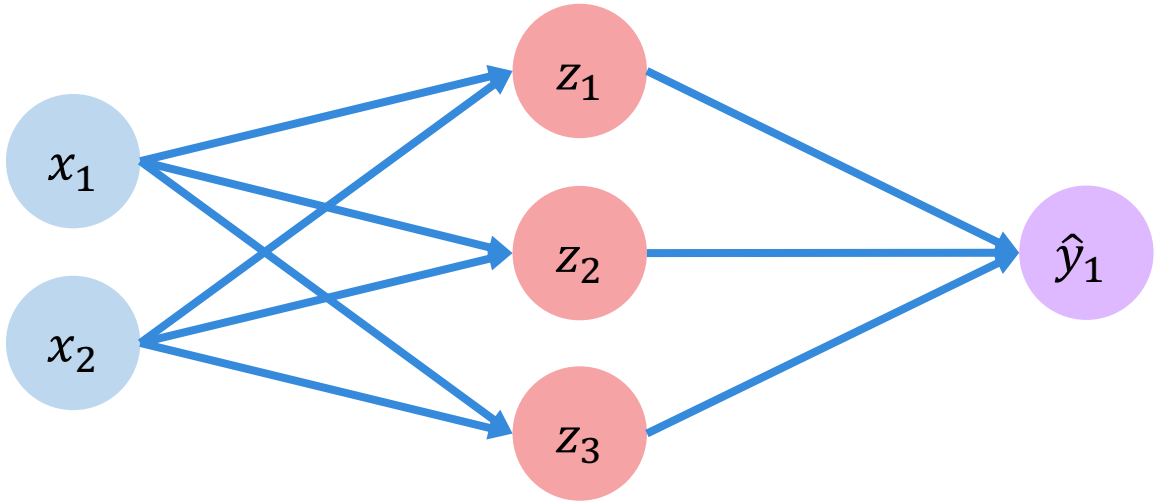
$$\boldsymbol{J}(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - f\left(x^{(i)}; \boldsymbol{W}\right)\right)$$

Actual Predicted Actual Predicted

Mean squared error loss

Mean squared error loss *can be used with regression models that output continuous real numbers*

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



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

$f(x)$ y

$$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$$

Final Grades (percentage)

Loss optimization

We want to find the network weights that ***achieve the lowest loss***

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

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

Remember:

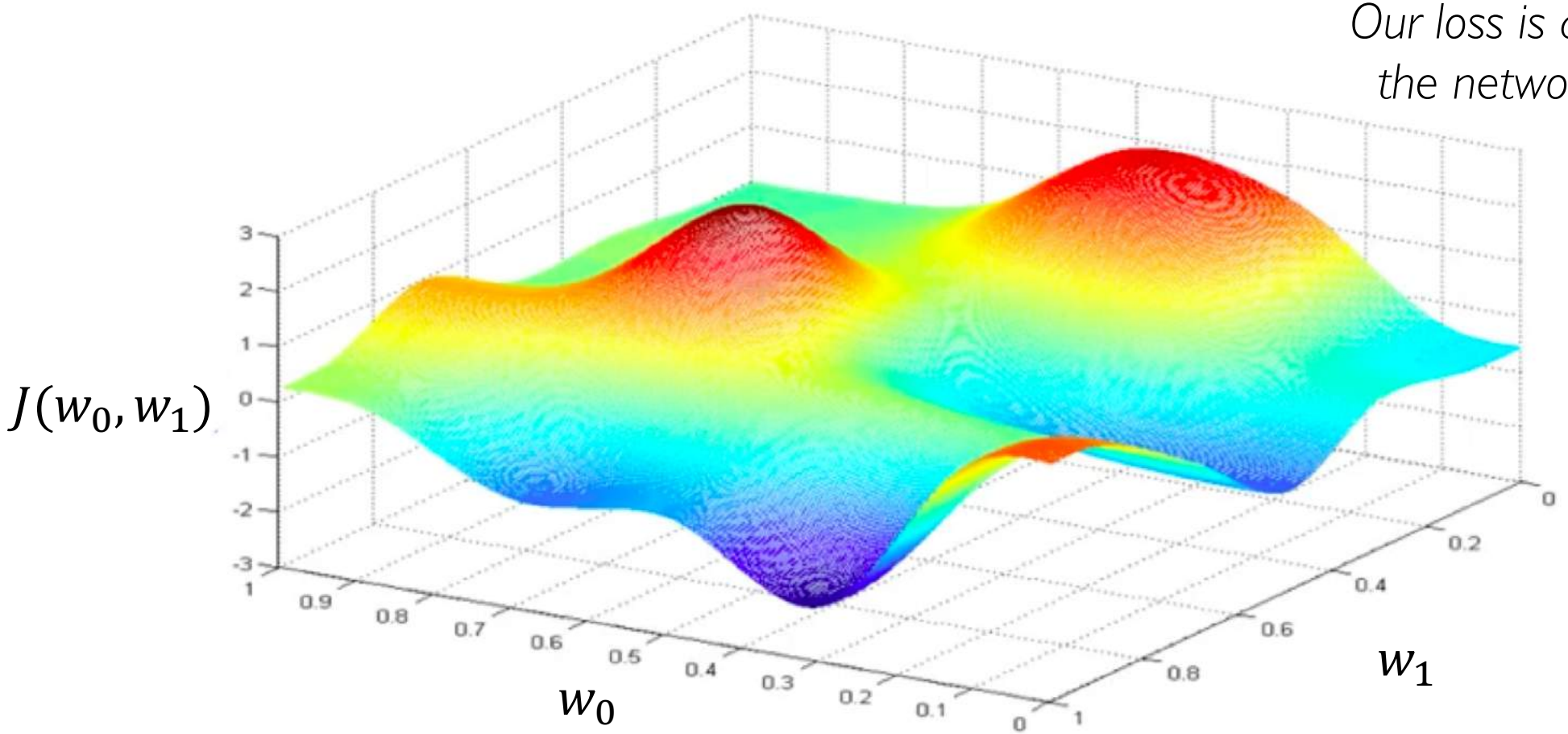
$$\boldsymbol{W} = \left\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\right\}$$

Loss optimization

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

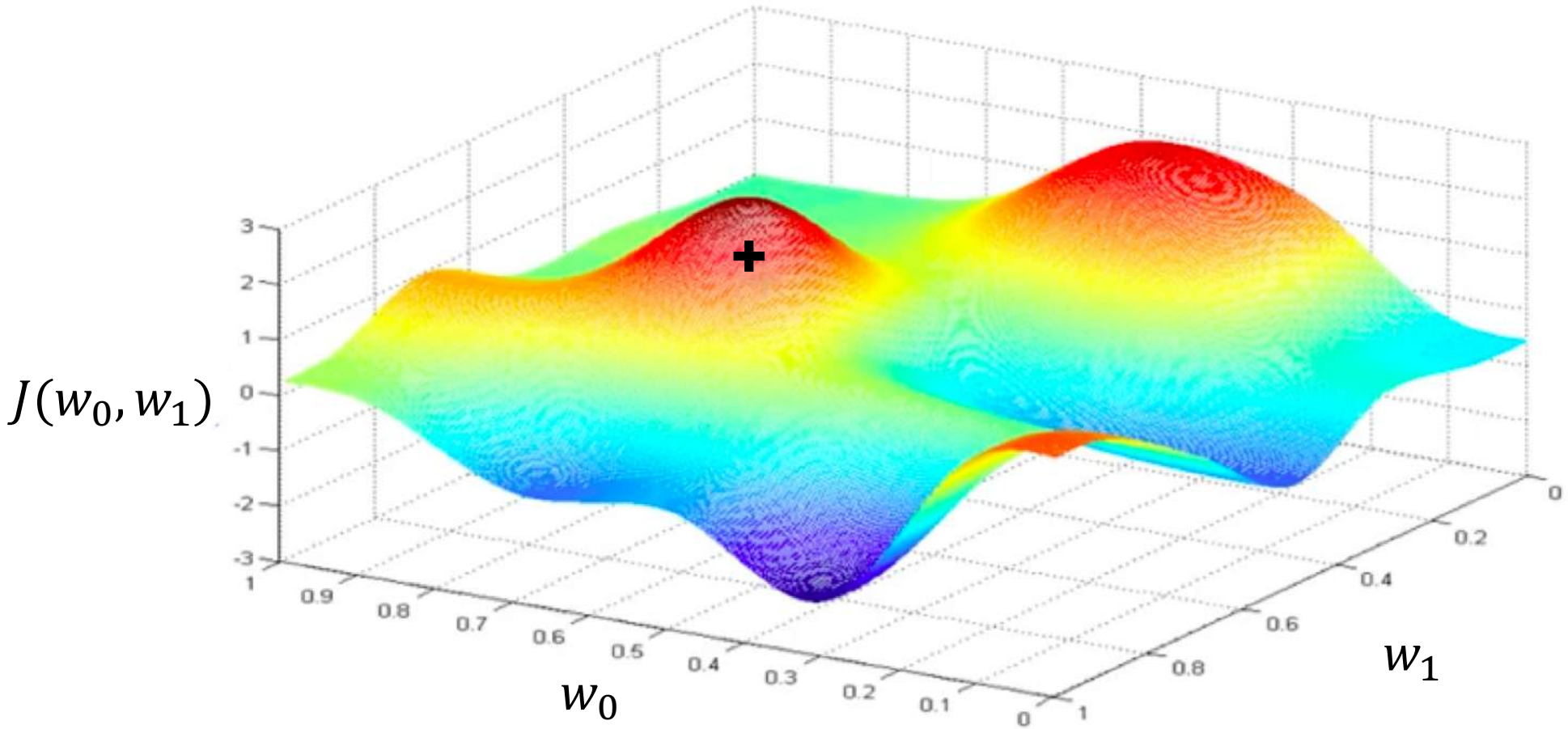
Remember:

Our loss is a function of the network weights!



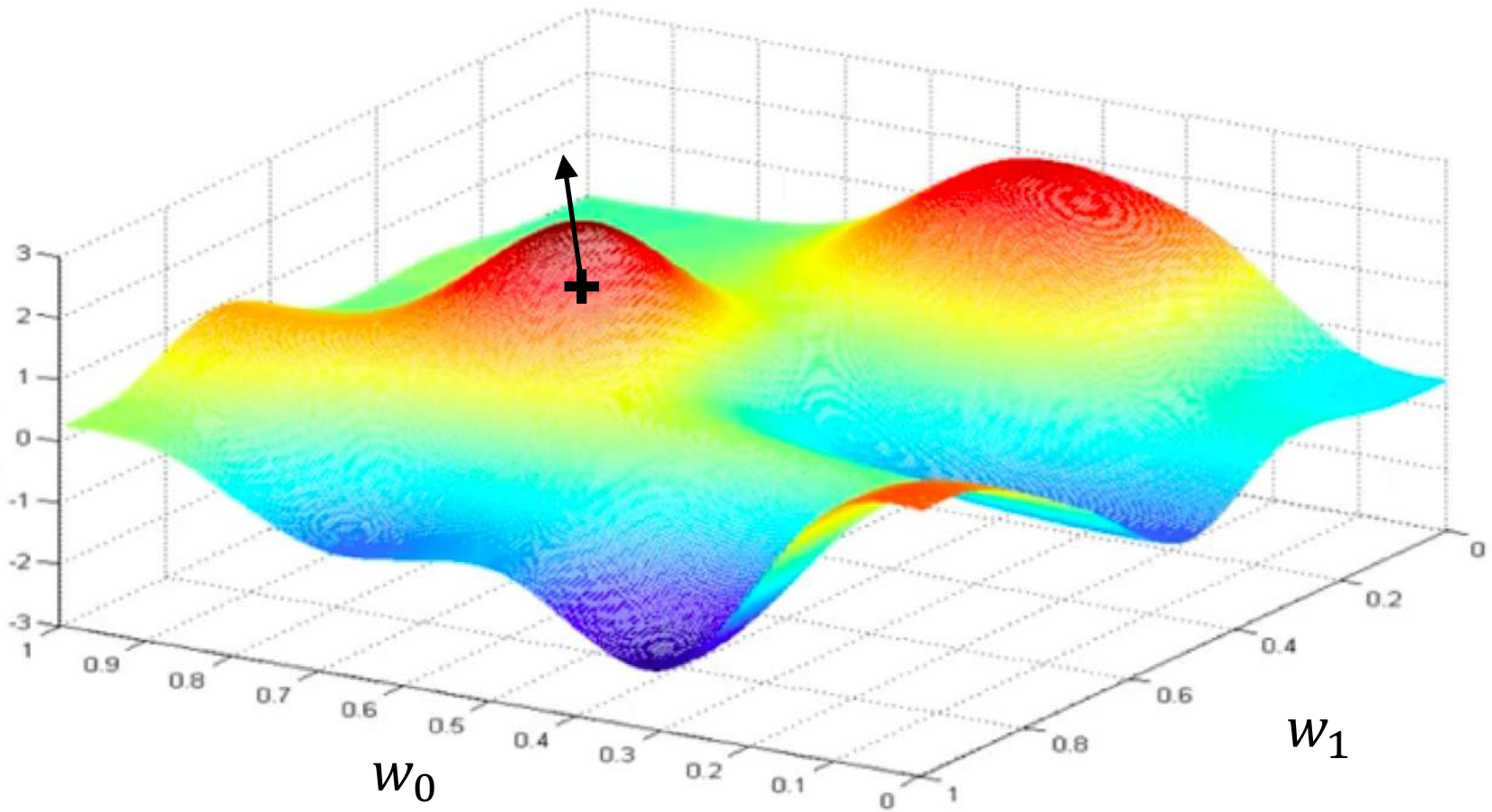
Loss optimization

Randomly pick an initial (w_0, w_1)



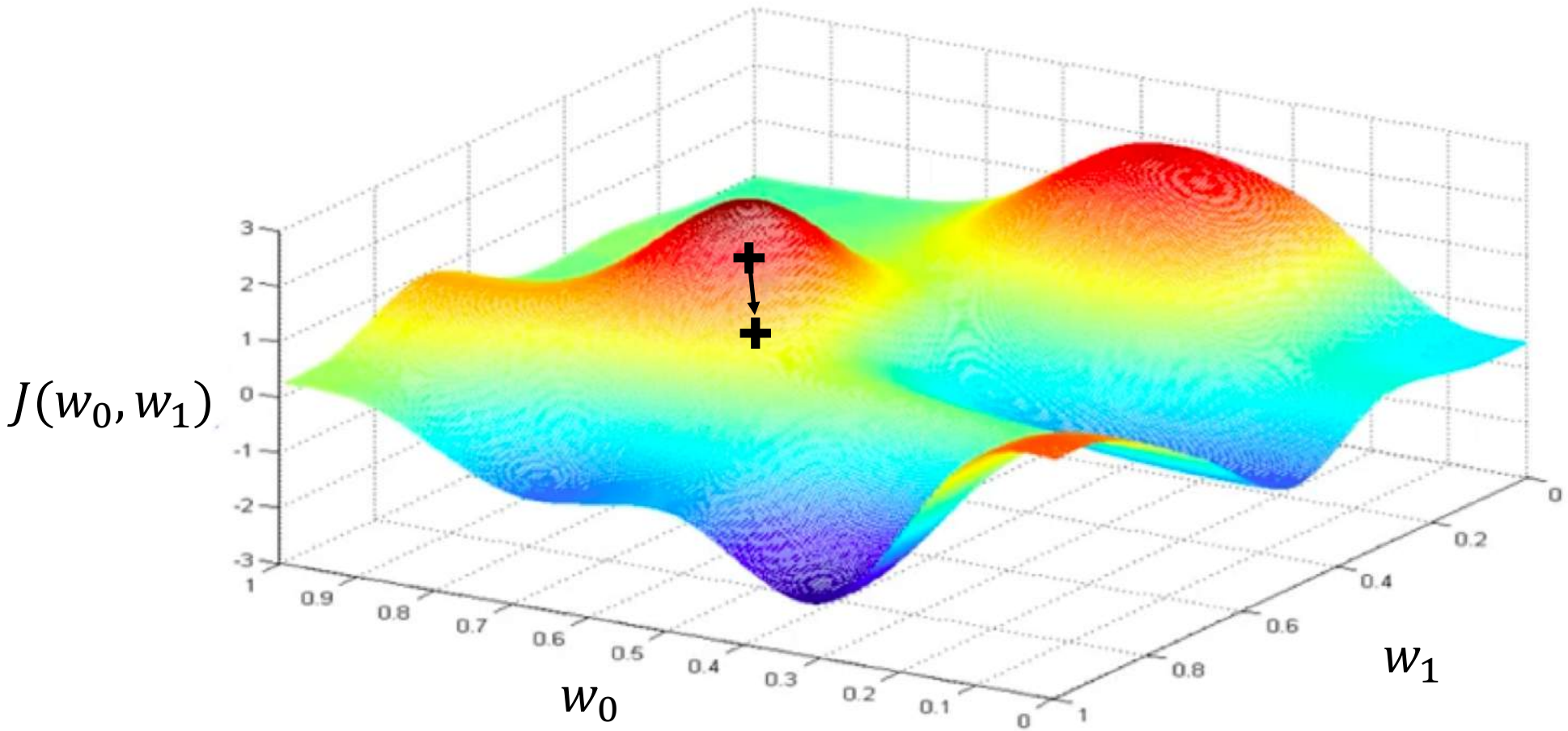
Loss optimization

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



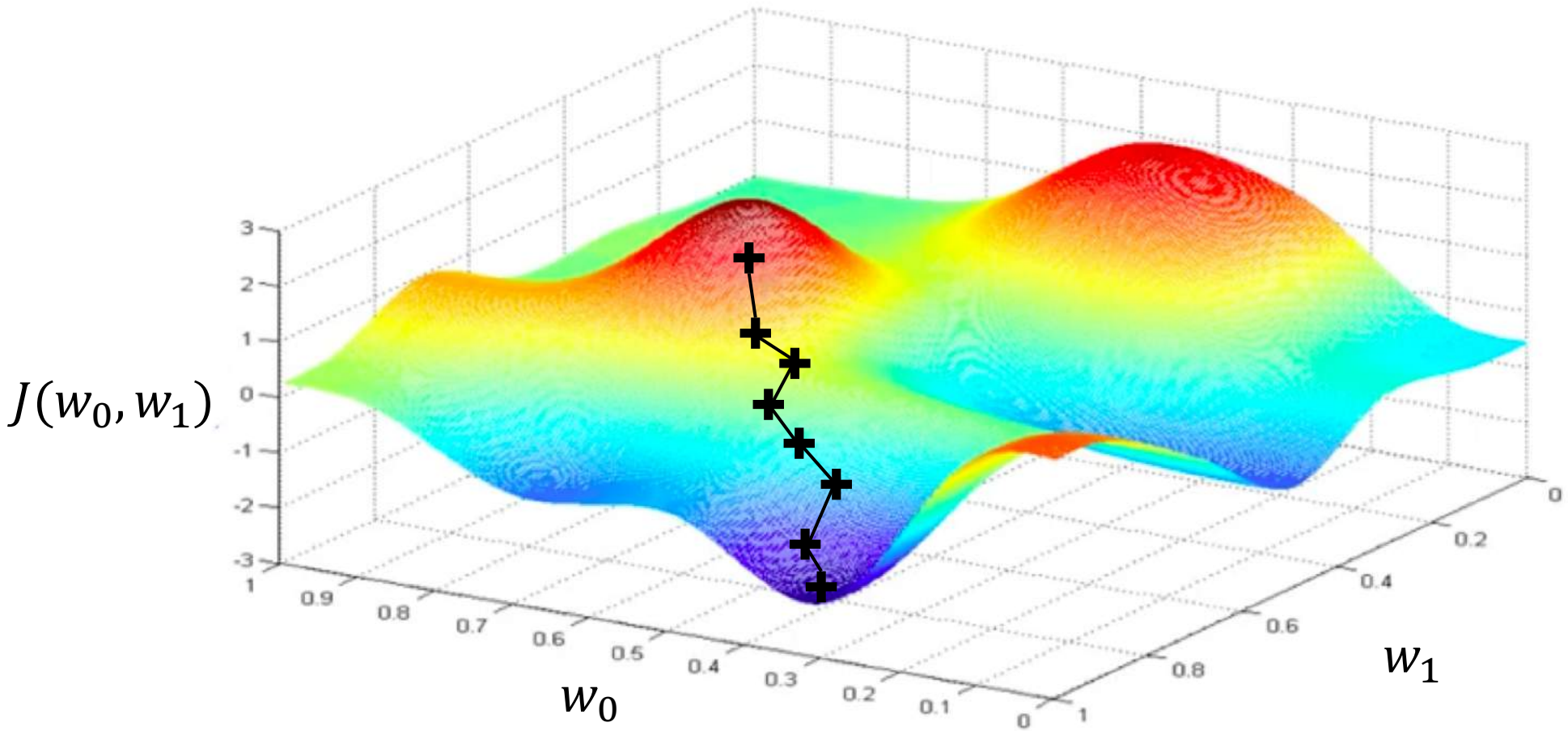
Loss optimization

Take small step in opposite direction of gradient



Loss optimization

Repeat until convergence

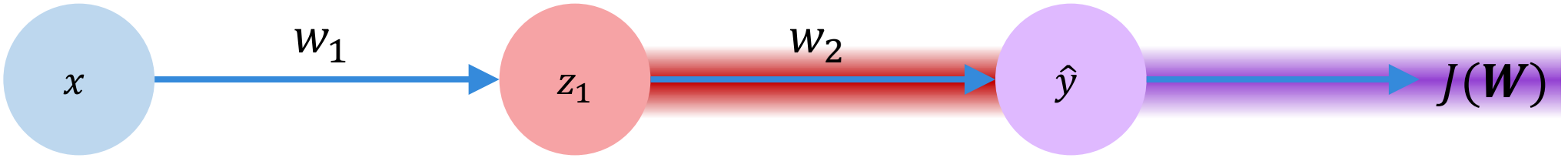


Gradient descent

Algorithm

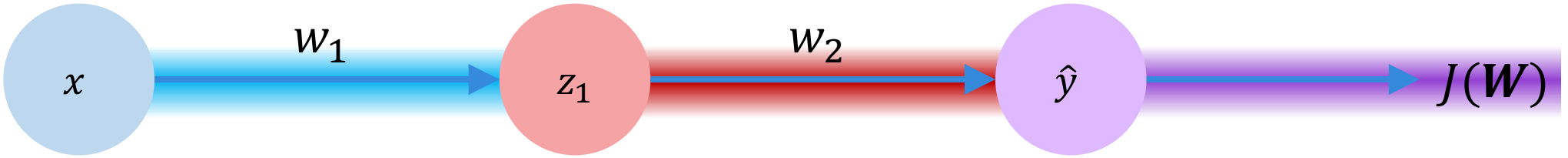
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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

Computing gradients: back propagation



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

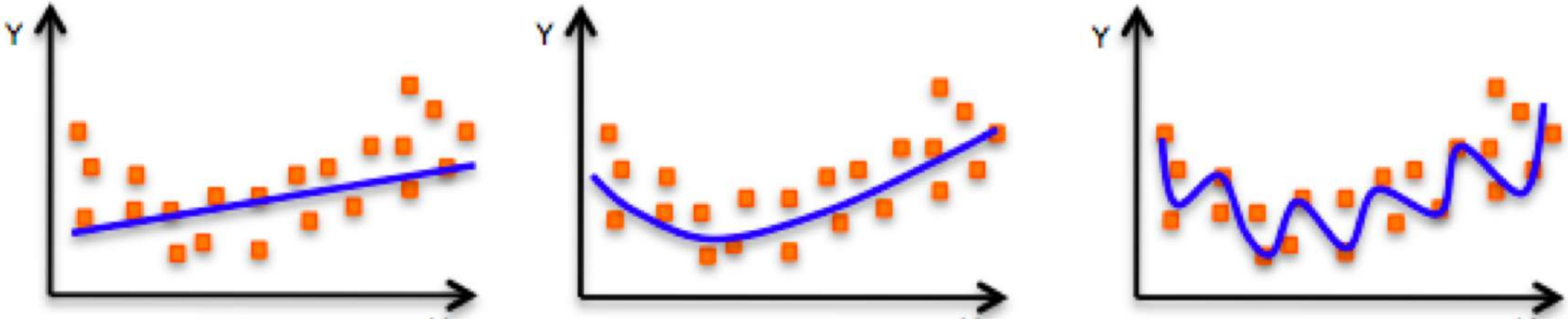
Computing gradients: back propagation



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

Repeat this for ***every weight in the network*** *using gradients from later layers*

The problem of overfitting



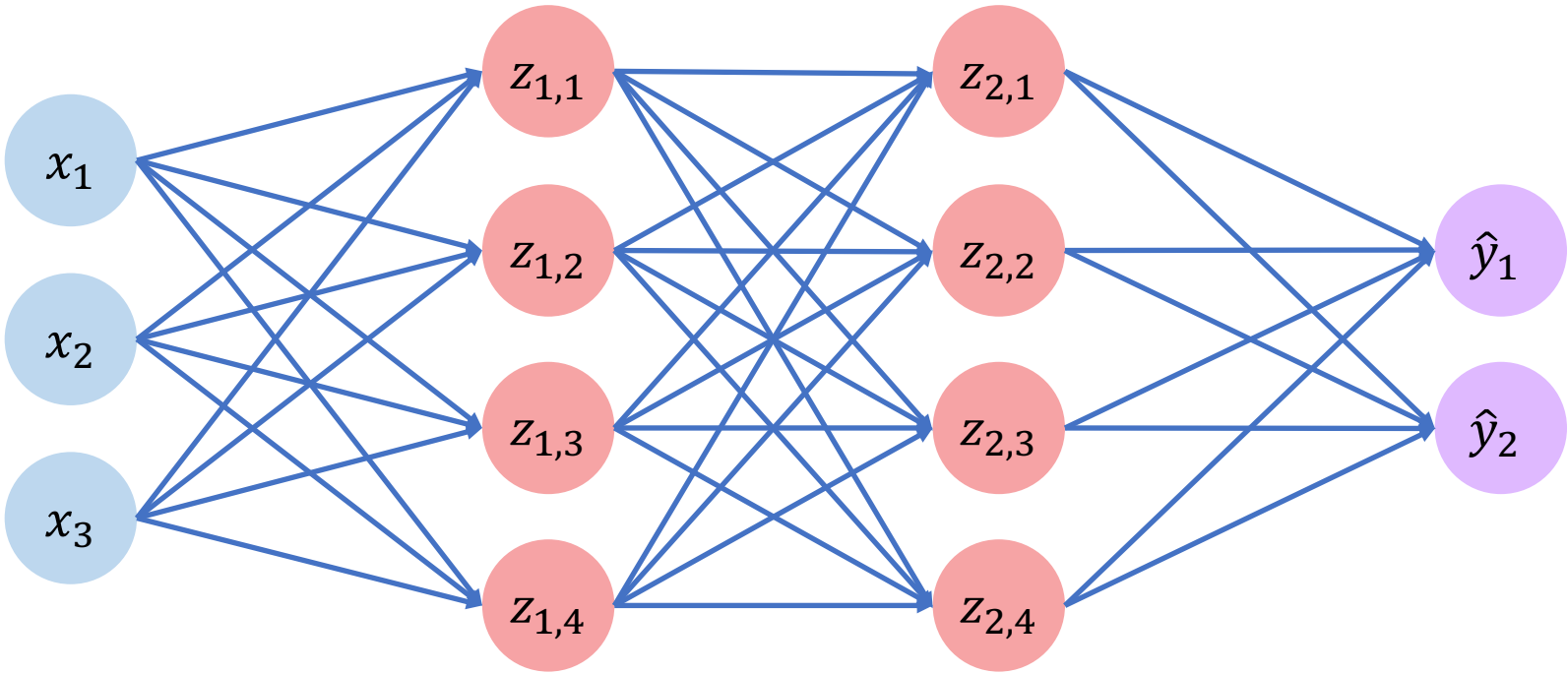
Underfitting
Model does not have capacity to fully learn the data

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

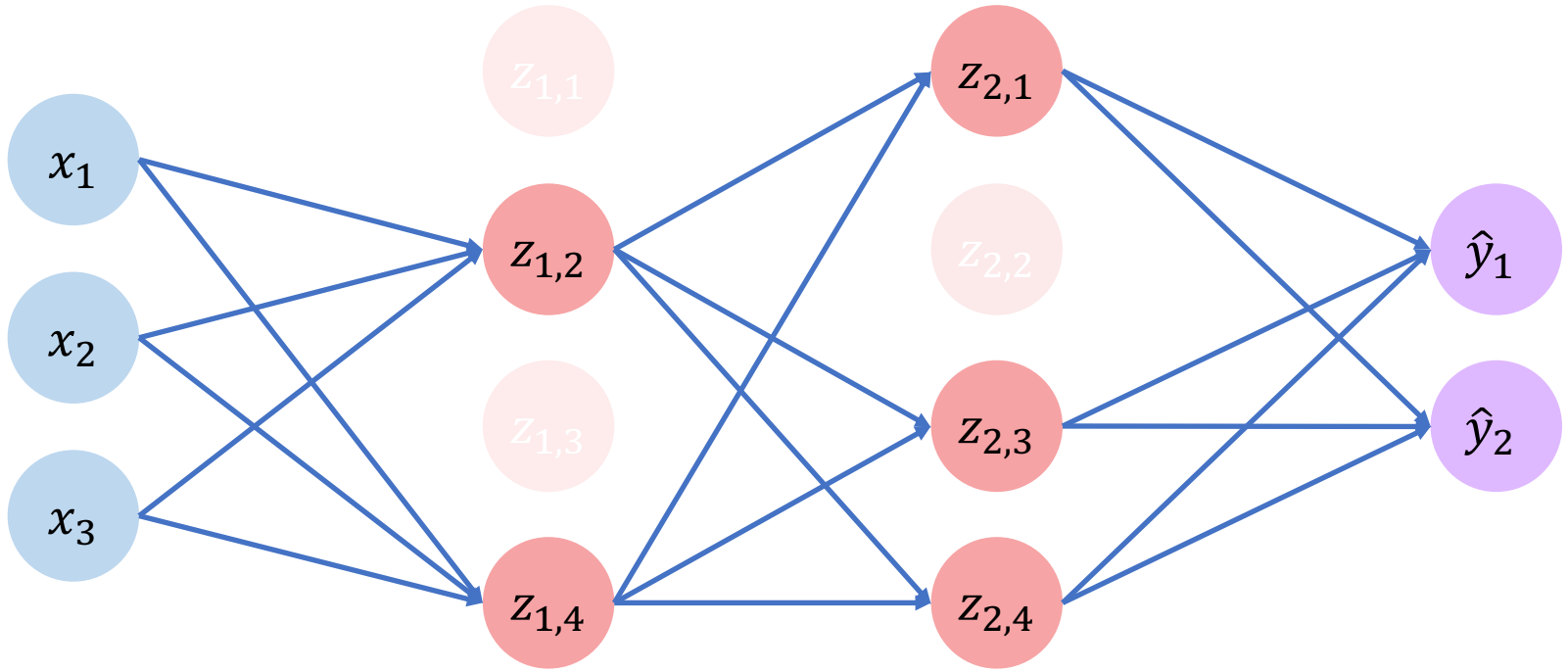
Regularization: dropout

- During training, randomly set some activations to 0



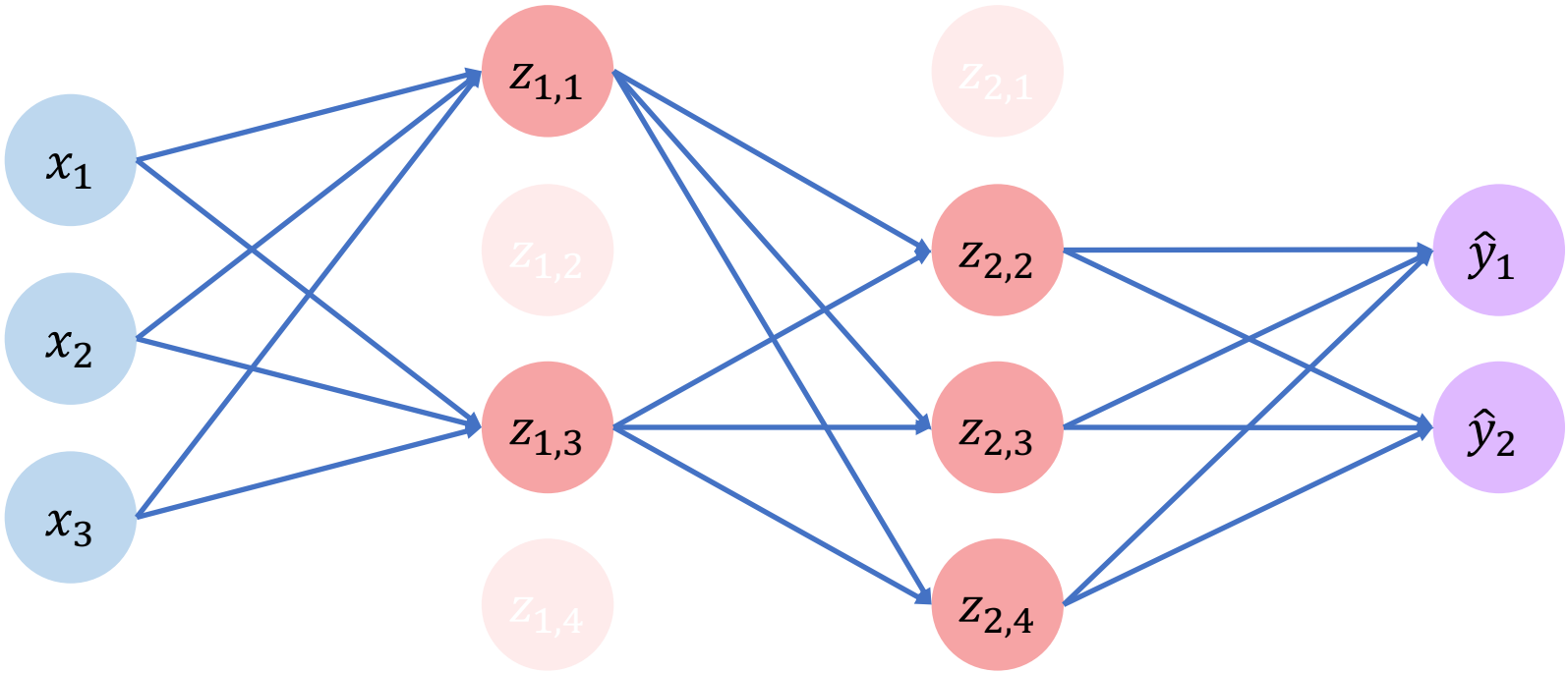
Regularization: dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node



Regularization: dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node



Regularization: early stopping

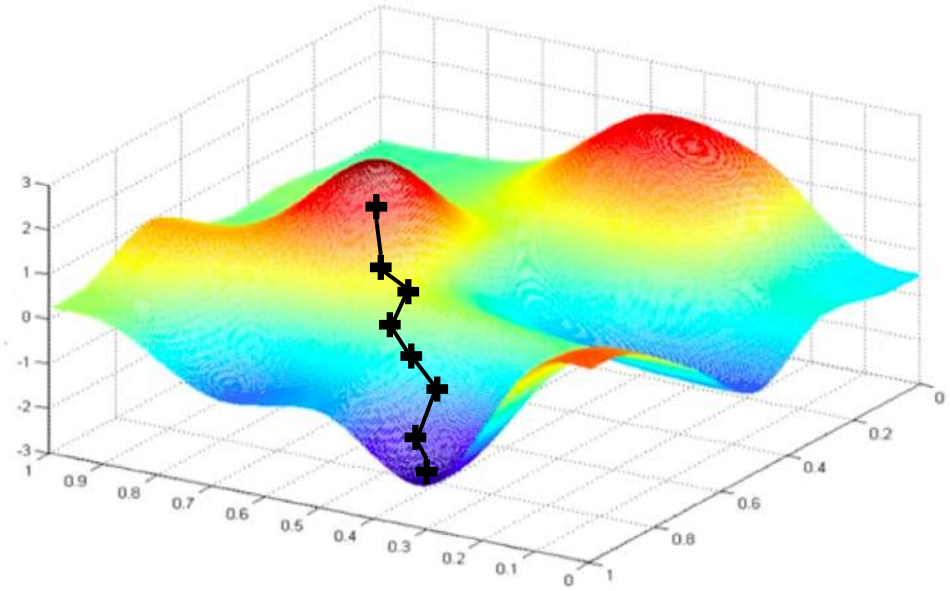
- Stop training before we have a chance to overfit

Mini-batches

Algorithm

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

Can be very computational to compute!

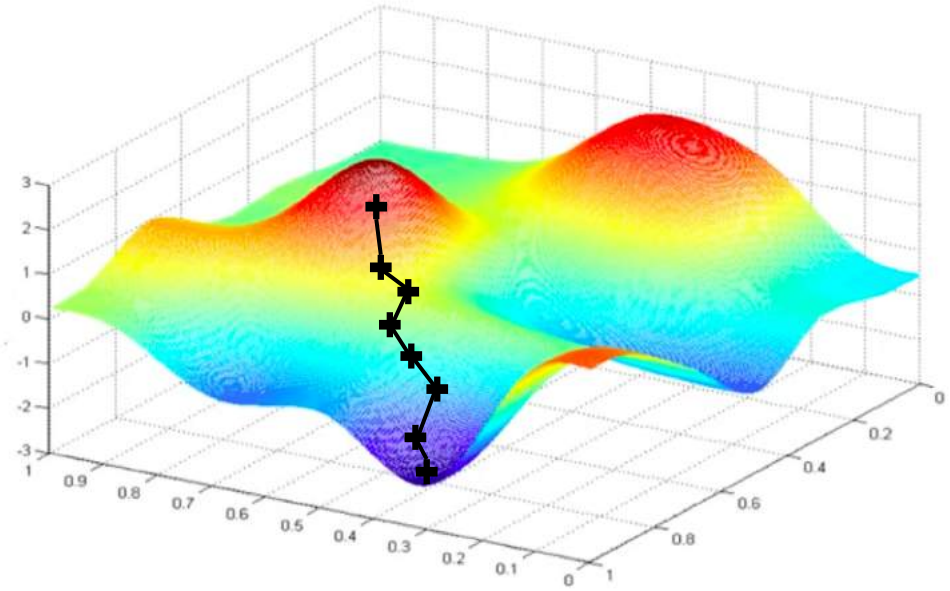


Mini-batches

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

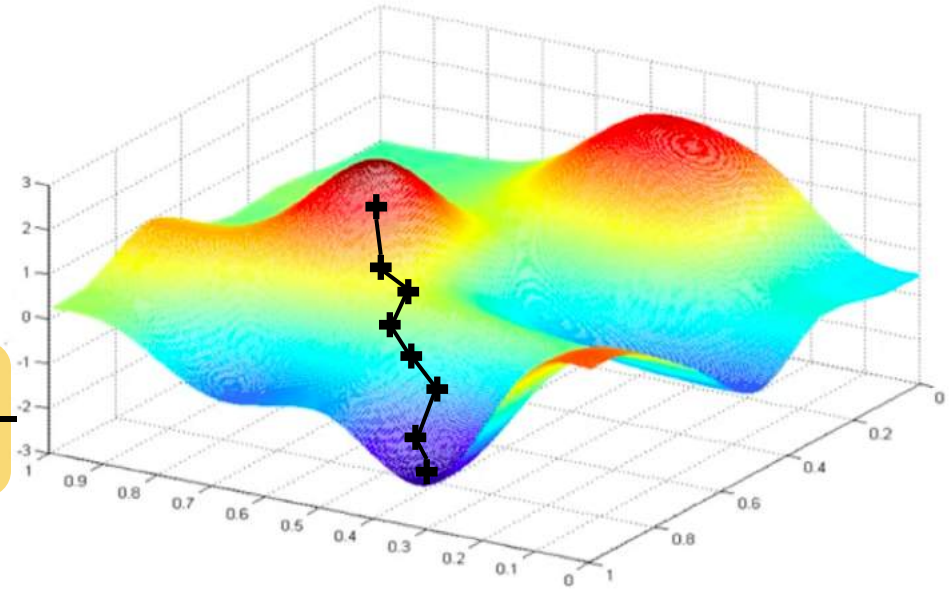
Easy to compute but **very noisy** (stochastic)!



Mini-batches

Algorithm

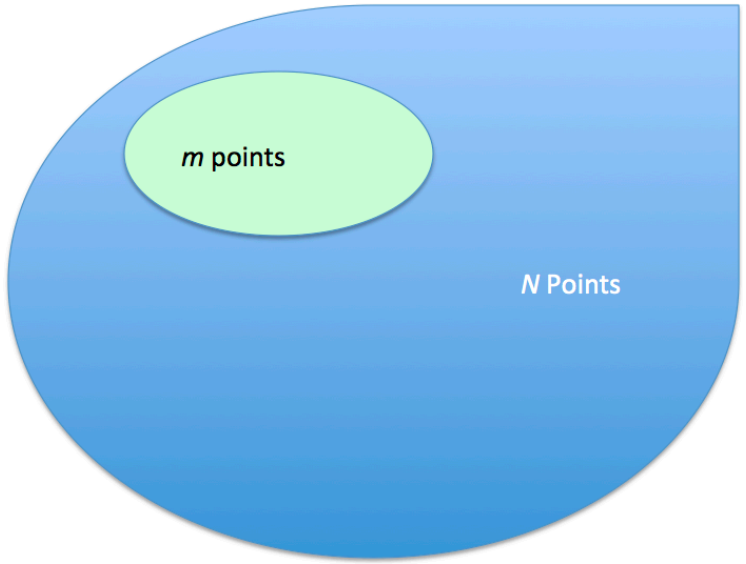
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

Batch normalization

- Performing normalization for each training mini-batch



Population statistics (with N points) and batch statistics (with m points)

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

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

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

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

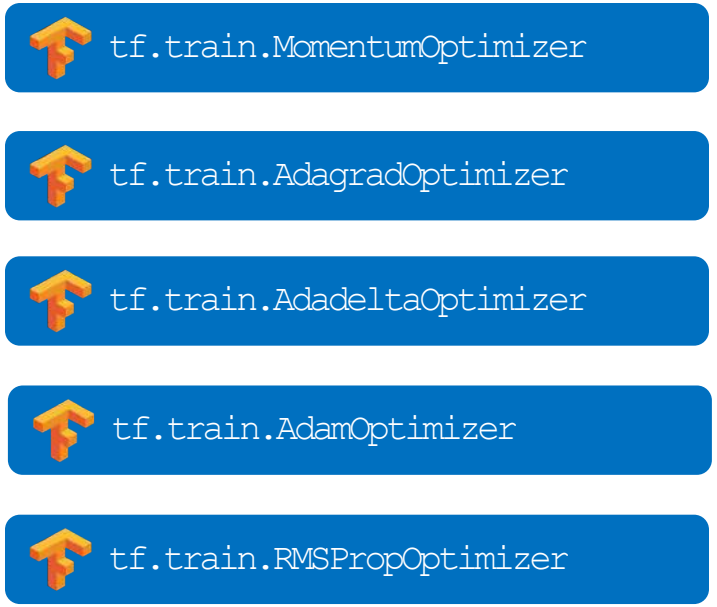
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Optimization

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp



Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Additional details: https://ruder.io/optimizing-gradient-descent/

Optimization

$V_{dW}=0,\ S_{dW}=0,\ V_{db}=0,\ S_{db}=0$

On iteration t (minibatch)

Compute dW, db using current minibatch.

moment estimate update:

$$\begin{cases} V_{dW} = \beta_1 V_{dW} + (1-\beta_1)dW, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1)db & \text{"momentum"} \\ S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2 & \text{"RMSprop"} \end{cases}$$

bias correction:

$$\begin{cases} V_{dW}^{corrected} = V_{dW}/(1-\beta_1^t), \quad V_{db}^{corrected} = V_{db}/(1-\beta_1^t) \\ S_{dW}^{corrected} = S_{dW}/(1-\beta_2^t), \quad S_{db}^{corrected} = S_{db}/(1-\beta_2^t) \end{cases}$$

Batch gradient descent: $W := W - \alpha dW, \quad b := b - \alpha db$

ADAM: $W := W - \alpha \dfrac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \varepsilon}, \quad b := b - \alpha \dfrac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$

Momentum: $W := W - \alpha V_{dW}^{corrected}, \quad b := b - \alpha V_{db}^{corrected}$

RMSprop: $W := W - \dfrac{\alpha dW}{\sqrt{S_{dW}} + \varepsilon}, \quad b := b - \dfrac{\alpha db}{\sqrt{S_{db}} + \varepsilon}$

α: needs to be tuned

$\beta_1 = 0.9$

$\beta_2 = 0.999$

$\varepsilon = 10^{-8}$

} usually fixed

Tutorials

- Google
  - https://www.tensorflow.org/tutorials/
- YouTube videos by Siraj Ravel
  - https://www.youtube.com/watch?v=2FmcHiLCwTU&vl=en
  - https://www.youtube.com/watch?v=H1AllrJ-_30
- Berkeley CS294
  - https://www.youtube.com/watch?v=xZKj7Z1CwHc&list=PLkFD6_40KJIxJMR-j5A1mkxK26gh_qg37&index=23&t=0s
- UNC BIOS735: https://biodatascience.github.io/statcomp/ml/nn.html
- …

What is deep learning?

Artificial Intelligence

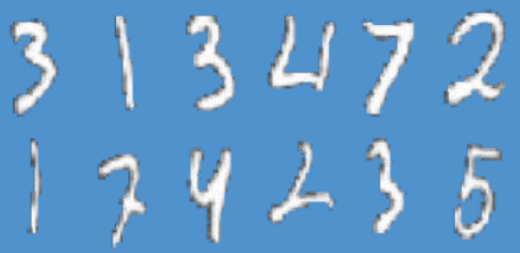
Any technique that enables computers to mimic human behavior

Machine Learning

Ability to learn without explicitly being programmed

Deep Learning

Extract patterns from data using neural networks



The Rise of Deep Learning