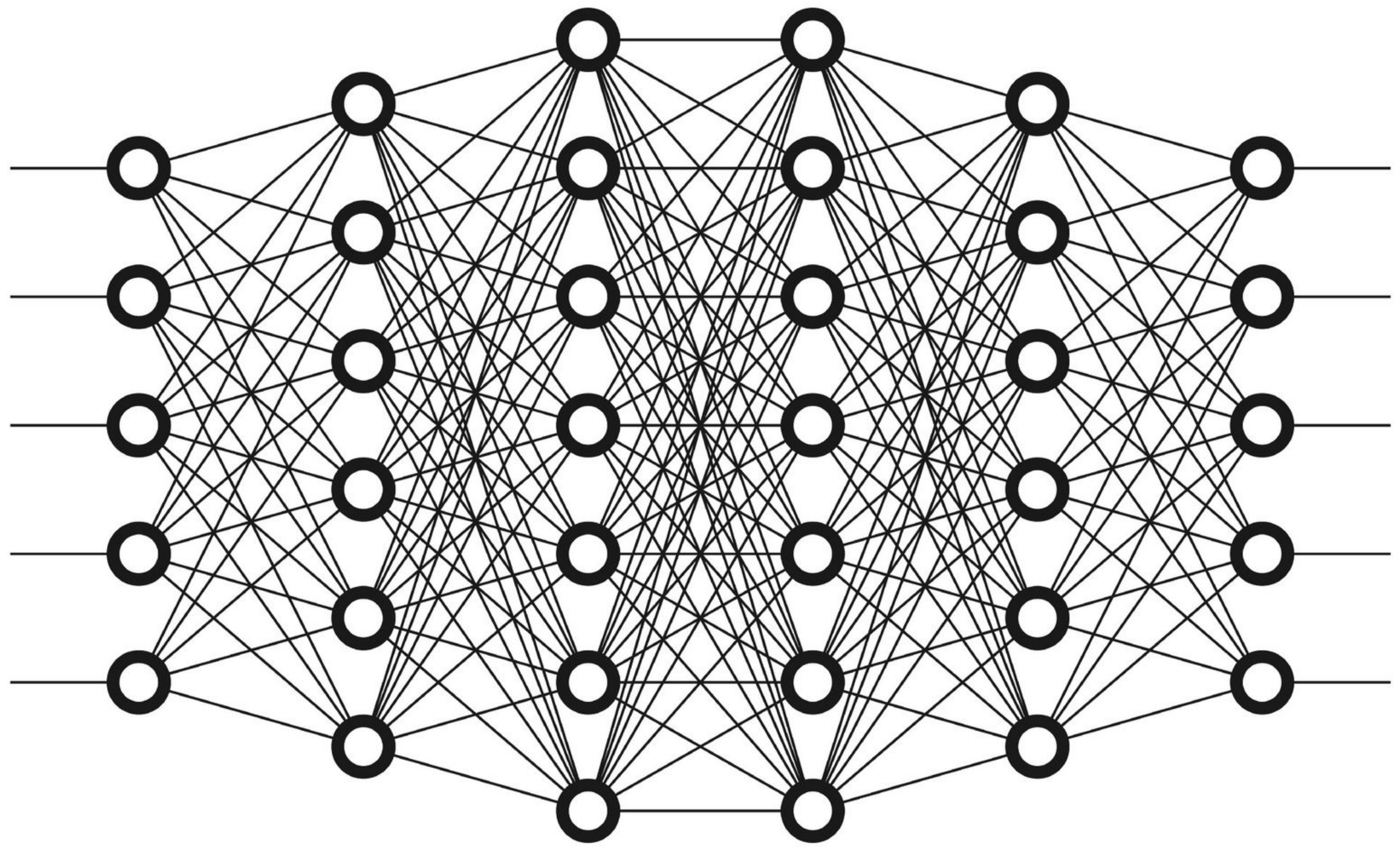
NATIONAL U

Neural Networks

CCDEPLRL: Deep Learning

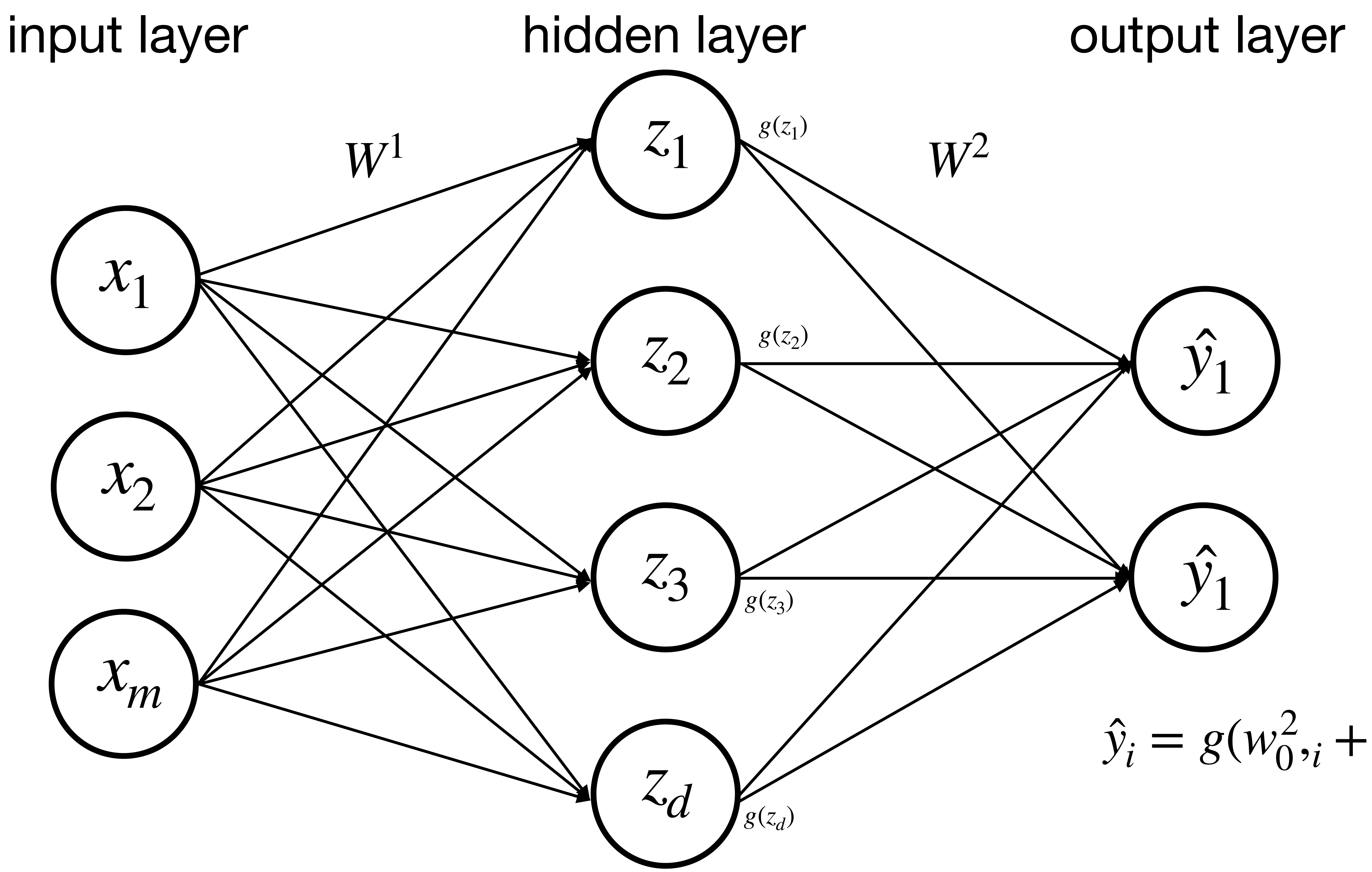


Joseph Jessie S. Oñate, MSc.

Neural Networks

Neural Network

Single Layer Neural Network



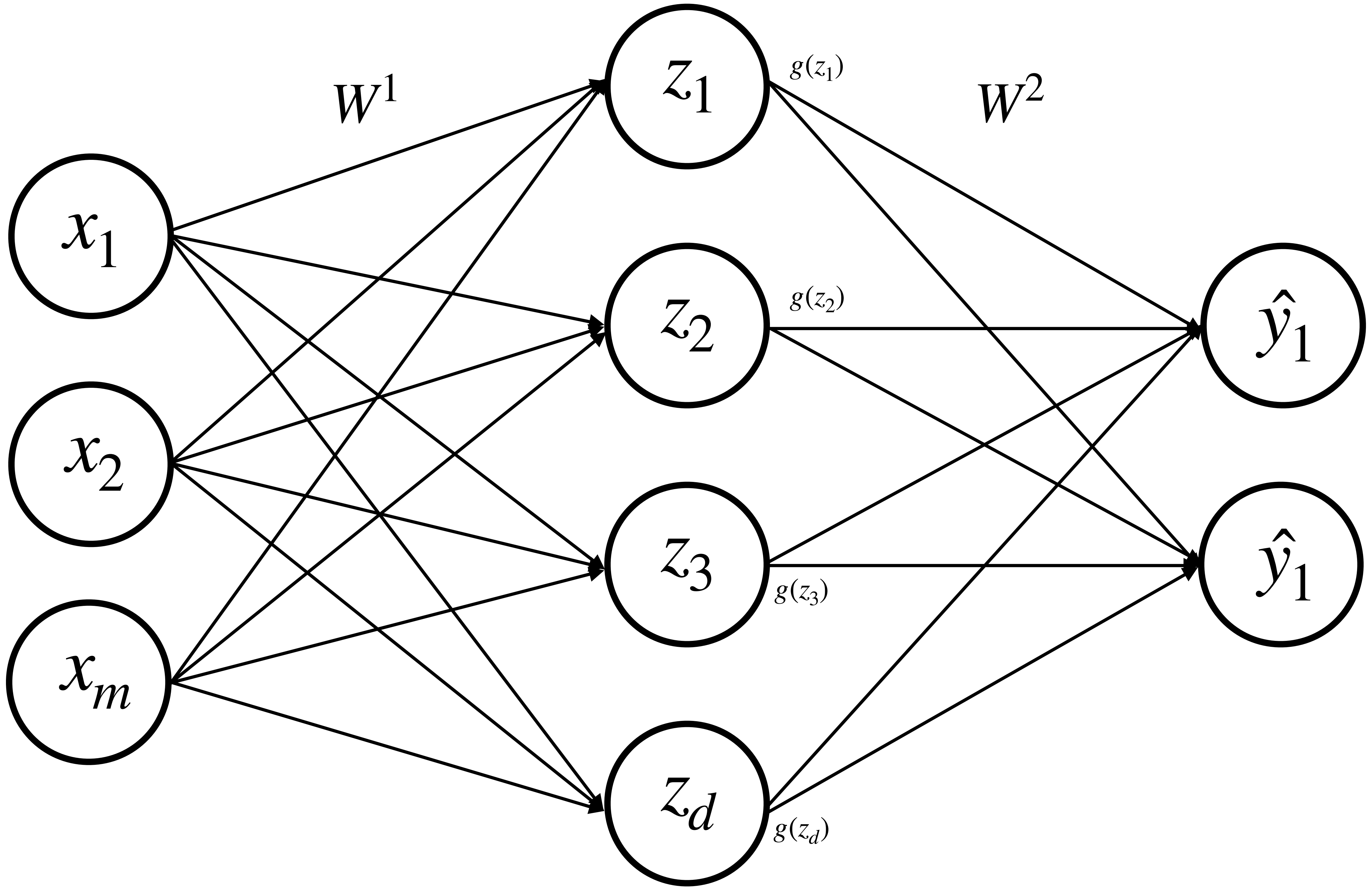
$$\hat{y}_i = g(w_0^2,_i + \sum_{j=1}^{d} g(z_j)w_j^2,_i)$$

$$z_i = w_0^1,_i + \sum_{j=1}^{m} x_j w_j^1,_i$$

Quantifying Loss

Quantifying Loss

The **loss** of our network measures the cost incurred from incorrect predictions.

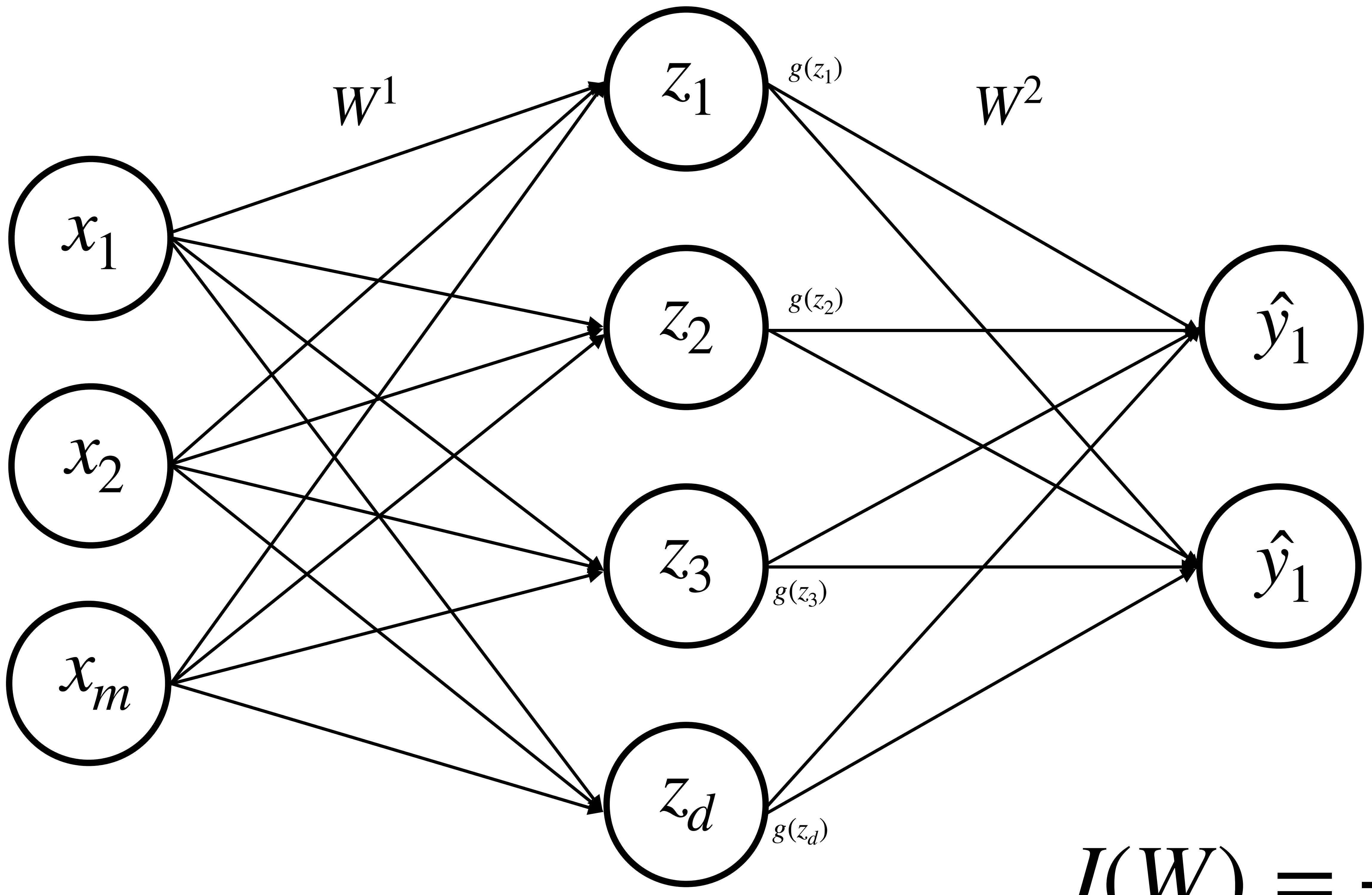


Actual: 1
Predicted: 0.1

$$L(\underline{f(x^{(i)}; W)}, \underline{y^{(i)}})$$

Empirical Loss

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



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

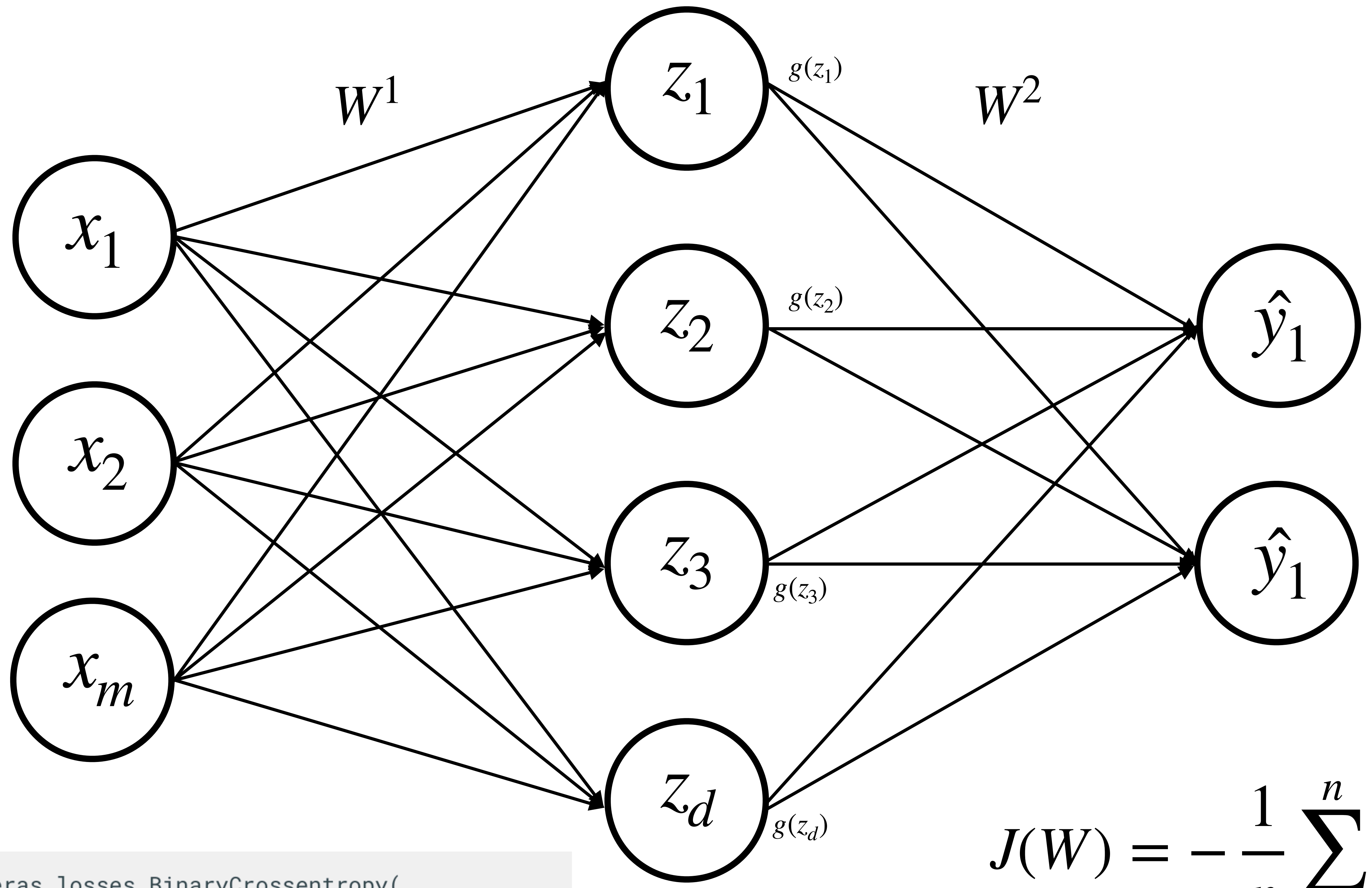
$$
J(W) = \frac{1}{n} \sum_{i=1}^{n} L(\underline{f(x^{(i)}; W)}, \underline{y^{(i)}})
$$

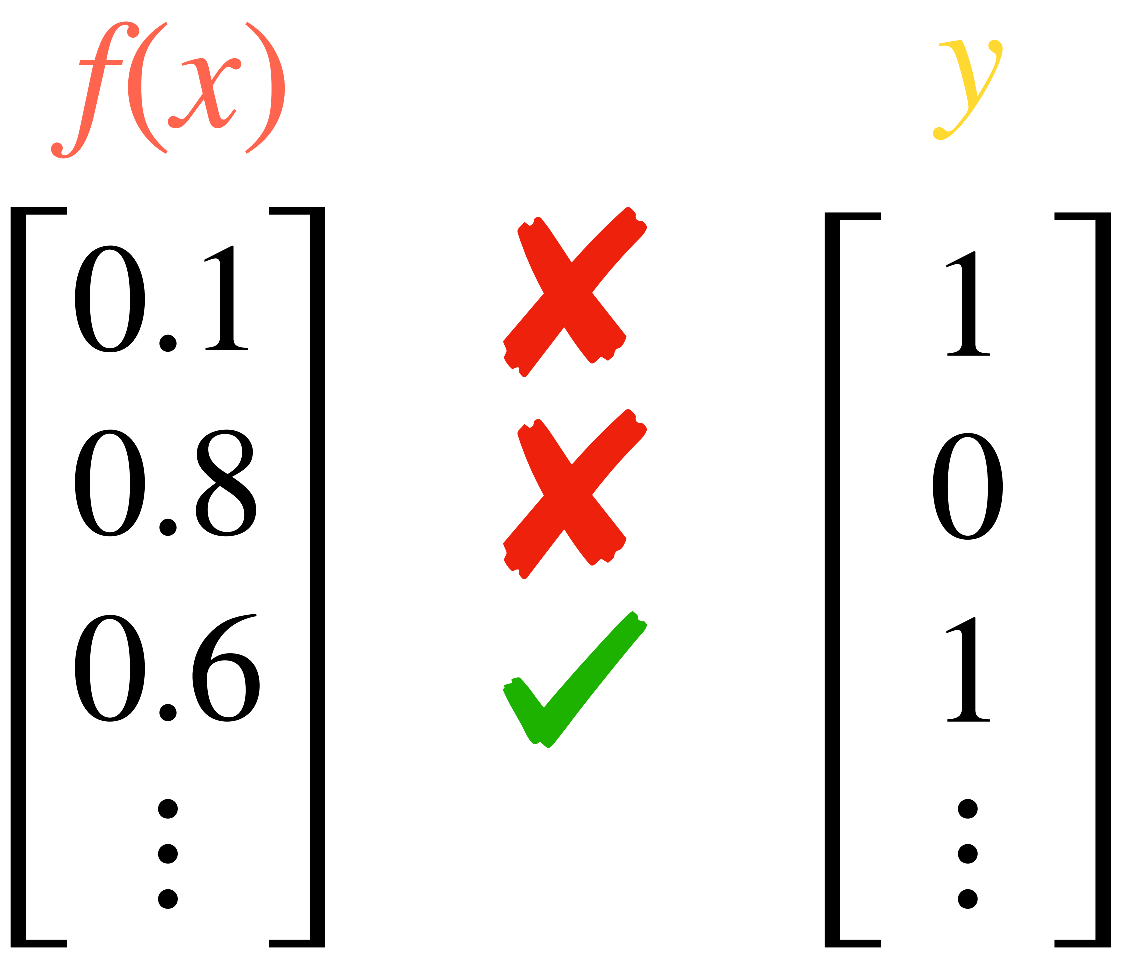
Also known as:

- 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



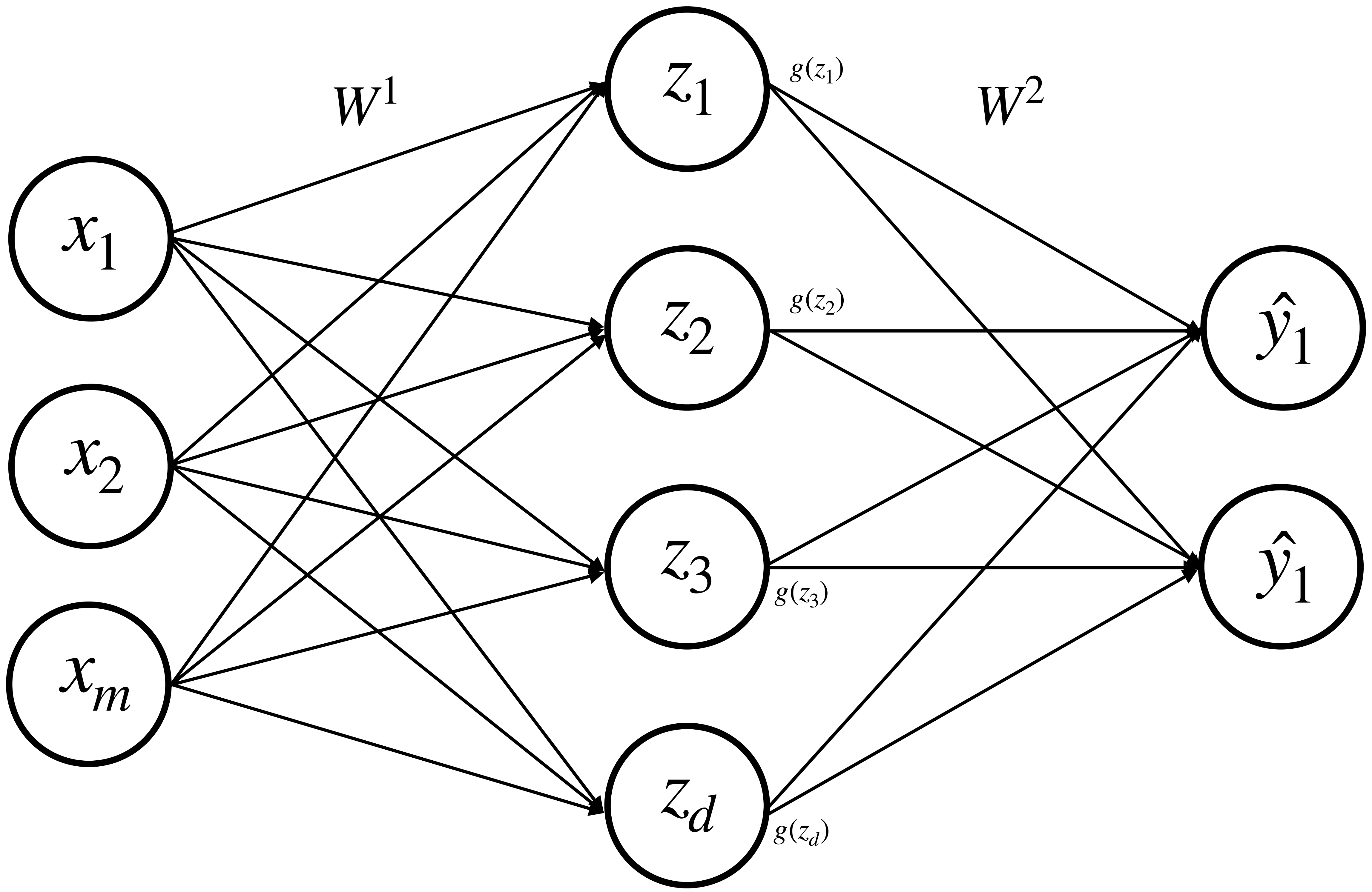


$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} y^{(1)} log(f(x^{(1)}; W)) + 1 - y^{(1)}) log(1 - f(x^{(1)}; W))$$

```
tf.keras.losses.BinaryCrossentropy(
    from_logits=False,
    label_smoothing=0.0,
    axis=-1,
    reduction='sum_over_batch_size',
    name='binary_crossentropy'
)
```

Mean Squared Error Loss

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



$$f(x) \quad \begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} \qquad y \quad \begin{bmatrix} 90 \\ 20 \\ 85 \\ \vdots \end{bmatrix}$$

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - f(x^{(i)}; W)\right)^2$$

```
tf.keras.losses.MeanSquaredError(
    reduction='sum_over_batch_size',
    name='mean_squared_error'
)
```

Training Neural Networks

Loss Optimization

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

$$W^* = \underset{W}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(f(x^{(i)}; W), y^{(i)})$$

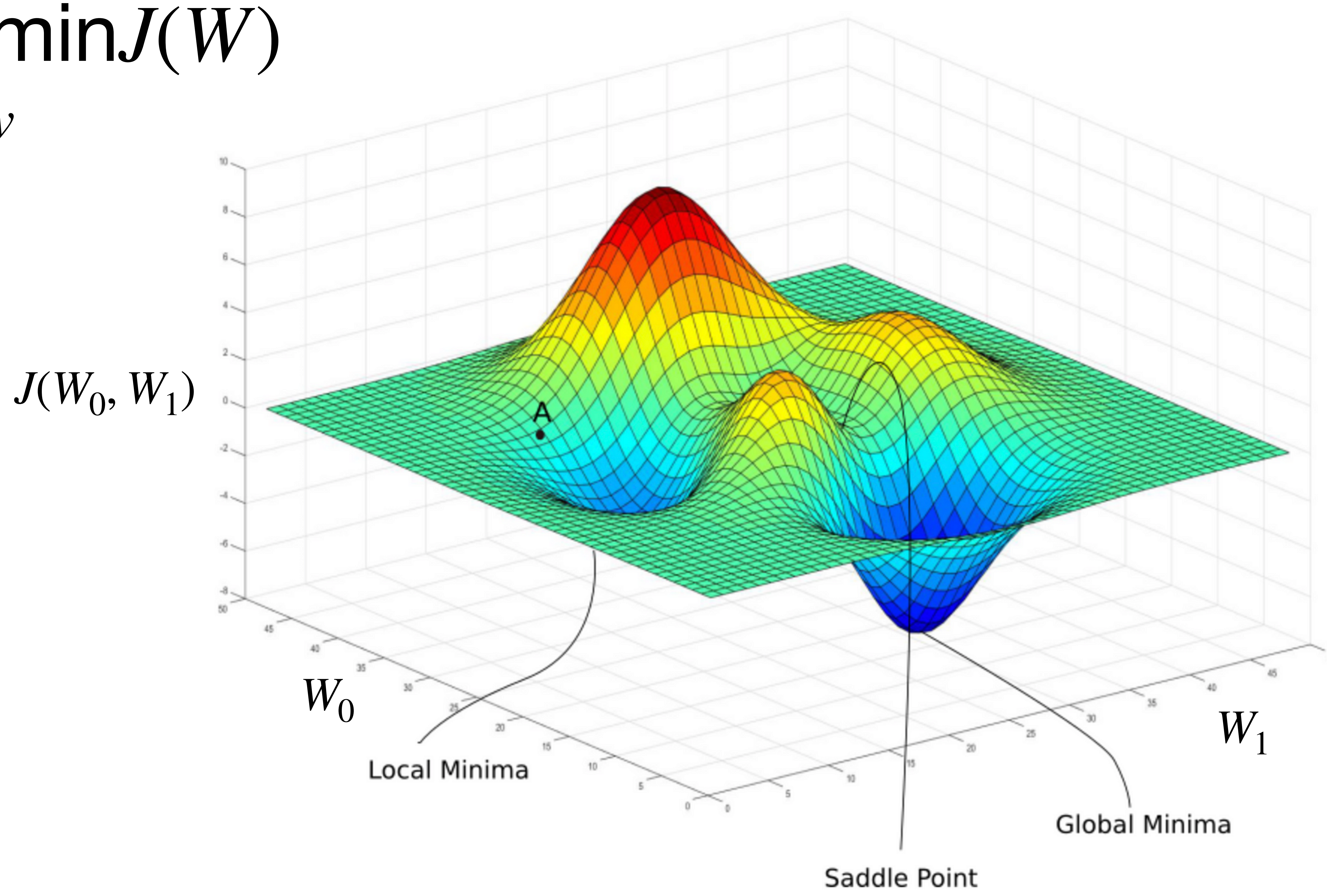
$$W^* = \underset{W}{\text{argmin}} J(W)$$

Loss Optimization

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

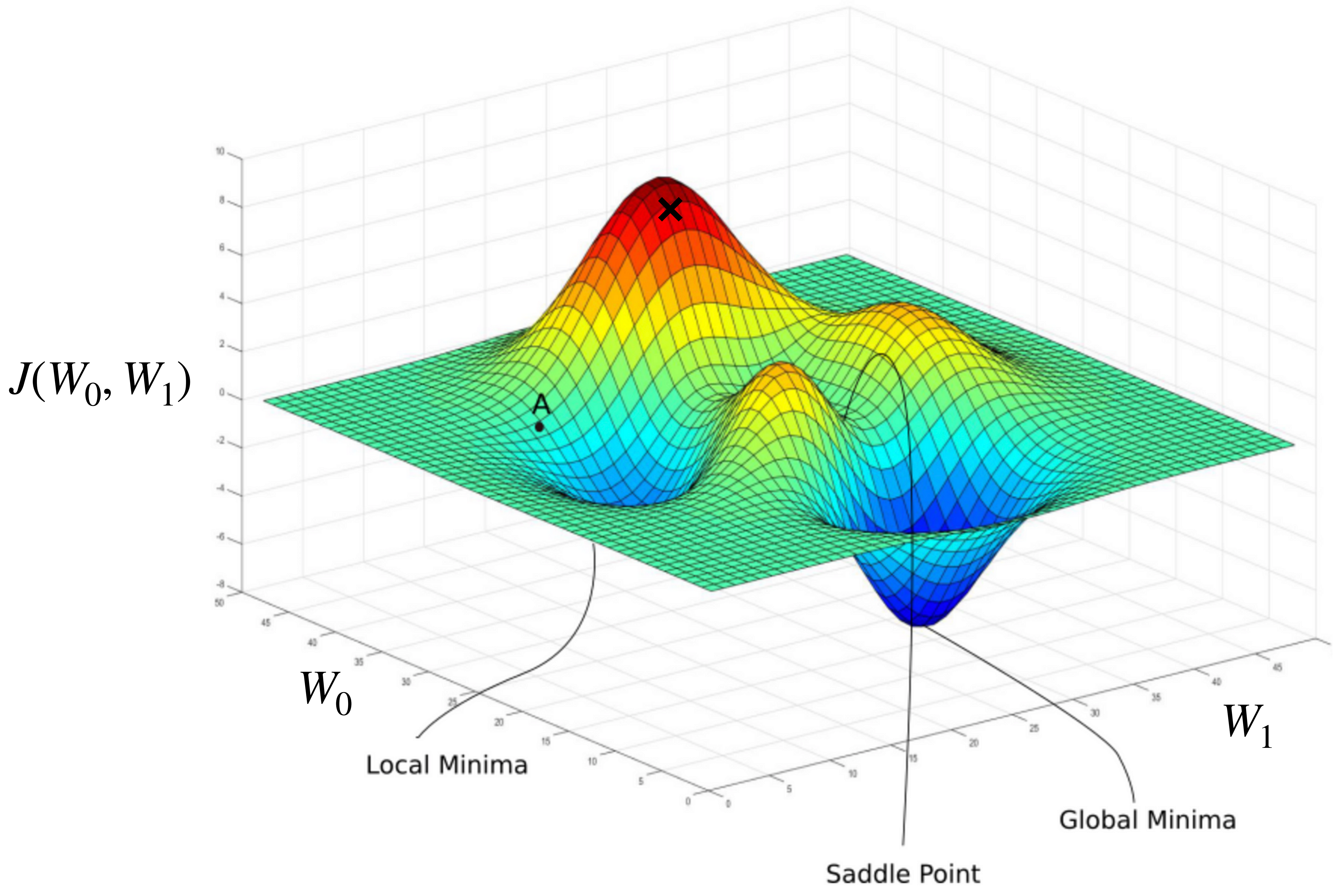
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

Remember:
Our loss is a function of
the network weights!



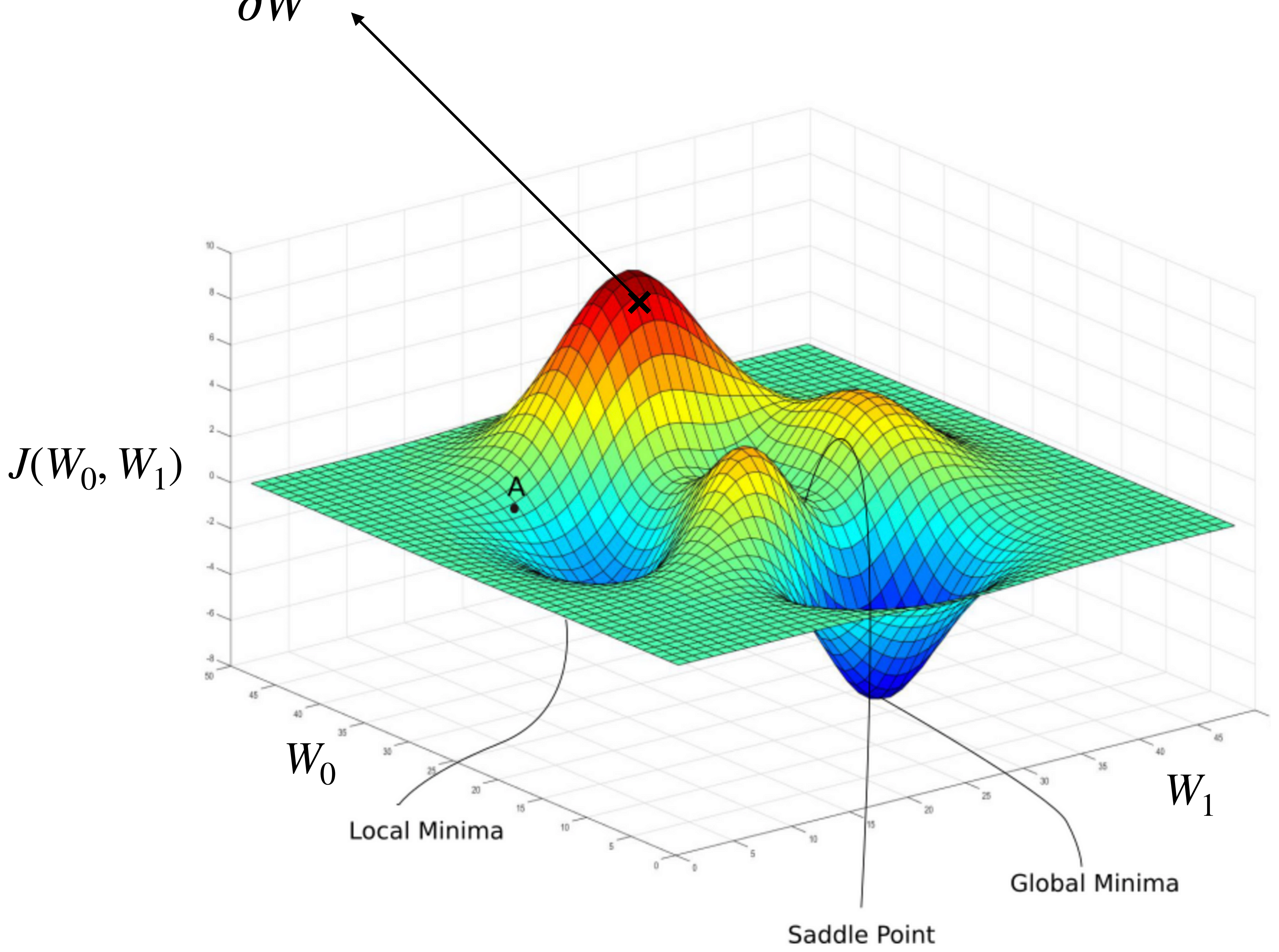
Loss **Optimization**

Randomly pick an initial *(W_0, W_1)*



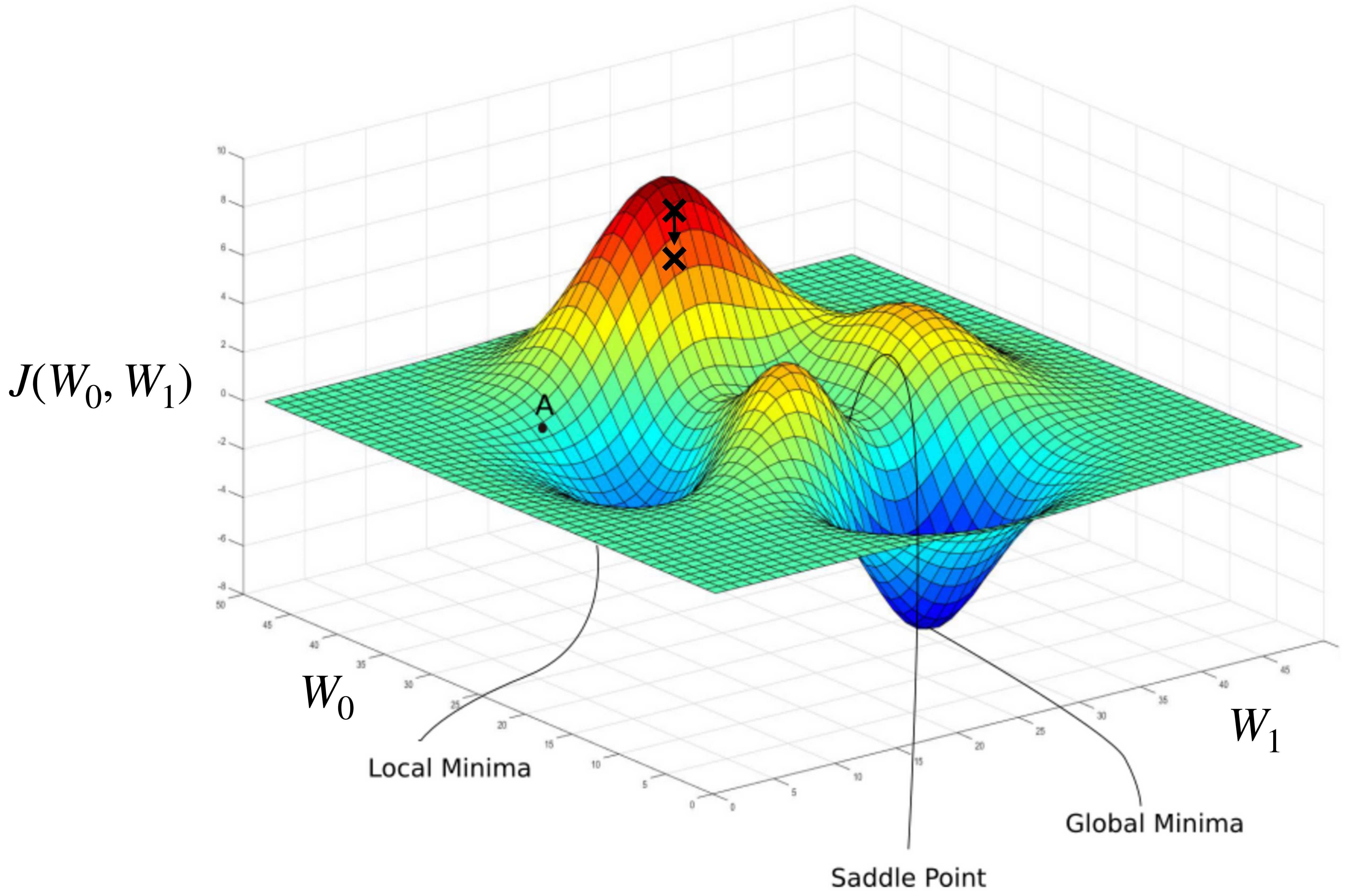
Loss Optimization

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



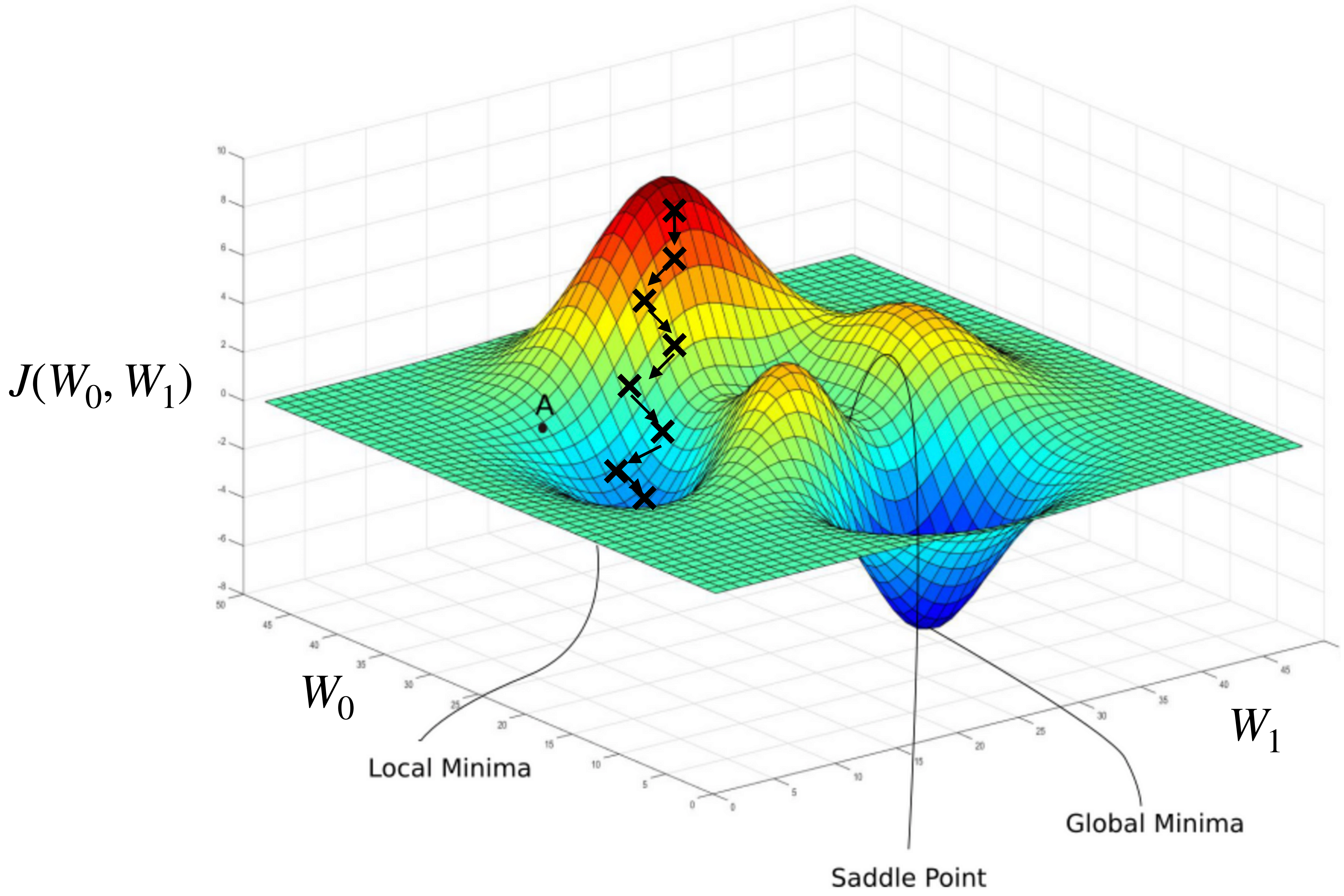
Loss Optimization

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:

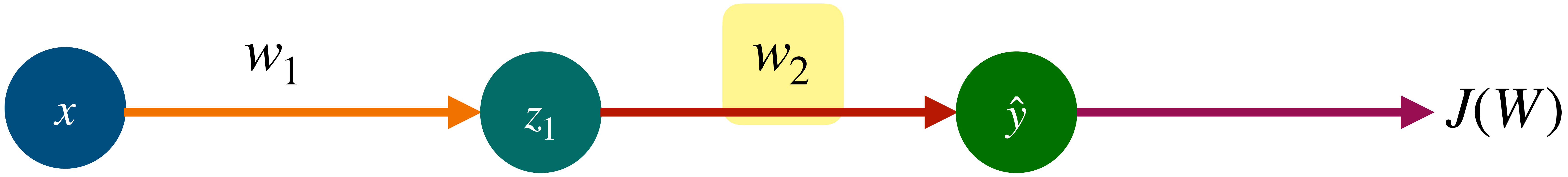
    3. Compute gradient, $\dfrac{\partial J(W)}{\partial W}$

    4. Update weights, $W \leftarrow W - \eta \dfrac{\partial J(W)}{\partial W}$

5. Return weights

Gradient Descent

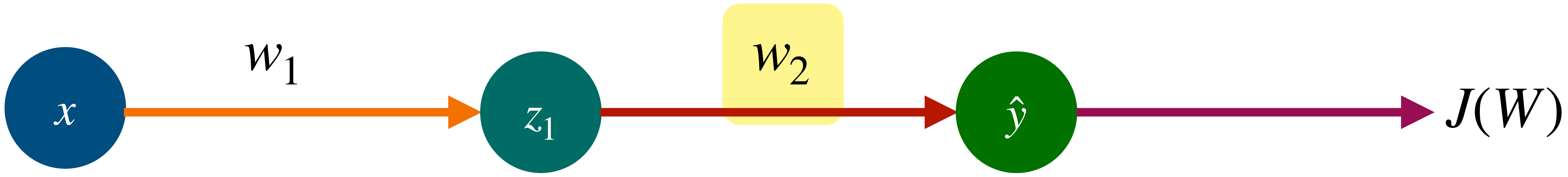
Backpropagation



How does a small change in one weight (ex. w_2) affect the final loss $J(W)$?

Gradient Descent

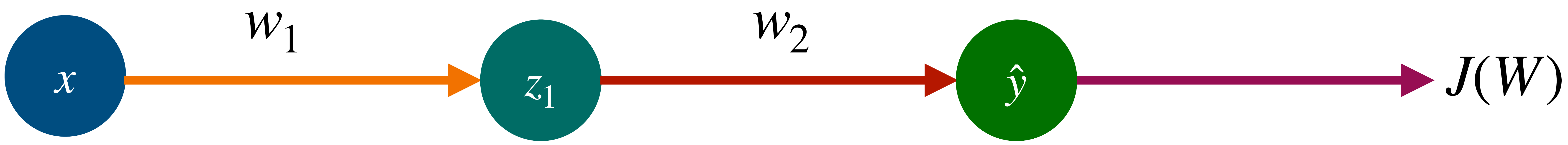
Backpropagation



$$\frac{\partial J(W)}{\partial w_2} =$$

Let's use the chain rule!

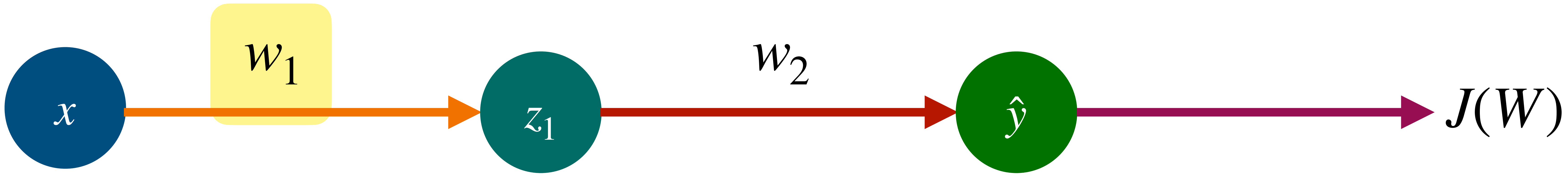
Gradient Descent
Backpropagation



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

Gradient **Descent**

Backpropagation



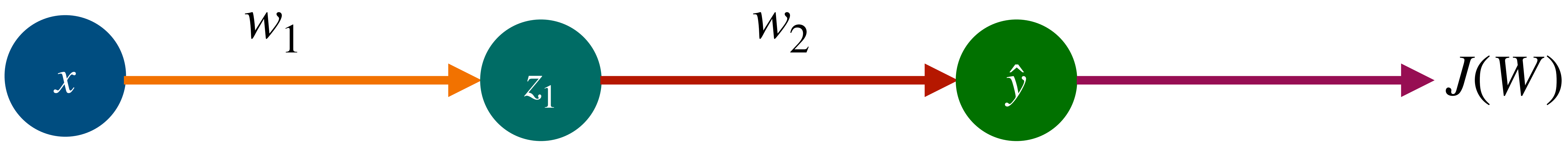
$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply the chain rule!

Apply the chain rule!

Gradient Descent

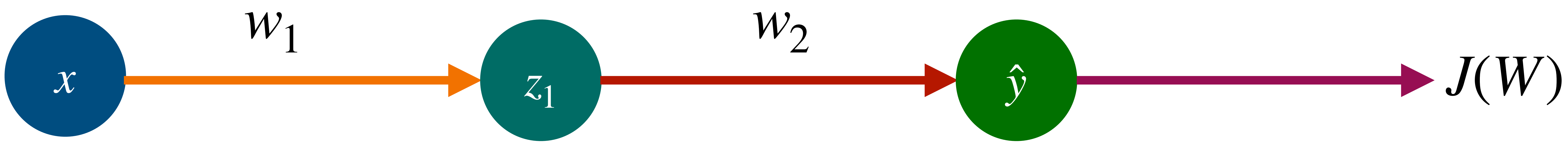
Backpropagation



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

Gradient Descent

Backpropagation



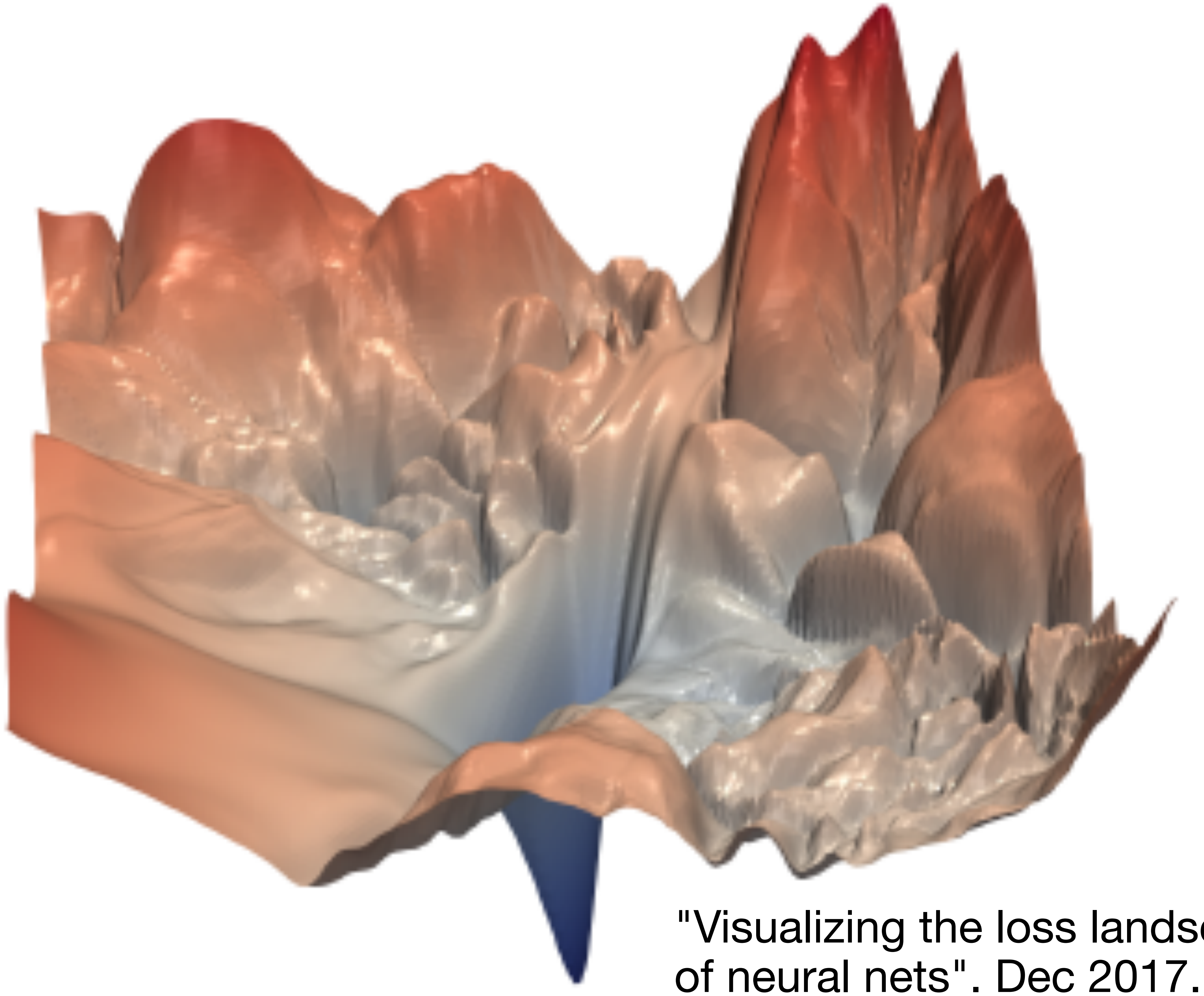
$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial \hat{y}}{\partial w_1}$$

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

Optimization in Neural Networks

Optimization in Neural Network

Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

Optimization in Neural Network

Loss Functions Can Be Difficult to Optimize

Remember:

Optimization through gradient descent

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

How can we set the learning rate?

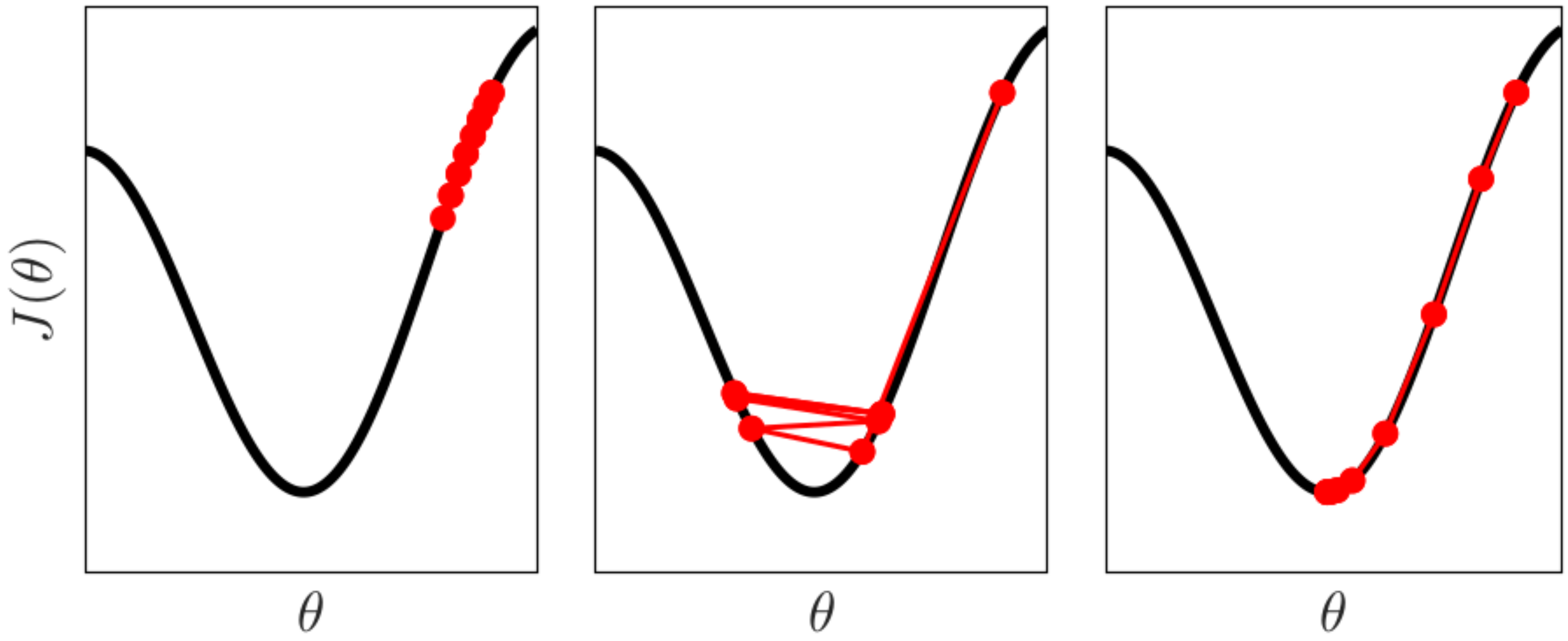
Optimization in Neural Network

Setting the Learning Rate

Small learning rate
converges slowly and gets stuck in false local minima

Large learning rates
overshoot, become unstable and diverge

Stable learning rates
converge smoothly and avoid local minima



Optimization in Neural Network

How to deal with this?

Idea I:

Try lots of different learning rates and see what works "just right"

Idea II:

Do something smarter!

Design an adaptive learning rate that "adapts" to the landscape

Optimization in Neural Network

Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
    - how large gradient is
    - how fast learning is happening
    - size of particular weights
    - etc...

Optimization in Neural Network

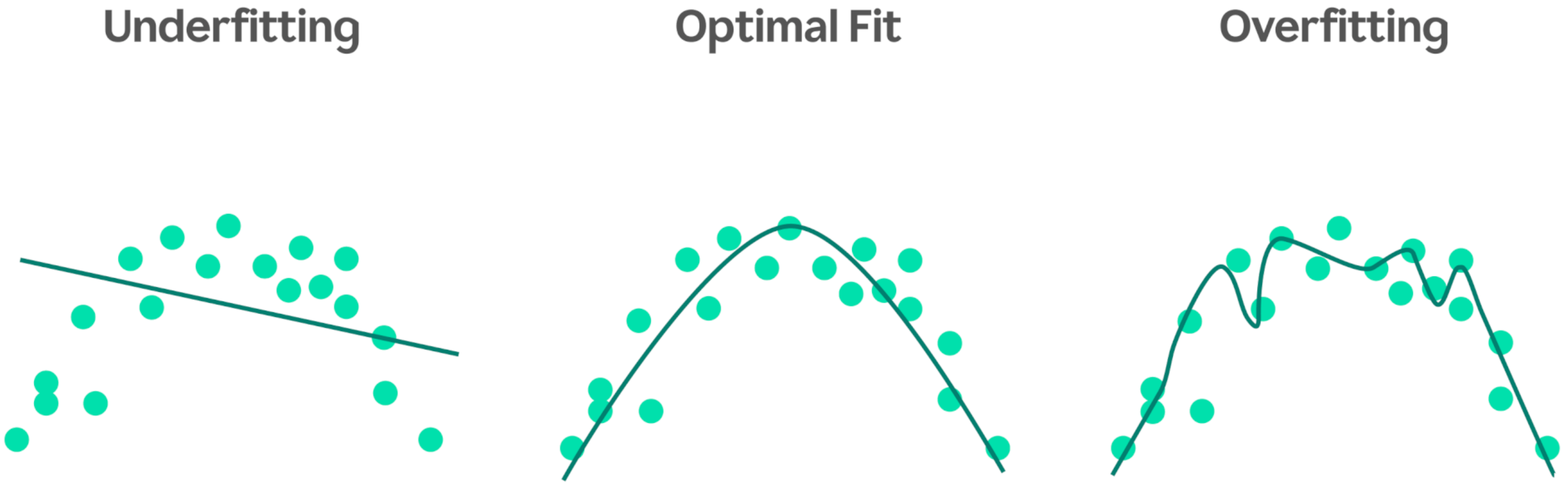
Gradient Descent Algorithms

Algorithm	TF Implementation
SGD	tf.keras.optimizers.SGD()
Adam	tf.keras.optimizers.Adam()
Adadelta	tf.keras.optimizers.Adadelta()
Adagrad	tf.keras.optimizers.Adagrad()
RMSprop	tf.keras.optimizers.RMSprop()
Others	https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Overfitting in Neural Networks

Overfitting in Neural Network

The Problem of Overfitting



Overfitting in Neural Network

Regularization

What is it?

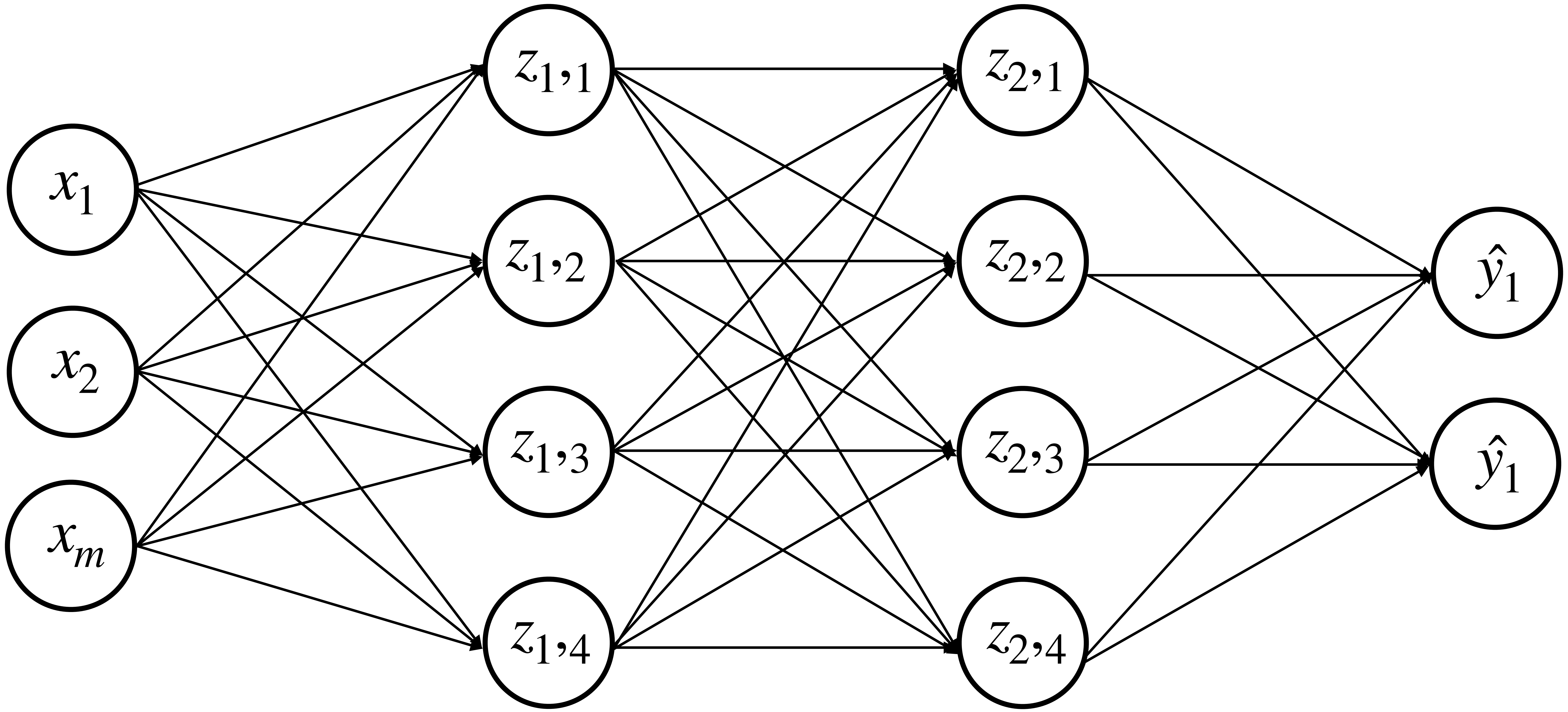
Technique that constrains our optimization problem to discourage complex models

Why do we need it?

Improve generalization of our model on unseen data

Overfitting in Neural Network

Regularization I: Dropout



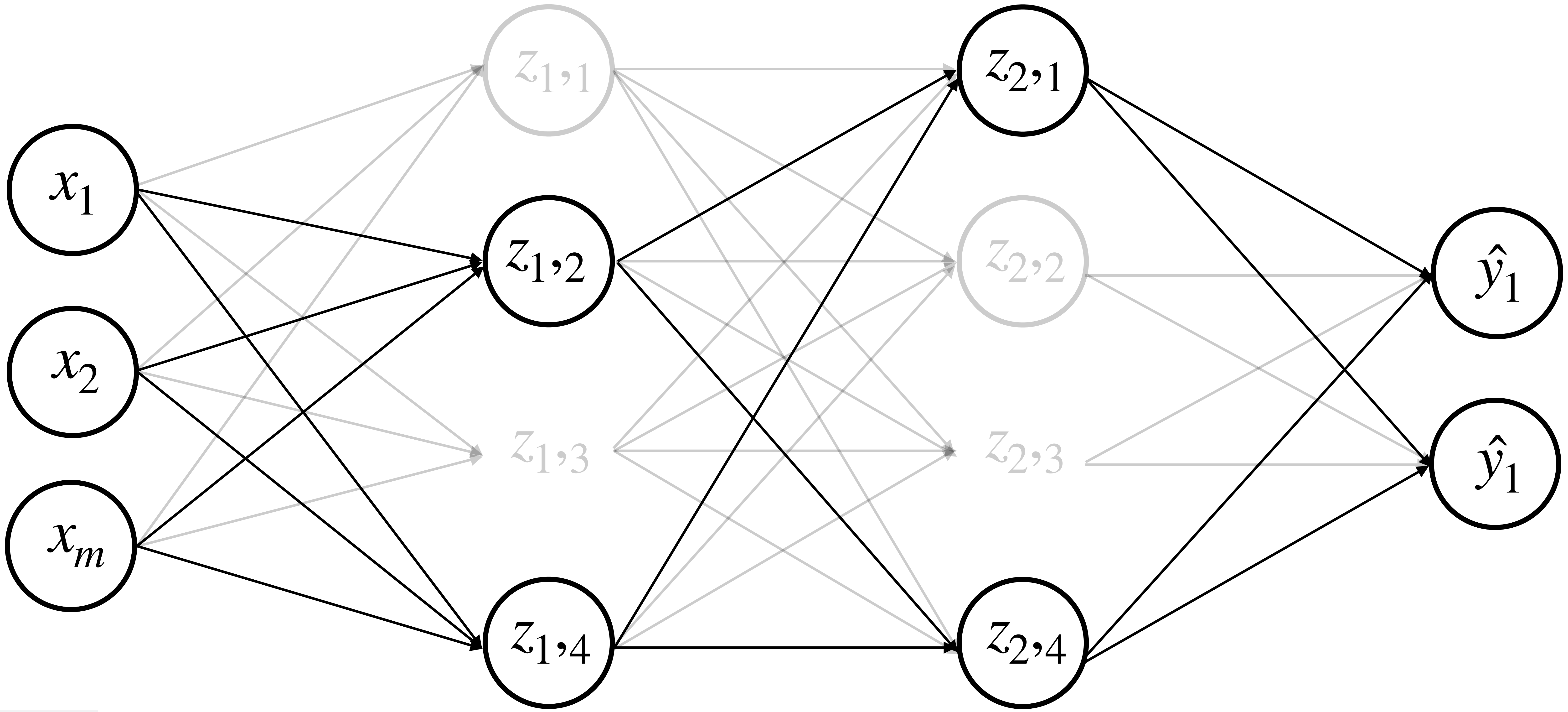
During training, randomly set some activations to 0

Overfitting in Neural Network

Regularization I: Dropout

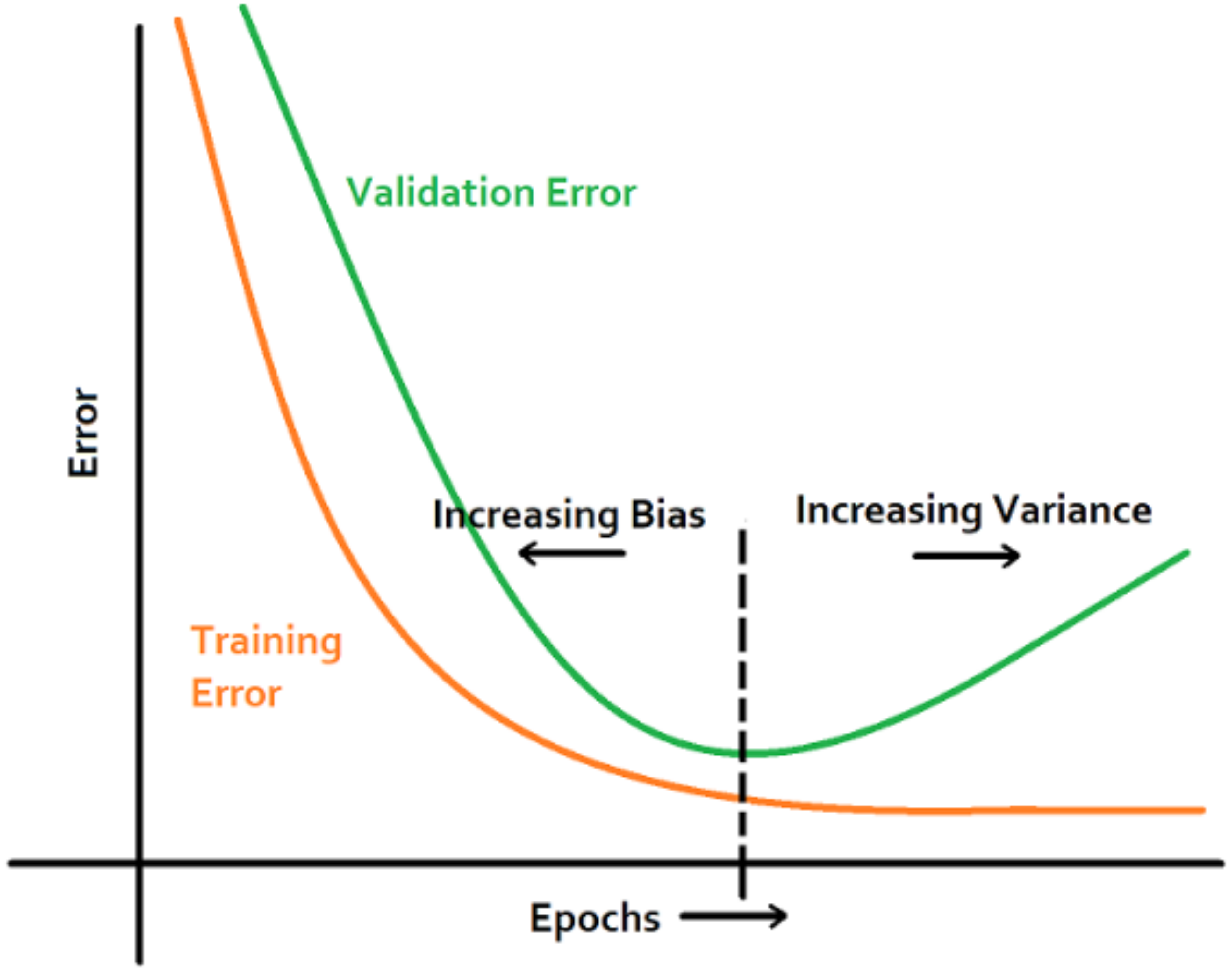
- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any | node

```
tf.keras.layers.Dropout(
    rate, noise_shape=None, seed=None, **kwargs
)
```



Overfitting in Neural Network

Regularization 2: Early Stopping




```
tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    min_delta=0,
    patience=0,
    verbose=0,
    mode='auto',
    baseline=None,
    restore_best_weights=False,
    start_from_epoch=0
)
```

Stop training before we have a chance to overfit

Thank you!

CCDEPLRL: Deep Learning