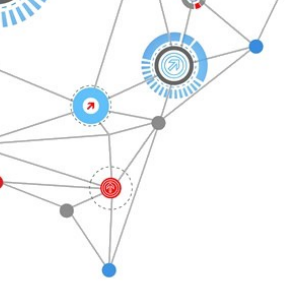
Claudio Mirabello

Intro to Neural Networks and Deep Learning

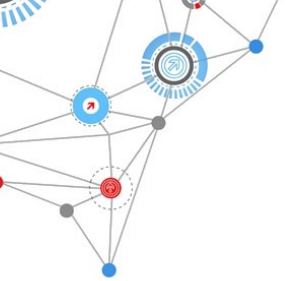
LINKÖPING UNIVERSITY

NBIS
NATIONAL BIOINFORMATICS INFRASTRUCTURE SWEDEN



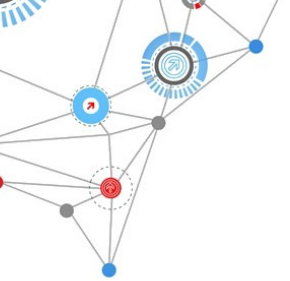
Resources

- MIT lectures on Deep Learning (http://introtodeeplearning.com/)
- TensorFlow Playground (https://playground.tensorflow.org)
- Keras Docs (https://keras.io)



Quick notes

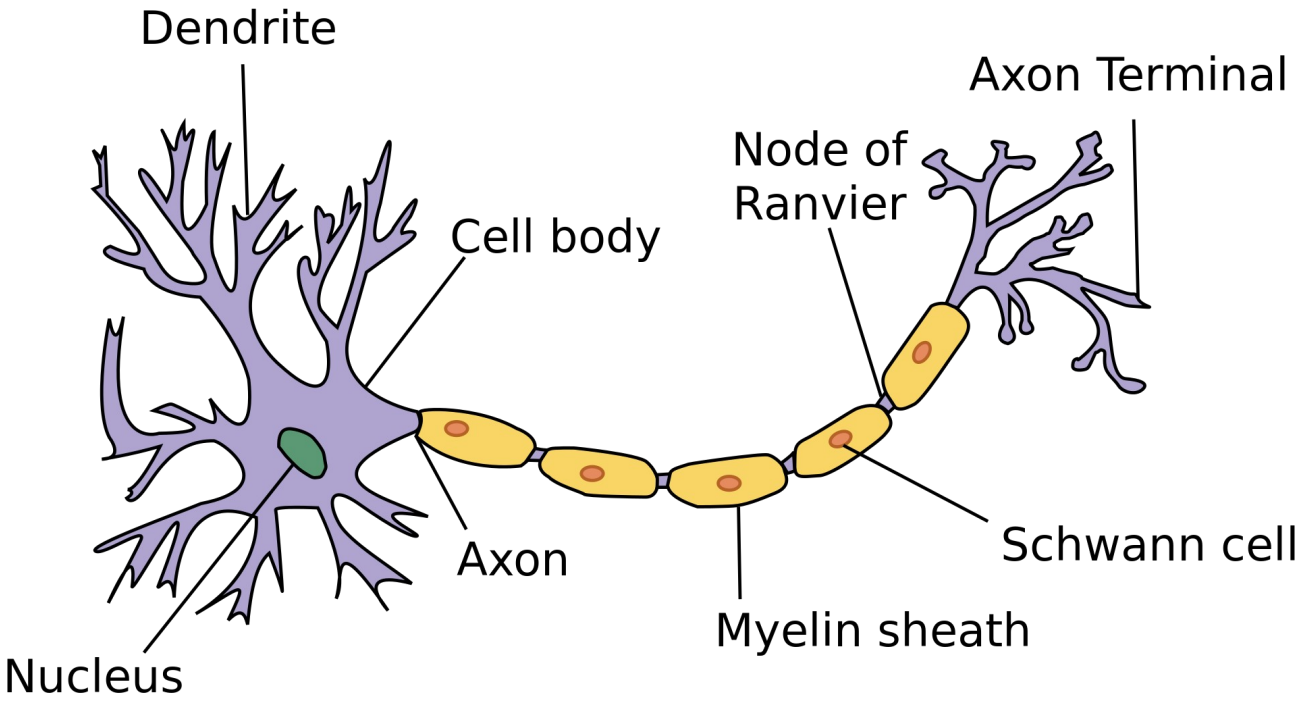
- All the code is in python, sorry R users! (you can still do everything we teach here in R as well)
- You must be tired of online lectures, so we will be taking plenty of breaks
- Interrupt at any time if you have questions, keep in mind that I might not see the chat
- Labs usually not about bioinformatics, not easy to build interesting datasets that can be processed in a seconds/minutes
- But everything you learn will be very much relevant to your research



Quick notes

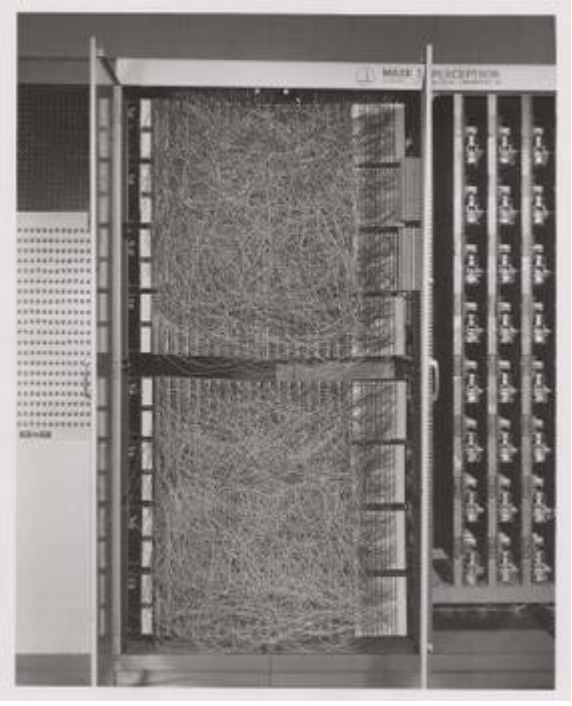
- We’re going to cover mostly classification because it is easy to visualize and the most common task
- Regression in the end is not that different, as we will see it is mostly about changing one line of code

This is a neuron

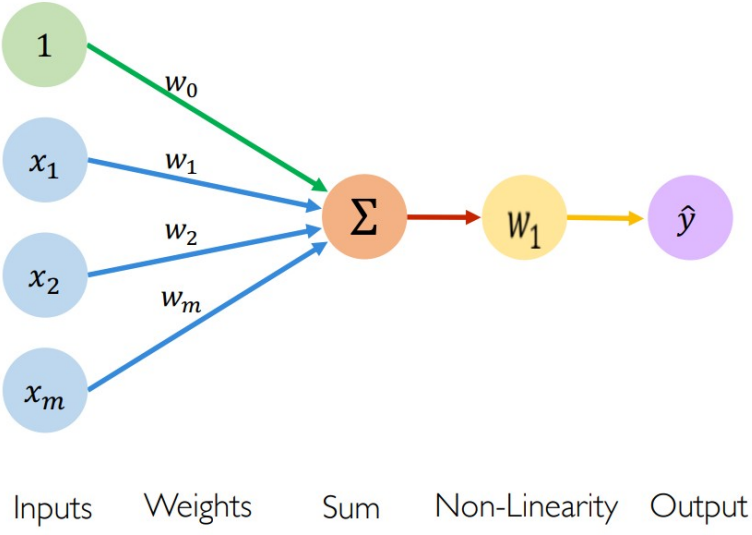


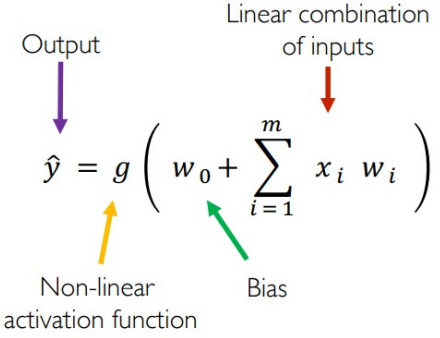
wikipedia.org

This is a perceptron (1958)

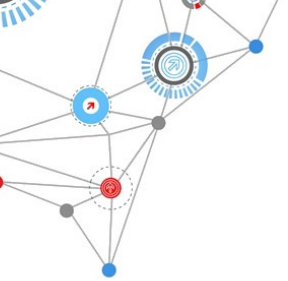


Mark I Perceptron machine
wikipedia.org





MIT “Intro to Deep Learning”

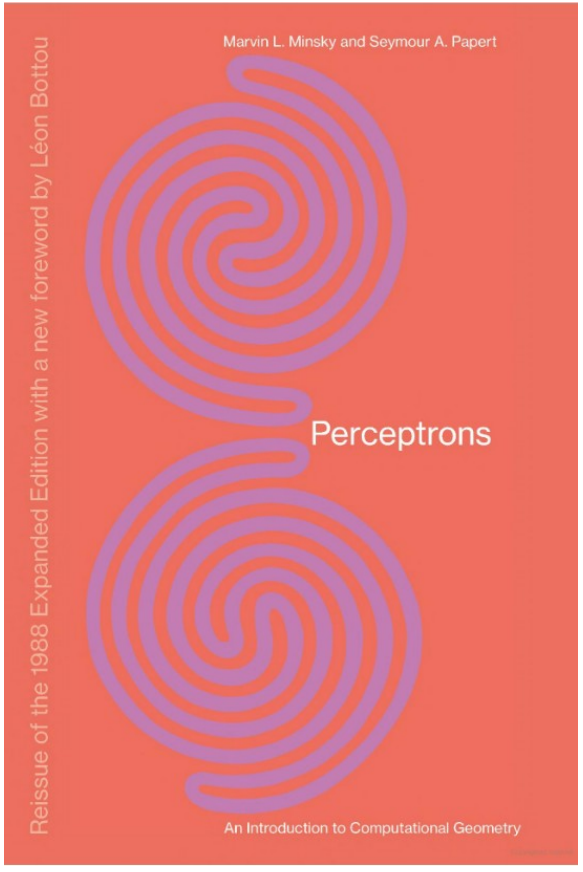


A lot was expected of perceptrons

- *"the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."*

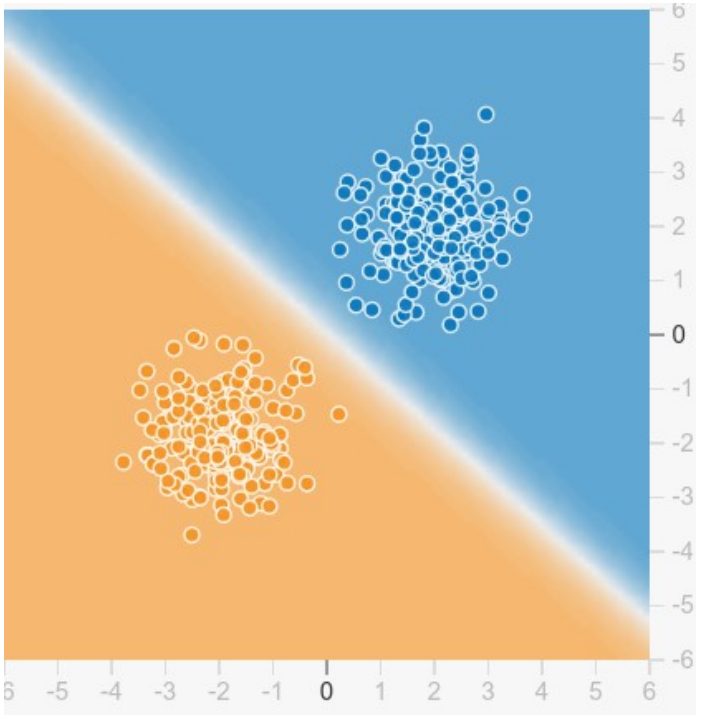
The New York Times

Perceptrons can only learn linearly separable classes



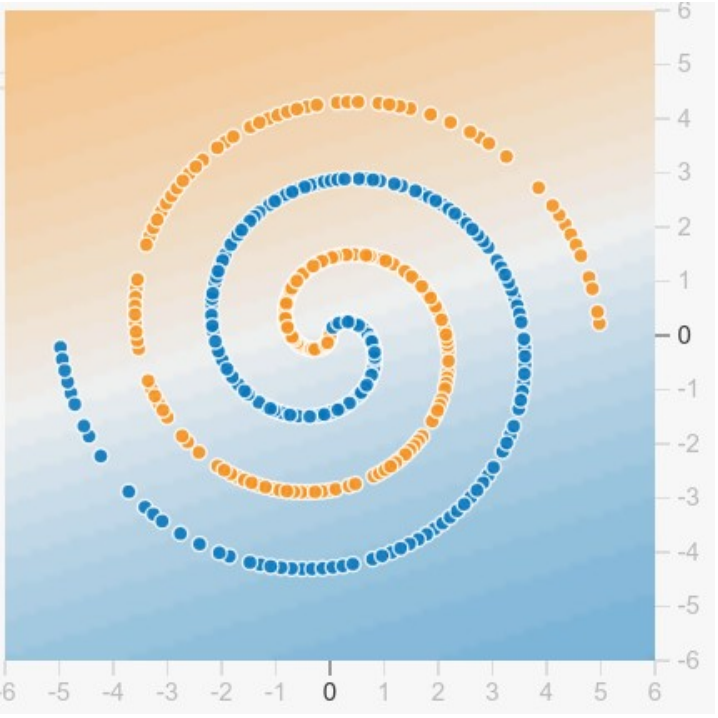
Minsky and Papert, 1969

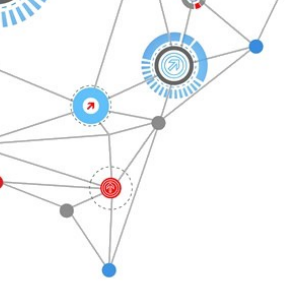
Perceptrons can only learn linearly separable classes



TensorFlow Playground

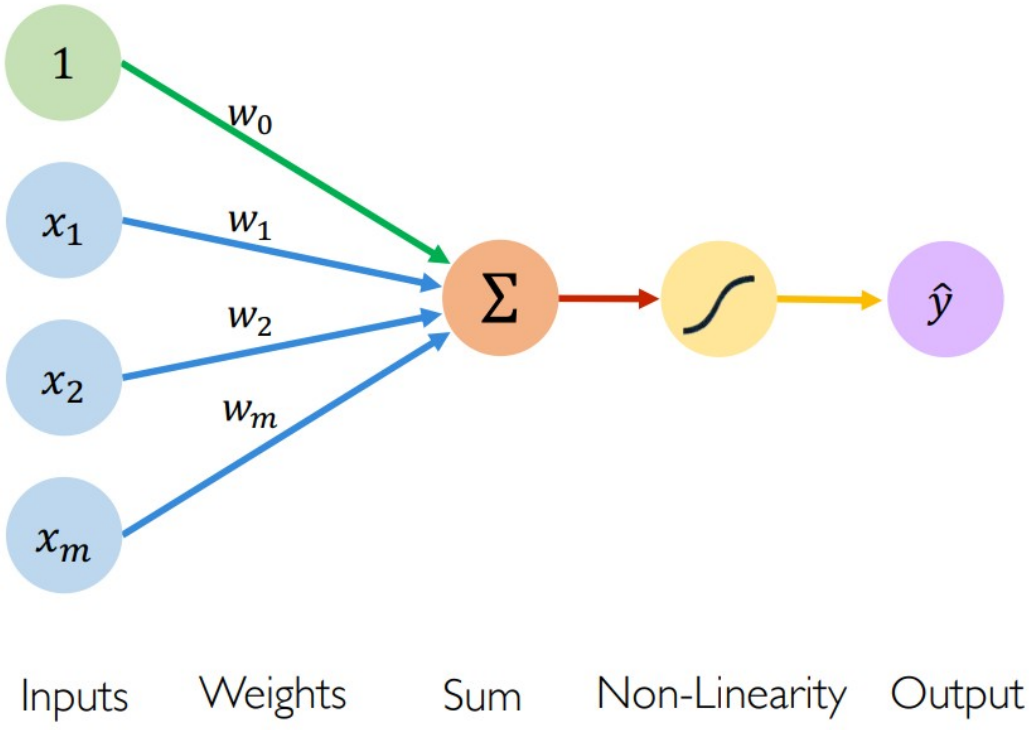
But sometimes you want to model non-linear functions





How do we make this non-linear then?
Two ingredients to add

Non-linear activation functions

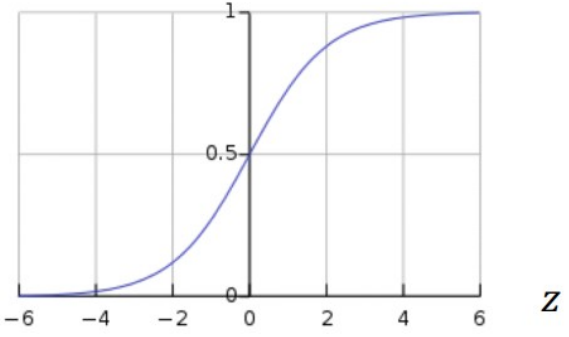


Activation Functions

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

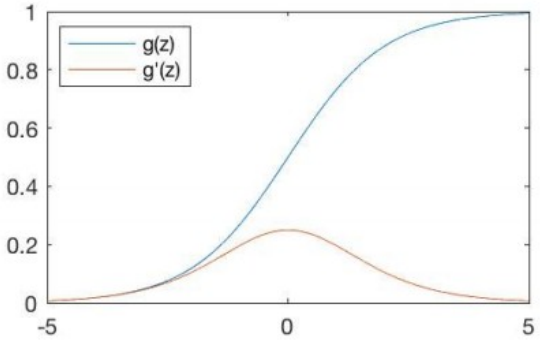
- Example: sigmoid function

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



Common activation functions

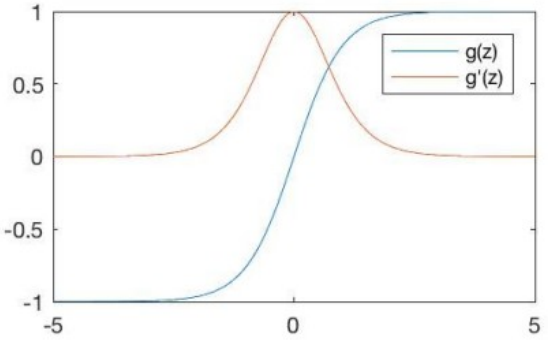
Sigmoid Function



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

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

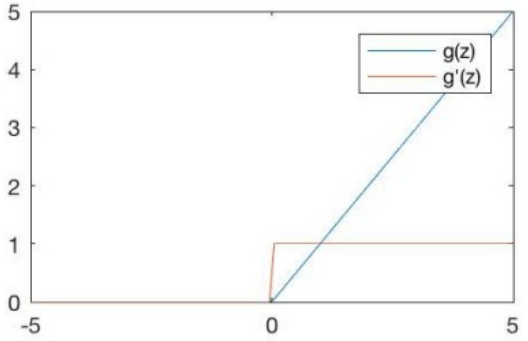
Hyperbolic Tangent



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

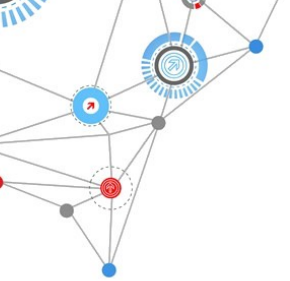
$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

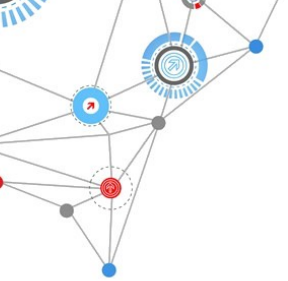
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$



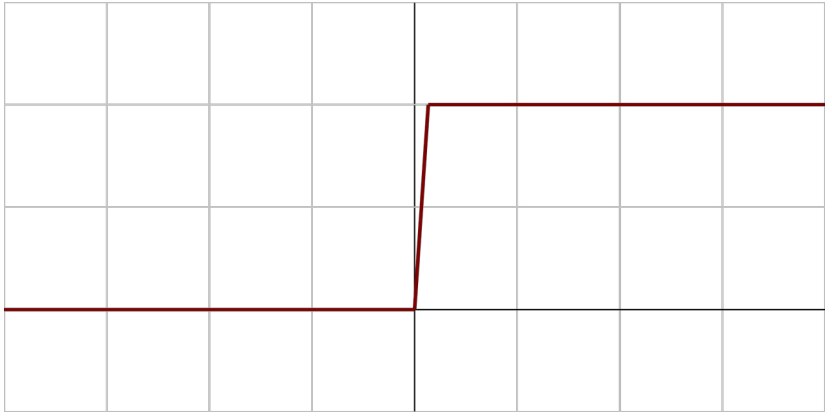
Special case: softmax

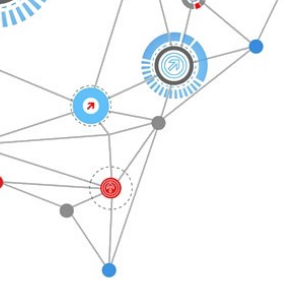
- Used in classification problems
- Given k classes, it decides which one is more likely
- One output per class, each output is assigned a probability from 0 to 1
- The sum of probabilities for all outputs is 1

$$\mathbf{g}(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \; for\; j = 1, \ldots, k$$



Wait a second, the perceptron already has a non-linear (step) activation function!

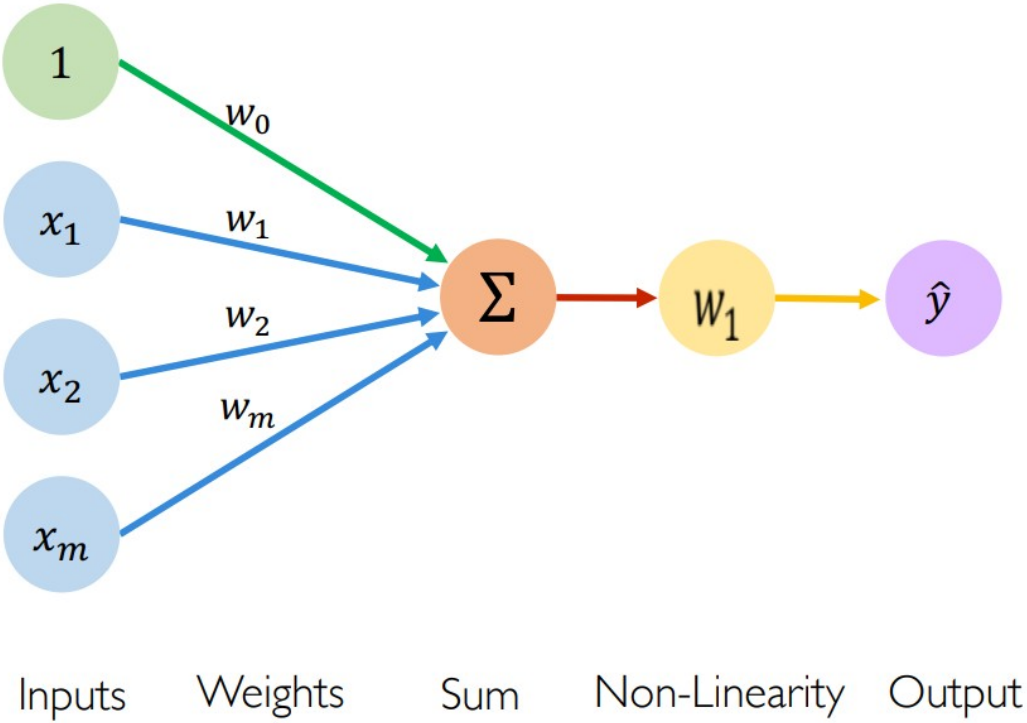


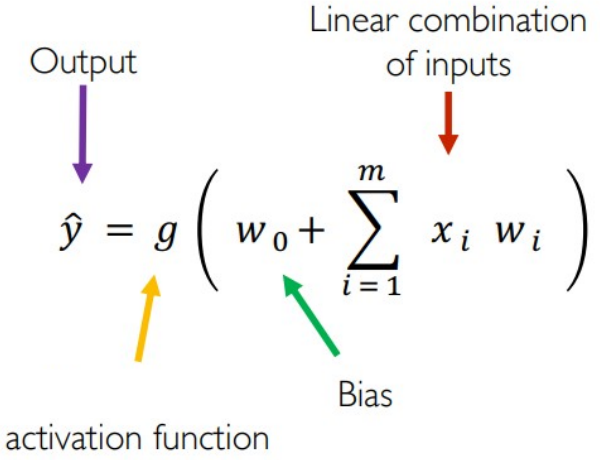


That was in the 50s

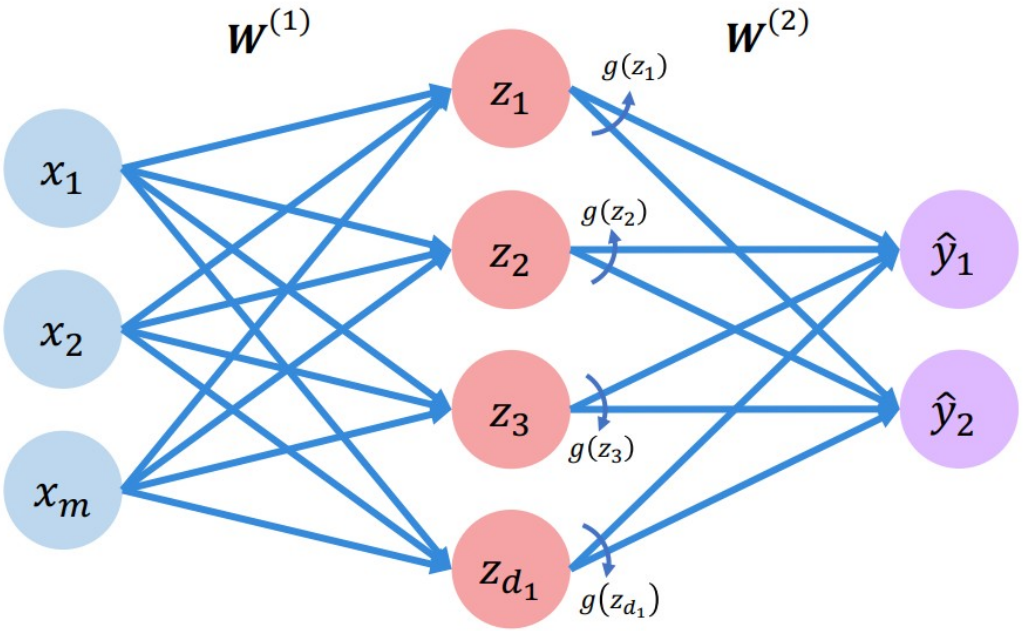
- Perceptron can't deal with many types of more complex patterns (XOR)
- Now we have huge, complex datasets (Big Data)
- Faster processors, more memory
- And a better understanding of learning processes

This is a perceptron





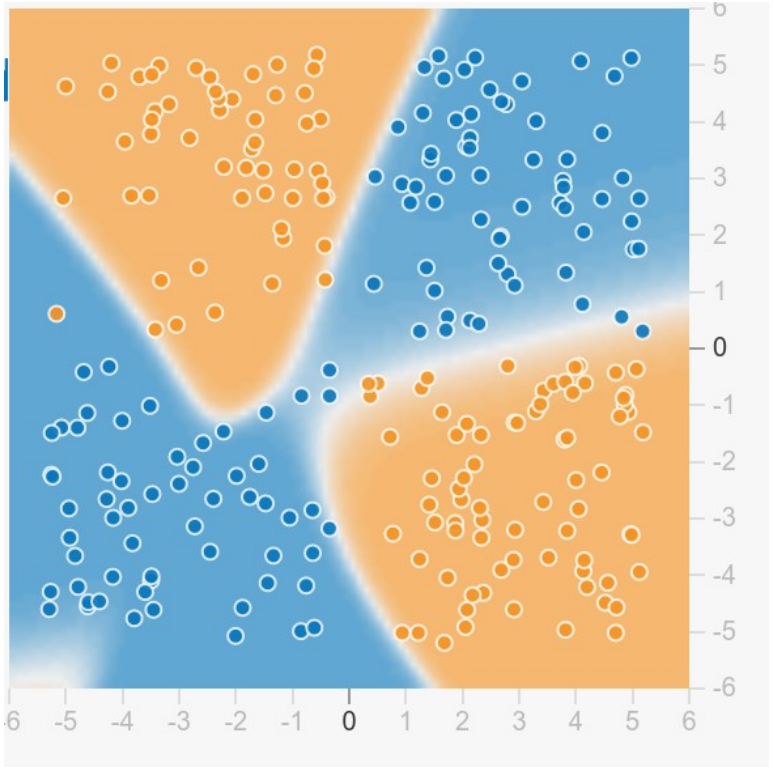
Multi-layer Perceptron (1986)



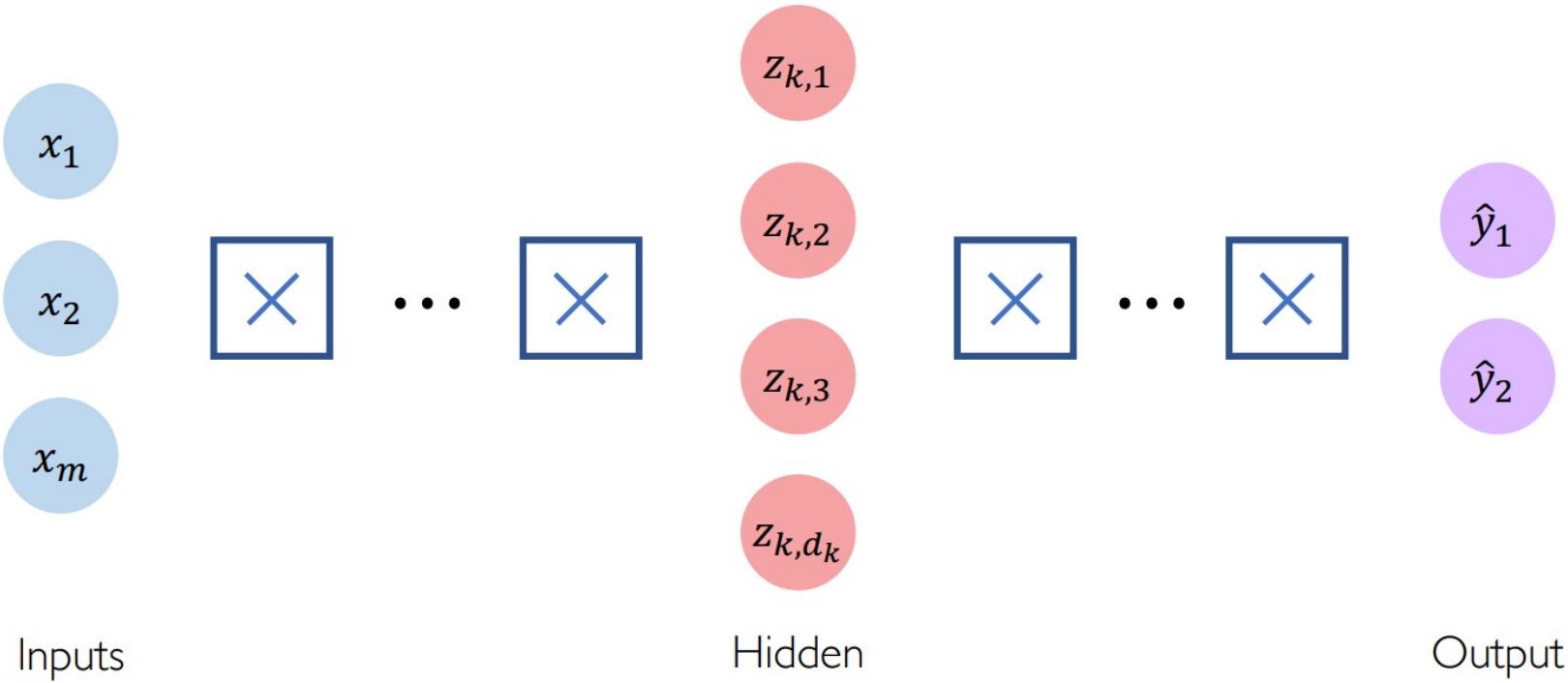
Inputs Hidden Final Output

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

Now we're getting somewhere

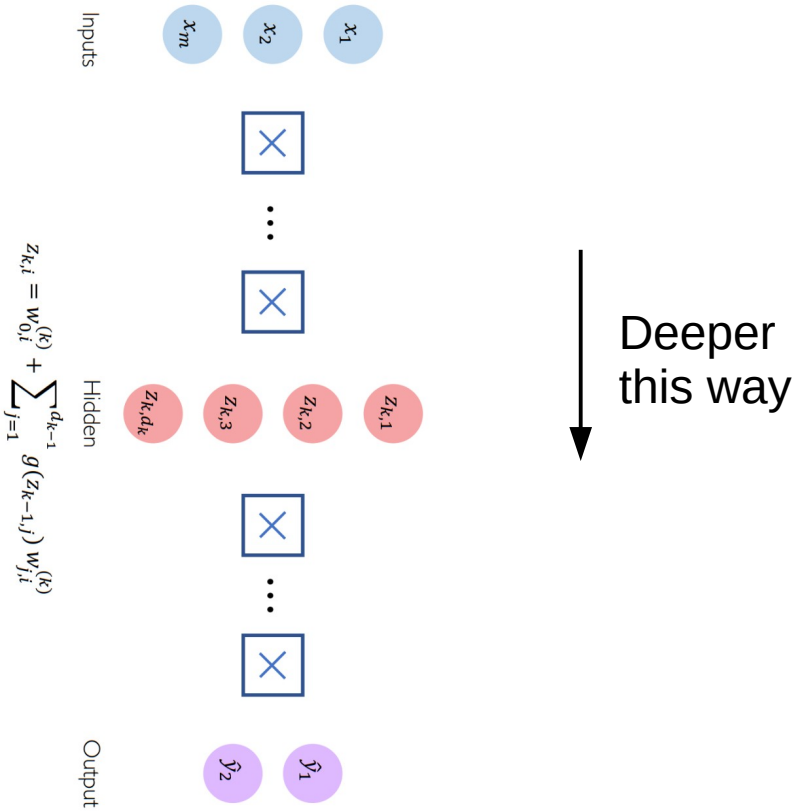


Why stop at one hidden layer?

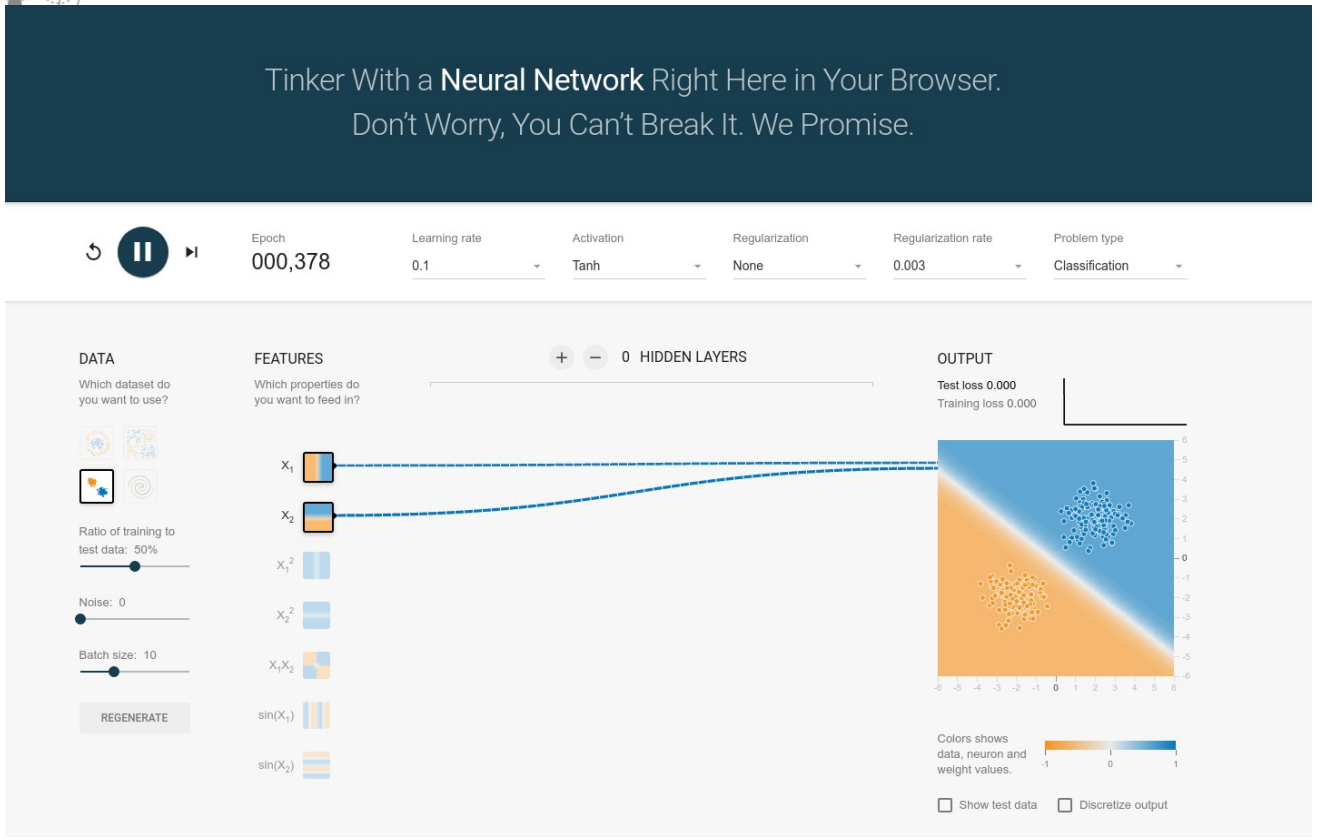


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Deep Networks are simply NNs with multiple hidden layers

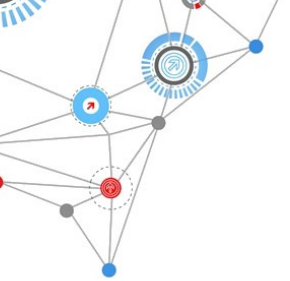


https://playground.tensorflow.org



Let's review:

- Perceptron
- XOR problem
- Activations
- Multi-layer perceptron

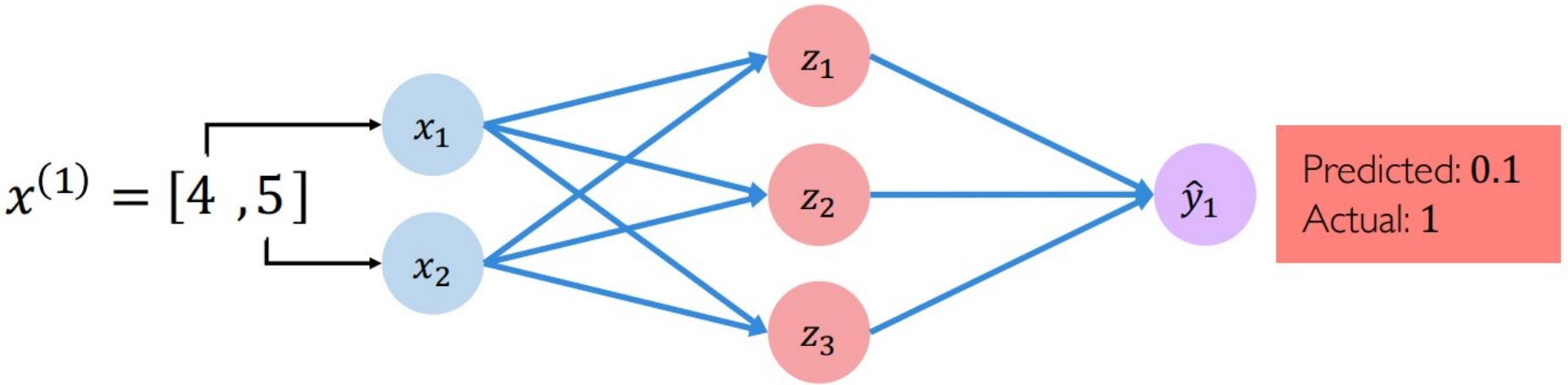


How do we decide which weights are optimal?

- A linear regressor's weights (coefficients) are calculated in closed form
- This can't be done if you have hidden layers and non-linear activations

How do we decide which weights are optimal?

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

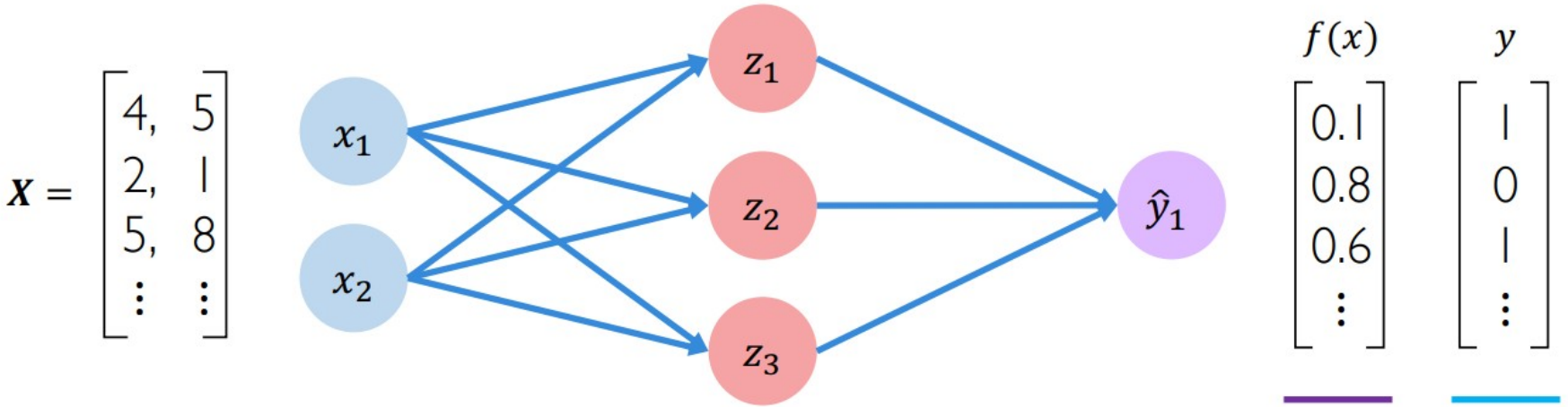


$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

How do we decide which weights are optimal?

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

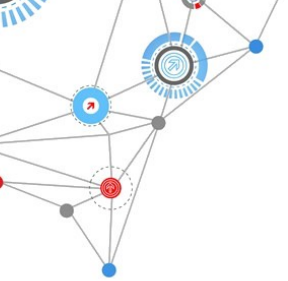


$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; W\right), y^{(i)}\right)$$

Predicted Actual

Also known as:

- Objective function
- Cost function
- Empirical Risk



Lower loss => better predictions

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

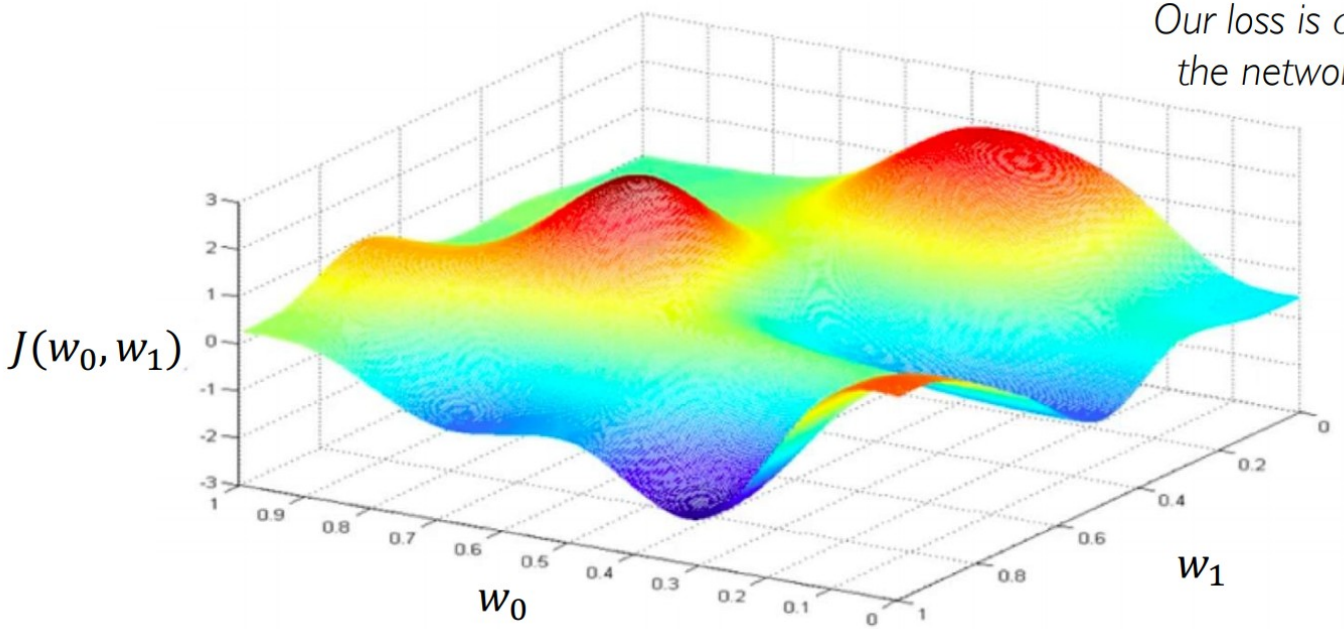
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

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

Minimizing loss

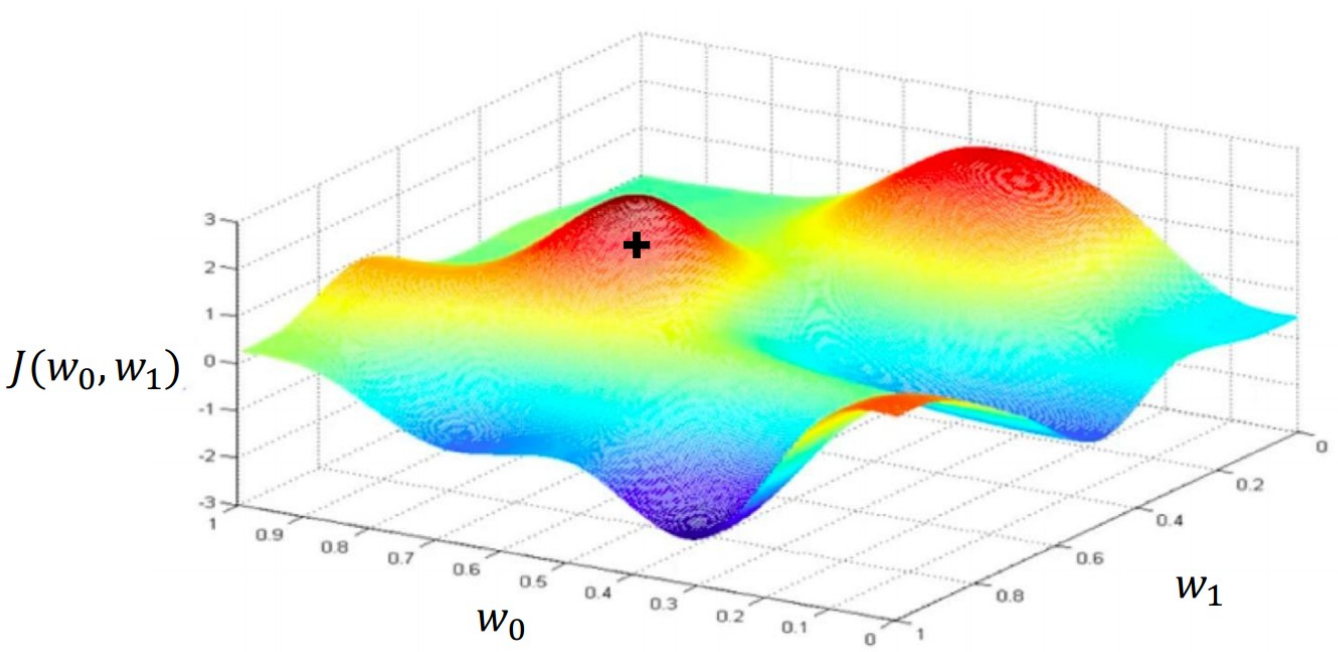
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

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



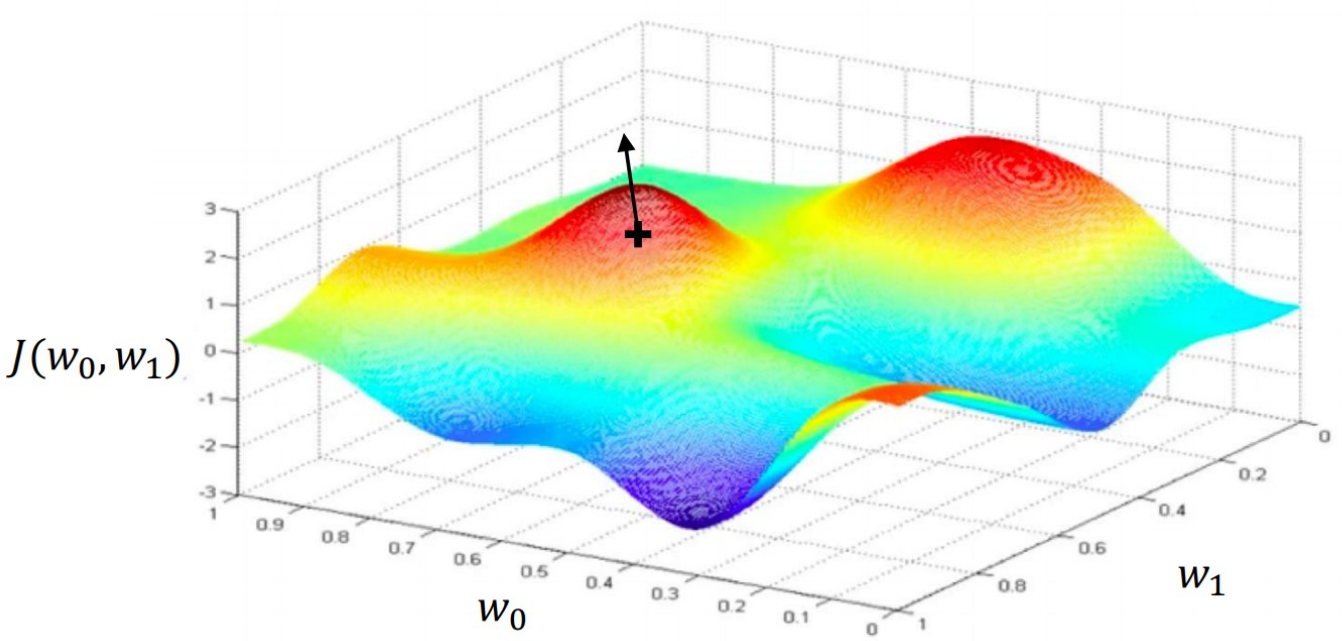
Minimizing loss

Randomly pick an initial (w_0, w_1)



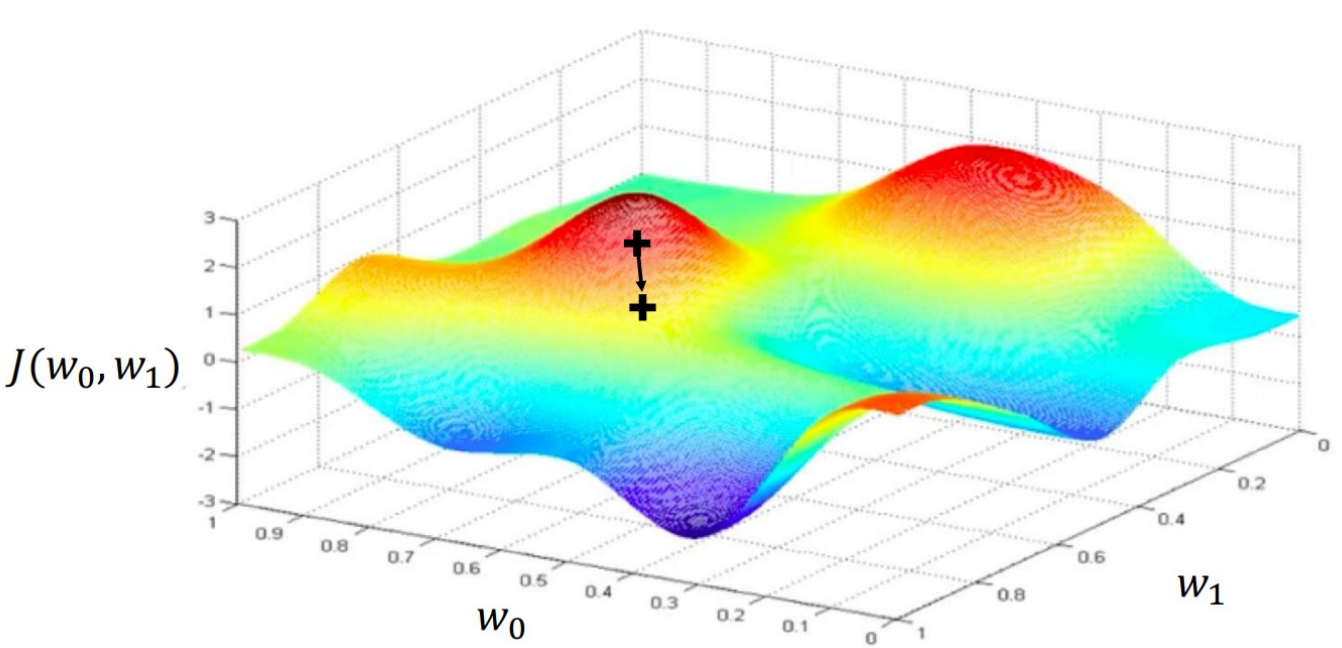
Minimizing loss

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



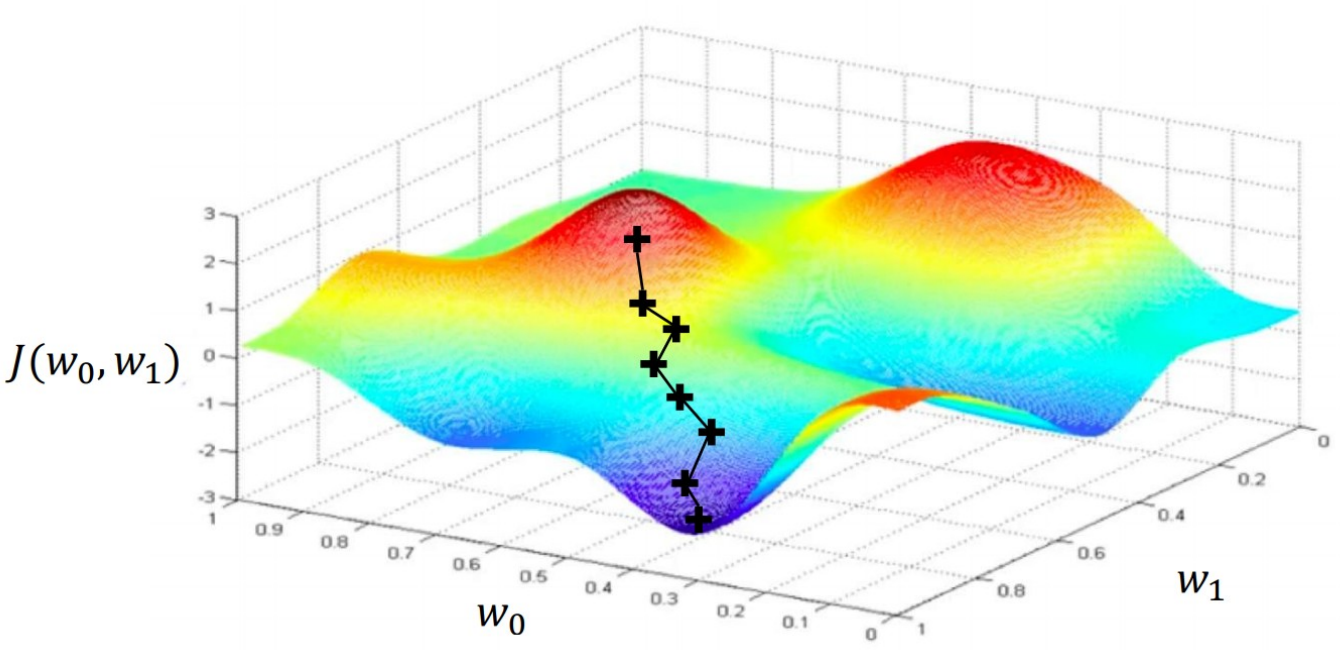
Minimizing loss

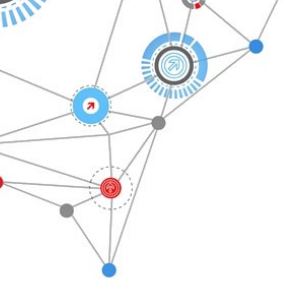
Take small step in opposite direction of gradient



Minimizing loss

Repeat until convergence





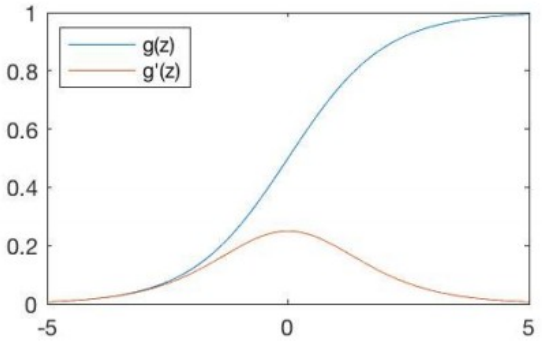
Gradient descent

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

Activation functions have to be differentiable!

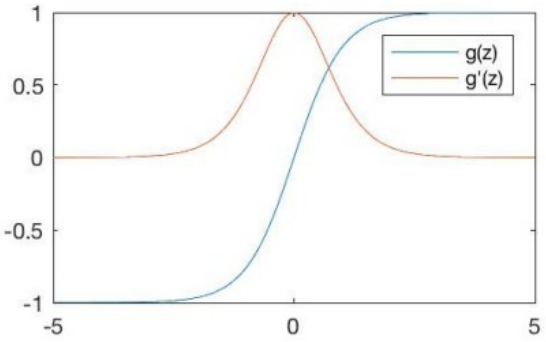
Sigmoid Function



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

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

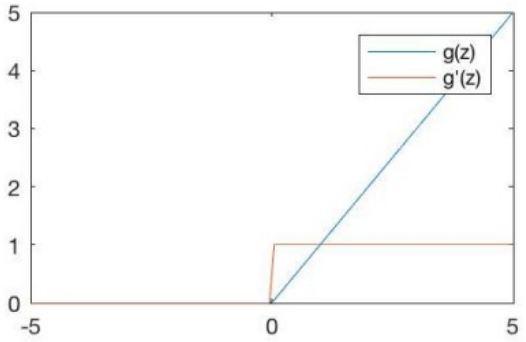
Hyperbolic Tangent



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

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

Rectified Linear Unit (ReLU)



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

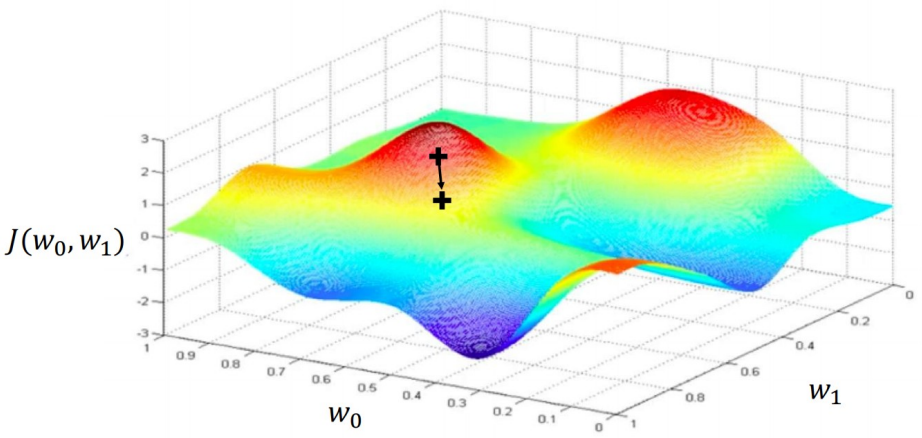
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

The learning rate η

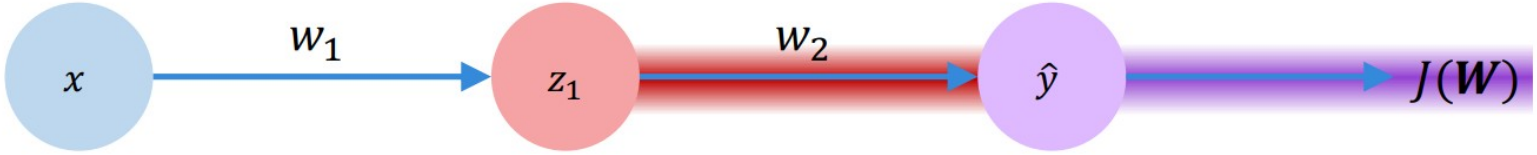
Algorithm

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

Take small step in opposite direction of gradient

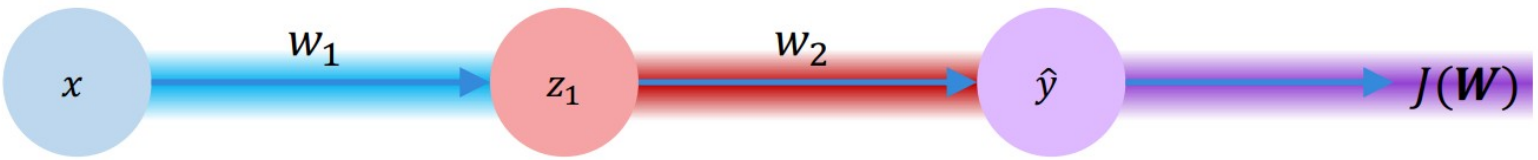


Backpropagation



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

Backpropagation

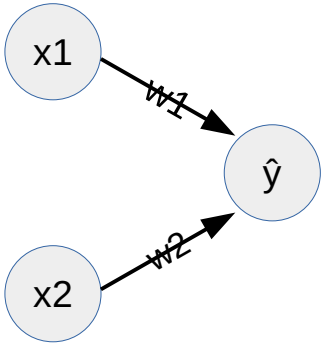


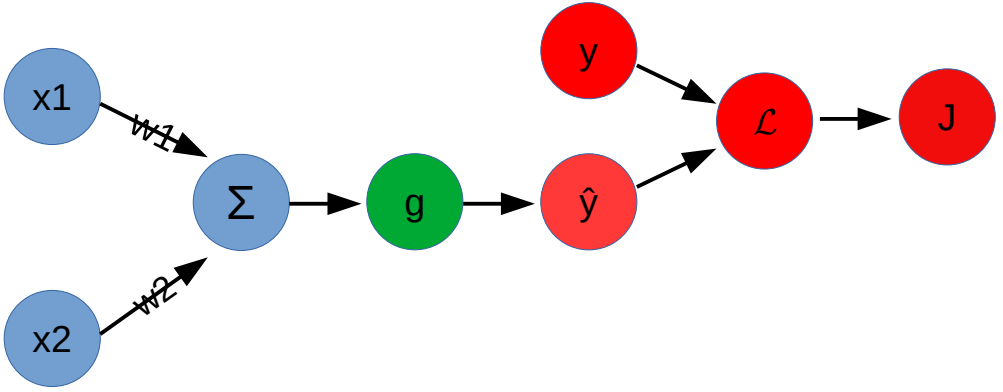
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Backpropagation example, step by step:
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

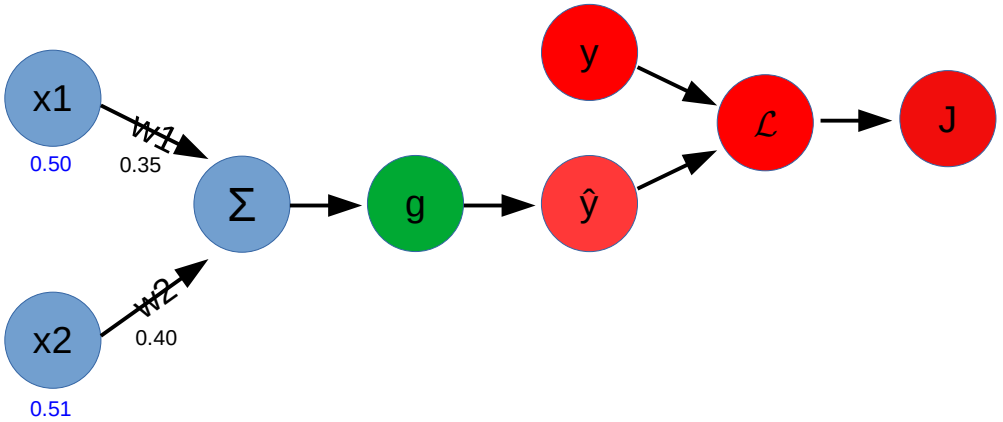
Practical example

- Simple network:
  - Two inputs [x1, x2]
  - Two weights [w1, w2]
  - No bias
  - Activation function g( )
  - One output ŷ
  - One label y
  - Loss function $\mathcal{L}$( )
  - Weight-dependent error J(W)





1. Forward pass

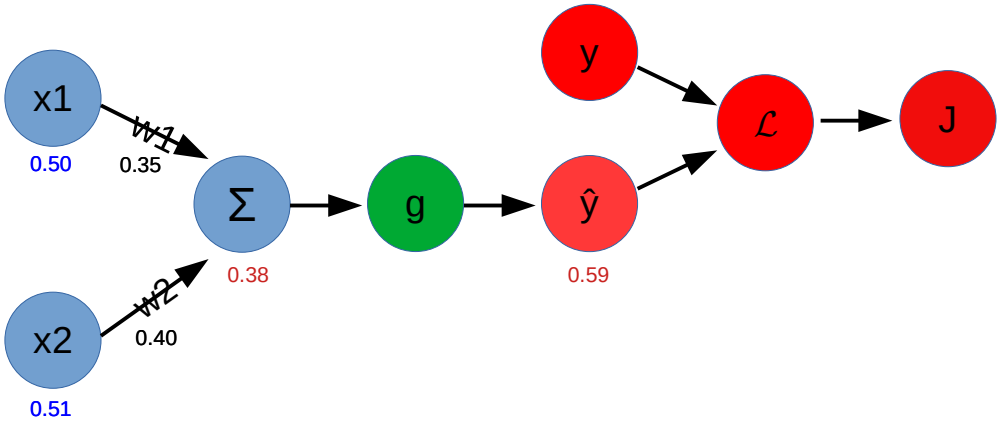


x1 = 0.50
x2 = 0.51

w1 = 0.35
w2 = 0.40

Σ = ?
ŷ = **g(Σ) =** $1/(1+e^{-\Sigma})$ = ?

1. Forward pass

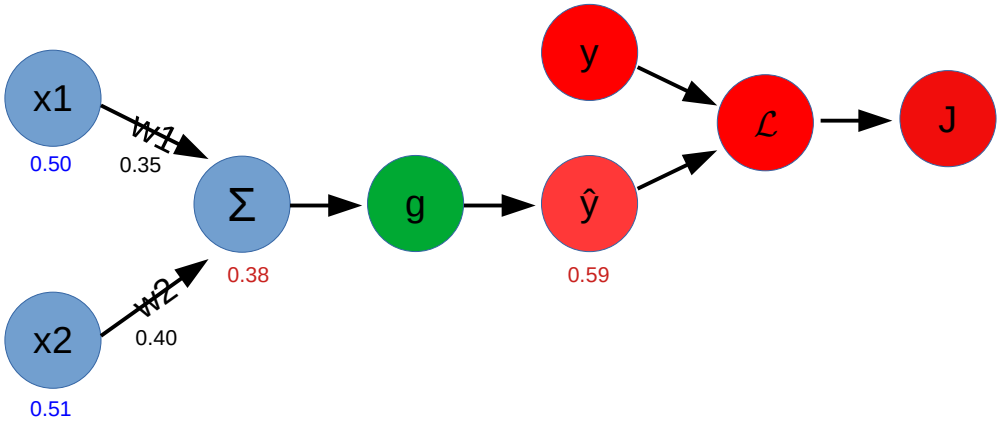


x1 = 0.50
x2 = 0.51

w1 = 0.35
w2 = 0.40

Σ = 0.35 * x1 + 0.4 * w2 = 0.38
ŷ = g(Σ) = $1/(1+e^{-\Sigma})$ = 0.59

2. Calculate Loss



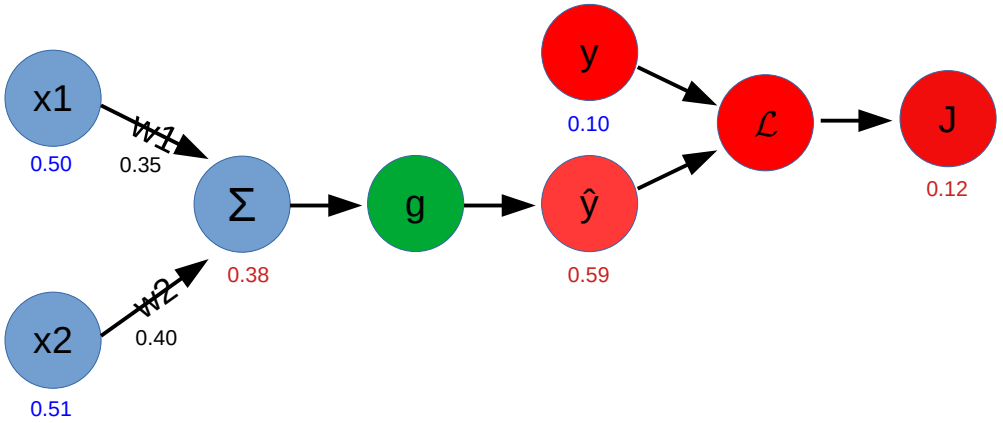
x1 = 0.50
x2 = 0.51

w1 = 0.35
w2 = 0.40

Σ = 0.35 * x1 + 0.4 * w2 = 0.38
ŷ = g(Σ) = $1/(1+e^{-\Sigma})$ = 0.59

Y = 0.10
J(W) = $\mathcal{L}(y, \hat{y}) = \frac{1}{2} * (y - \hat{y})^2$ = ?

2. Calculate Loss



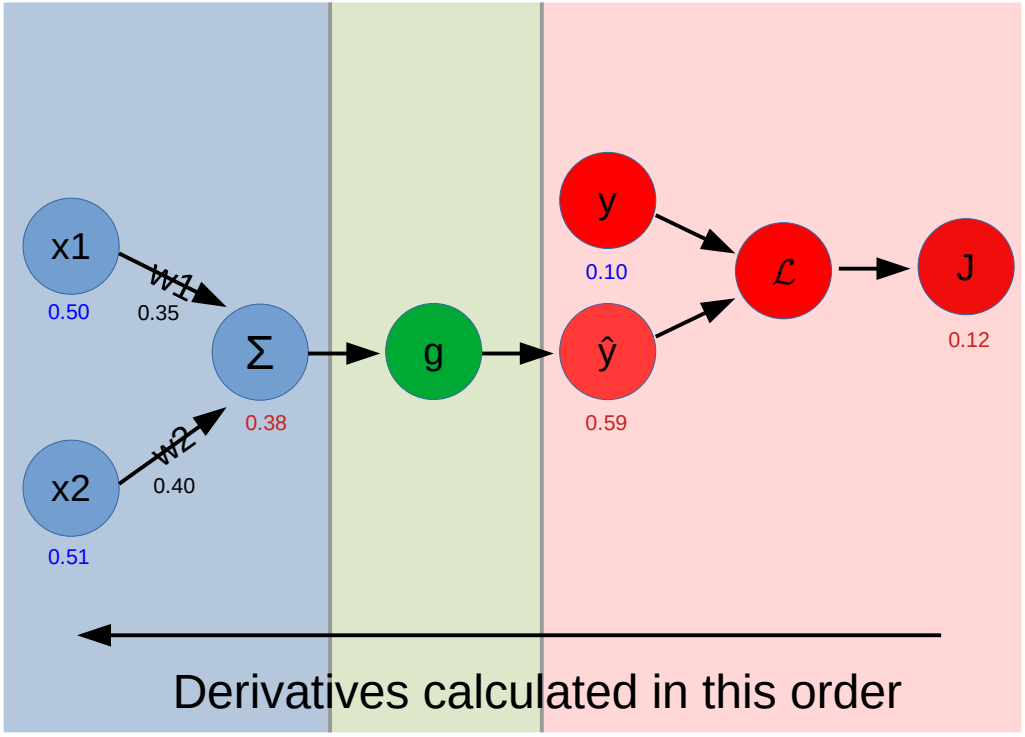
x1 = 0.50
x2 = 0.51

w1 = 0.35
w2 = 0.40

Σ = 0.35 * x1 + 0.4 * w2 = 0.38
ŷ = g(Σ) = $1/(1+e^{-\Sigma})$ = 0.59

Y = 0.10
J(W) = $\mathcal{L}(y, \hat{y}) = \frac{1}{2} * (y - \hat{y})^2$ = 0.12

3. Backpropagate the error



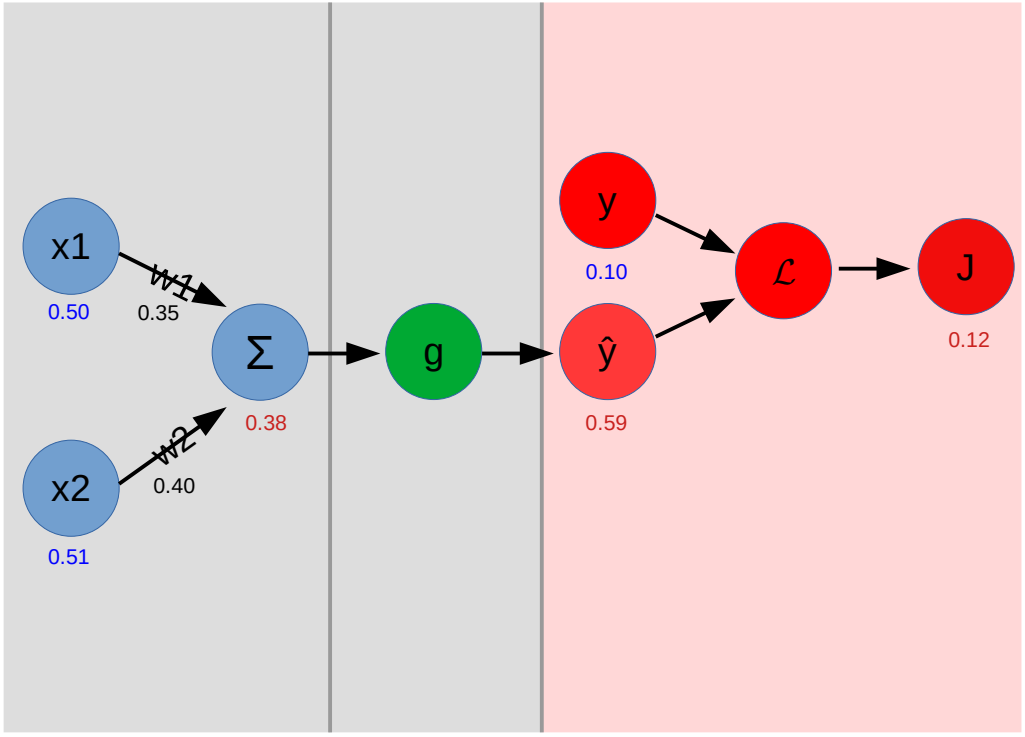
$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = \partial J(W) / \partial \hat{y}$ (loss)

$* \partial \hat{y} / \Sigma$ (activation)

$* \partial \Sigma / \partial w1$ (weight)

3. Backpropagate the error: loss



$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = \partial J(W) / \partial \hat{y}$ (loss)

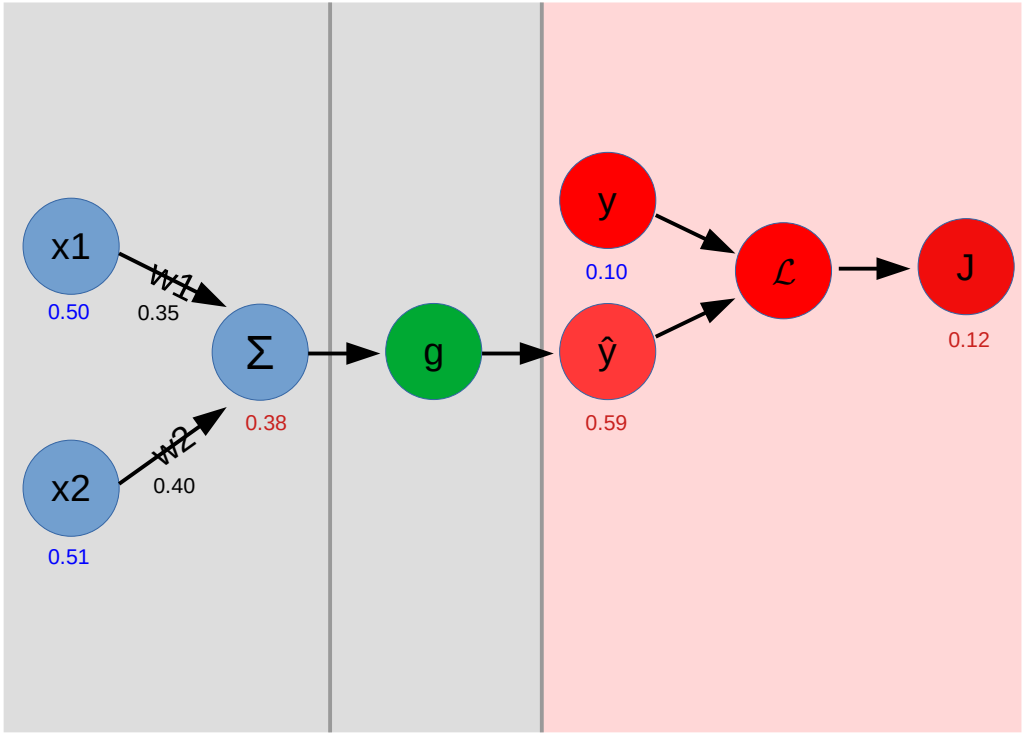
$* \partial \hat{y} / \Sigma$ (activation)

$* \partial \Sigma / \partial w1$ (weight)

$J(W) = \mathcal{L}(y, \hat{y}) = ½ * (y - \hat{y})^2$

$\partial J(W) / \partial \hat{y} = \partial \mathcal{L}(y, \hat{y}) / \partial \hat{y} = ?$

3. Backpropagate the error: loss



$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = \partial J(W) / \partial \hat{y}$ (loss)

$* \partial \hat{y} / \Sigma$ (activation)

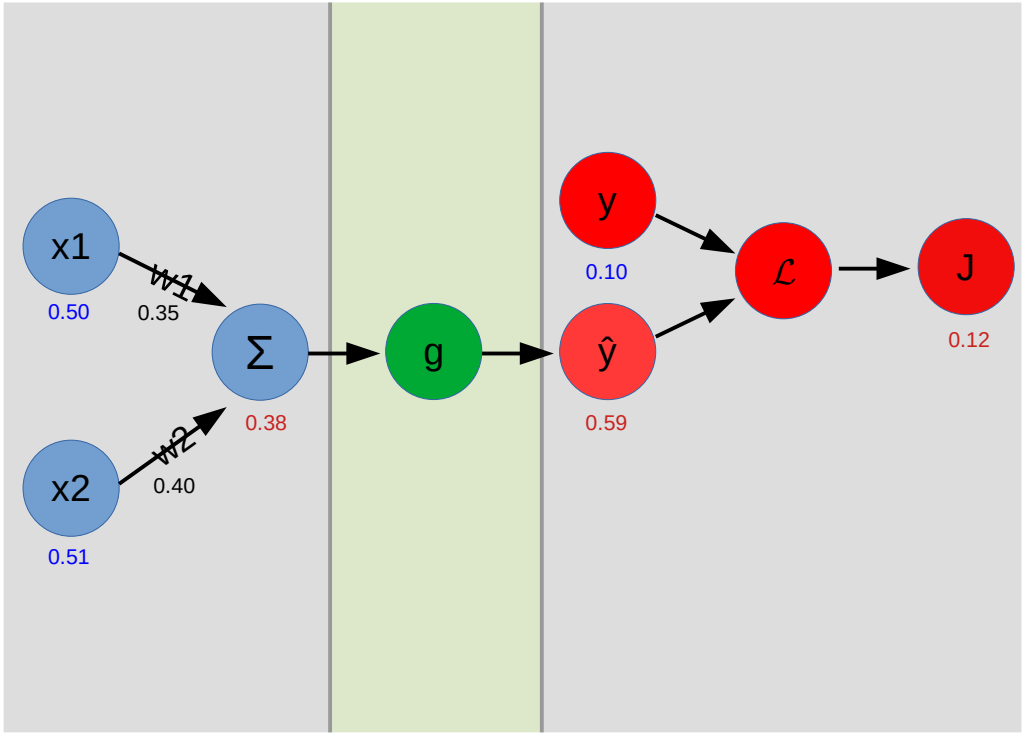
$* \partial \Sigma / \partial w1$ (weight)

$J(W) = \mathcal{L}(y, \hat{y}) = ½ * (y - \hat{y})^2$

$\partial J(W) / \partial \hat{y} = 2 * ½ * (y - \hat{y}) * -1$

$= -y + \hat{y} = -0.1 + 0.59 = 0.49$

3. Backpropagate the error: activation



$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

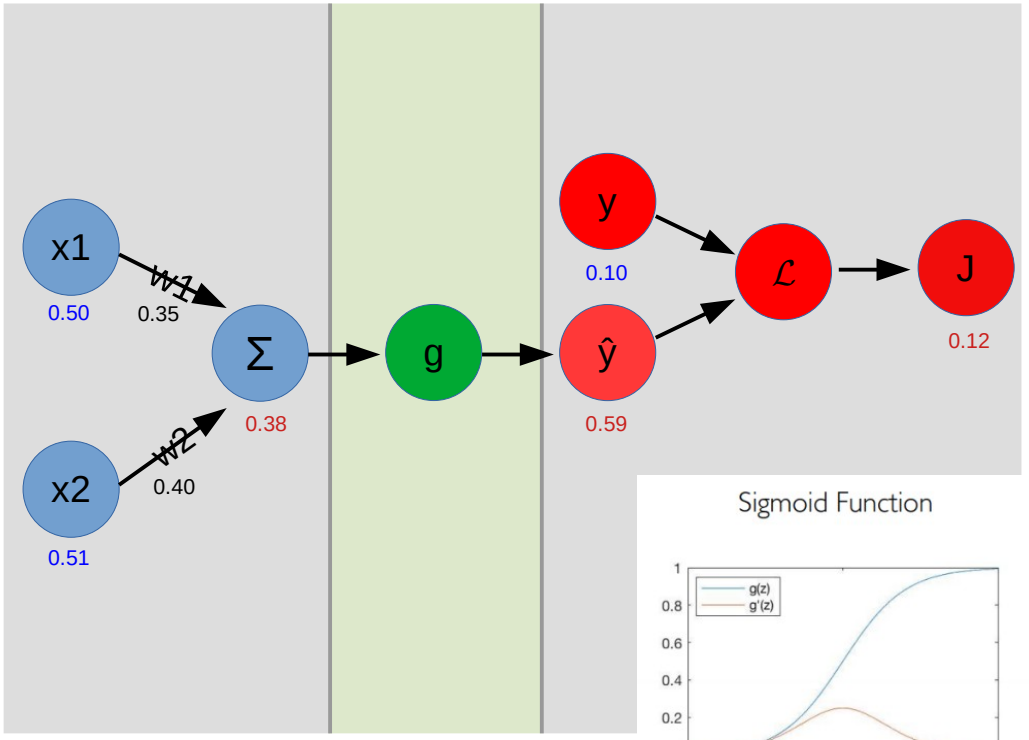
$* \partial \hat{y} / \partial \Sigma$ (activation)

$* \partial \Sigma / \partial w1$ (weight)

$\hat{\mathbf{y}} = g(\Sigma) = 1/(1+e^{-\Sigma})$

$\partial \hat{y} / \partial \Sigma = ?$

3. Backpropagate the error: activation



$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 =$ 0.49 (loss)

$* \partial \hat{y} / \partial \Sigma$ (activation)

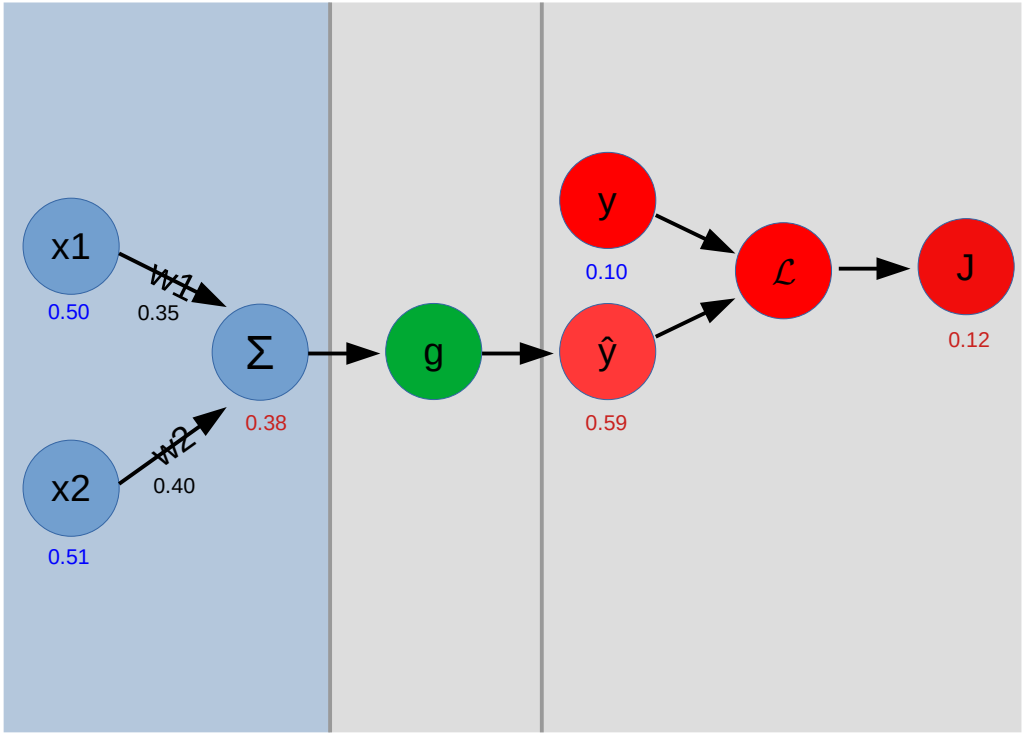
$* \partial \Sigma / \partial w1$ (weight)

$\hat{\mathbf{y}} = g(\Sigma) = 1/(1+e^{-\Sigma})$

$\partial \hat{y} / \partial \Sigma = 1/(1+e^{-\Sigma}) * (1 - 1/(1+e^{-\Sigma}))$

$= 0.24$

3. Backpropagate the error: weight



$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

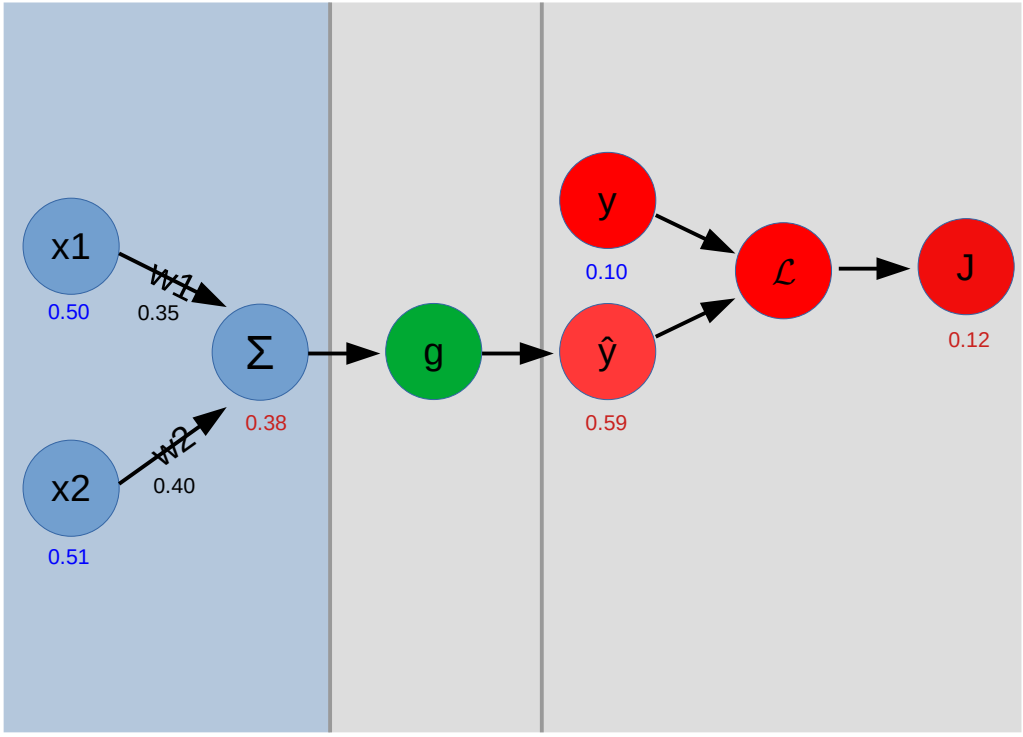
$* 0.24$ (activation)

$* \partial \Sigma / \partial w1$ (weight)

$\Sigma = X * W = x1 * w1 + x2 * w2$

$\partial \Sigma / \partial w1 = ?$

3. Backpropagate the error: weight



$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

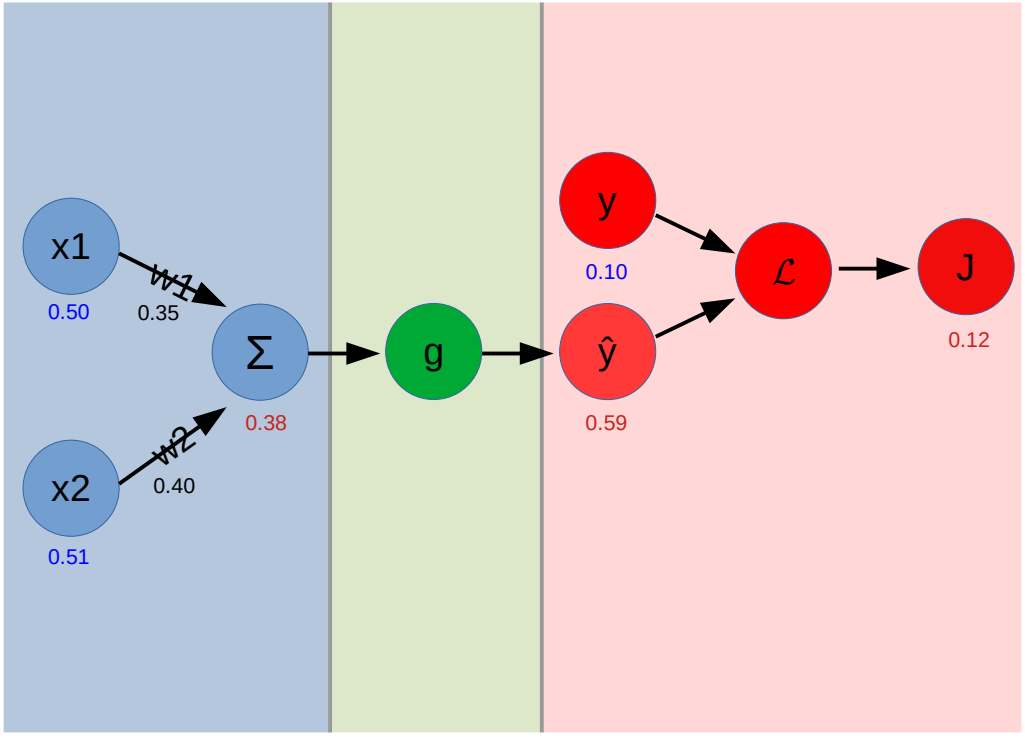
$* 0.24$ (activation)

$* \partial \Sigma / \partial w1$ (weight)

$\Sigma = X * W = x1 * w1 + x2 * w2$

$\partial \Sigma / \partial w1 = x1 + 0 = 0.5$

4. Weight update



J(W) = ℒ(g(X * W))

∂J(W)/∂w1 = 0.49 (loss)

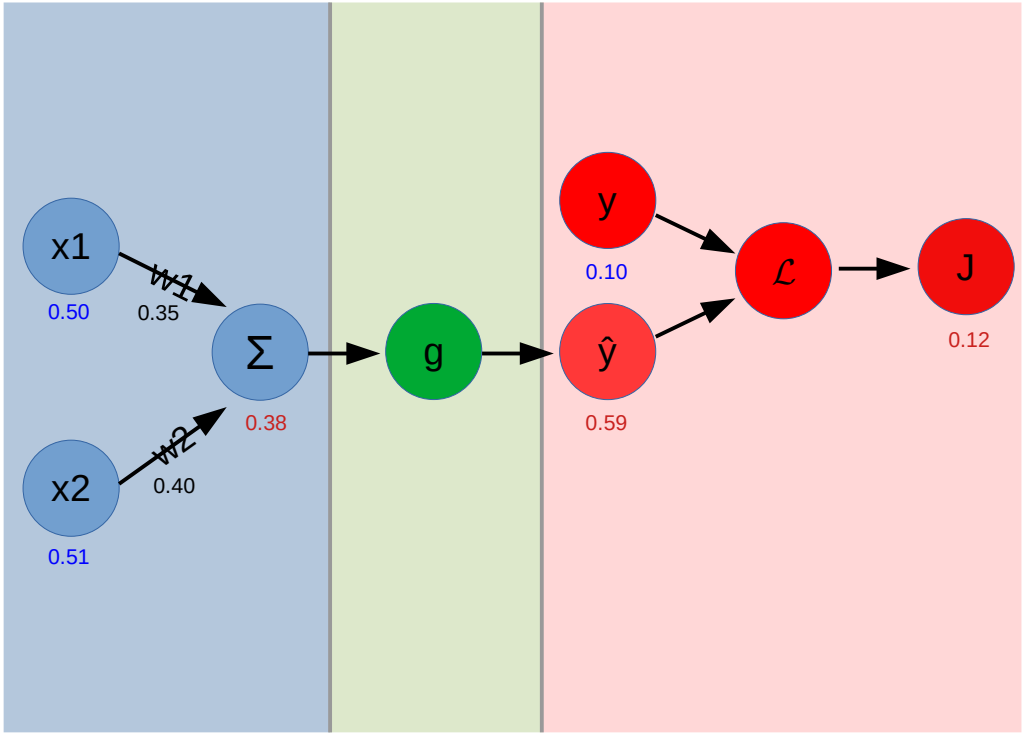
* 0.24 (activation)

* 0.5 (weight)

= 0.06 (gradient)

w1’ = ?

4. Weight update



$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

$* 0.24$ (activation)

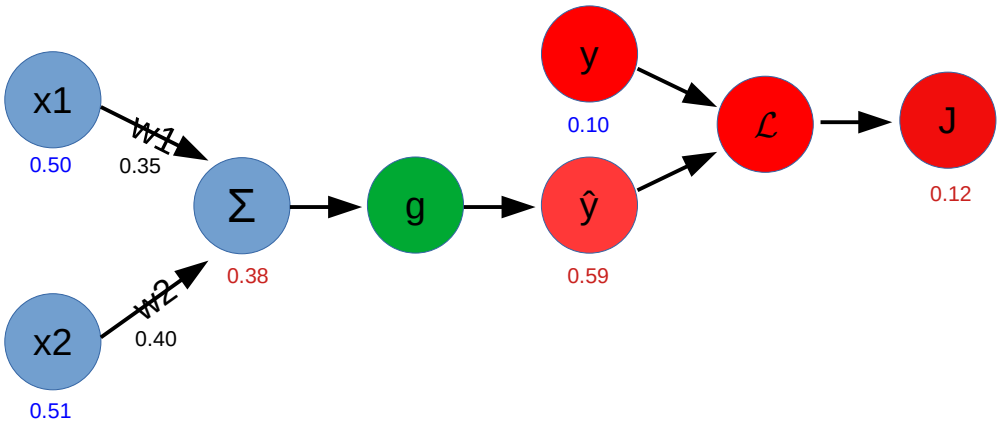
$* 0.5$ (weight)

$= 0.06$ (gradient)

$w1' = w1 - \eta * 0.06 = 0.35 - 0.06 = 0.29$

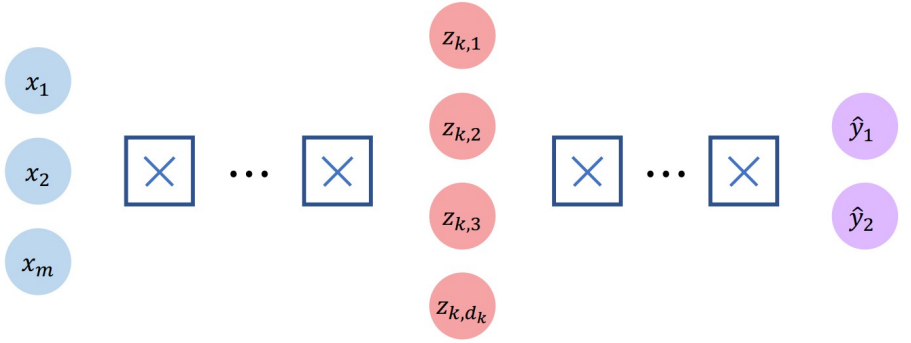
Exercise

- Can you calculate the weight update for w2? How many new gradients do you need to calculate?
- What is the new predicted output? Has the error gone down?
- What if I had another layer before this one?



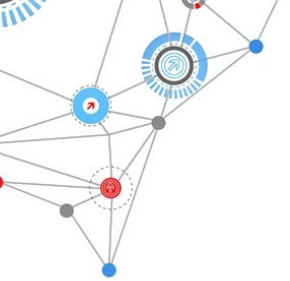
Gradient vanishing

- What happens if we backpropagate on a network with many (N > k > 1) hidden layers?



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

$$\partial E/\partial W1 = \partial J/\partial \hat{Y} * \partial \hat{Y}/\partial \Sigma_N * \partial \Sigma_N/\partial W_N * \ldots * \partial Z_k/\partial \Sigma_k * \partial \Sigma_k/\partial W_k * \ldots \partial Z_1/\partial \Sigma_1 * \partial \Sigma_1/\partial X_1$$



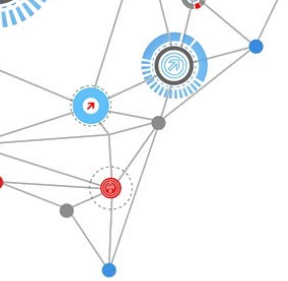
Gradient vanishing

- These are all “zero-point-somethings” multiplied by each other
- So the gradient becomes smaller by orders of magnitudes as we go back more and more layers until it’s so small that the network is stuck

$$\partial E/\partial W1 = \partial J/\partial \hat{Y} * \partial \hat{Y}/\partial \Sigma_N * \partial \Sigma_N/\partial W_N * \ldots * \partial Z_k/\partial \Sigma_k * \partial \Sigma_k/\partial W_k * \ldots \partial Z_1/\partial \Sigma_1 * \partial \Sigma_1/\partial X_1 = O(10^{-N})$$

initial w1 = 0.5
optimal w1 = -0.2
5-layer gradient ~0.00001

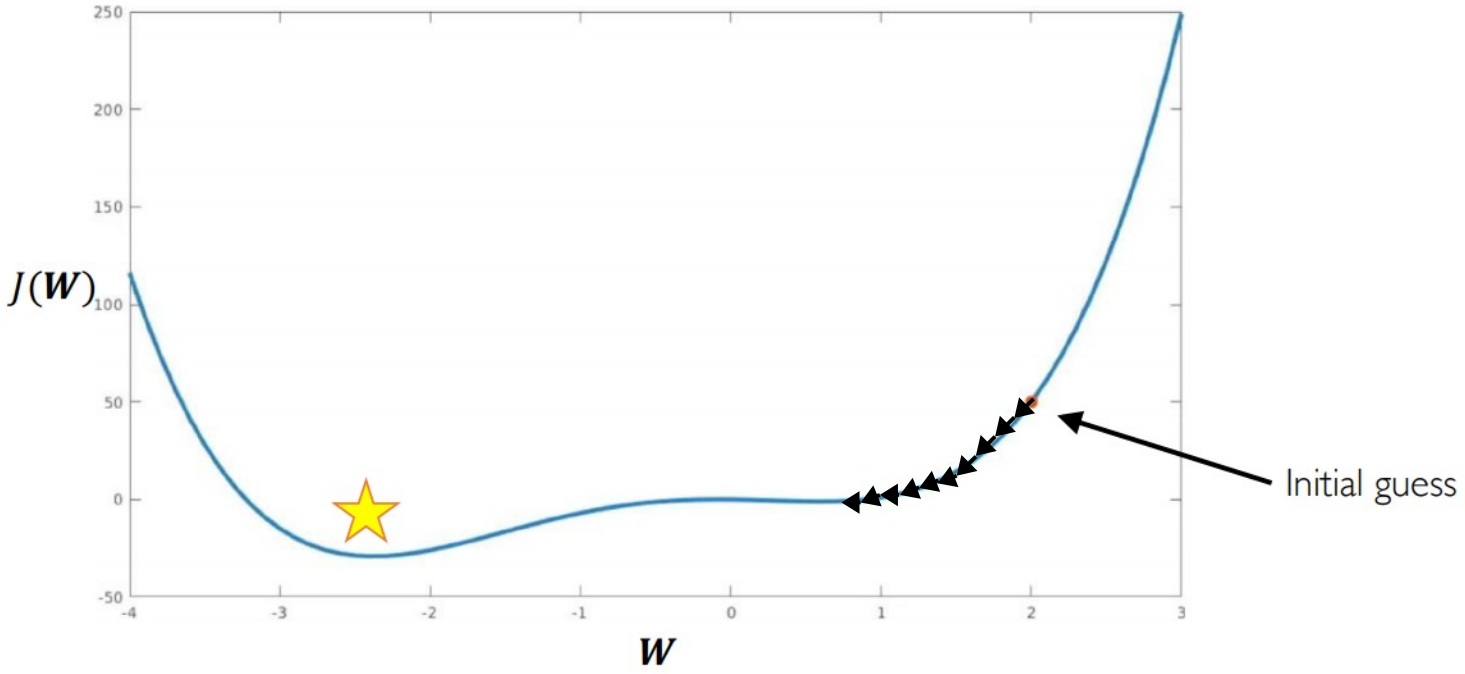
How many iterations do we need to get from 0.5 to -0.2?



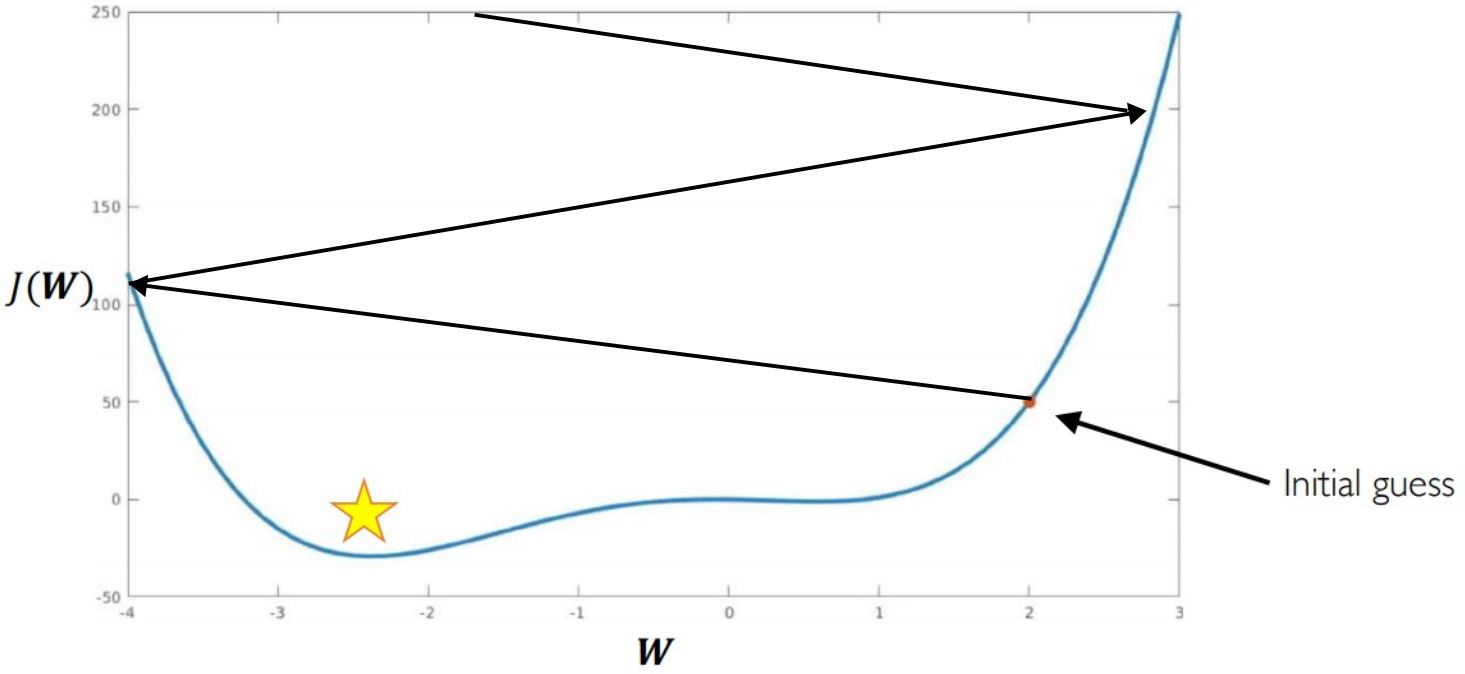
Other issues

- Finding the right learning rate (η, *eta*) can be tricky
- Computing the loss over the full set at every iteration is slow

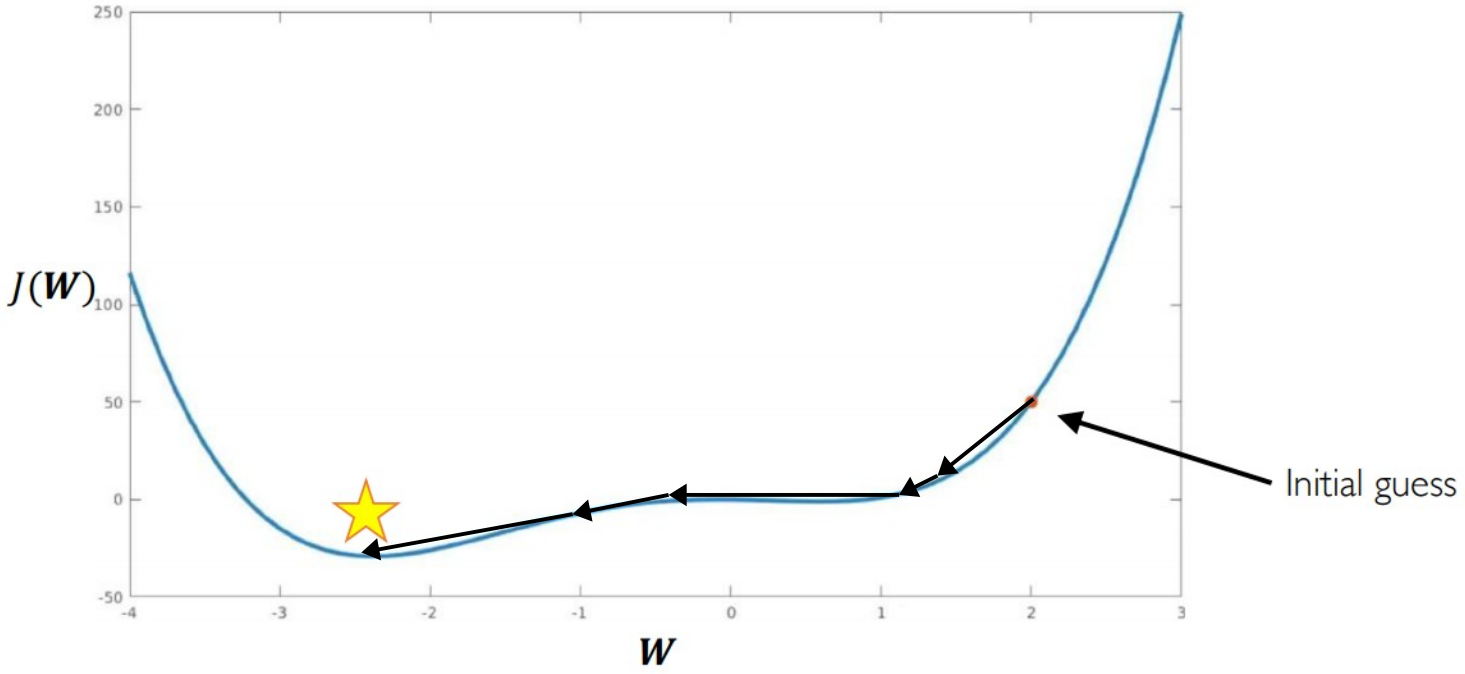
Too small: slow, gets stuck



Too large: fast, might explode



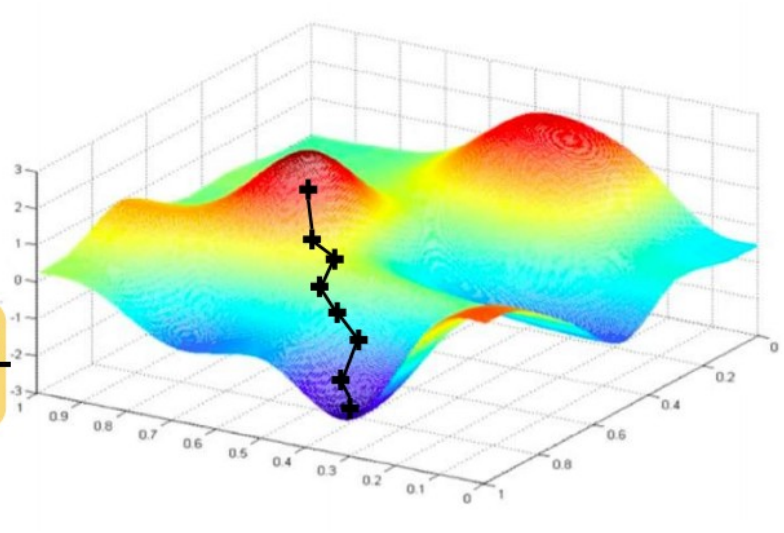
Adaptive learning rate



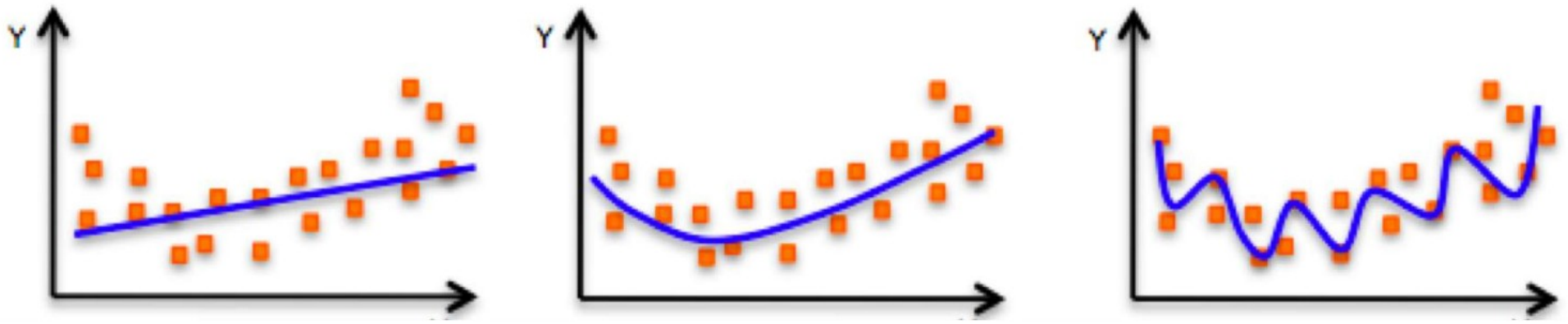
Stochastic gradient descent

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, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Return weights



Other issues: under/overfitting



Underfitting
Model does not have capacity to fully learn the data

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

Regularization to limit overfitting

L1 (Lasso):

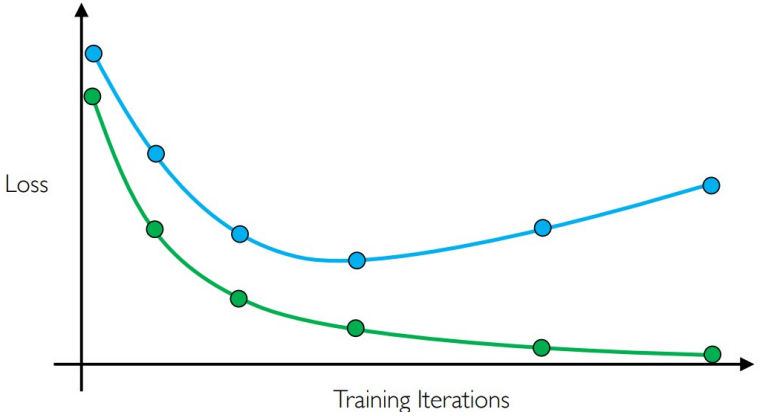
$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} |\theta_i|$$

L2 (Ridge):

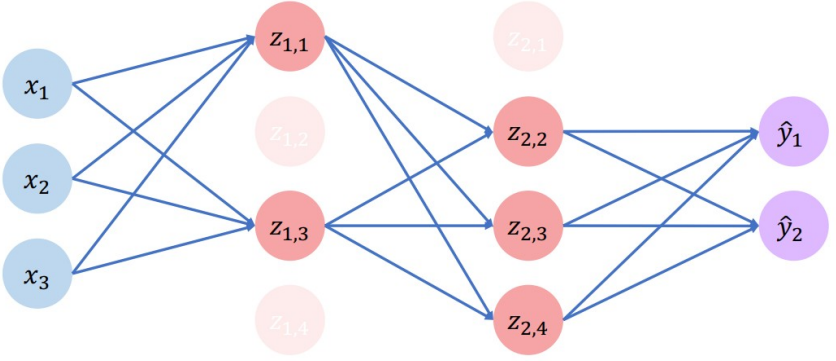
$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

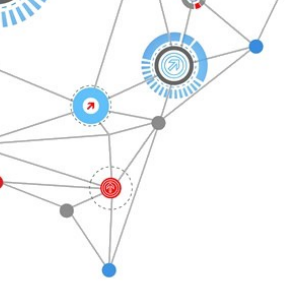
Regularization to limit overfitting

Early stopping



Dropout





Before we continue

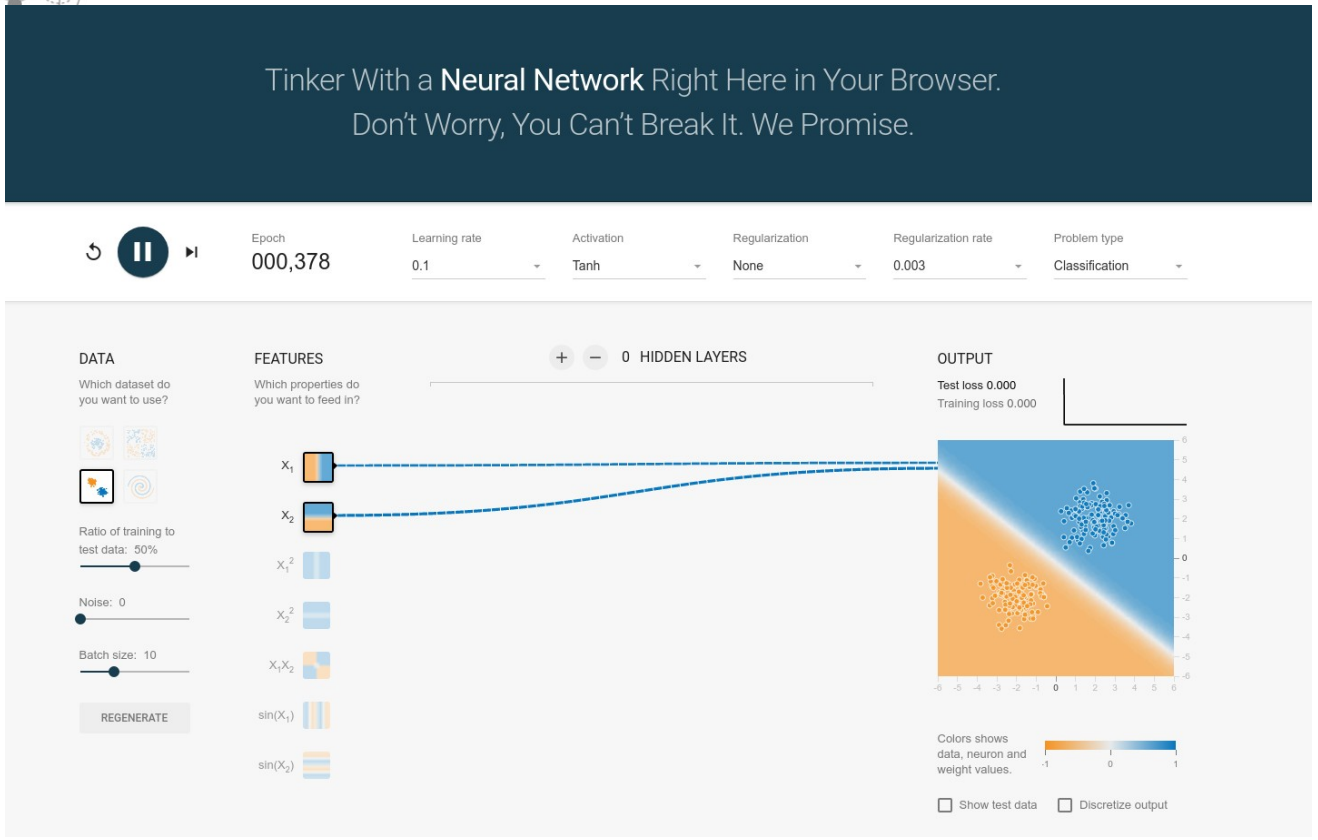
- Go to: https://bitbucket.org/clami66/deep_learning_course
- Clone the repository with git

  (git clone git@bitbucket.org:clami66/deep_learning_course.git

  git clone https://clami66@bitbucket.org/clami66/deep_learning_course.git)
- Or use the “Download Repository” option

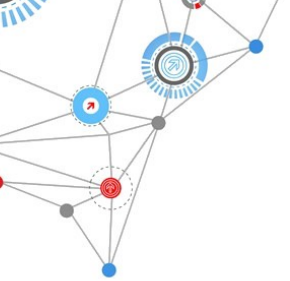
  (left menu → Downloads → Download Repository)
- You will get a Jupyter Notebook as well as the latest version of the slides

https://playground.tensorflow.org



Let's review:

- Learning rate
- Minibatches
- Regularization
- Under/Overfitting



Visual cortex studies (‘50s)

- Hubel & Wiesel (1959) hypothesized that the visual cortex is made of two types of cells:
  - Simple cells recognize fixed patterns such as edges
  - Complex cells can do the same task, but in a spatially invariant fashion
  - A model was proposed where simplex and complex cells where combined in a cascade

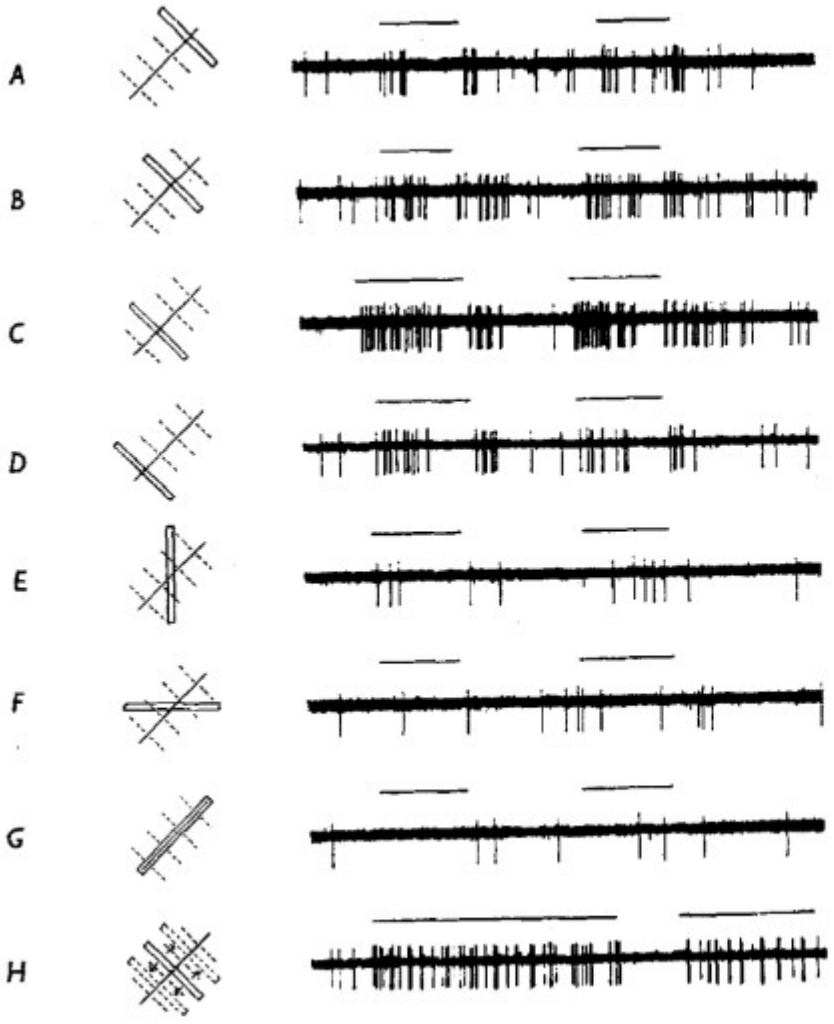


Fig. 4. Responses of a cell with a complex field to stimulation of the left (contralateral) eye with a slit ⅛ × 2½°. Receptive field was in the area centralis and was about 2 × 3° in size. A–D, ⅛° wide slit oriented parallel to receptive field axis. E–G, slit oriented at 45 and 90° to receptive-field axis. H, slit oriented as in A–D, is on throughout the record and is moved rapidly from side to side where indicated by upper beam. Responses from left eye slightly more marked than those from right (Group 3, see Part II). Time 1 sec.

Wiesel, Hubel, Torsten (2005)

Neocognitron (1979)

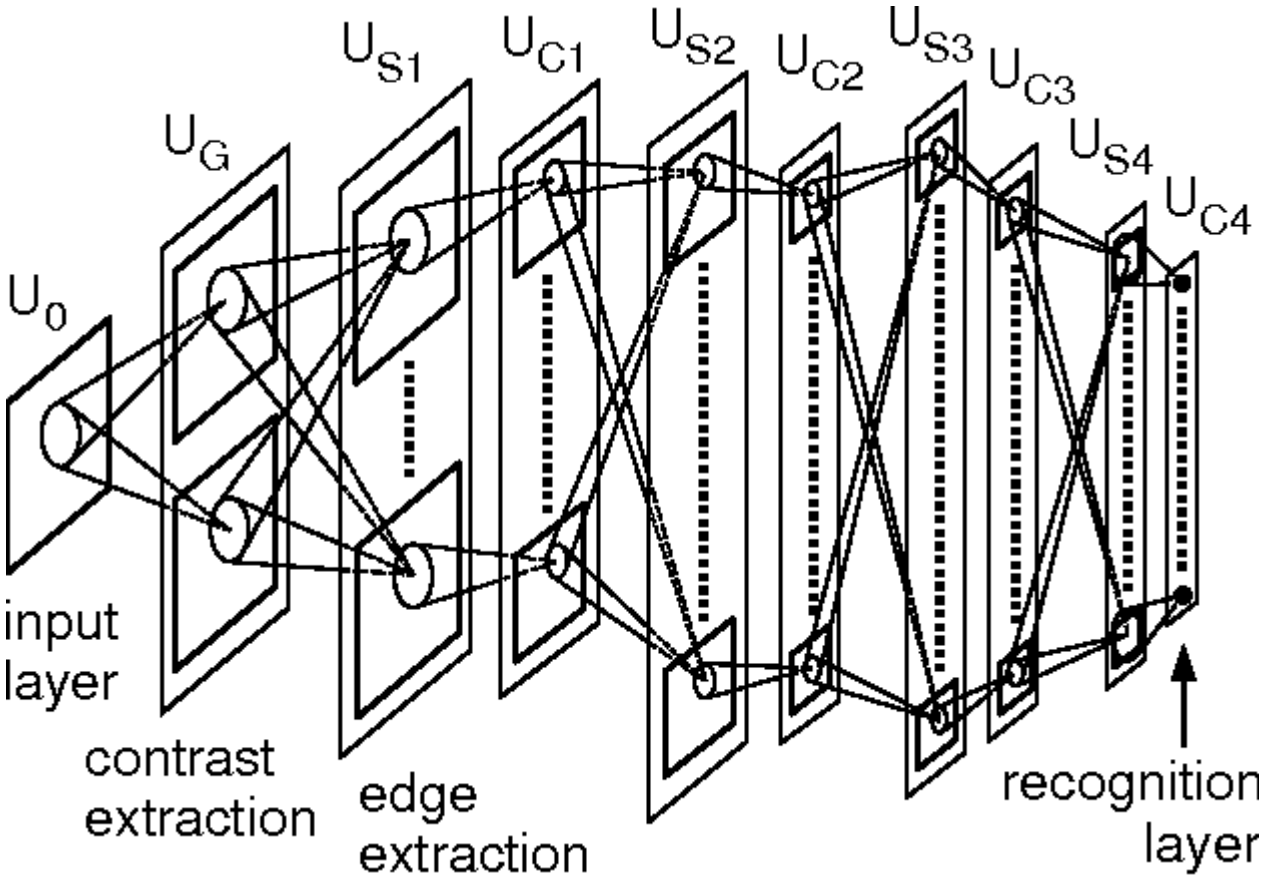
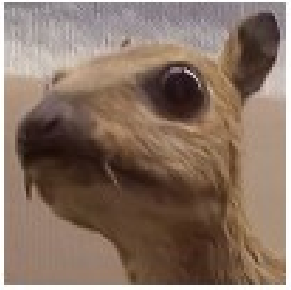


Image adapted from Fukushima, 1979

Image processing: kernels

Input image



Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map

Image processing: kernels

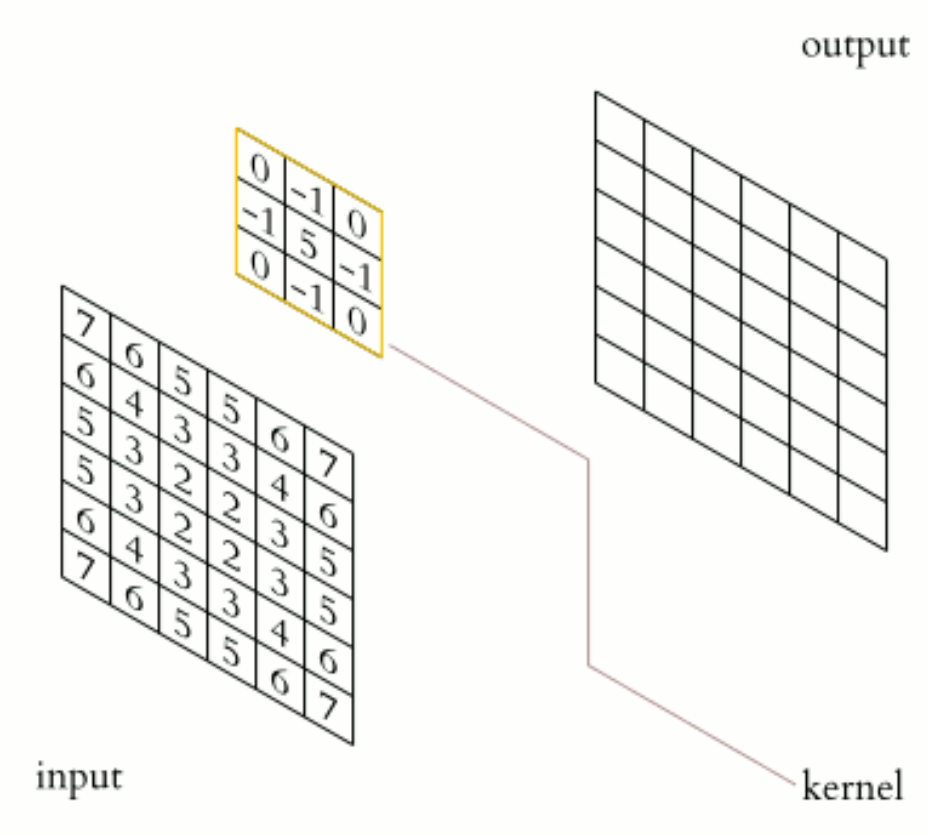
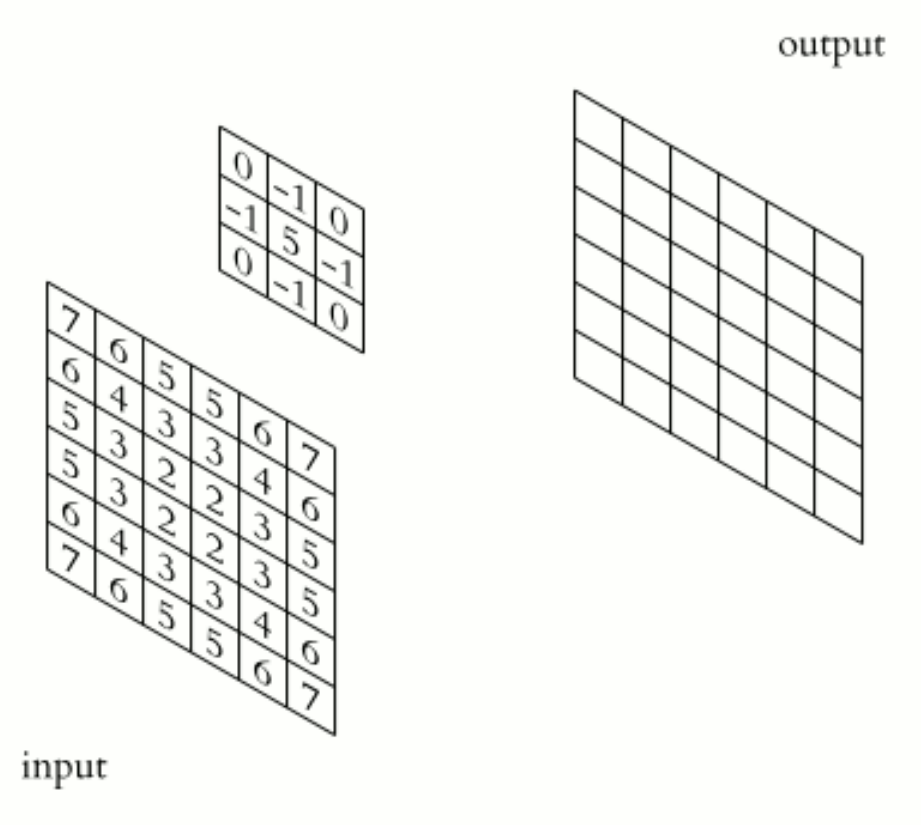
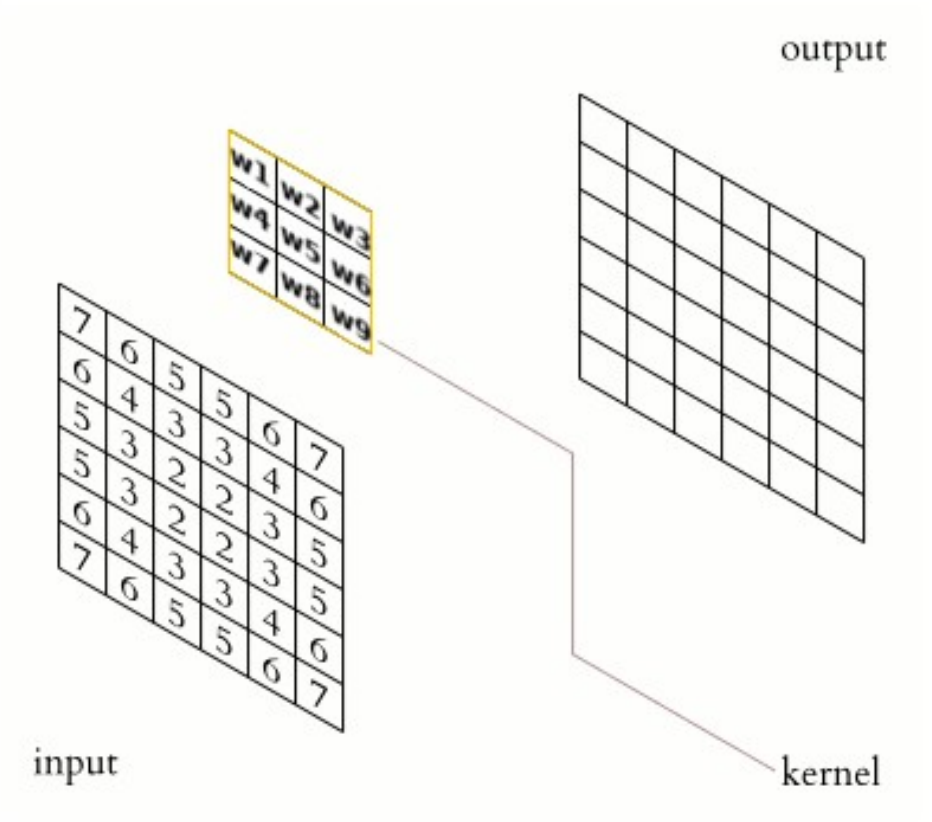


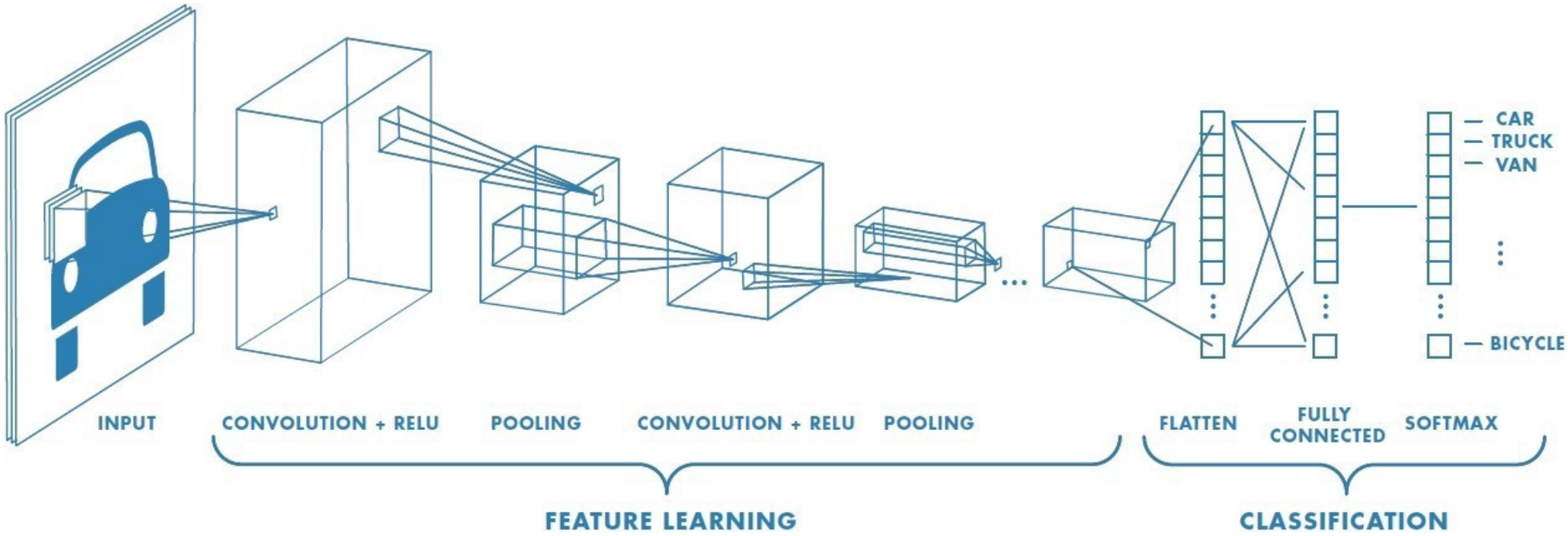
Image processing: kernels



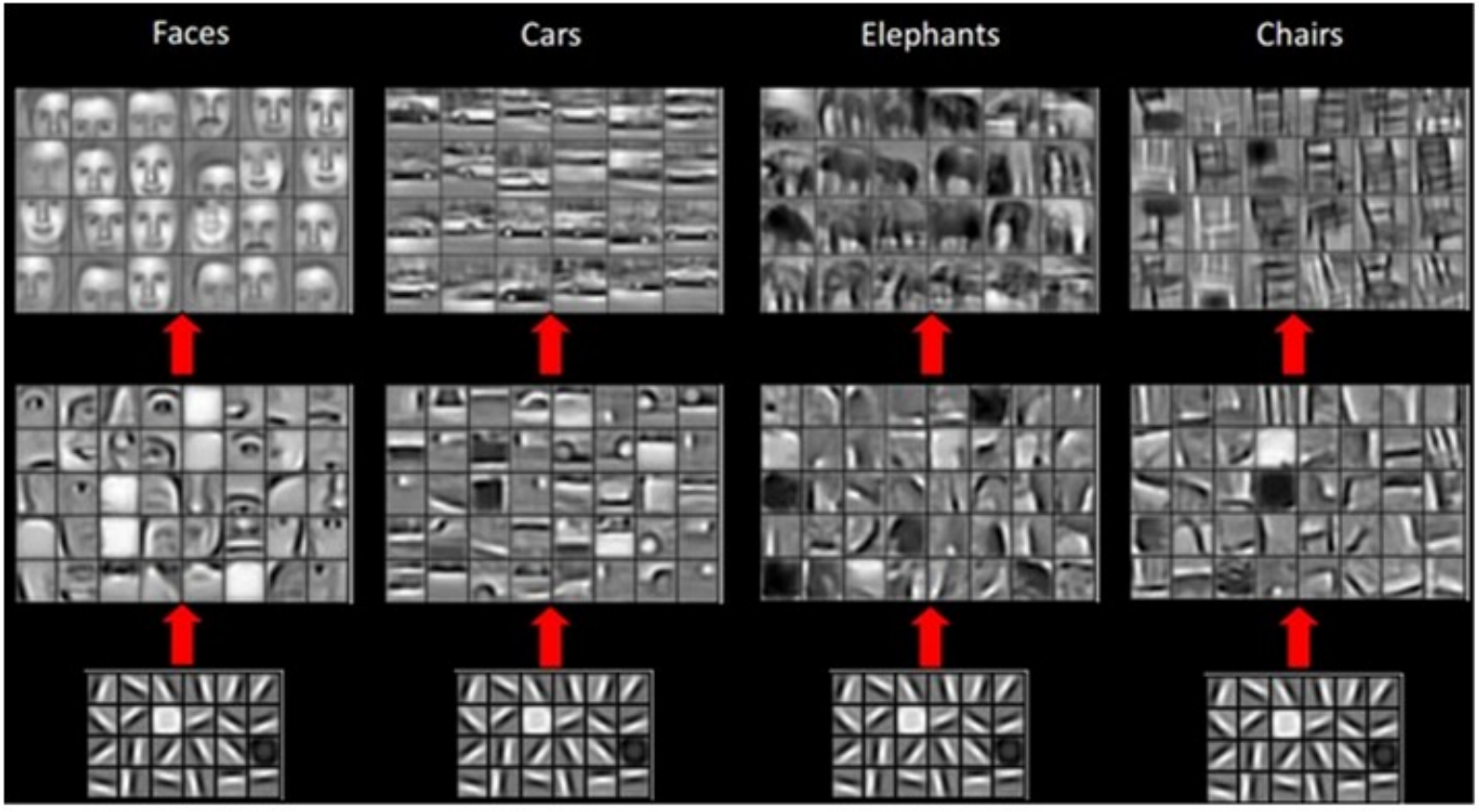
Convnets apply adaptive filters to the inputs



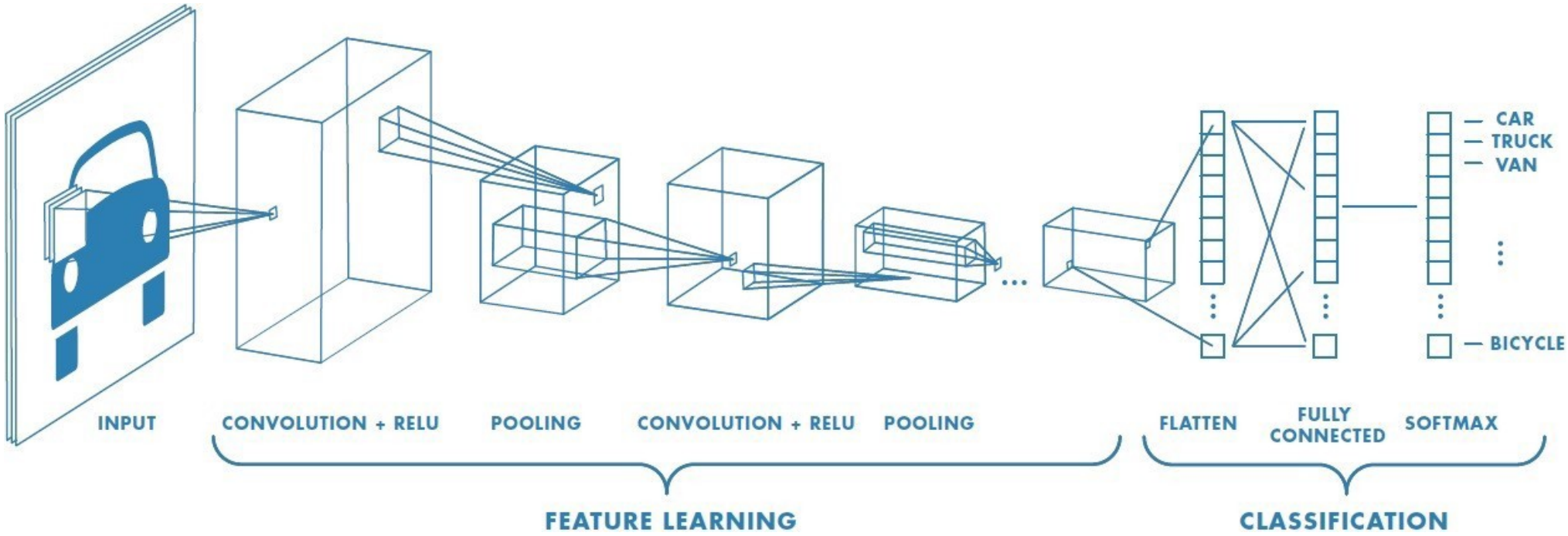
Convolutional Neural Networks (1995)



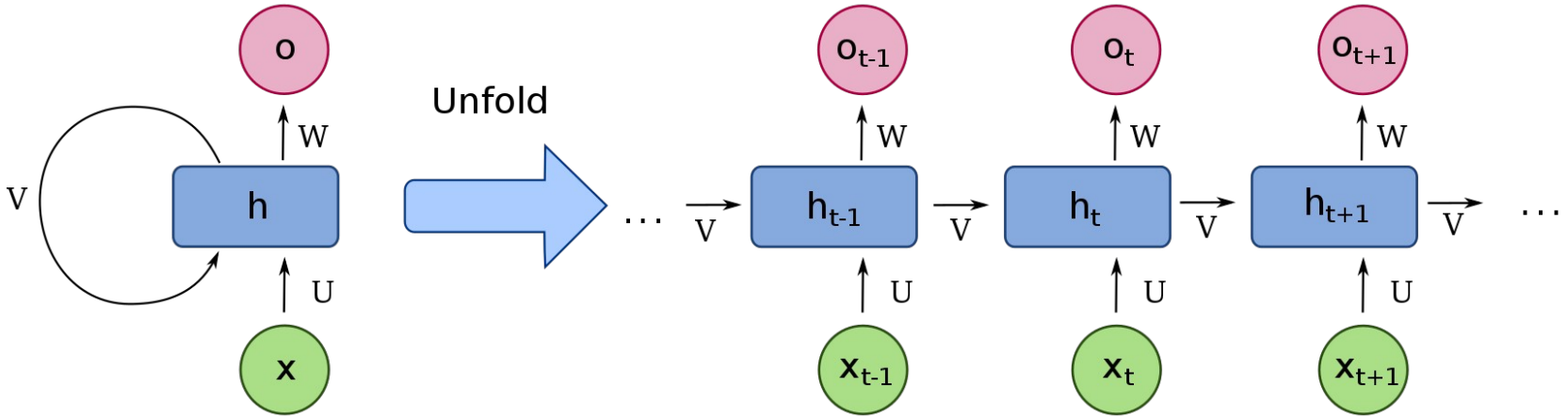
Convnets apply adaptive filters to inputs



Convolutional Neural Networks (1995)

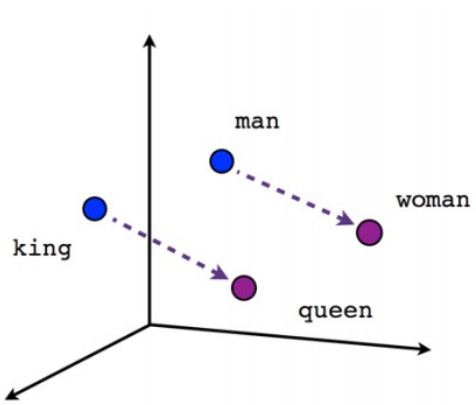


Recurrent Neural Networks

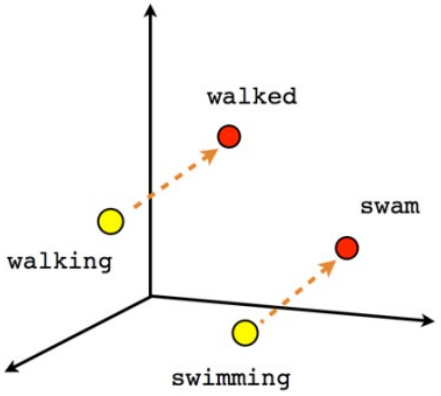


- Connections between nodes form a directed graph along a temporal sequence
- Actually always the same neurons repeated over and over (weight sharing)
- RNNs work very well when your data is like a time series (audio, text, DNA bases...)

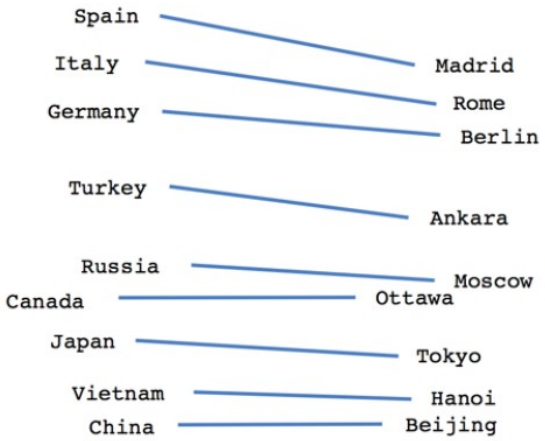
Word embeddings (word2vec)



Male-Female



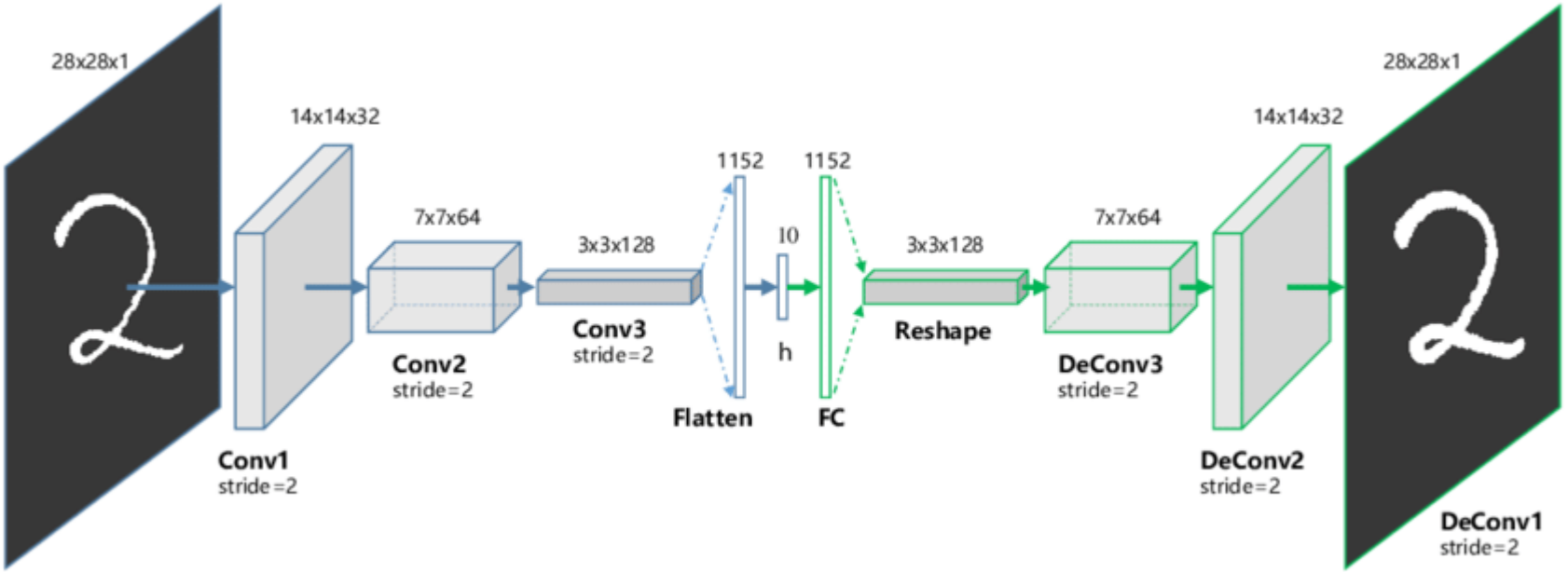
Verb tense



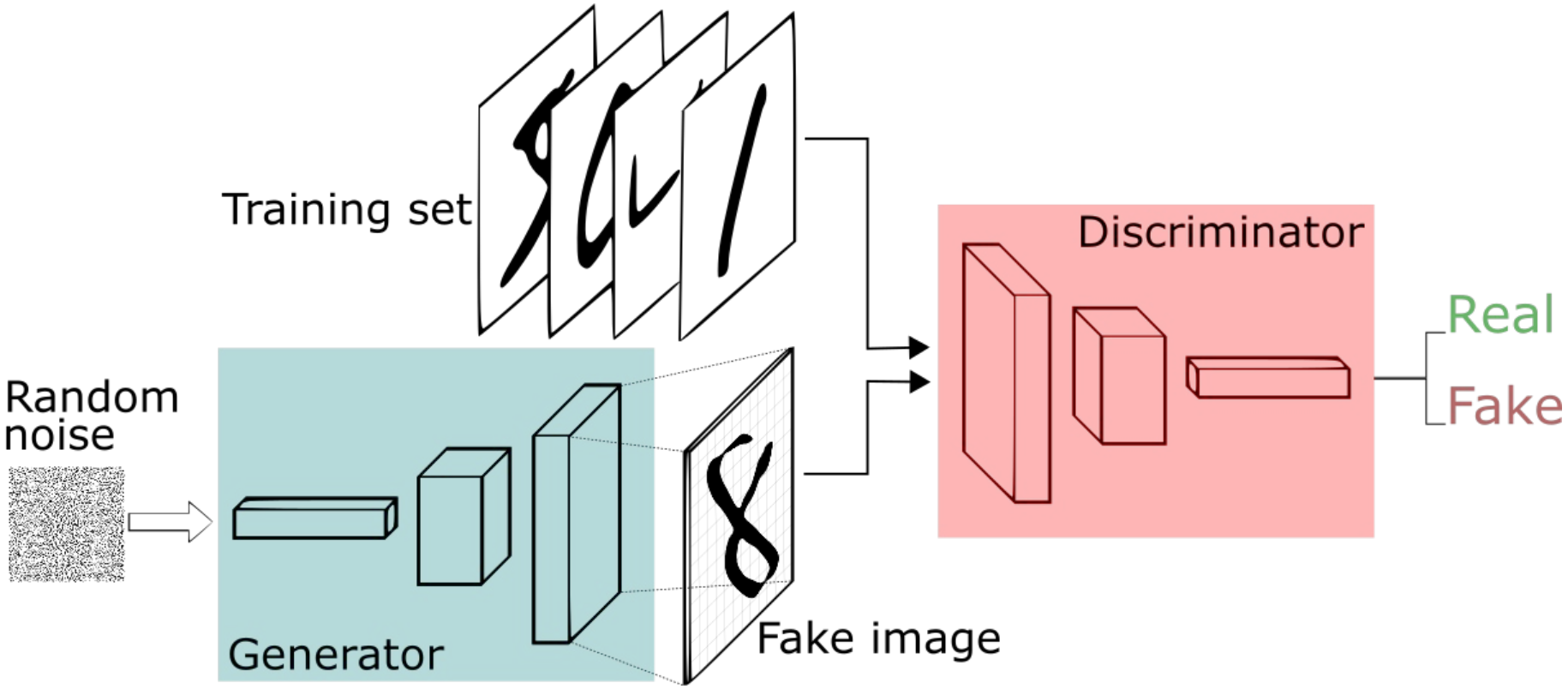
Country-Capital

- Simple two-layer NN that assigns to categorical inputs a floating point vector
- Inputs that are closely associated will be close in the output space
- It can also find analogies

Autoencoders



Generative Adversarial Network (GAN)



Fantastic beasts and where to find them (2016)

Is this the real life? (2019)

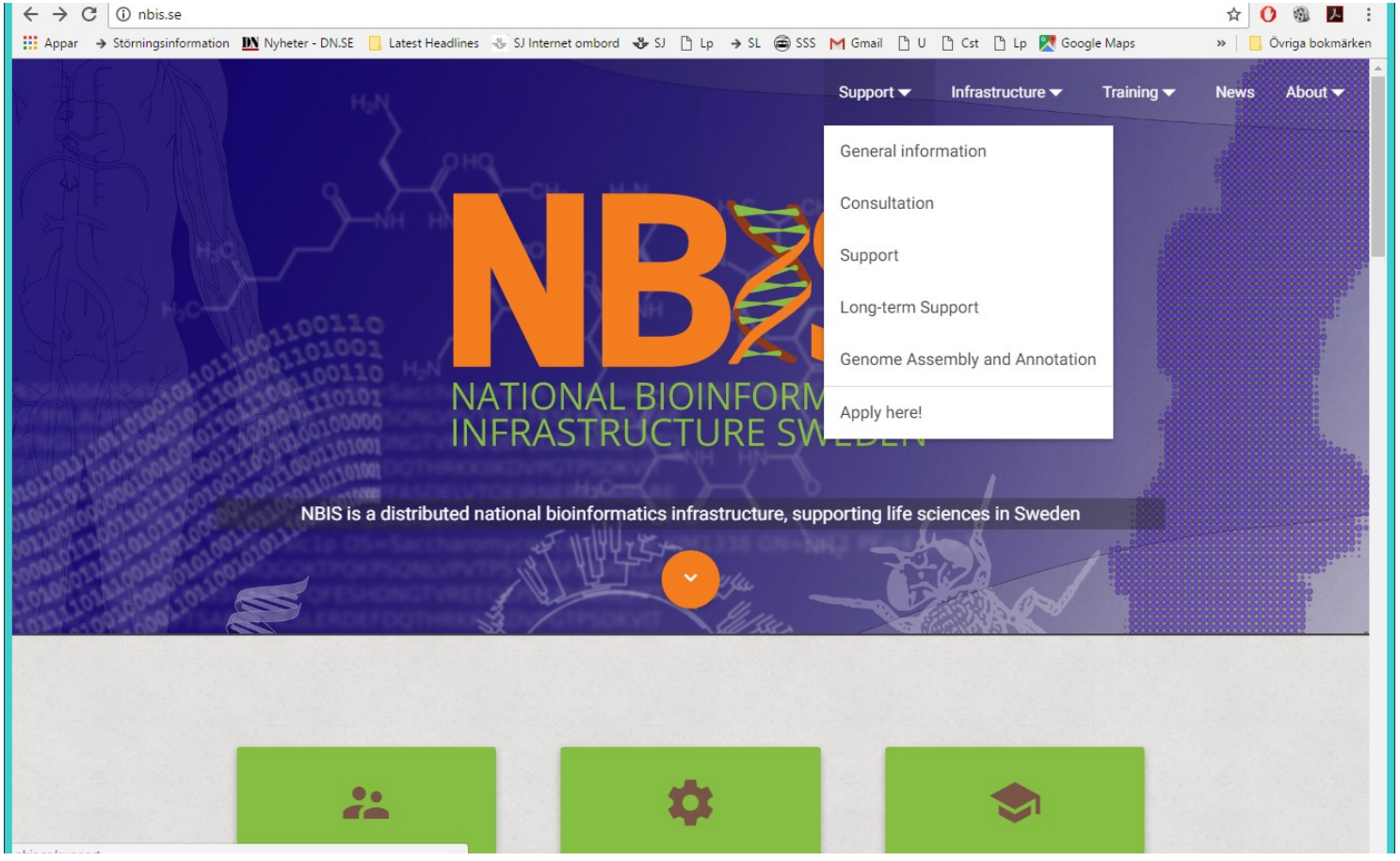
NVIDIA, 2019

https://thispersondoesnotexist.com/

National Bioinformatics Infrastructure Sweden

- Drop-in sessions on Zoom every Tuesday at 14
- Project support (user fee 800 kr/h)
- Long-term support (free; selection by peer review)
- Partner projects (cost: part-time employment)
- PhD advisory program (one call per year)
- Numerous workshops for PhD students and scientists (Neural Networks and Deep learning workshop: Jan 2022)

NBIS
NATIONAL BIOINFORMATICS INFRASTRUCTURE SWEDEN


nbis.se
Appar
Störningsinformation
Nyheter - DN.SE
Latest Headlines
SJ Internet ombord
SJ
Lp
SL
SSS
Gmail
U
Cst
Lp
Google Maps
Övriga bokmärken
Support
Infrastructure
Training
News
About
General information
Consultation
Support
Long-term Support
Genome Assembly and Annotation
Apply here!
NATIONAL BIOINFORM
INFRASTRUCTURE SW
NBIS is a distributed national bioinformatics infrastructure, supporting life sciences in Sweden

Expertise

Genomics

Systems biology

Proteomics

Machine Learning

Multi-omics integration

Metabolomics

Phylogenomics

Biostatistics

Computational method development