Introduction to Deep Learning and Neural Network

Kuk-Jin Yoon

Visual Intelligence Lab.
Department of Mechanical Engineering

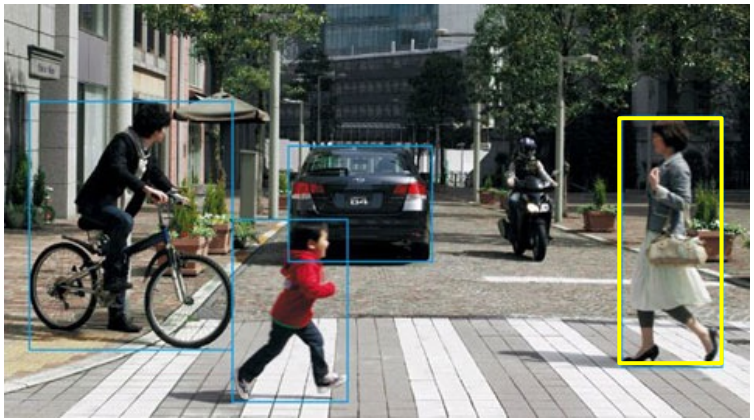
KAIST

The Core of Visual Intelligence

- Different levels of image primitives
  - Pixel: basic primitive
  - Region: a set of connected pixels
  - Object: a set of (connected) regions
  - Scene: a set of objects

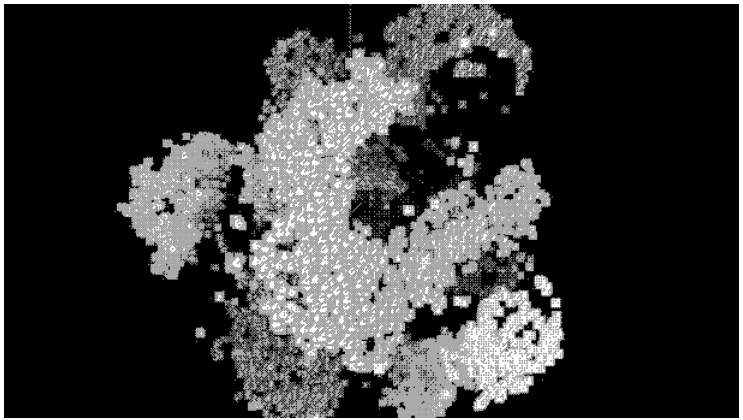
- How to represent the image primitives (feature extraction) and how to process (e.g., matching, classifying) in the latent space (classification)
  - Feature extraction, mapping, and classification (or matching)
  - Deep neural networks are superior to conventional methods

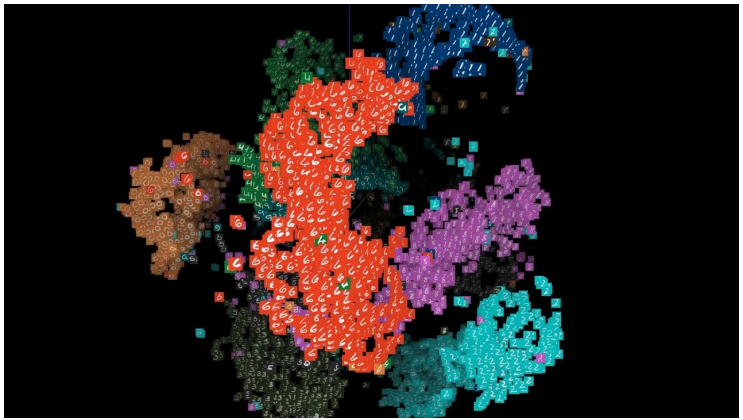
Object Detection/Classification



Representation: feature extraction

Classification in the feature space

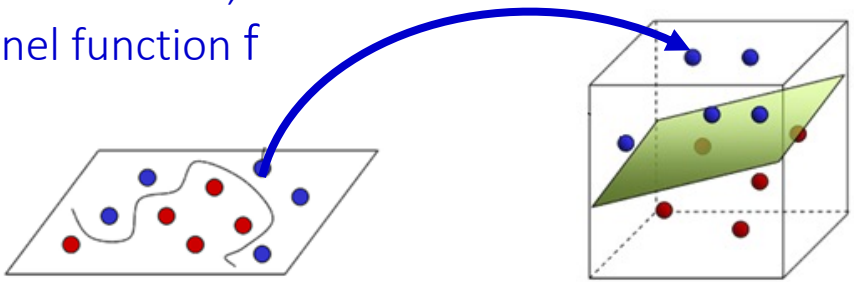




Classification in **Feature** space

- In machine learning and pattern recognition, a feature is an individual measurable property or characteristic of a phenomenon being observed
- Choosing informative, discriminating and independent features is a crucial step for effective algorithms in machine learning
- **Kernel function** which enable the data in raw representation to operate in a high-dimensional, implicit "**feature" space**

Coordinate transfer,
i.e., Kernel function f



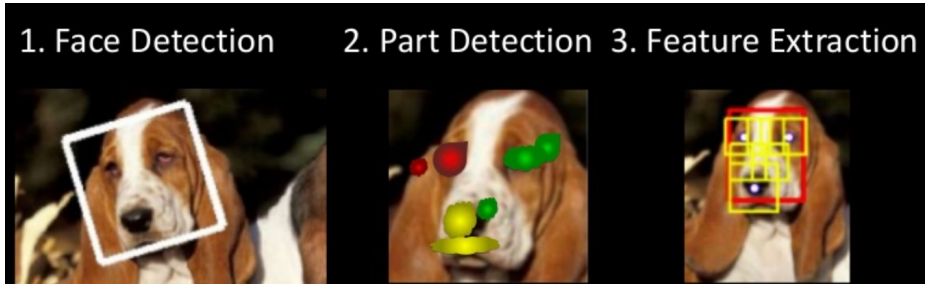
Input space

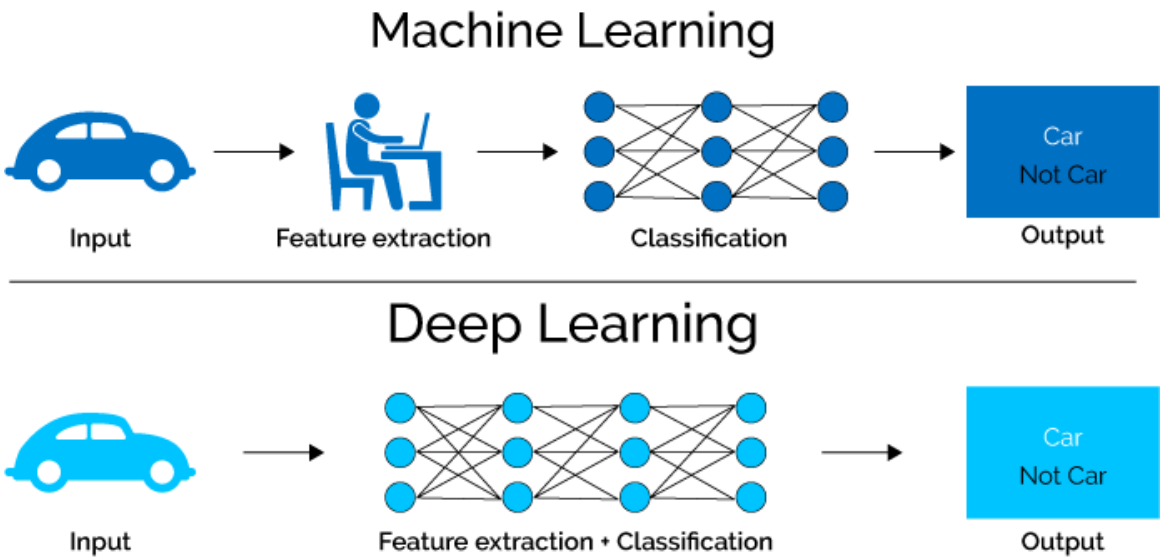
Feature space

✓ Then find SVM = maximize the distance from separation lines(planes)

[images from google]

Deep Learning

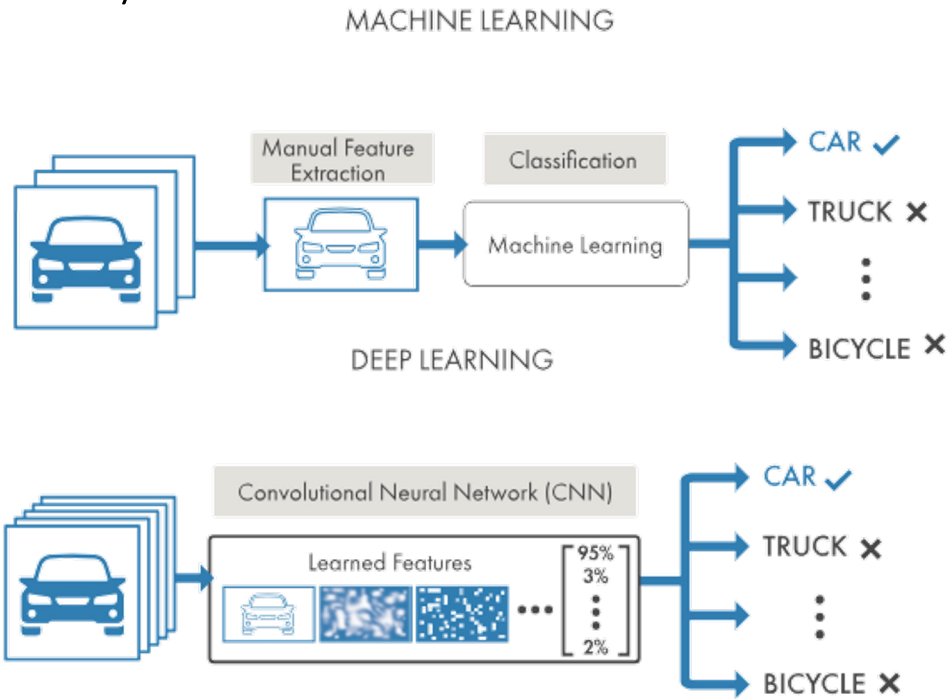




http://www.umiacs.umd.edu/~kanazawa/
https://medium.com/swlh/ill-tell-you-why-deep-learning-is-so-popular-and-in-demand-5aca72628780

Machine Learning vs. Deep Learning

- In machine learning, you manually choose features and a classifier to sort images. With deep learning, feature extraction and modeling steps are automatic (so called, end-to-end learning).
- A key advantage of deep learning networks is that they often continue to improve as the size of your data increases.



https://www.mathworks.com/discovery/deep-learning.html#withmatlab

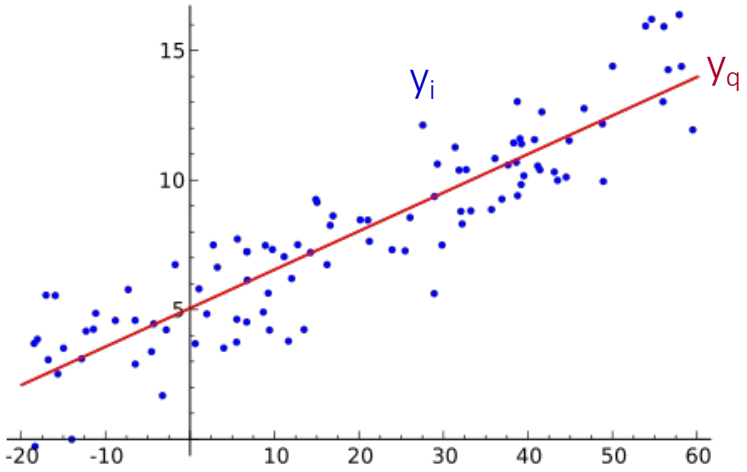
Machine Learning vs. Deep Learning

	Machine Learning	Deep Learning
Training dataset	Small	Large
Choose your own features	Yes	No
# of classifiers available	Many	Few
Training time	Short	Long

From Matlab machine learning tutorial

NEURAL NETWORK

Mathematical Formulation of Regression



- Model/feature (= regression function)

$$y_\theta(x_i) = \theta^T x_i + b$$

- Loss function (= cost function)

$$J(\theta) = \sum_{i=1} \left(y_\theta(x_i) - y_i \right)^2$$

- Mean Squared Error (MSE)

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \bar{y}_i \right)^2$$

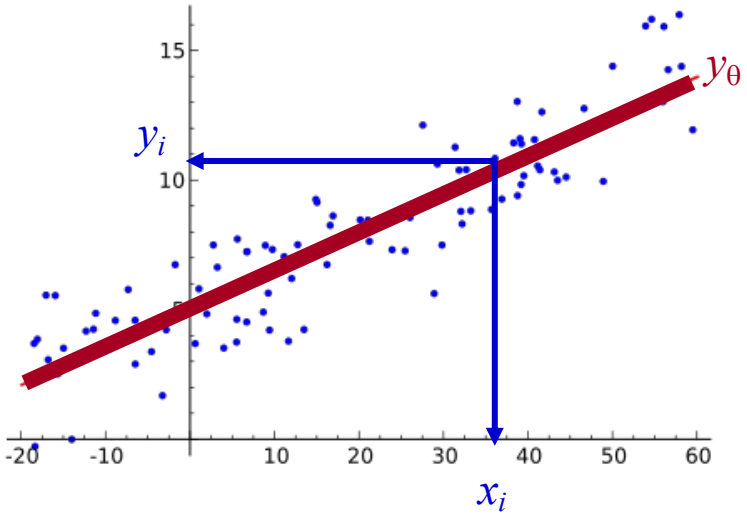
- Optimization

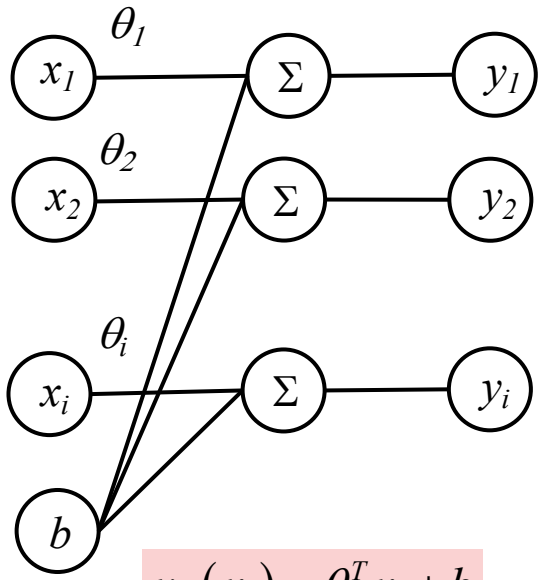
$$\theta^* = \arg\min_\theta J(\theta)$$

KAIST [images from google]

Machine Learning

- Try your own definition





$$y_\theta(x_i) = \theta^T x_i + b$$

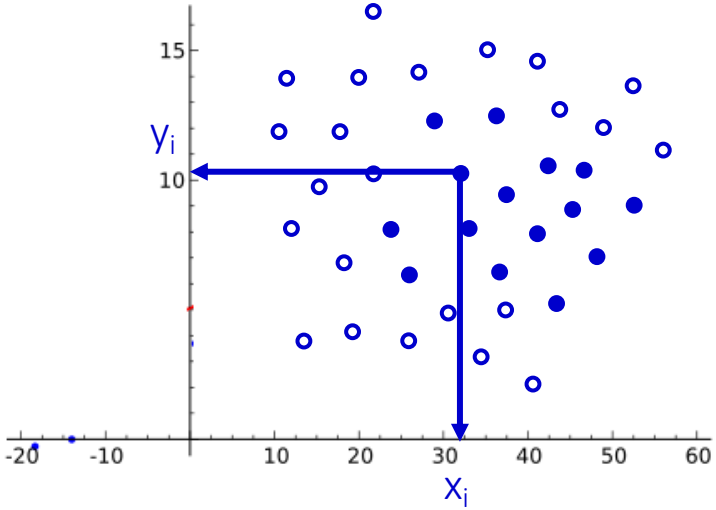
- Loss function (= cost function)

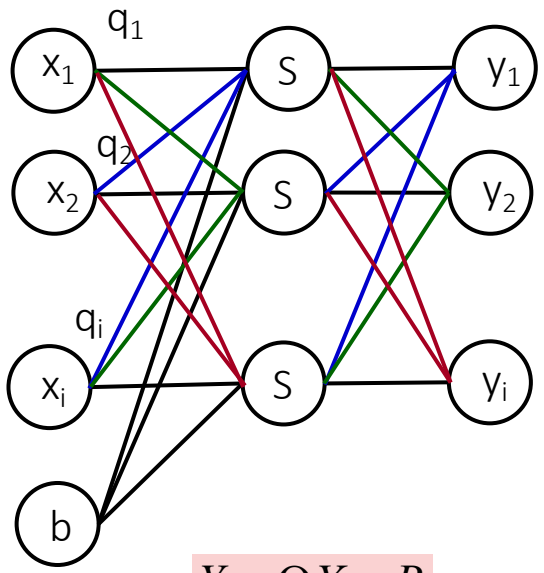
$$J(\theta) = \sum_{i=1} \left(y_\theta(x_i) - y_i \right)^2$$

- Mean Squared Error (MSE)

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \bar{y}_i \right)^2$$

Machine Learning ⊃ Neural Network





$$Y = \Theta X + B$$

- Loss function (= cost function)

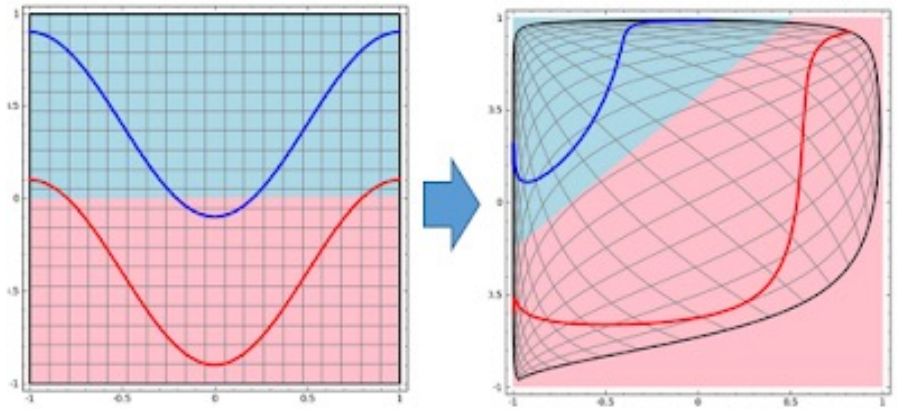
$$J(\theta) = \sum_{i=1} \left(y_\theta(x_i) - y_i \right)^2$$

- Mean Squared Error (MSE)

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \bar{y}_i \right)^2$$

Neural Network

- Neural networks are structures that are repeatedly built up with linear fitting and nonlinear transformation or activation to distinguish data

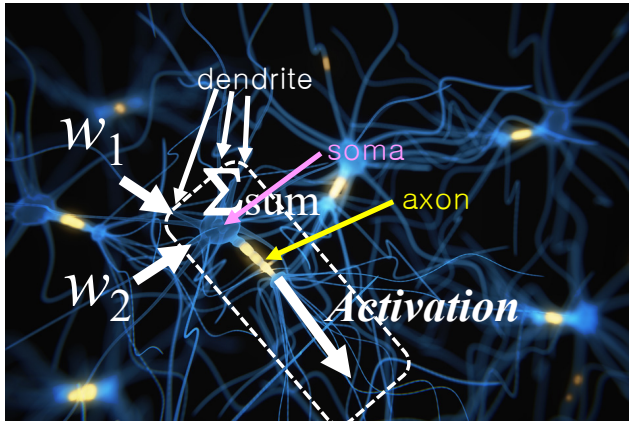


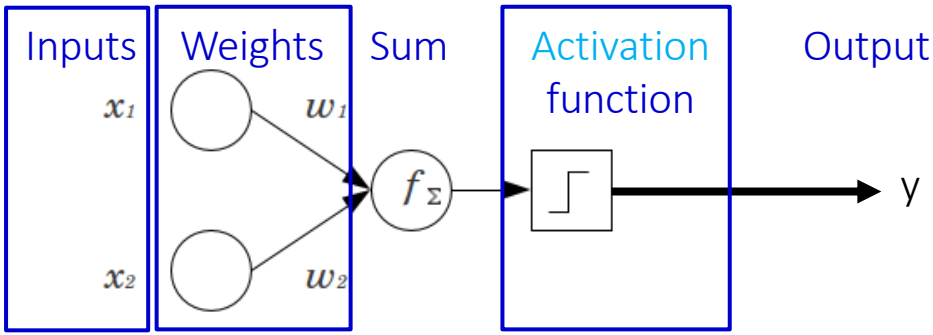
(colah's blog)

- We can not perfectly separate the space into two sub-spaces just by using linear classifiers (left), whereas we can do that after non-linearly transforming the space into another space (right) .
- Neural networks process the data by repeatedly performing the above steps.

http://t-robotics.blogspot.com/2015/05/deep-learning.html#.W2ke9SgzY2w

Perceptron: a basic unit of neural network



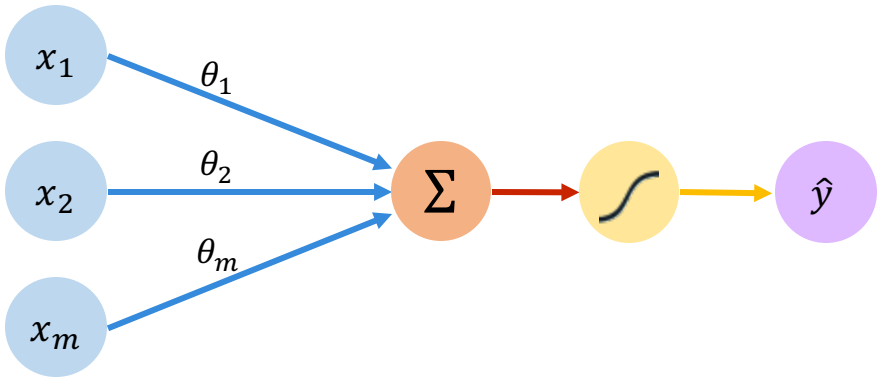


Activation function
- Step function
- Sigmoid(=logistic) function
- ReLU
- Softmax

node So an artificial neuron simply calculates a "weighted sum" of its input and then decides whether it should be "fired" or not (= activation)

KAIST [images from google]

The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

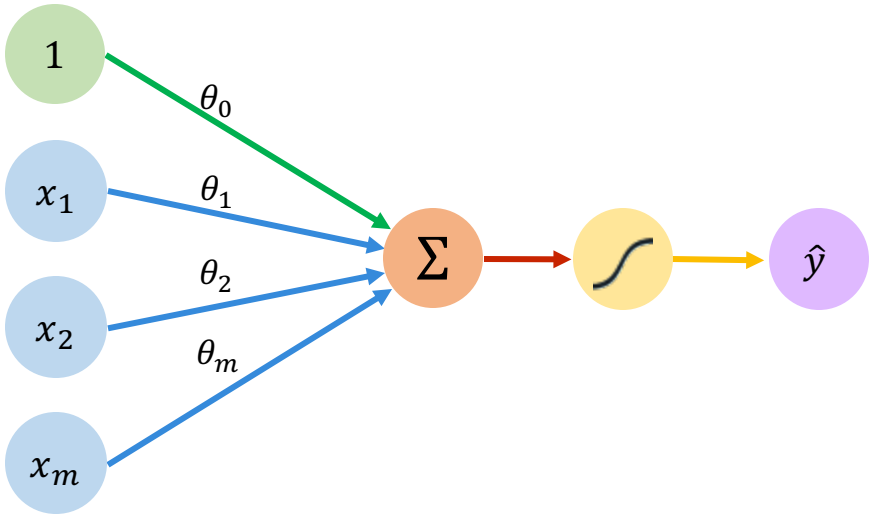
Output

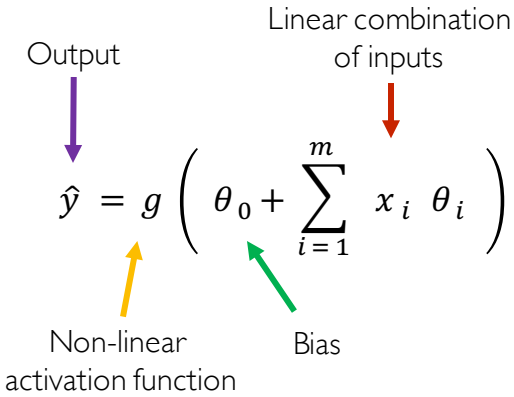
Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i \, \theta_i\right)$$

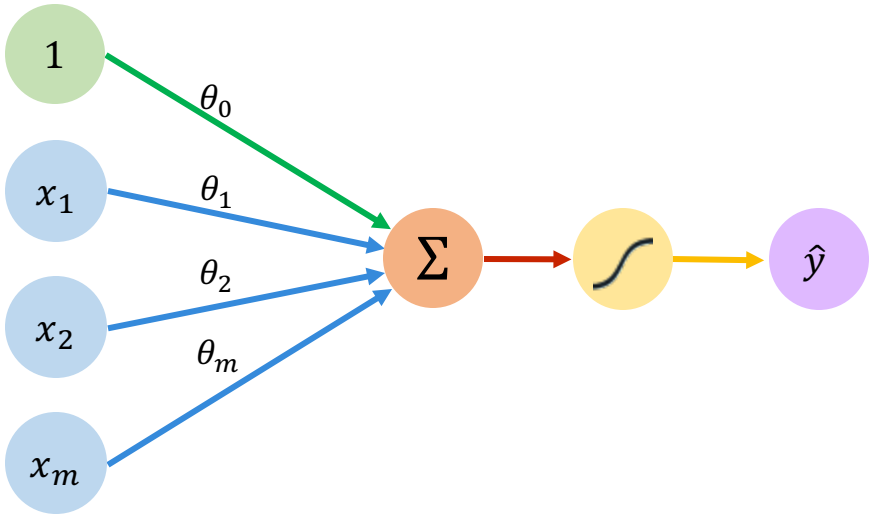
Non-linear activation function

The Perceptron: Forward Propagation





The Perceptron: Forward Propagation

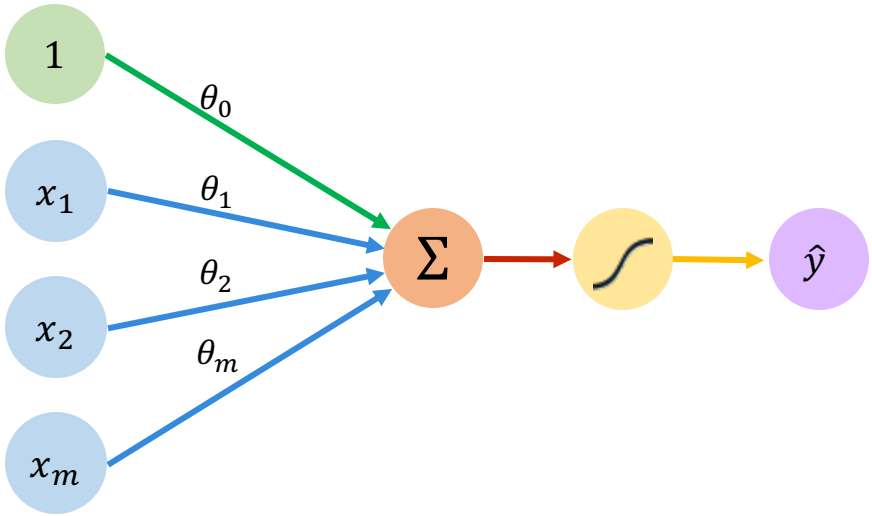


$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i\ \theta_i\right)$$

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix}$

The Perceptron: Forward Propagation

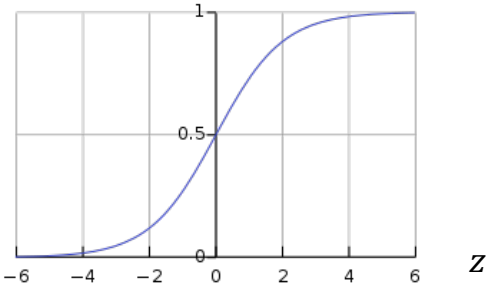


Activation Functions

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta}\right)$$

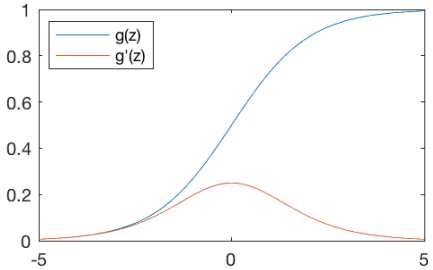
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function

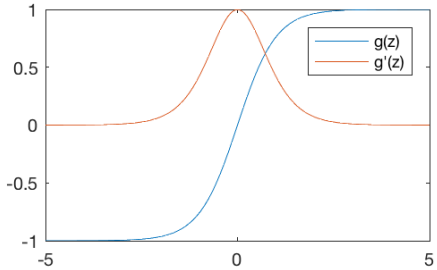


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

tf.nn.sigmoid(z)

Hyperbolic Tangent

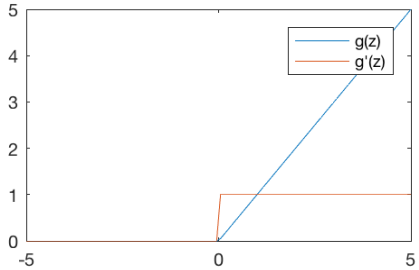


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

tf.nn.tanh(z)

Rectified Linear Unit (ReLU)



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

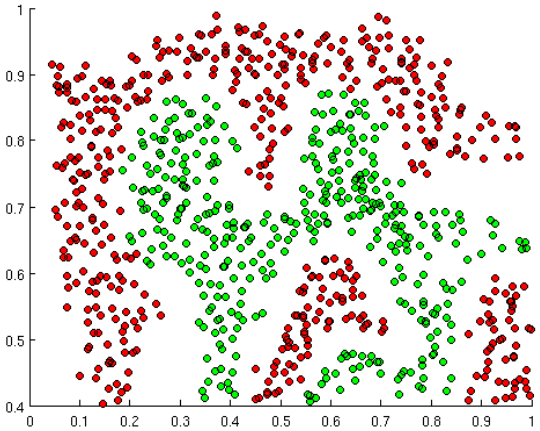
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

tf.nn.relu(z)

NOTE: All activation functions are non-linear

Importance of Activation Functions

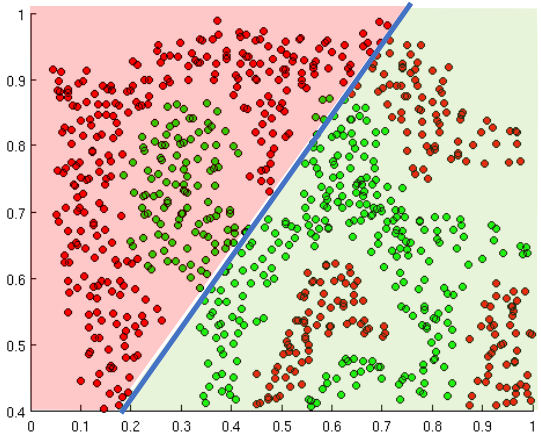
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



What if we wanted to build a Neural Network to distinguish green vs red points?

Importance of Activation Functions

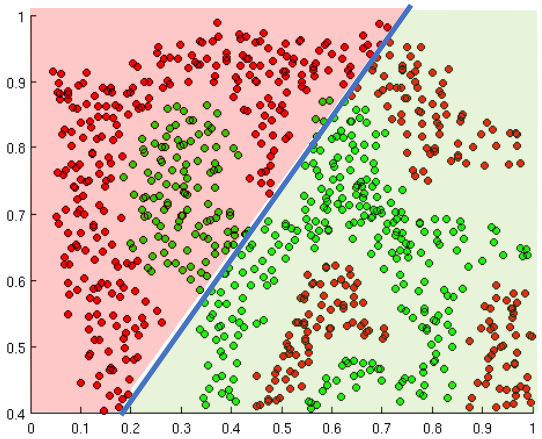
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



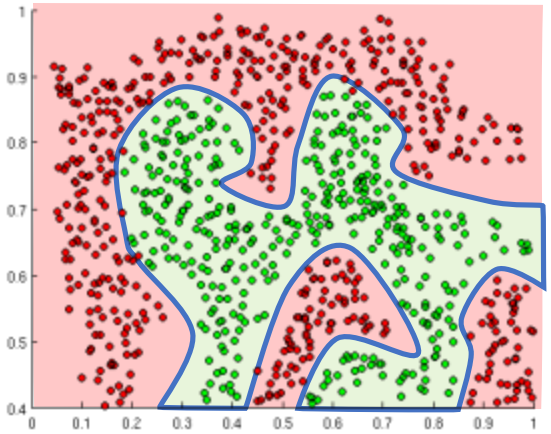
Linear Activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

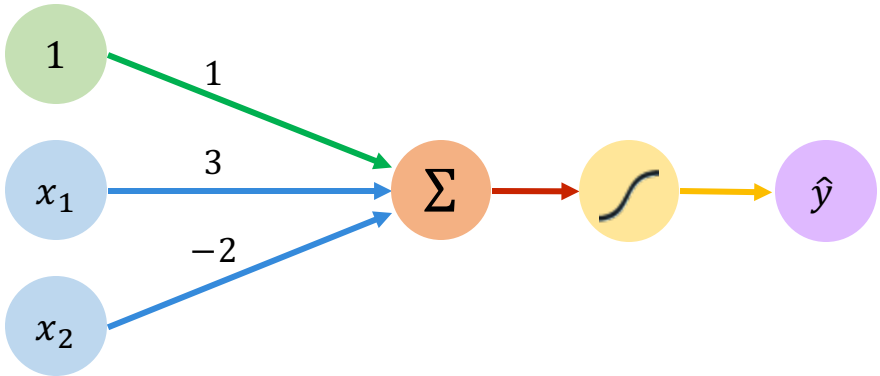


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

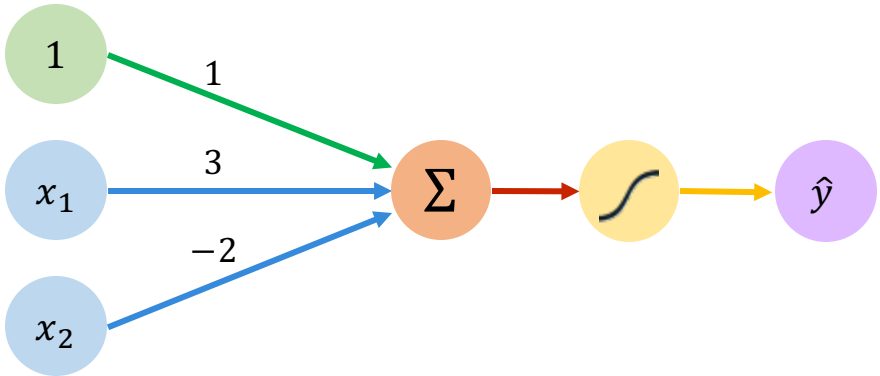
The Perceptron: Example



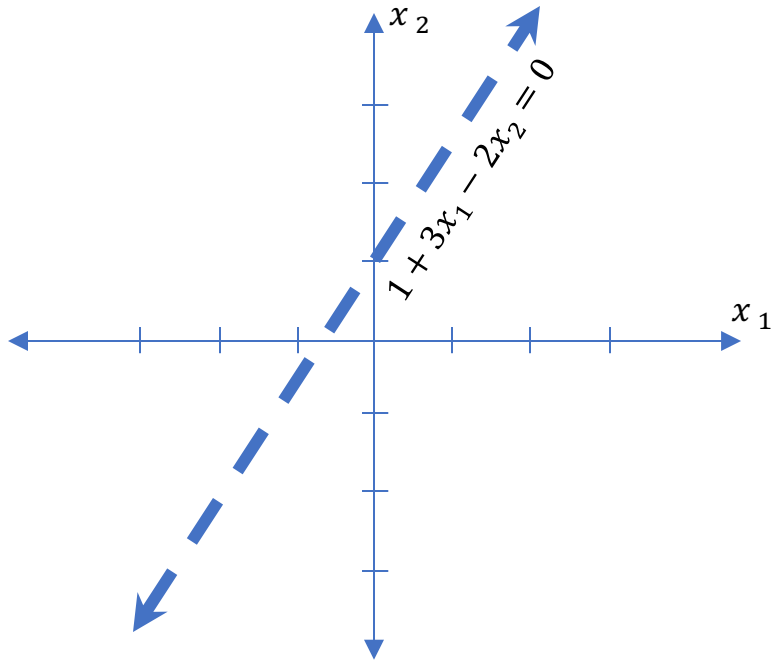
We have: $\theta_0 = 1$ and $\boldsymbol{\theta} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}})$$

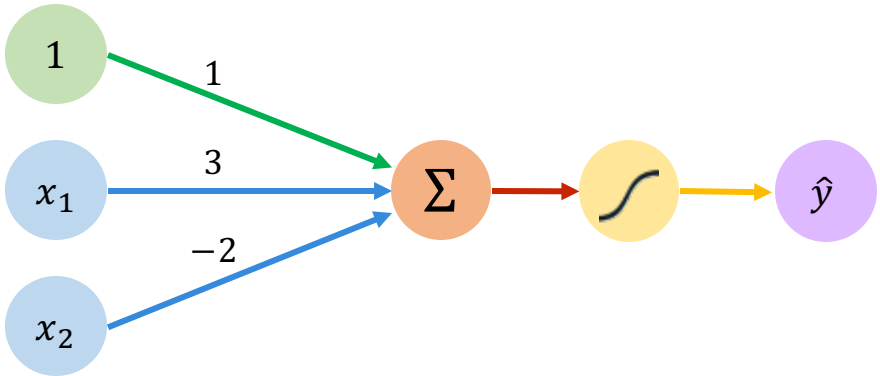
The Perceptron: Example



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



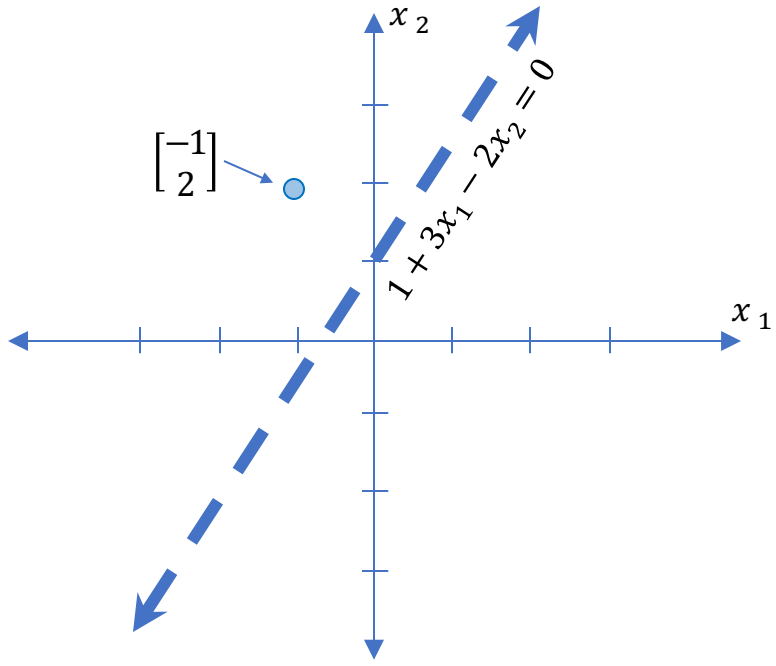
The Perceptron: Example



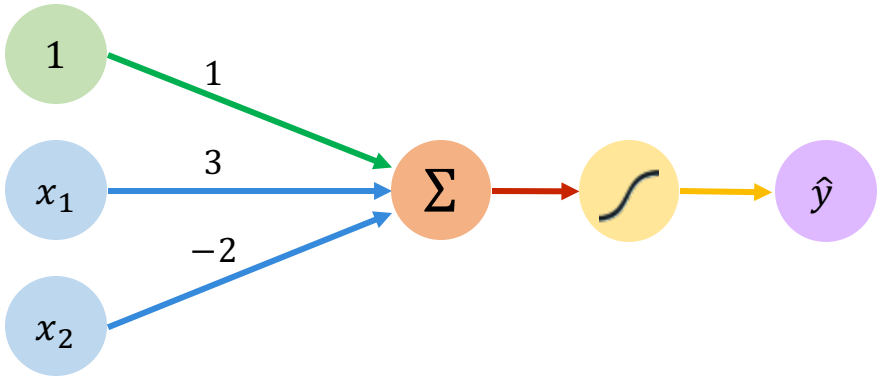
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

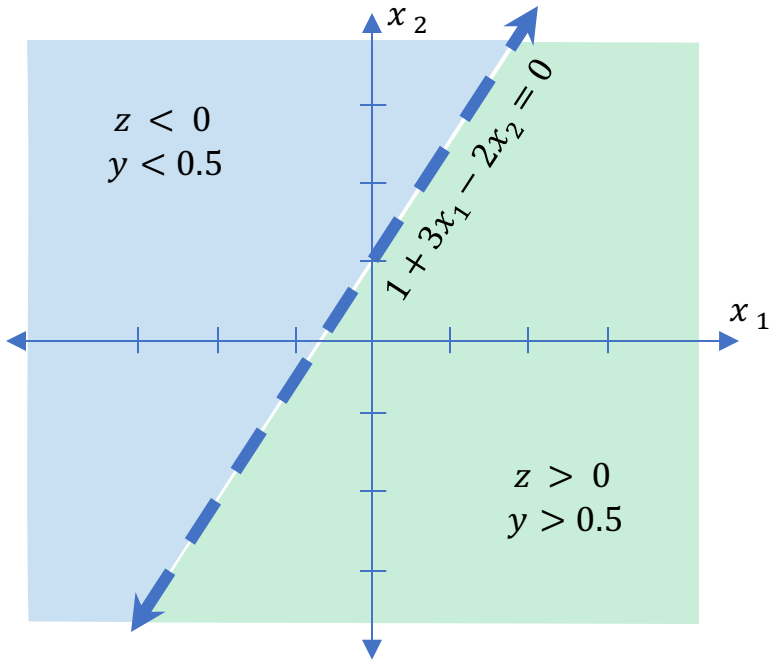
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The Perceptron: Example



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

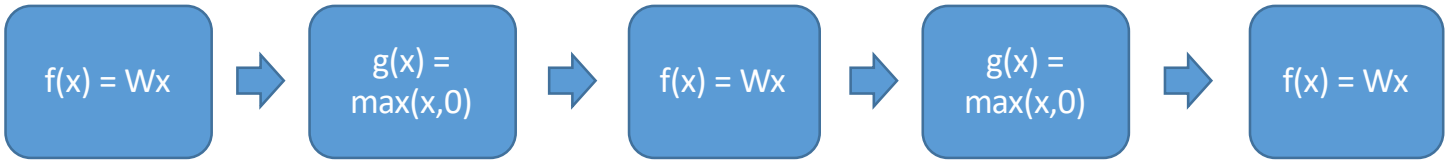


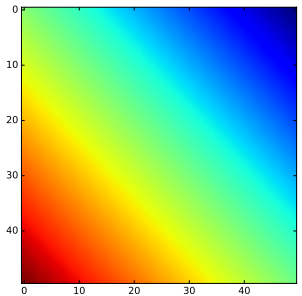
Building Neural Networks with Perceptrons

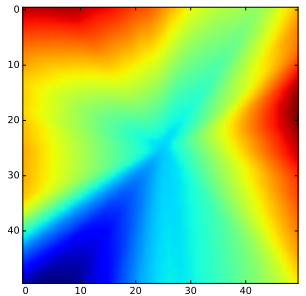
Lecture note from MIT 6.S191 Introduction to Deep Learning

Multilayer Perceptrons

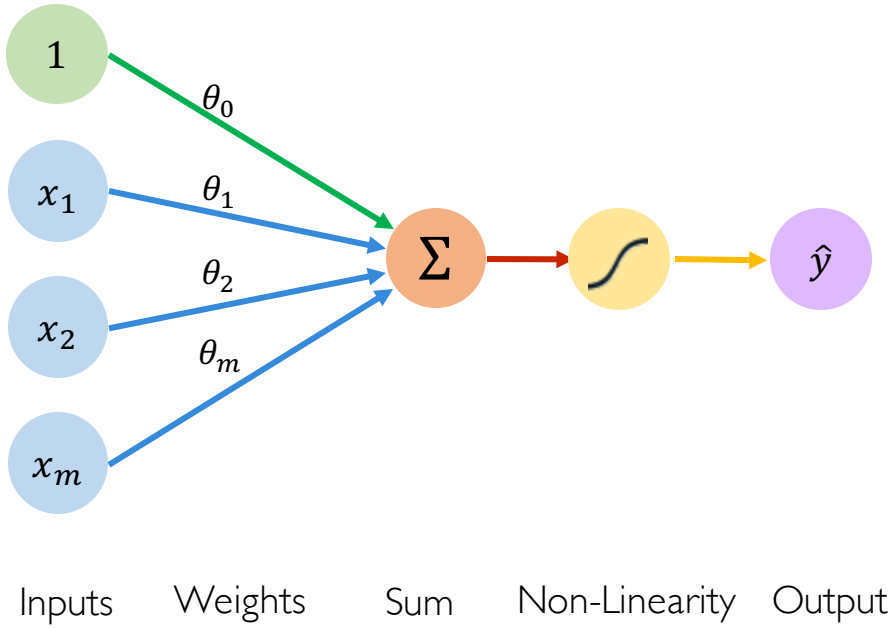
- Key idea: build complex functions by composing simple functions
- Caveat: simple functions must include non-linearities
- W(U(Vx)) = (WUV)x



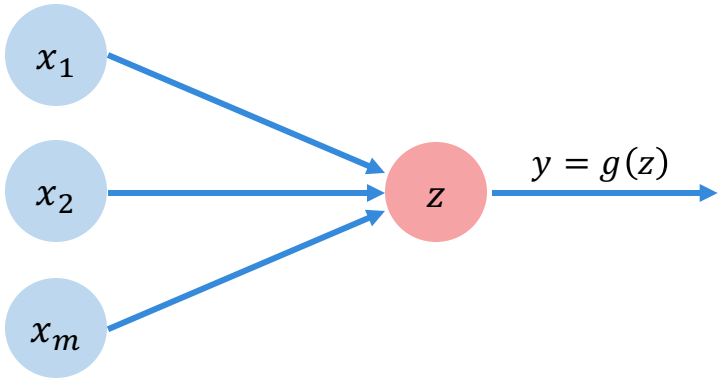




The Perceptron: Simplified

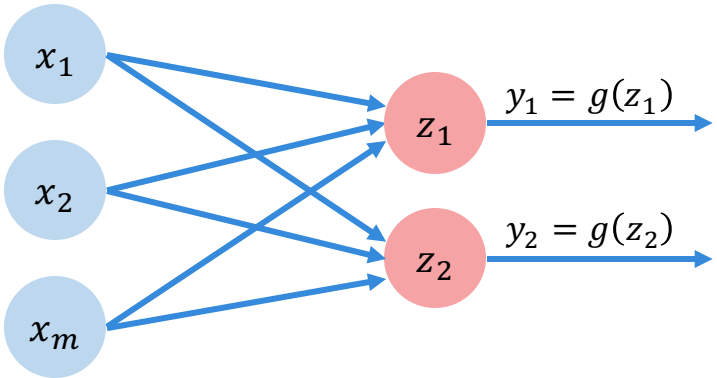


The Perceptron: Simplified



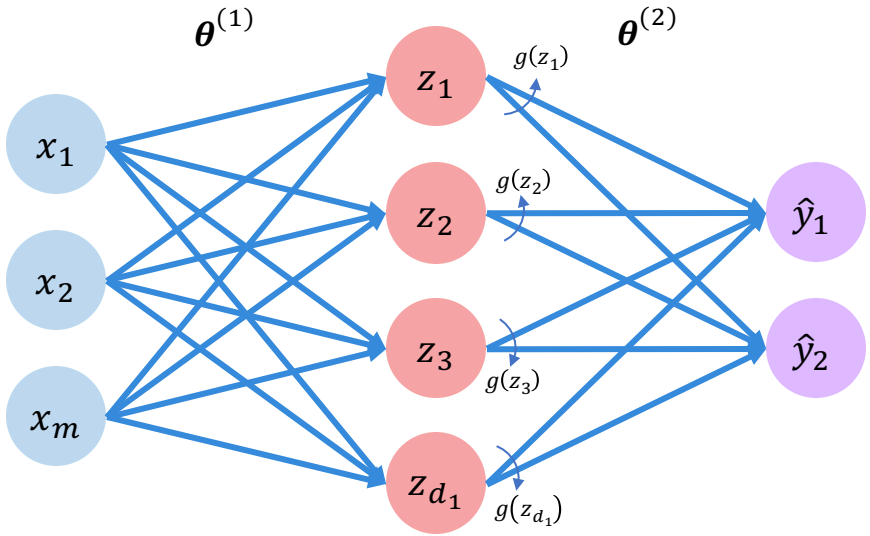
$$z = \theta_0 + \sum_{j=1}^{m} x_j \, \theta_j$$

Multi Output Perceptron



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

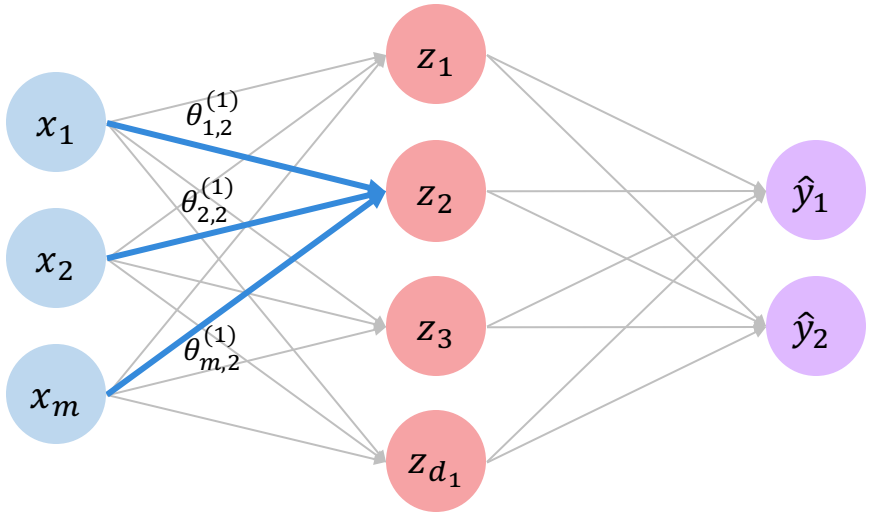
Single Layer Neural Network



Inputs Hidden Final Output

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

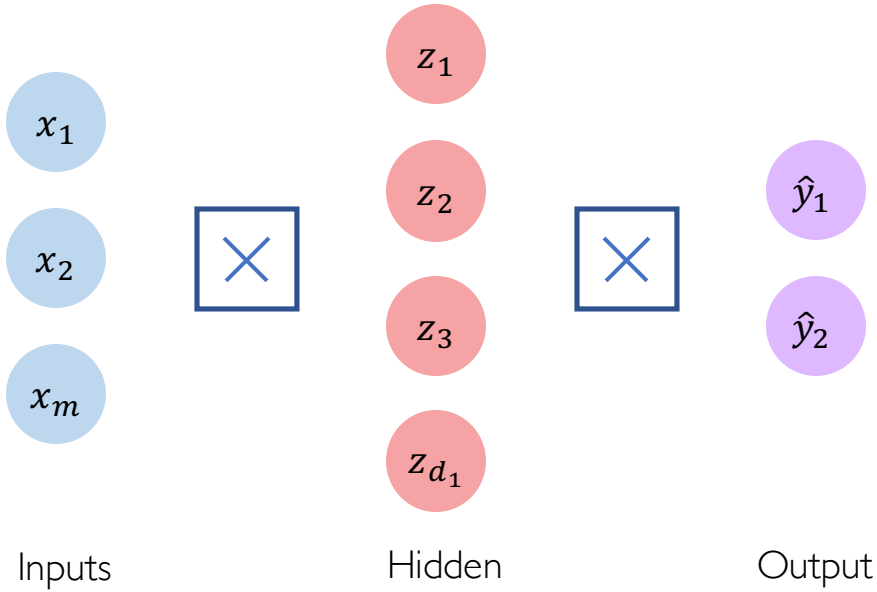
Single Layer Neural Network



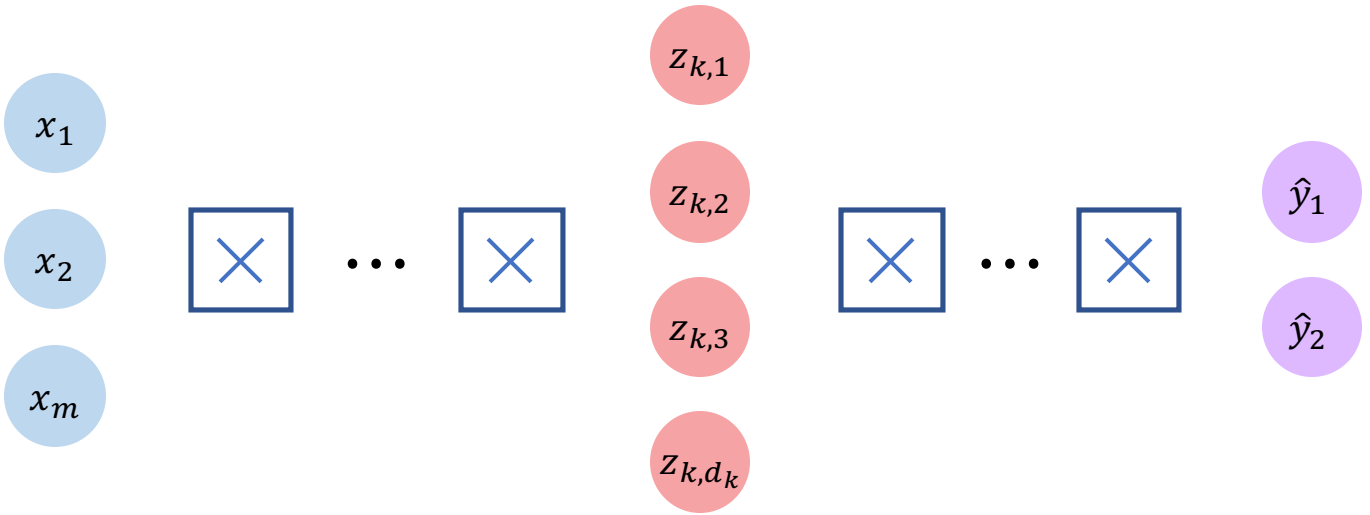
$$z_2 = \theta_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, \theta_{j,2}^{(1)}$$

$$= \theta_{0,2}^{(1)} + x_1 \, \theta_{1,2}^{(1)} + x_2 \, \theta_{2,2}^{(1)} + x_m \, \theta_{m,2}^{(1)}$$

Multi Output Perceptron

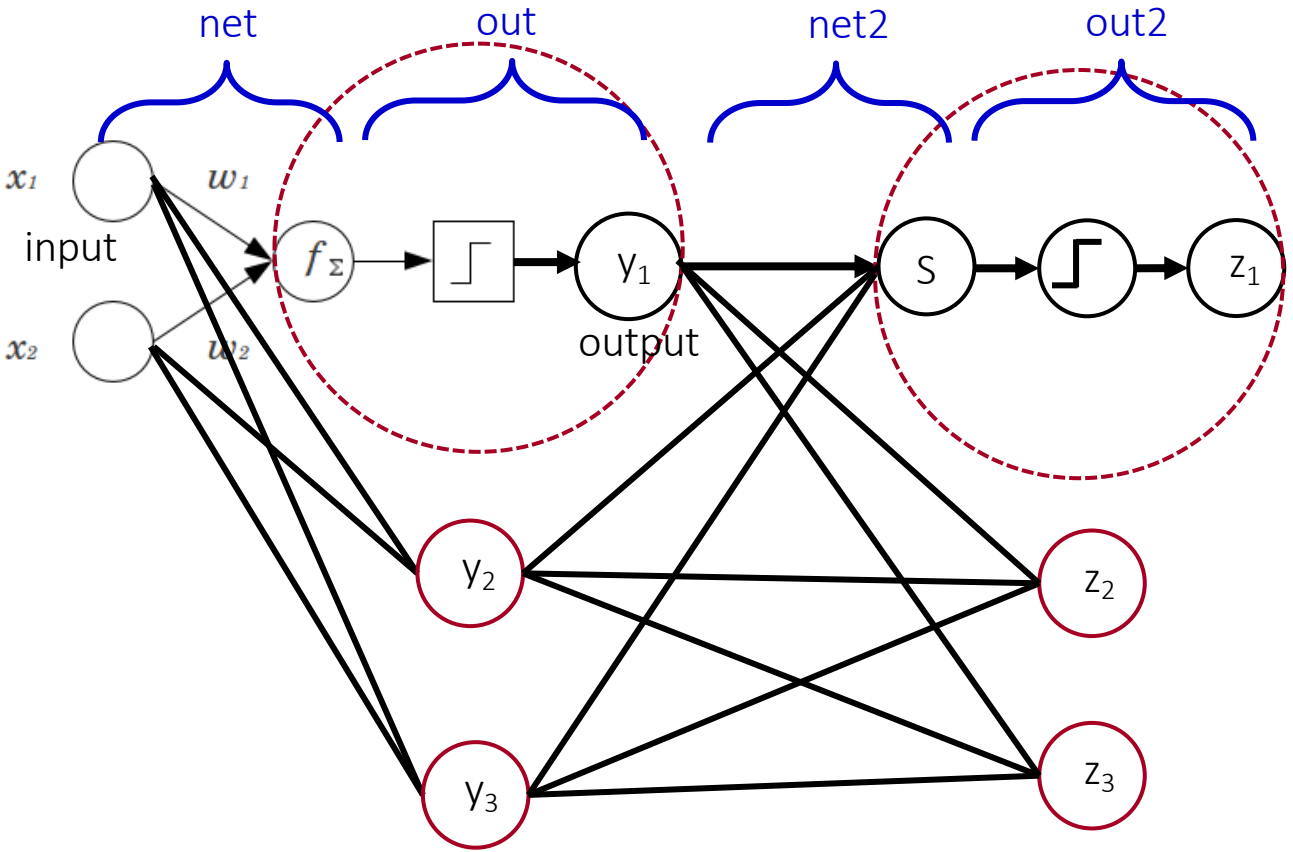


Deep Neural Network



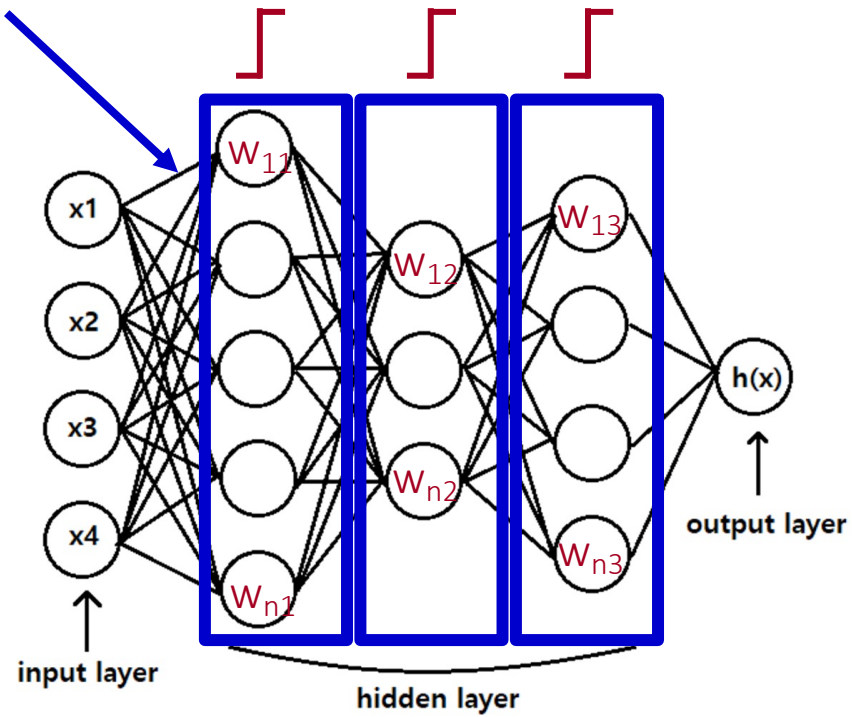
$$z_{k,i} = \theta_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, \theta_{j,i}^{(k)}$$

Multilayer Perceptron



Multilayer Perceptron

Fully connected



Q: how to find w's?

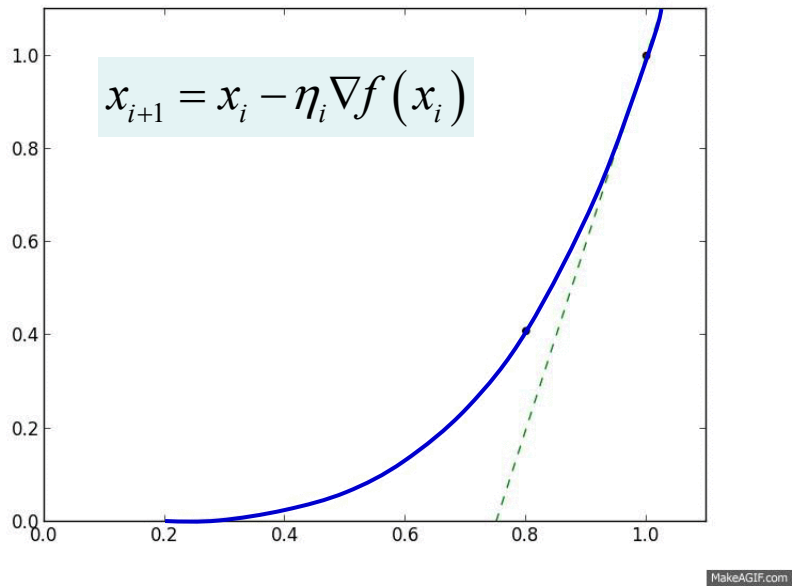
A: Tune 'w' to minimize the error btw. the true outputs and the estimated outputs ≡ loss function

[images from google]

Tune weight w to minimize the loss function

- Recall, Gradient descent

  Find x^* such that min $f(x^*)$



- Backpropagation

$$w' = w - \eta \nabla f(w)$$

Q: What is the loss function f(w)?

A: Error between the NN output & true values

Loss Optimization

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

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

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

Loss Optimization

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

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

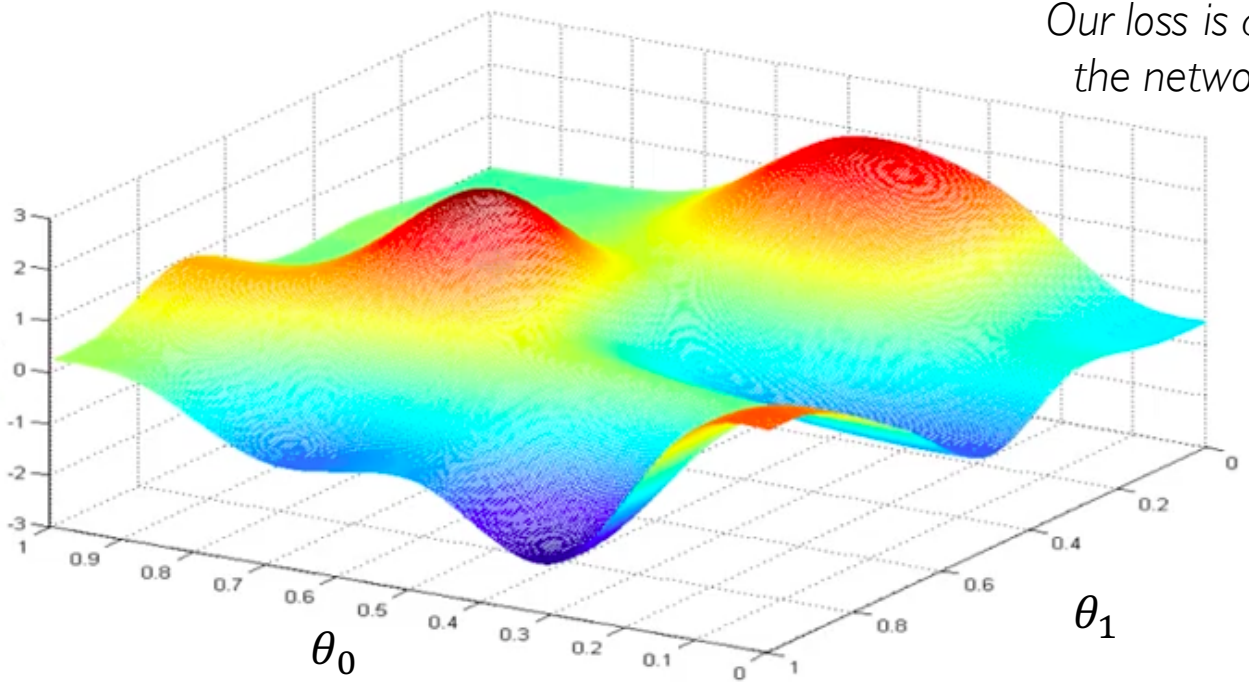
$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta})$$

Remember:
$\boldsymbol{\theta} = \{\theta^{(0)}, \theta^{(1)}, \dots\}$

Loss Optimization

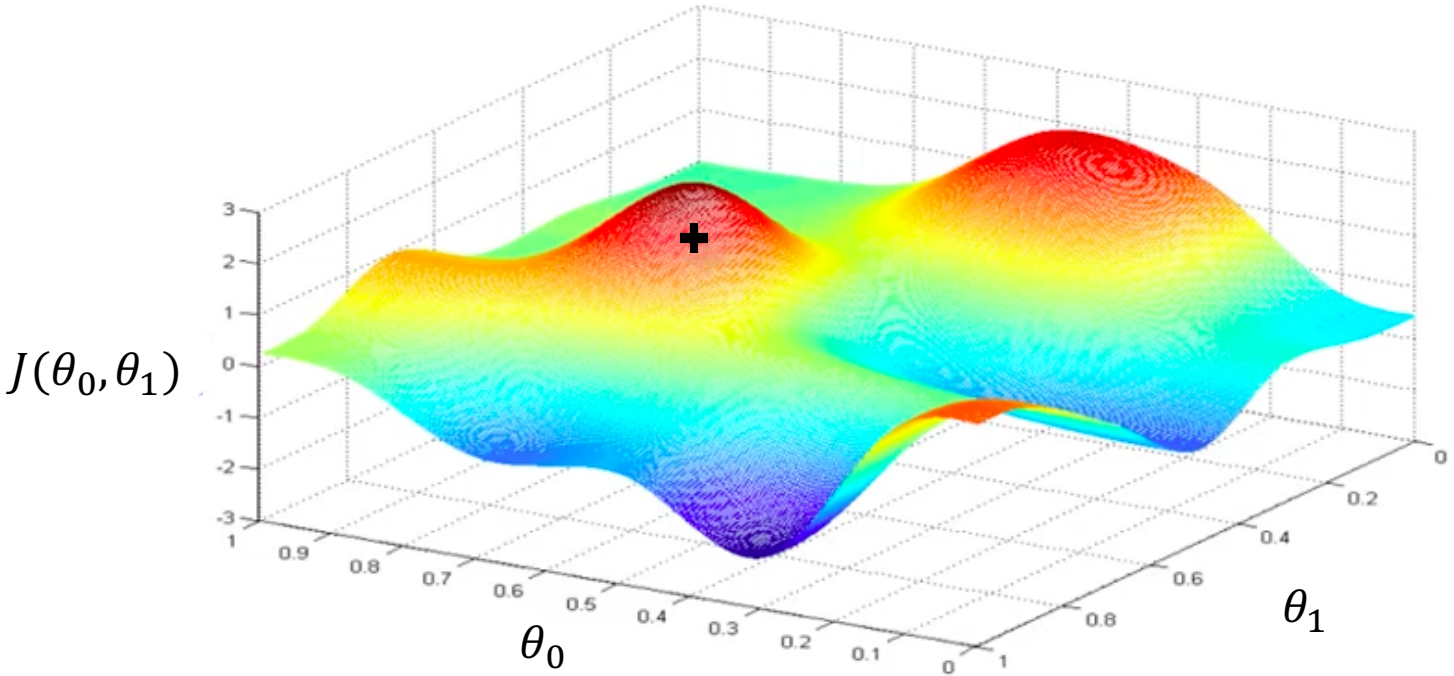
$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta})$$

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



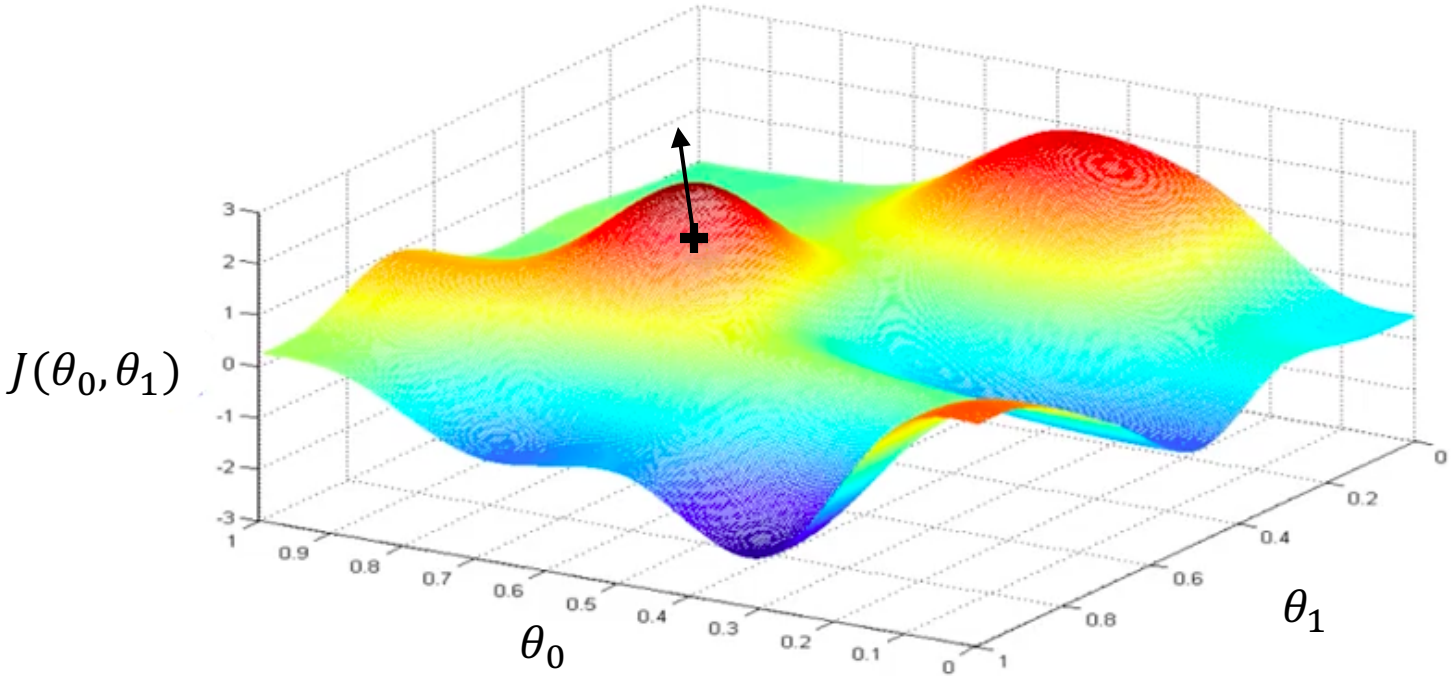
Loss Optimization

Randomly pick an initial (θ_0, θ_1)



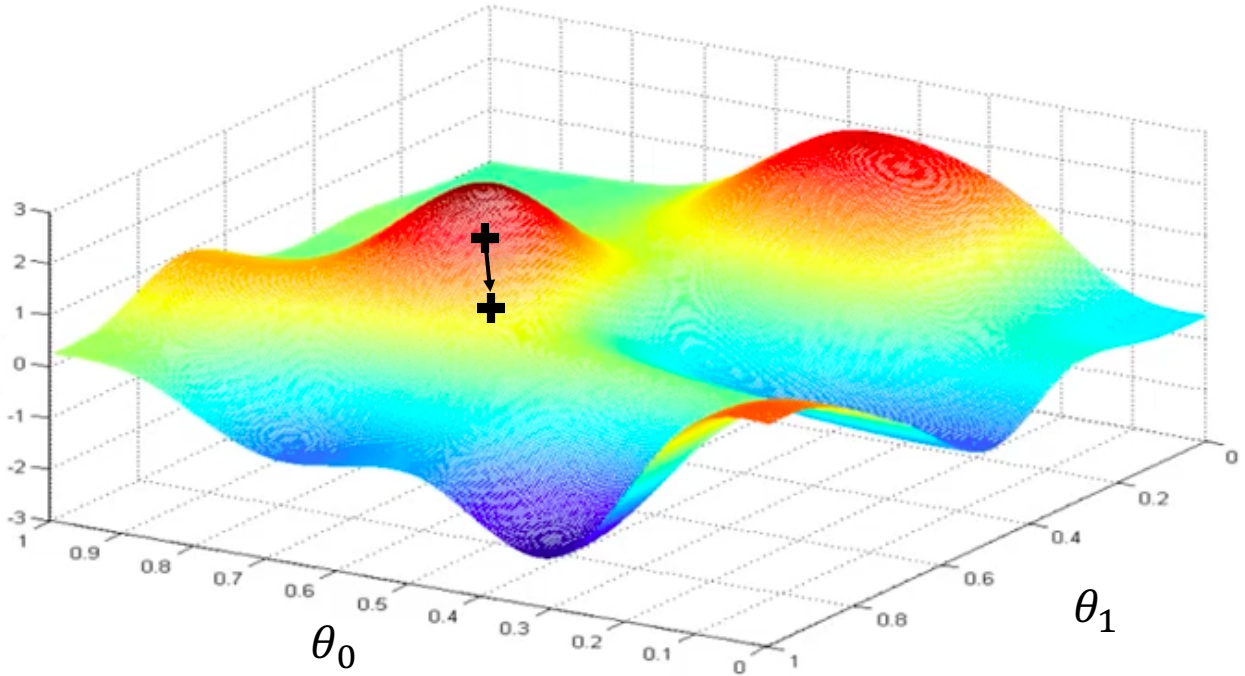
Loss Optimization

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



Loss Optimization

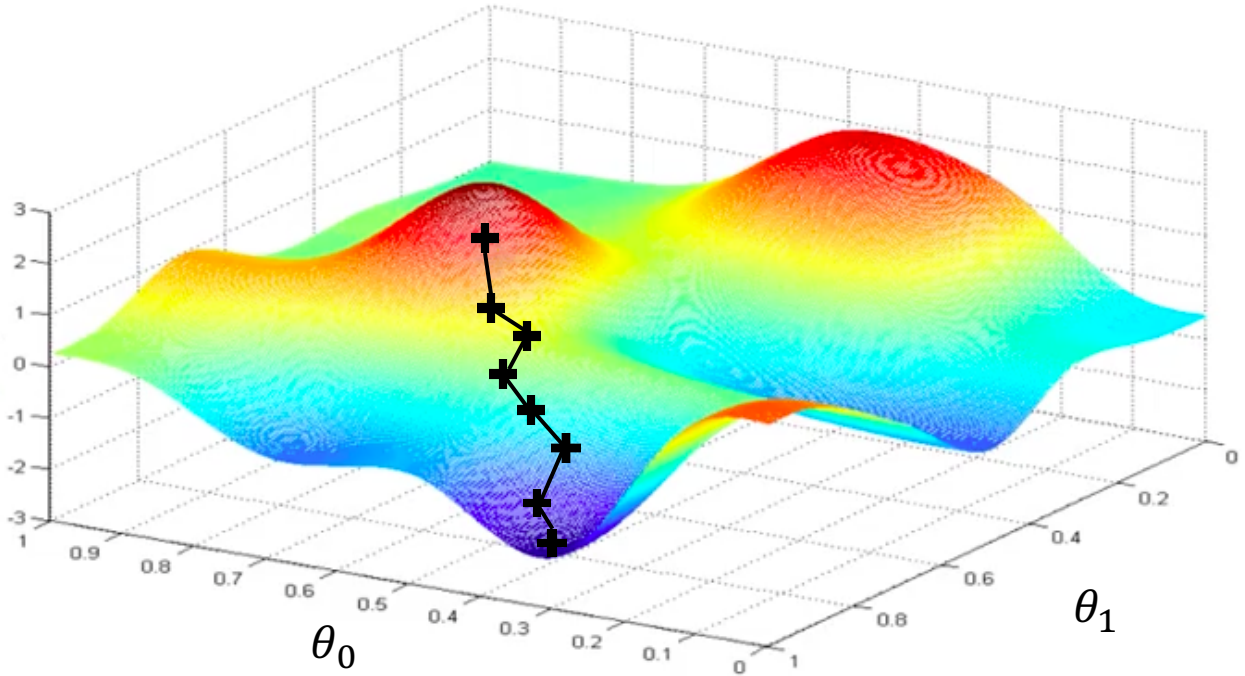
Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence

$J(\theta_0, \theta_1)$



Gradient Descent

Algorithm

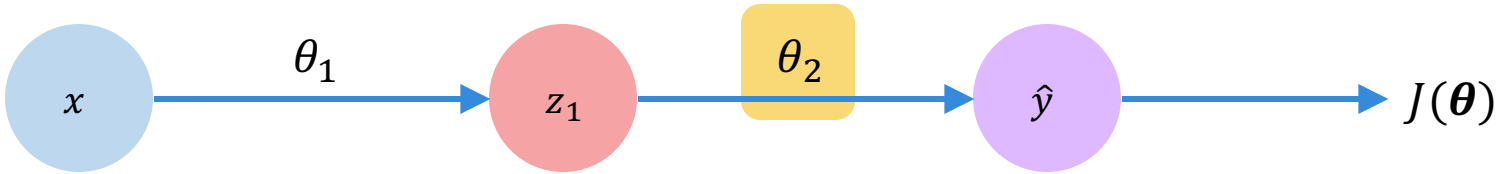
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

```
weights = tf.random_normal(shape, stddev=sigma)
```

```
grads = tf.gradients(ys=loss, xs=weights)
```

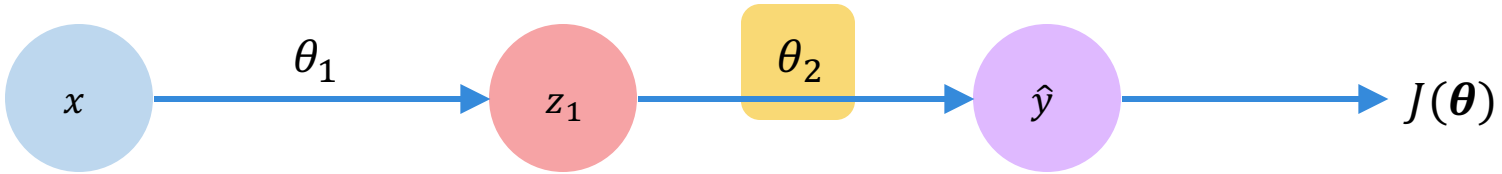
```
weights_new = weights.assign(weights - lr * grads)
```

Computing Gradients: Backpropagation



How does a small change in one weight (ex. $\boldsymbol{\theta_2}$) affect the final loss $\boldsymbol{J(\theta)}$?

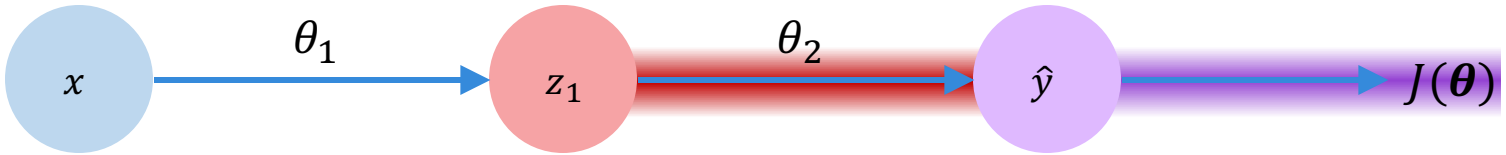
Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} =$$

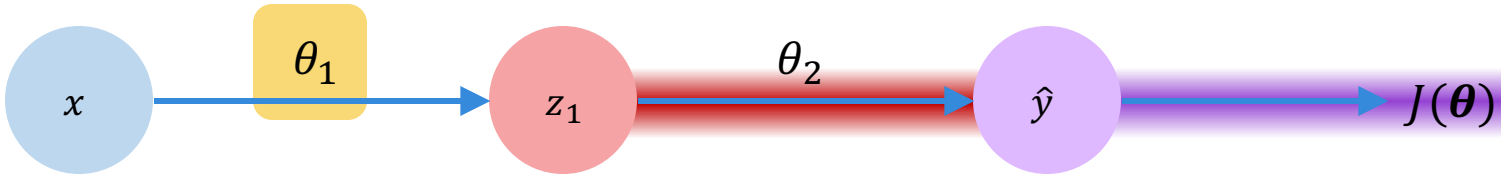
Let's use the chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta_2}$$

Computing Gradients: Backpropagation

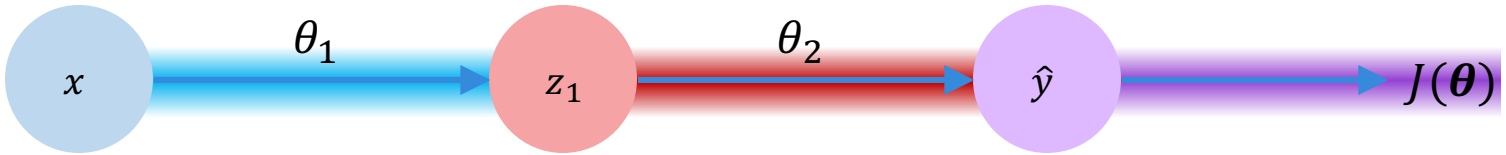


$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta_1}$$

Apply chain rule!

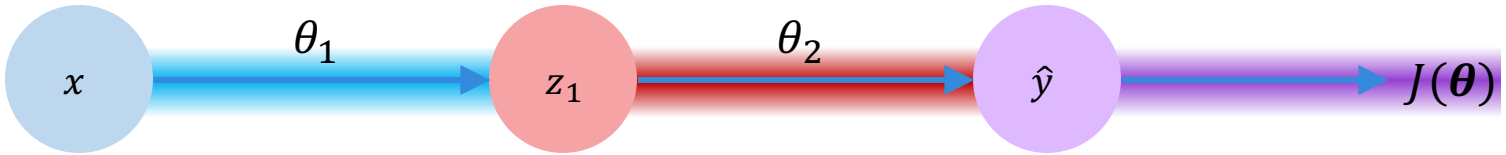
Apply chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$

Computing Gradients: Backpropagation

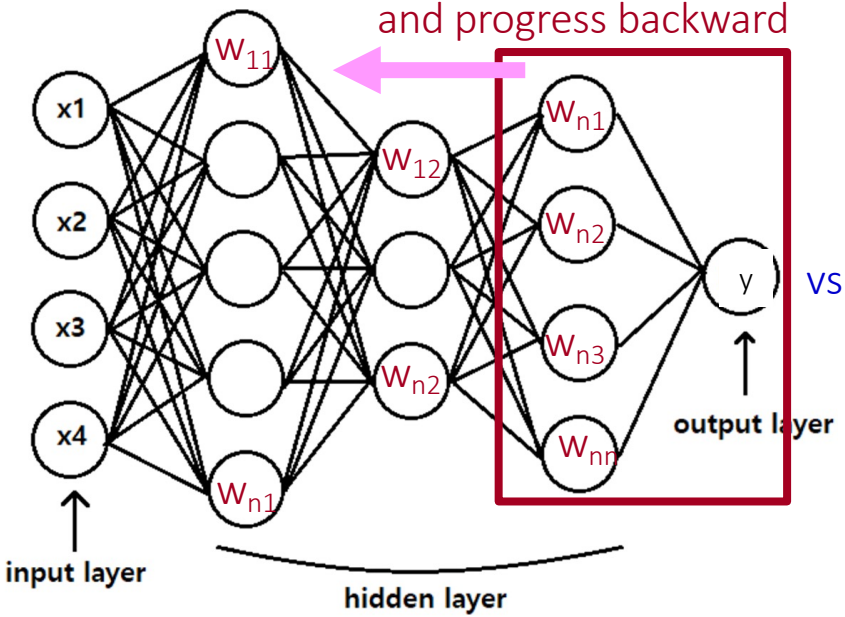


$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$

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

Backpropagation

- To tune 'w', starting from the outer most output, and progress backward



vs. y_{true}

$$f(w) = f\left(y_{true} - y_{NN}\right)^2$$

$$= f\left(y_{true} - Wx\right)^2$$

Adjust "w" to minimize loss function f

$$w' = w - \eta \nabla f(w)$$

Q: What is the loss function f(w)?

A: Error between the NN output & true values

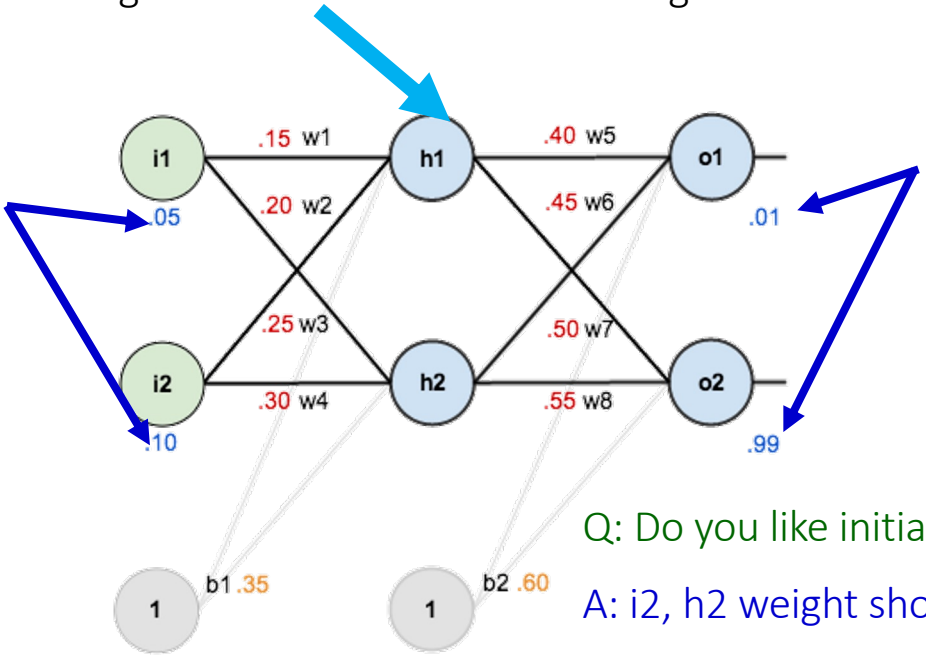
Backpropagation

- Backpropagation is a method used to **calculate a gradient of the loss function** to adjust the **weights** of nodes in the network.

Ex) Here are the initial weights, the biases, and training inputs/outputs

Suppose using an activation function with logistic function $y(x) = \frac{1}{1 + Ae^{Bx}}$

Given inputs 0.05 and 0.10



We want the neural network to output 0.01 and 0.99.

Q: Do you like initial guess?

A: i2, h2 weight should be greater

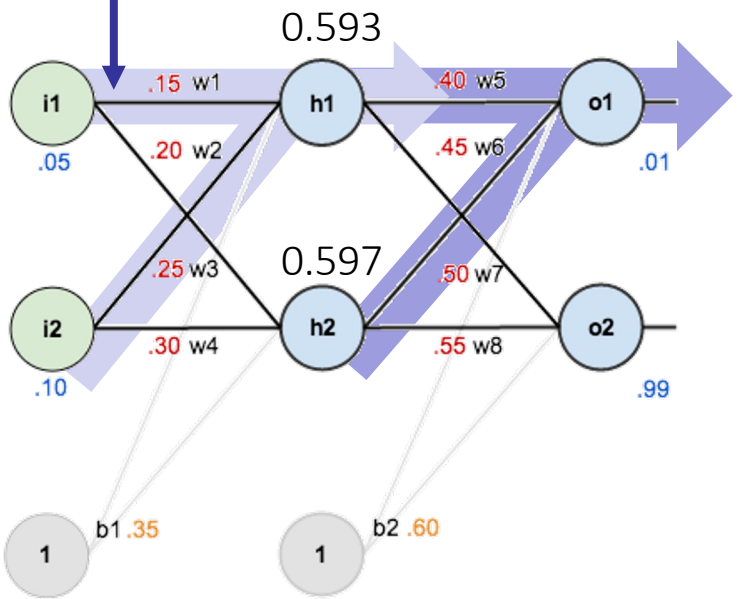
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Backpropagation

- Here's how we calculate the total net input for h1:

$$net_{h1} = w1 * i1 + w2 * i2 + b1 * 1 = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

- Using logistic function to get the output of h1: $out_{h_1} = \frac{1}{1+e^{-h_1}} = \frac{1}{1+e^{-0.3775}} = 0.593$
- Carrying out the same process for h2 we get: $out_{h2} = 0.597$



Repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs

+ (plus)

Activation function

Backpropagation

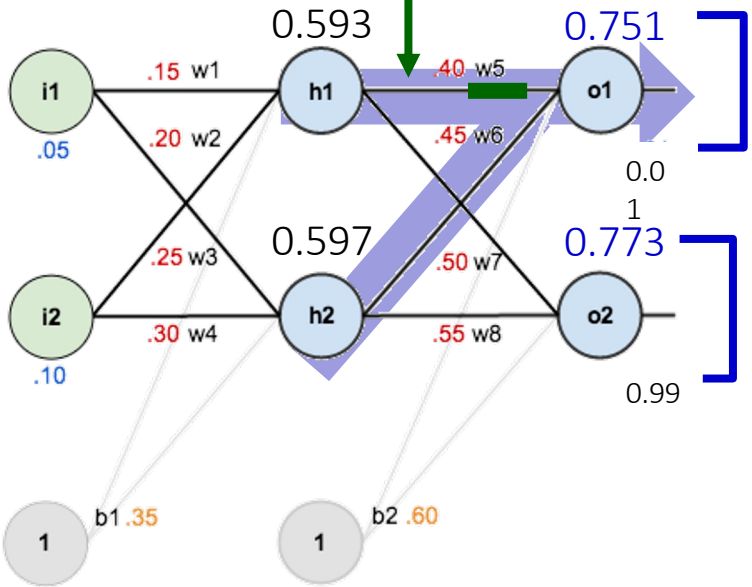
- Here's the output for o1:

$net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1 = 0.4 * 0.593 + 0.45 * 0.597 + 0.6 * 1 = 1.106$

$$out_{o_1} = \frac{1}{1+e^{-net_{o_1}}} = \frac{1}{1+e^{-1.106}} = 0.751$$

- Similarly, $out_{o2} = 0.773$

Q: how to change w5?



- Then calculate the loss function

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{o_1} = \frac{1}{2}(0.01 - 0.751)^2 = 0.275$$

Q: what affects to out_{o1}?

$$E_{o_2} = 0.024$$

$$E_{total} = E_{o_1} + E_{o_2} = 0.298$$

Q: Which output component contributes to error more?

Backward pass

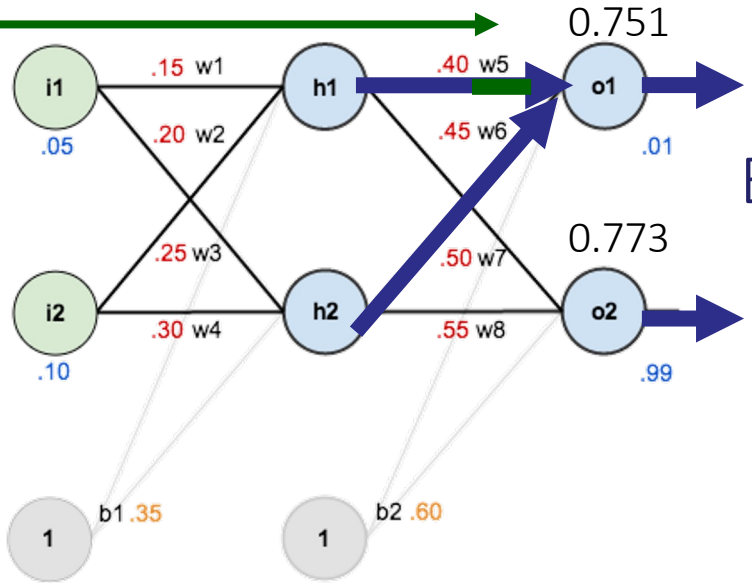
Q: We want to know how much a change in w5 affects the total error?

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} \cdot \frac{\partial out_{o_1}}{\partial net_{o_1}} \cdot \frac{\partial net_{o_1}}{\partial w_5}$$

w_5 affects to E_{total} through o_1 by network + activation function

$$\frac{\partial E_{total}}{\partial out_{o_1}} = -(0.01 - o_1) = -(0.01 - 0.751) = 0.741$$

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = \frac{\partial}{\partial net_{o_1}}\left(\frac{1}{1+e^{-net_{o_1}}}\right) = 0.187$$



$$= \left(1 - \frac{1}{1+e^{-net_{o_1}}}\right)\left(\frac{1}{1+e^{-net_{o_1}}}\right) = (1-0.751)0.751$$

E_{total}

$$E_{o_1} = \frac{1}{2}(0.01 - 0.751)^2 = 0.275$$

out_{o1}

$$E_{o_2} = 0.024$$

$$E_{total} = E_{o_1} + E_{o_2} = 0.298$$

Backward pass

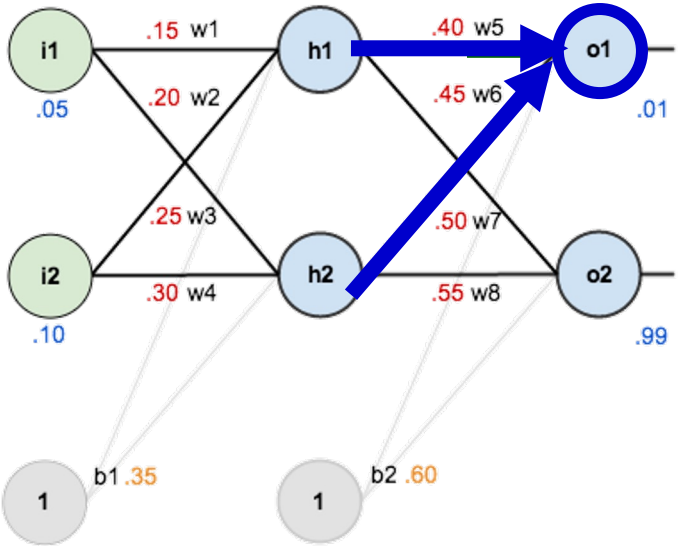
- Finally, how much does the total net input of o1 change with respect to w5?

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} \cdot \frac{\partial out_{o_1}}{\partial net_{o_1}} \cdot \frac{\partial net_{o_1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial out_{o_1}} = -(0.01 - o_1) = -(0.01 - 0.751) = 0.741$$

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = \frac{\partial}{\partial net_{o_1}}\left(\frac{1}{1+e^{-net_{o_1}}}\right) = 0.187$$

Recall, $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$



$$\Rightarrow \frac{\partial net_{o_1}}{\partial w_5} = out_{h_1} = 0.593$$

- Putting all together

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} \cdot \frac{\partial out_{o_1}}{\partial net_{o_1}} \cdot \frac{\partial net_{o_1}}{\partial w_5}$$

$$= 0.741 \cdot 0.187 \cdot 0.593 = 0.082$$

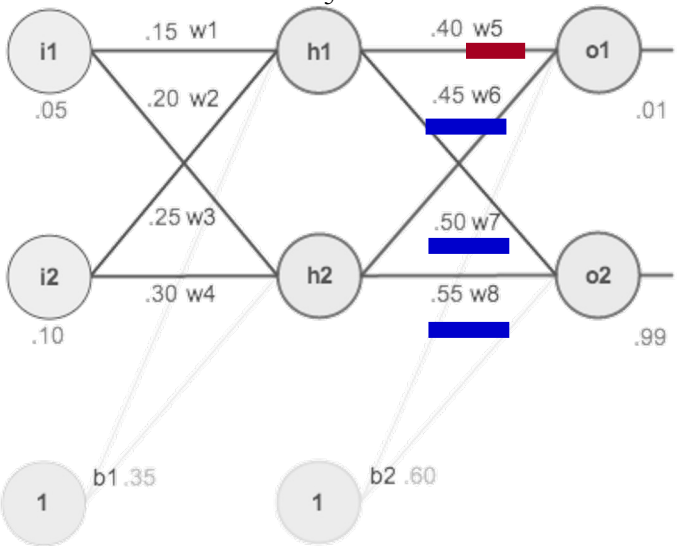
Backward pass

- Update weight: to decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta \cdot \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 \cdot 0.082 = 0.359$$

Reduce weight w5 to reduce error (= loss function)

$$\frac{\partial E_{total}}{\partial w_5} = 0.082$$



We can repeat this process to get the new weights w6, w7, and w8:

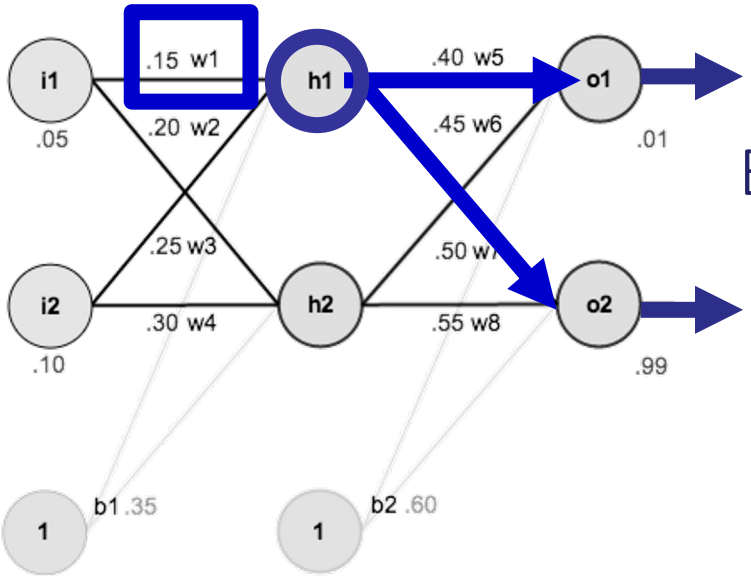
$$w_6^+ = 0.409$$
$$w_7^+ = 0.511$$
$$w_8^+ = 0.561$$

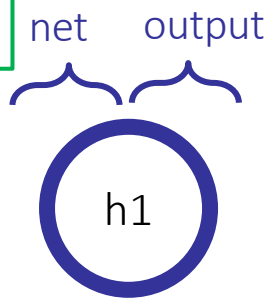
Hidden layer

- Next, we'll continue the backwards pass by calculating new values for w1, w2, w3, w4.

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} \cdot \frac{\partial out_{h_1}}{\partial net_{h_1}} \cdot \frac{\partial net_{h_1}}{\partial w_1} \quad \Rightarrow \frac{\partial E_{total}}{\partial w_1} = 0.00044$$

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} \cdot \frac{\partial E_{o_2}}{\partial out_{h_1}}$$





E_{total}

Hidden layer

- We can now update all of our weights w1, w2, w3, w4.

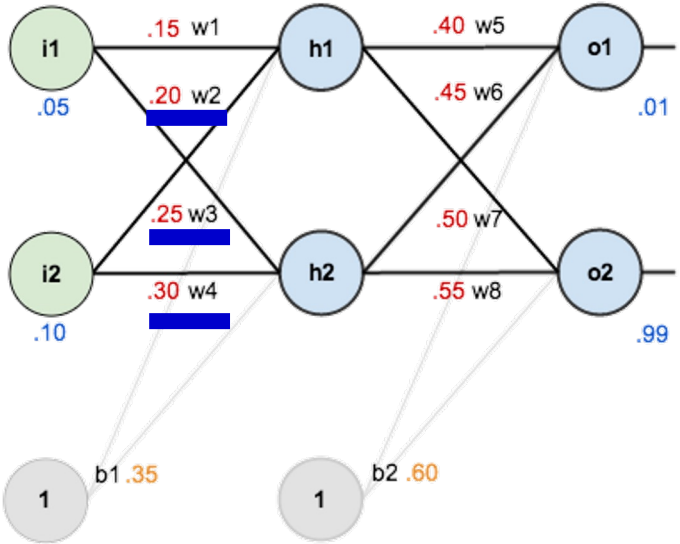
$$w_1^+ = w_1 - \eta \cdot \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 \cdot 0.00044 = 0.1497$$

$$w_2^+ = 0.1996$$

$$w_3^+ = 0.2498$$

$$w_4^+ = 0.2995$$

Reduce weight w1 (very slightly) to reduce error (= loss function)

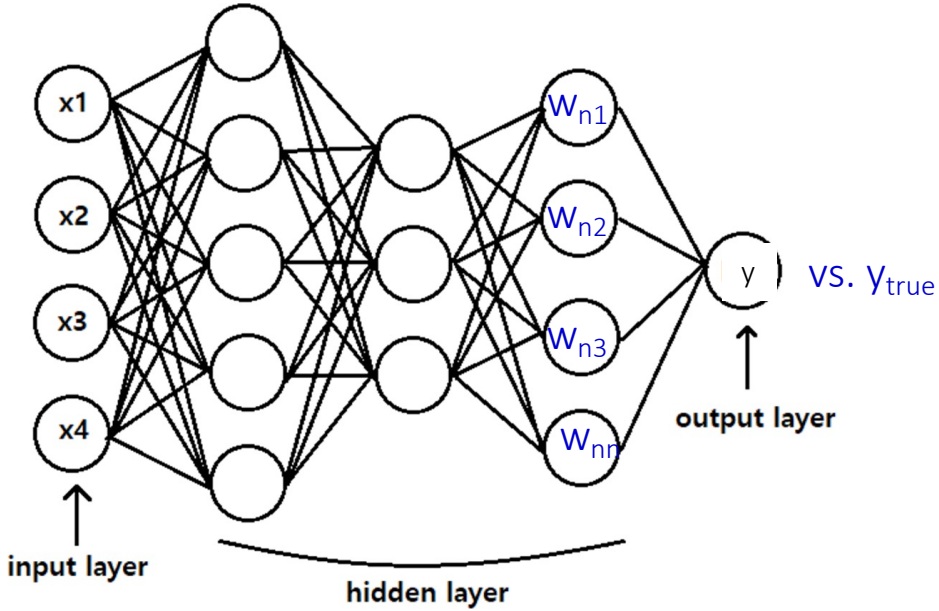


- Original network error E_{total} = 0.298
- After the first round of backpropagation, E_{total} = 0.291

It seems to be very small change, but after training (repeating) this 10^4 times (with different batches), for example, the error plummets to 0.0000351085.

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

Recall, Backpropagation

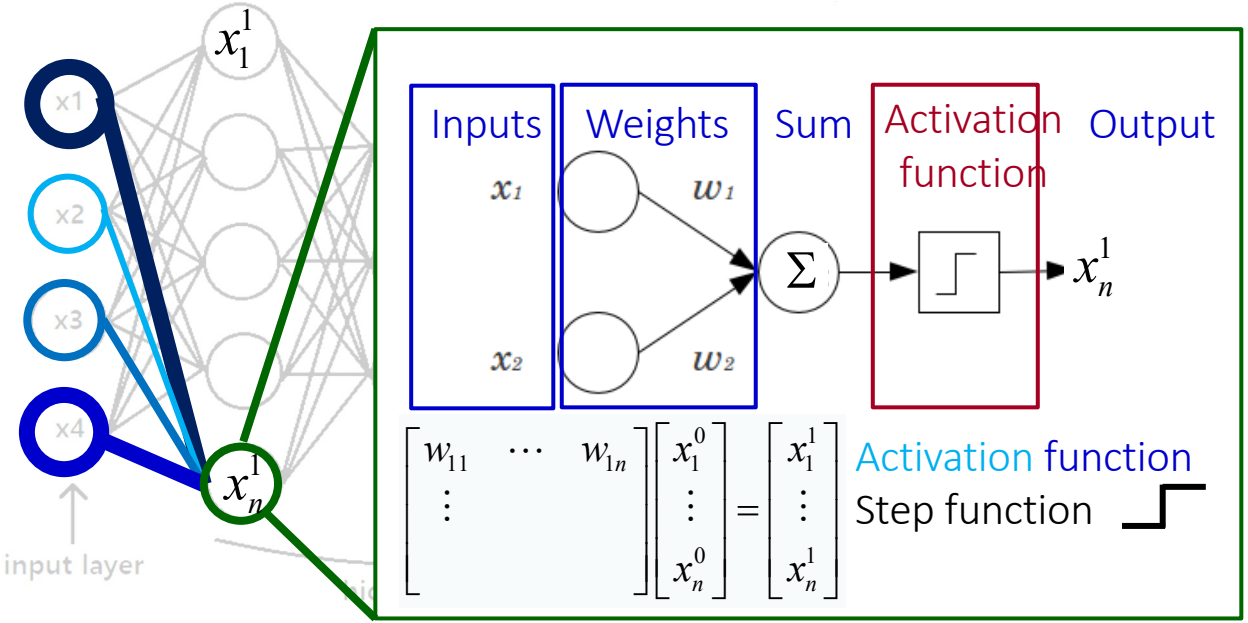


$$w' = w - \eta \nabla f(w)$$

- Derivative of output errors
- Output is a weighted sum of inputs with activation function

Images from google

Activation function in NN

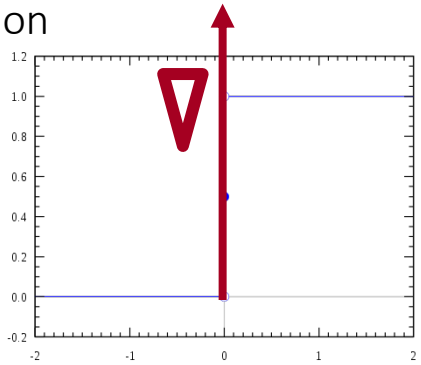


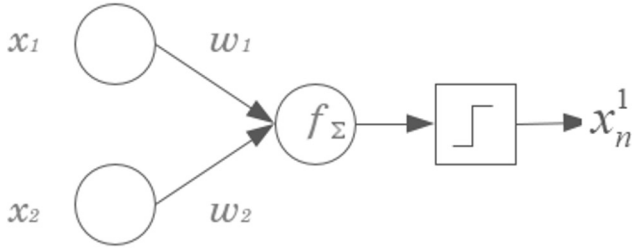
$$w' = w - \eta \nabla f(w)$$

- Derivative of output errors
- Output is a weighted sum of inputs with activation function

Activation function

Step function



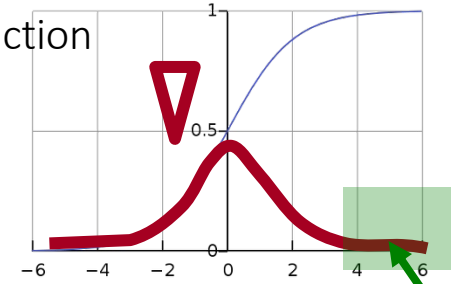


$$x_n^1 = s\left(w_{1,1}x_1^0 + w_{1,2}x_2^0 + \cdots + w_{1,n}x_n^0 \right) + b_1$$

Bias

Activation function, e.g. logistic

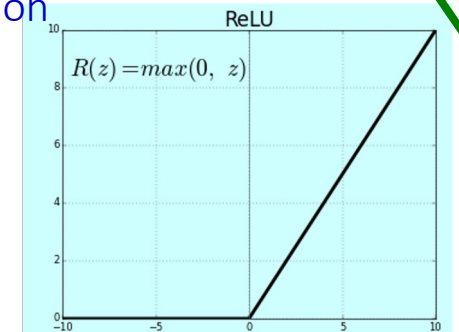
Logistic function



To update the weight w, take a derivative of f, i.e., activation function

$$w' = w - \eta \nabla f(w)$$

ReLU function



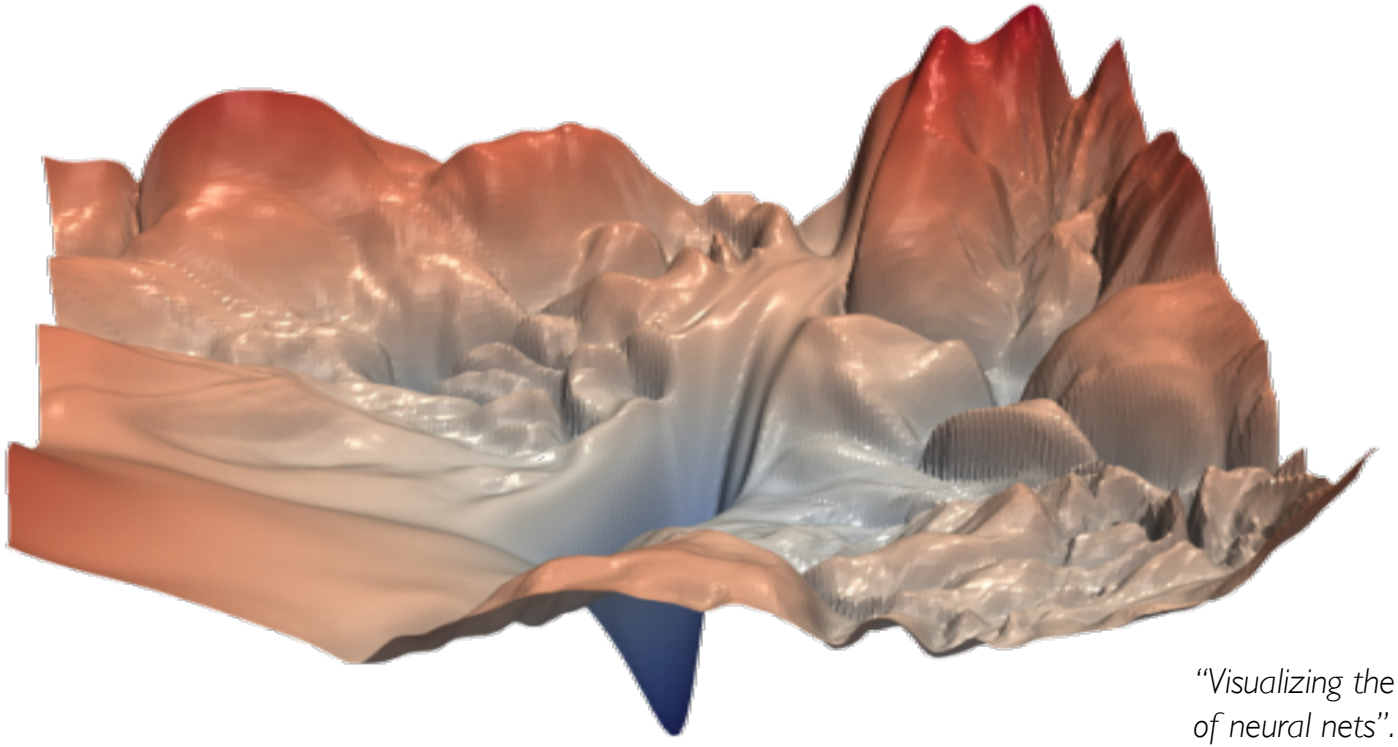
- Update of the weight (~ proportional to the partial derivative ∇ of the error function) would not occur if the gradient will be vanishingly small. This ⇒ 'Vanishing gradient' problem network from further training.

Images from google

Neural Networks in Practice: Optimization

Lecture note from MIT 6.S191 Introduction to Deep Learning

Training Neural Networks is Difficult



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

Loss Functions Can Be Difficult to Optimize

Remember:

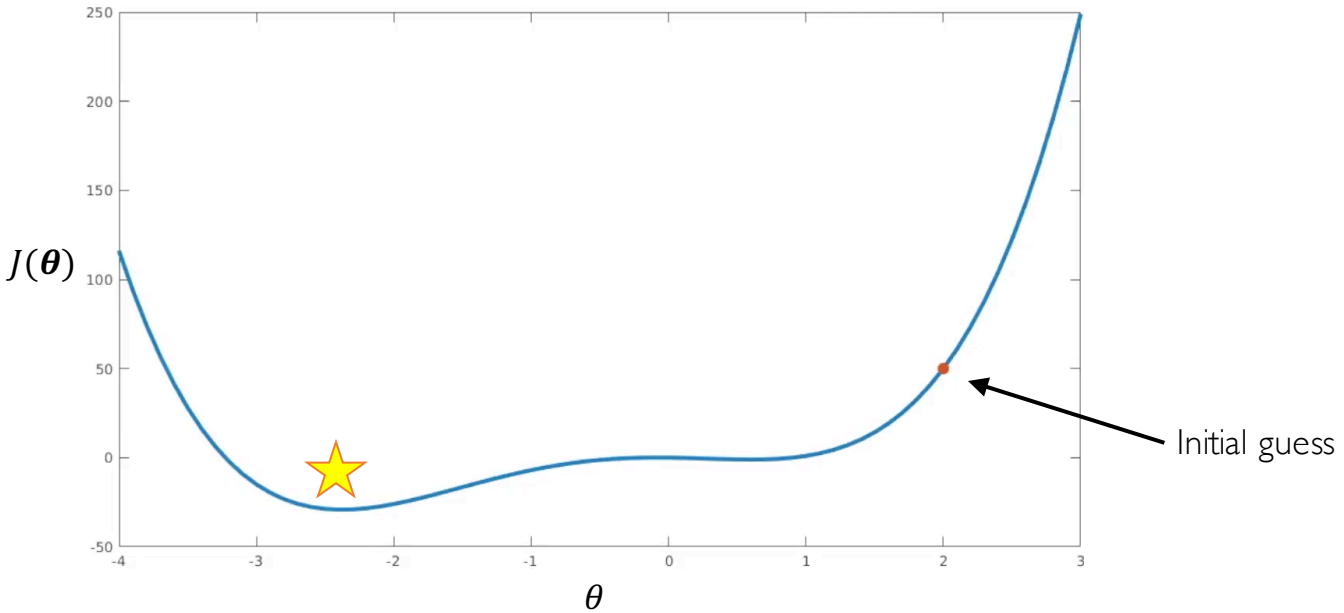
Optimization through gradient descent

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

How can we set the learning rate?

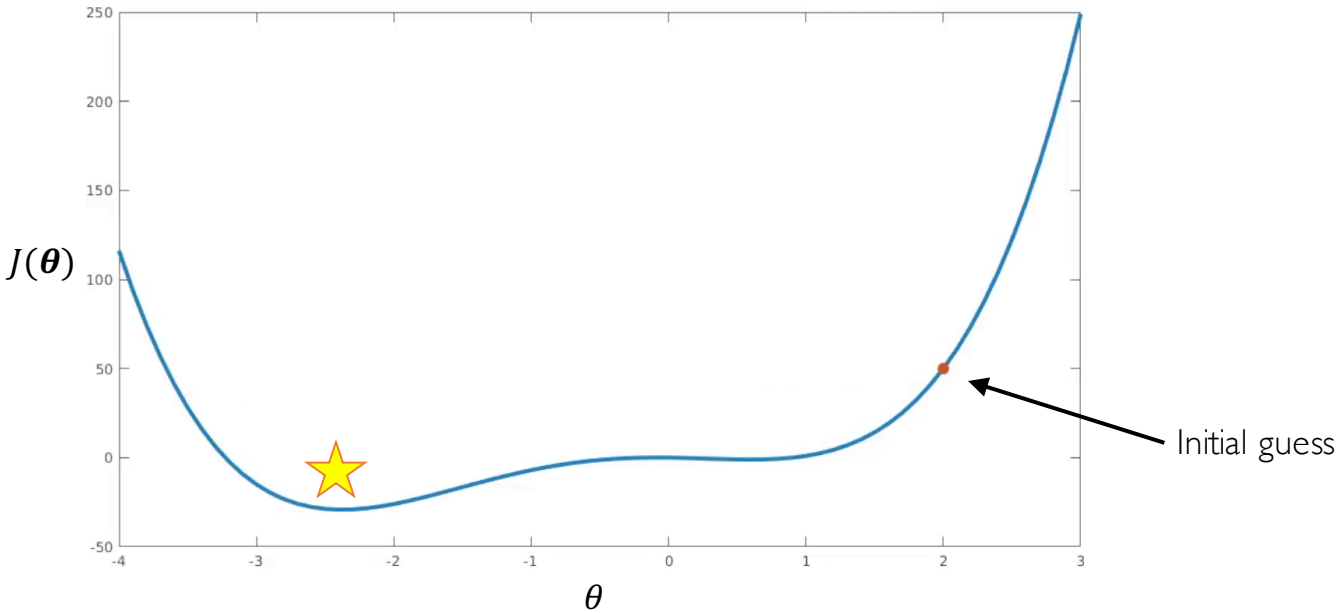
Setting the Learning Rate

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



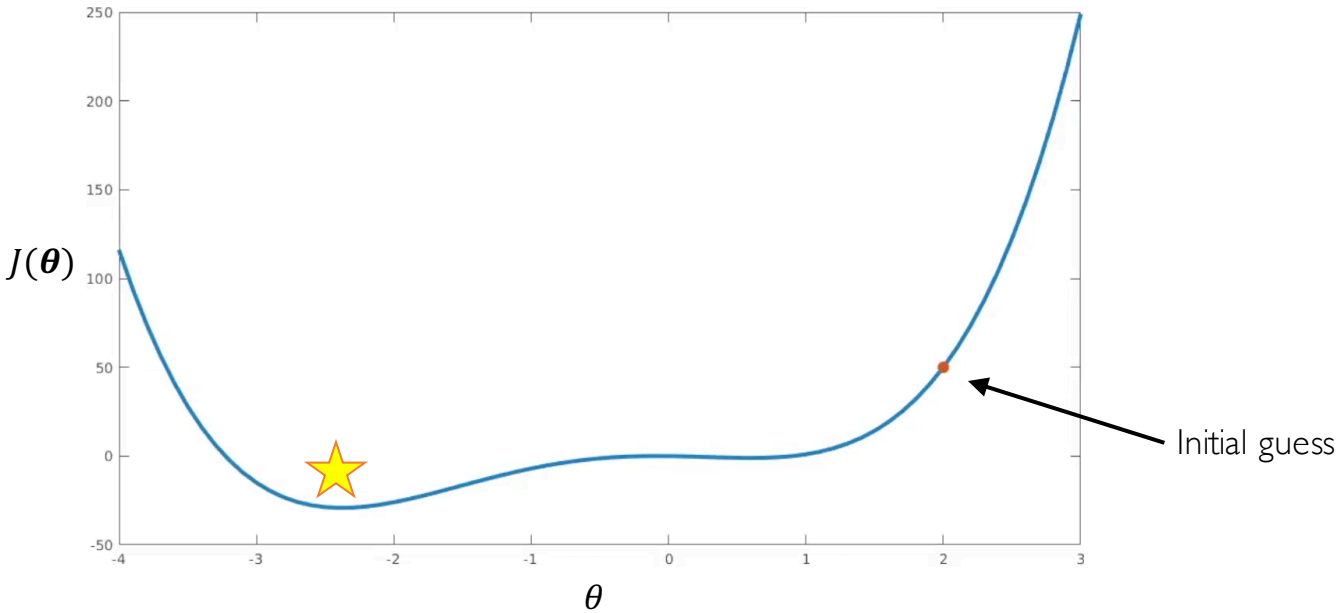
Setting the Learning Rate

Large learning rates *overshoot, become unstable and diverge*



Setting the Learning Rate

Stable learning rates *converge smoothly and avoid local minima*



How to deal with this?

Idea 1:

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

How to deal with this?

Idea 1:

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

Idea 2:

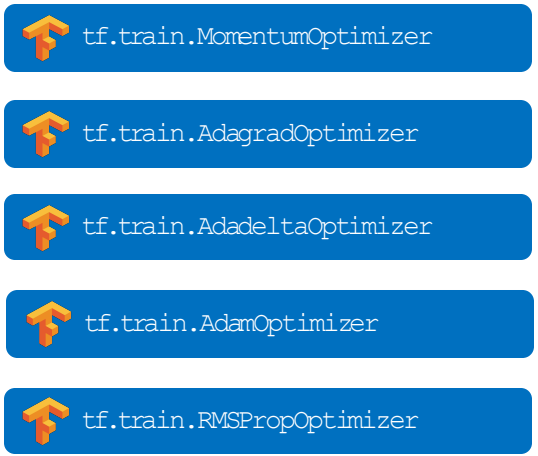
Do something smarter!
Design an adaptive learning rate that "adapts" to the landscape

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...

Adaptive Learning Rate Algorithms

- 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: http://ruder.io/optimizing-gradient-descent/

Neural Networks in Practice: Mini-batches

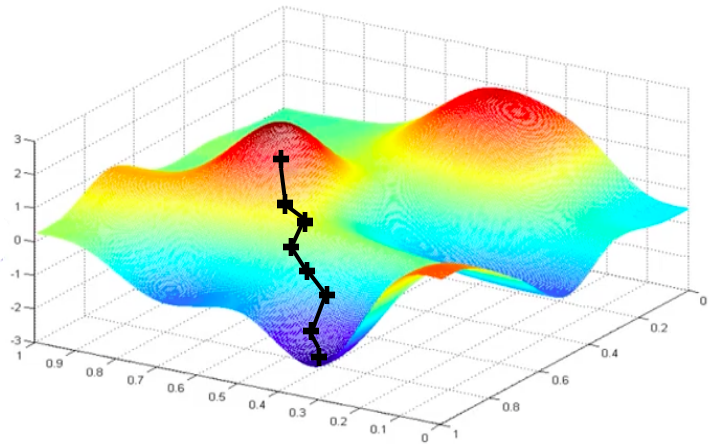
Lecture note from MIT 6.S191 Introduction to Deep Learning

Gradient Descent

Algorithm

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

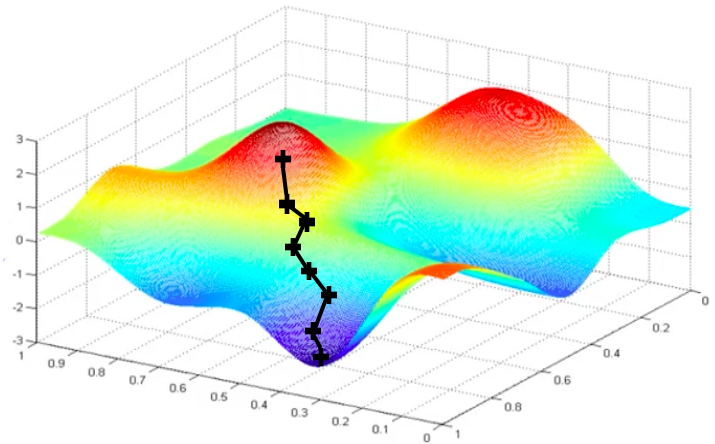
Can be very computational to compute!



Stochastic Gradient Descent

Algorithm

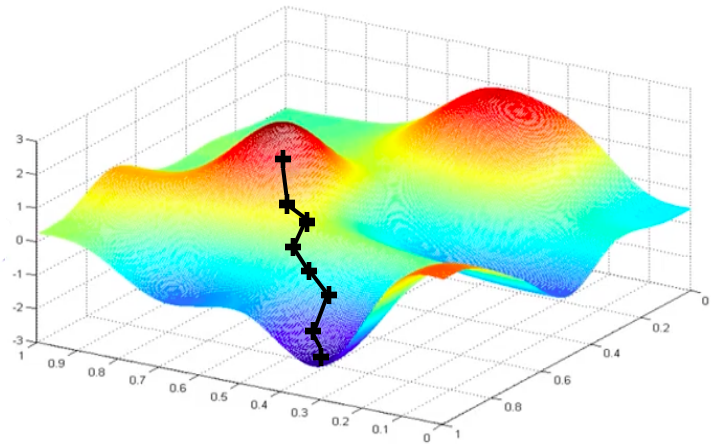
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights



Stochastic Gradient Descent

Algorithm

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

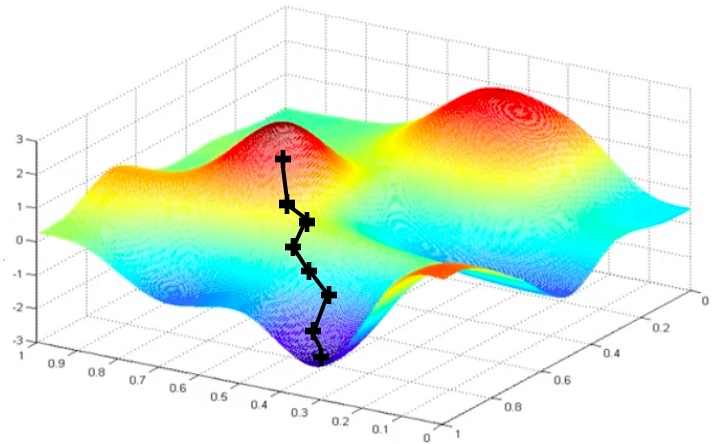


Easy to compute but
very noisy
(stochastic)!

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



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

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

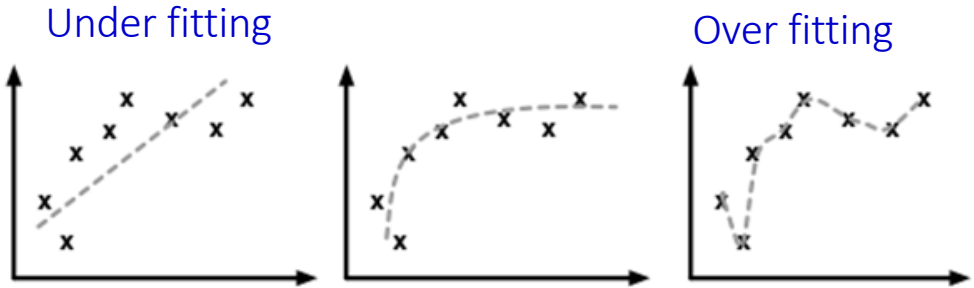
Mini-batches lead to fast training!

Can parallelize computation + achieve significant speed increases on GPU's

Neural Networks in Practice: Overfitting

Lecture note from MIT 6.S191 Introduction to Deep Learning

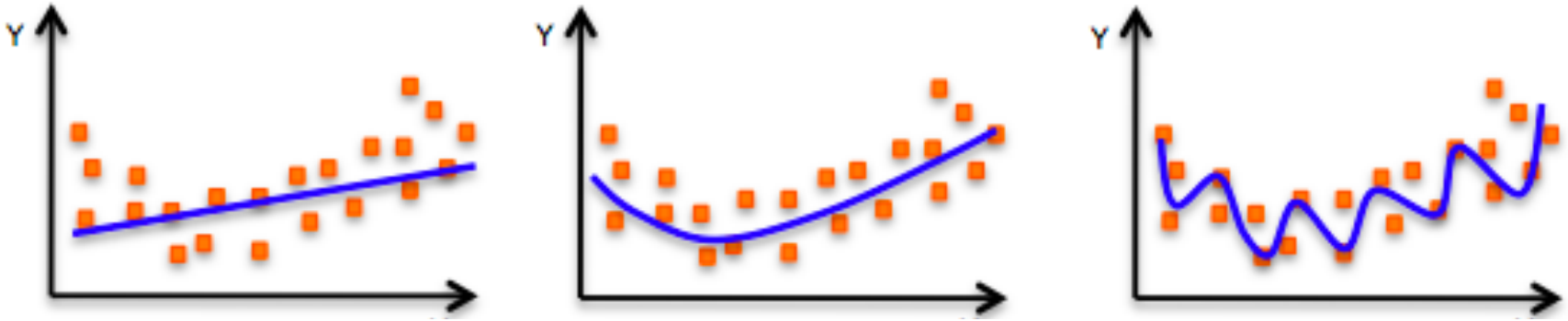
Overfitting with Training Data



These are not cats because it's different from what has been trained

Q: How to avoid 'overfitting'?

The Problem of Overfitting

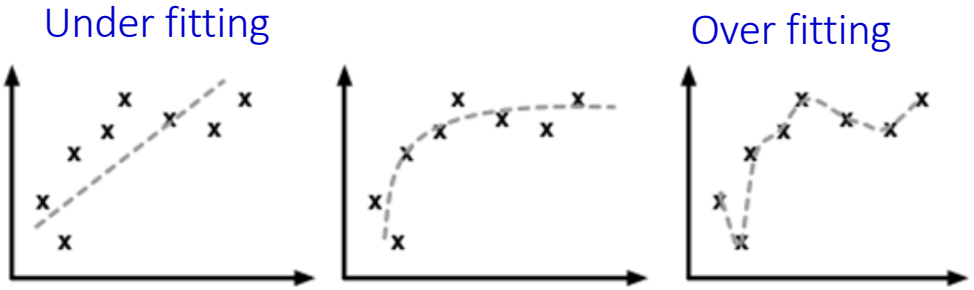


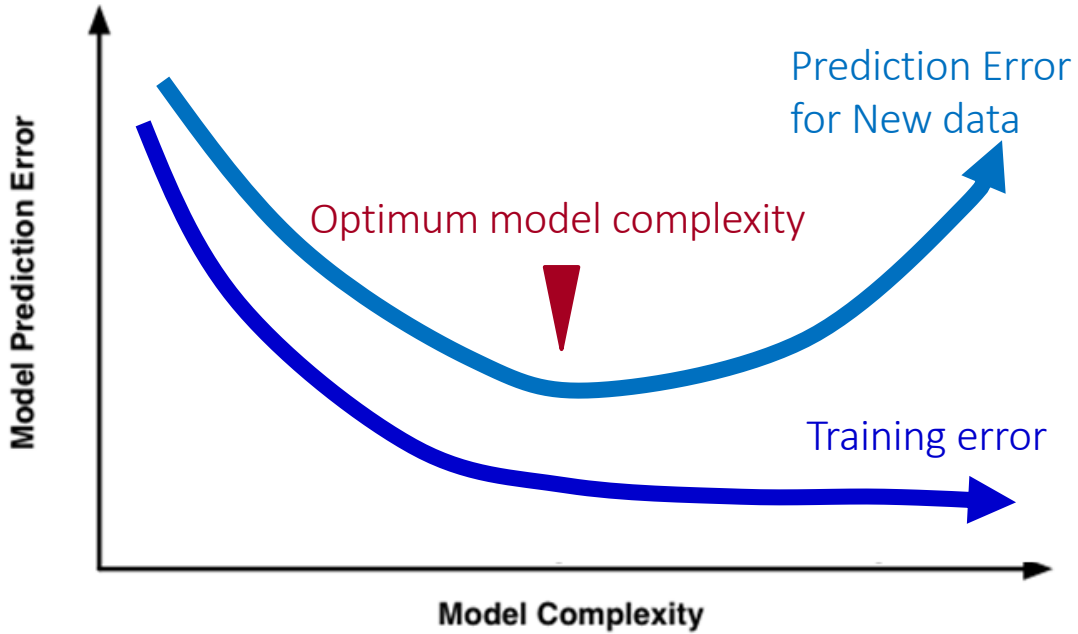
Underfitting
Model does not have capacity to fully learn the data

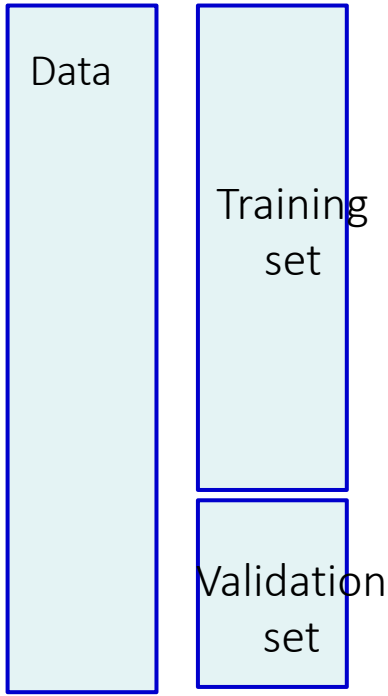
Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

Training & Test (and Validation)



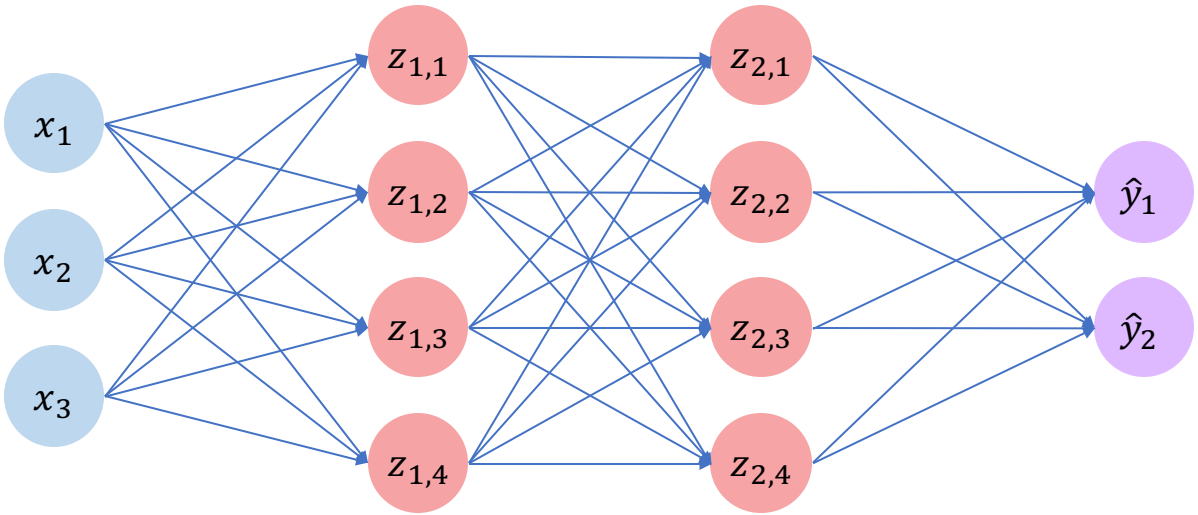




https://www.quora.com/What-is-generalization-in-machine-learning

Regularization 1: Dropout

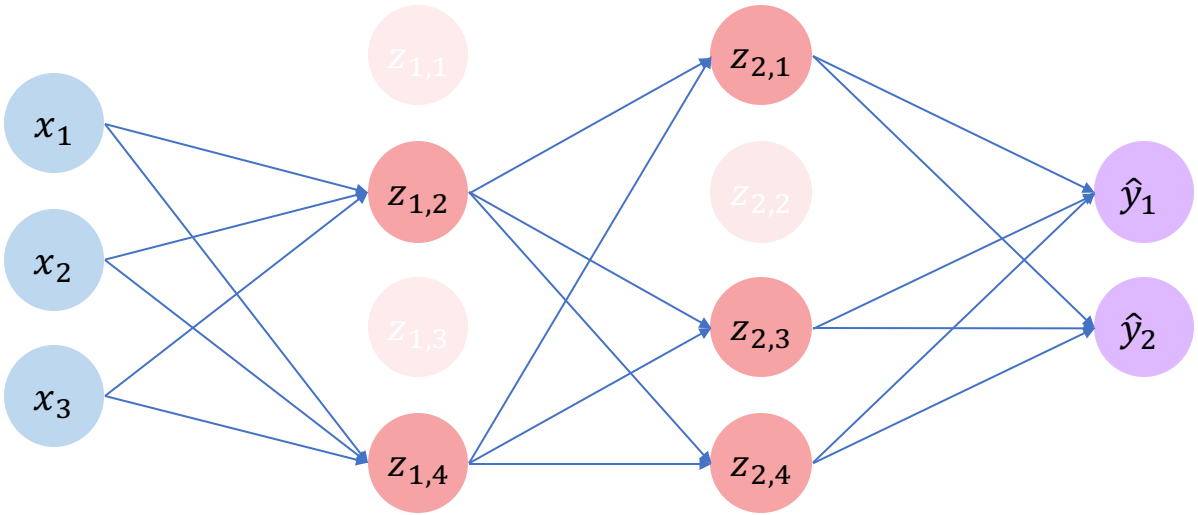
- During training, randomly set some activations to 0



Regularization 1: 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

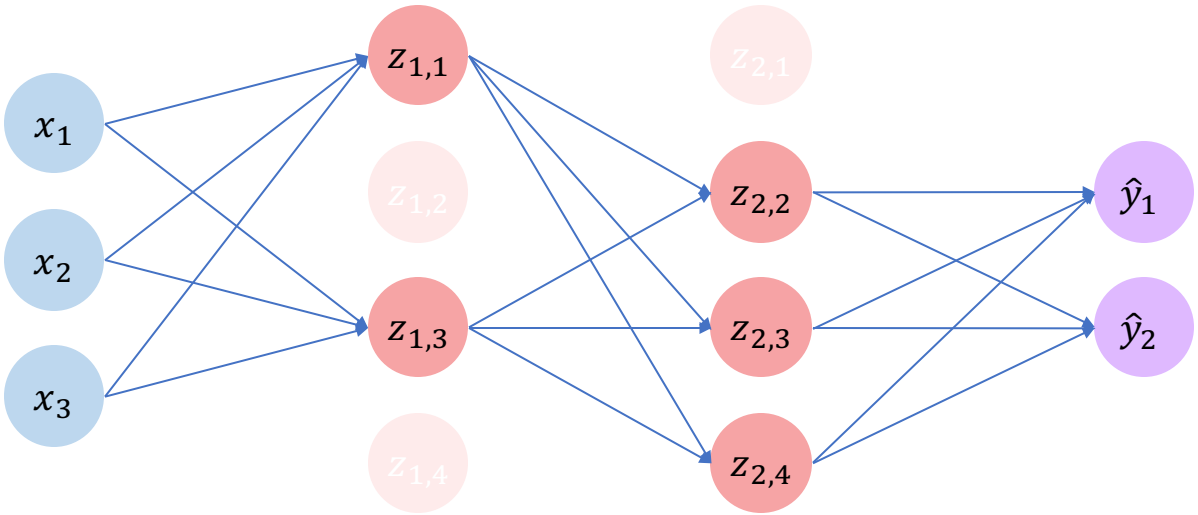
```
tf.nn.dropout(hiddenLayer, p=0.5)
```



Regularization 1: Dropout

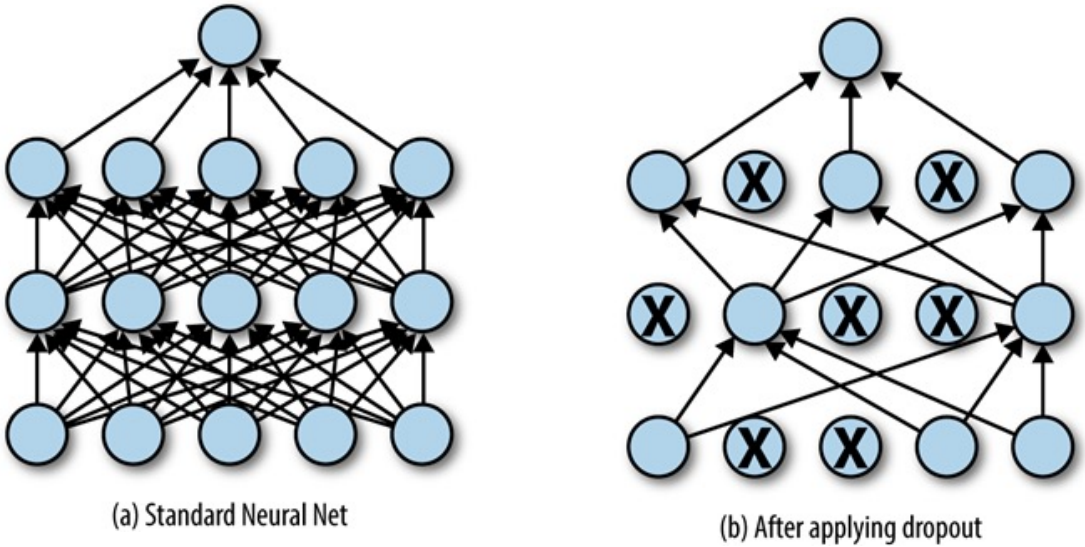
- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
tf.nn.dropout(hiddenLayer, p=0.5)
```



Dropout

- **Dropout** is a form of regularization that randomly drops some proportion of the nodes that feed into a fully connected layer.
- This helps prevent the net from relying on one node in the layer too much.
- Here, dropping a node means that its contribution to the corresponding activation function is set to 0. Since there is no activation contribution, the gradients for dropped nodes drop to zero as well.



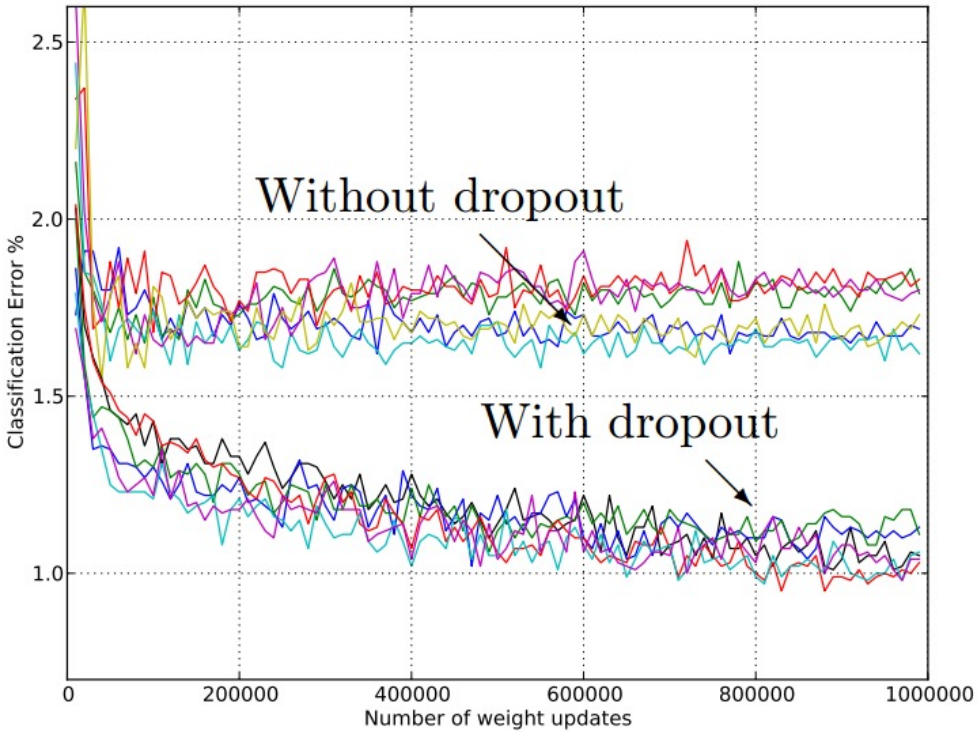
(a) Standard Neural Net

(b) After applying dropout

https://www.safaribooksonline.com/library/view/tensorflow-for-

Dropout

- Bellow we have a classification error (not including loss), observe that the test/validation error is smaller using dropout



https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/dropout_layer.html

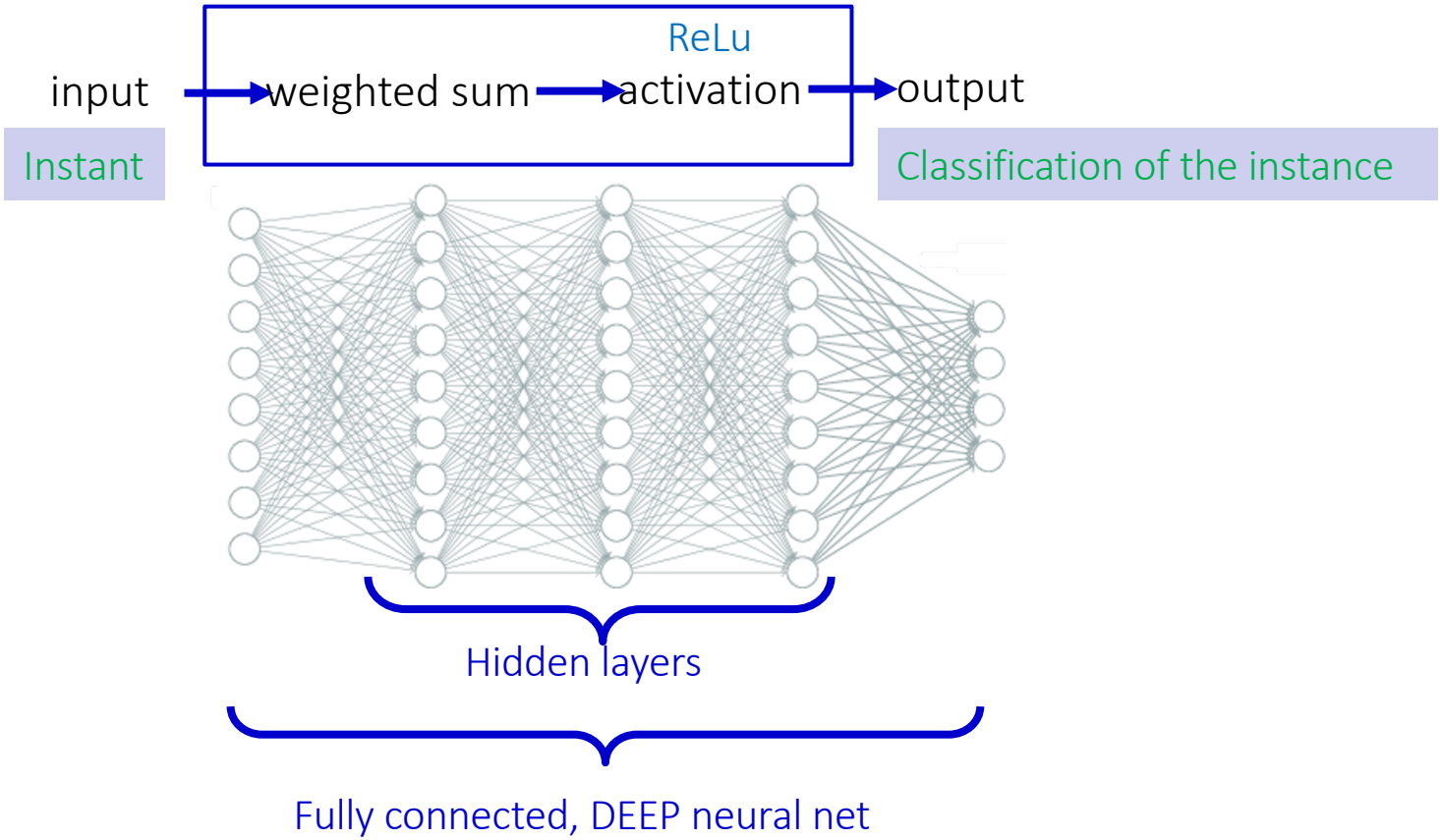
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

CONVOLUTIONAL NEURAL NETWORK

KAIST

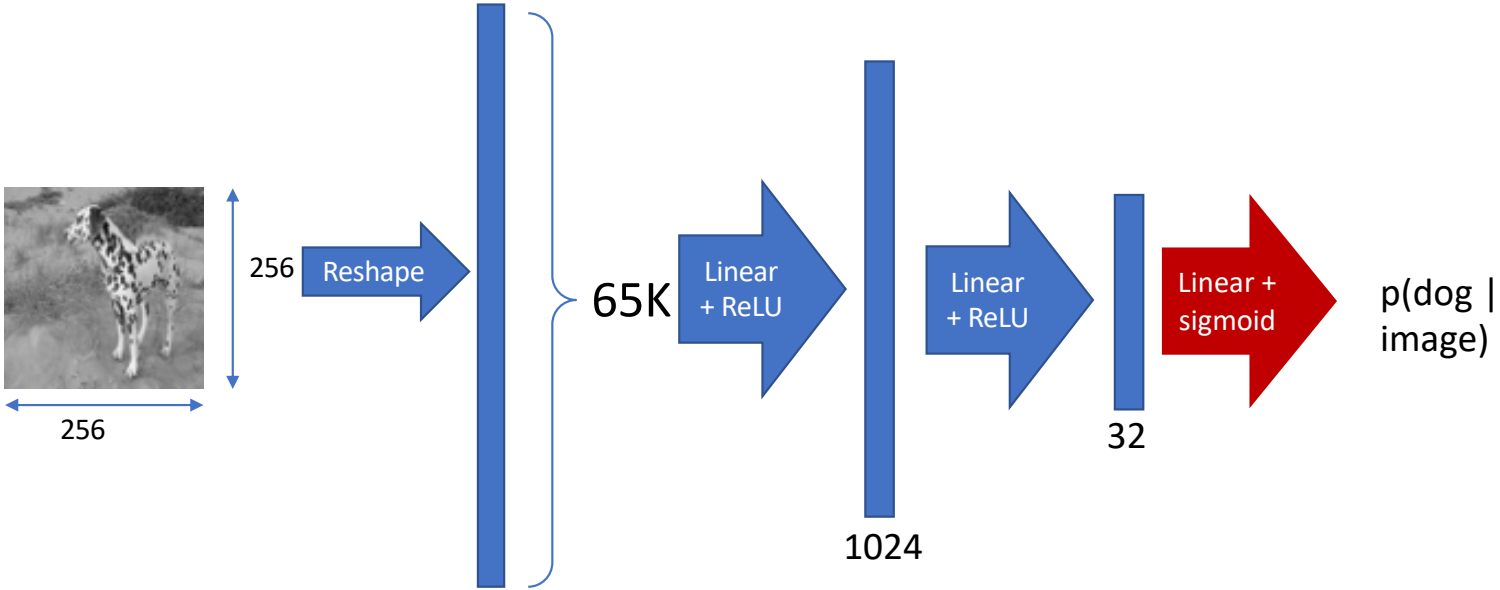
Artificial NN, i.e., fully connected NN



Images from google

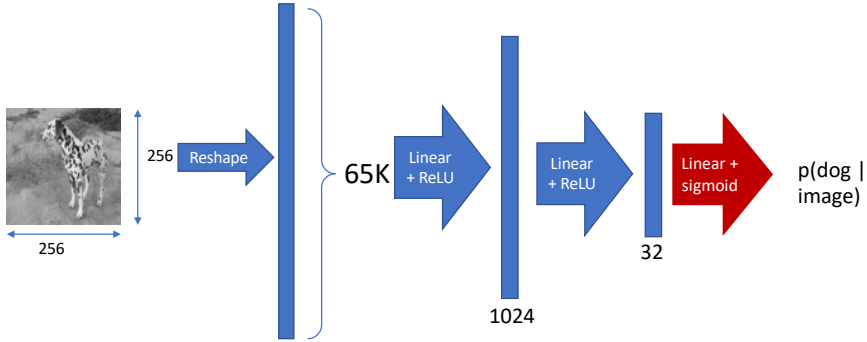
Multilayer perceptron on images

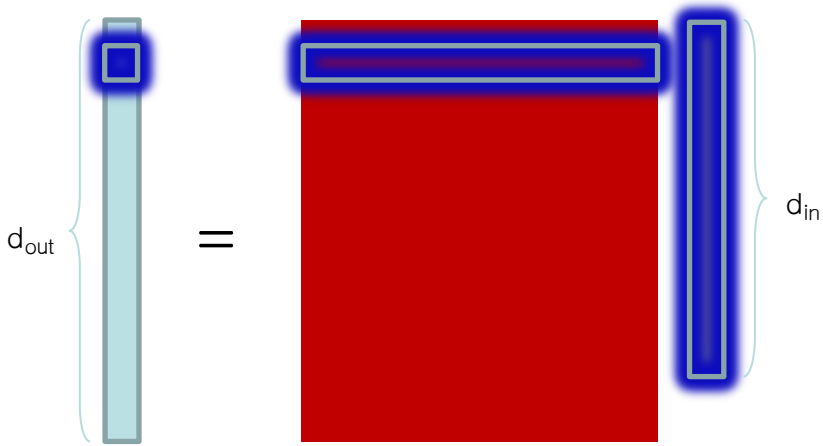
- An example network for cat vs dog



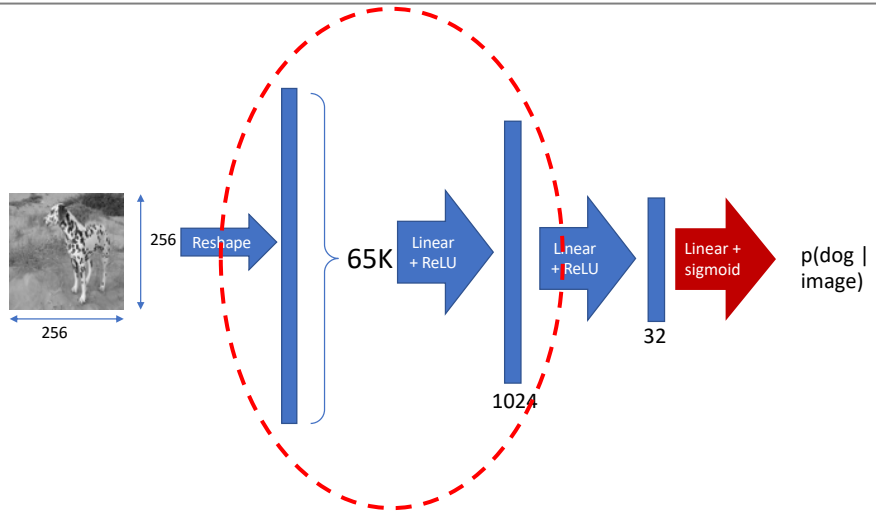
The linear function for images

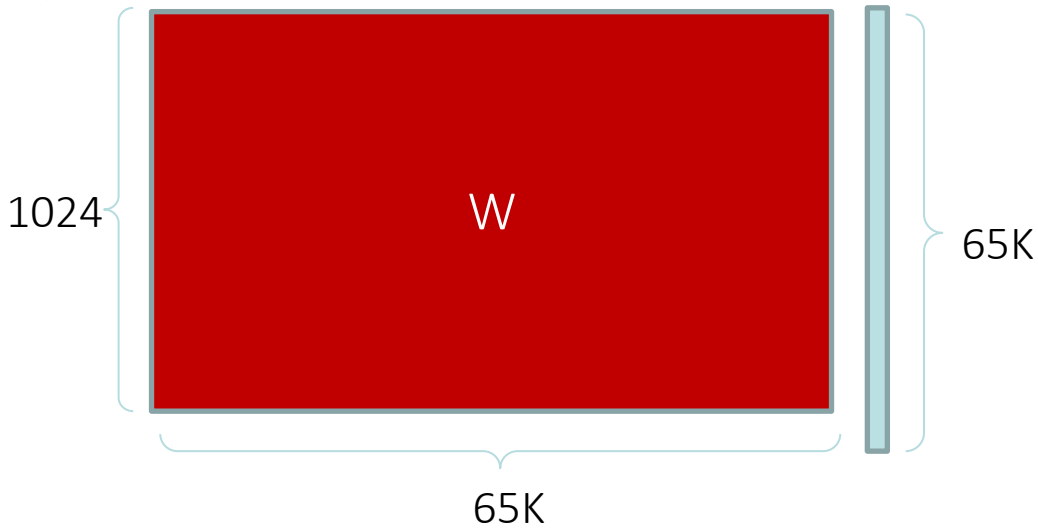
- y = Wx + b
- How many parameters does a linear function have?





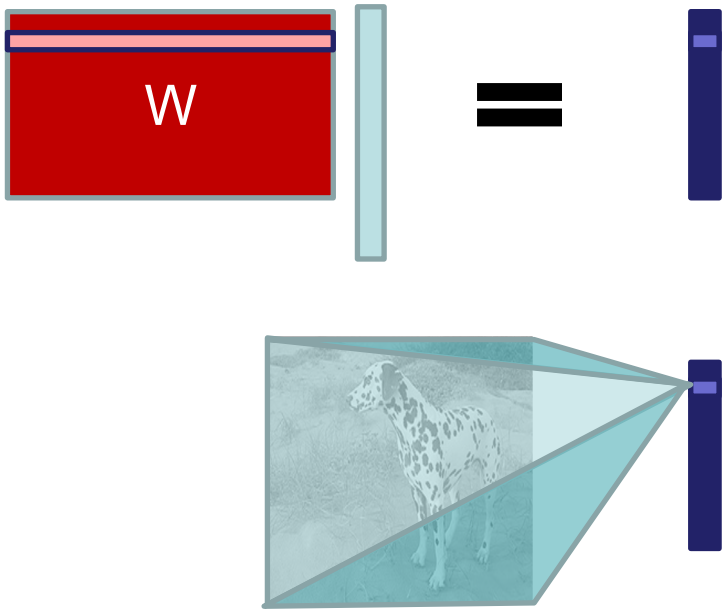
The linear function for images





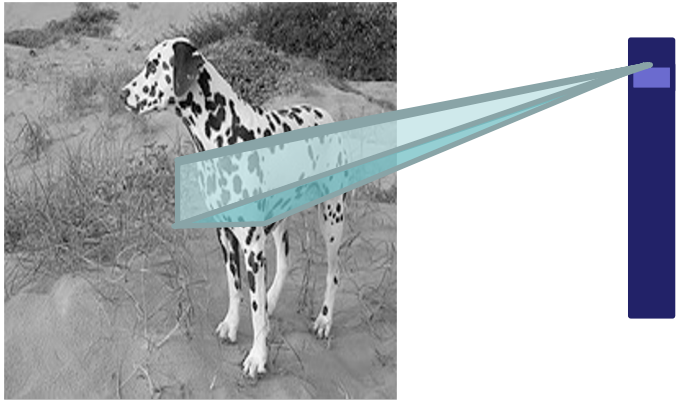
Reducing parameter count

- A single “pixel” in the output is a weighted combination of *all* input pixels

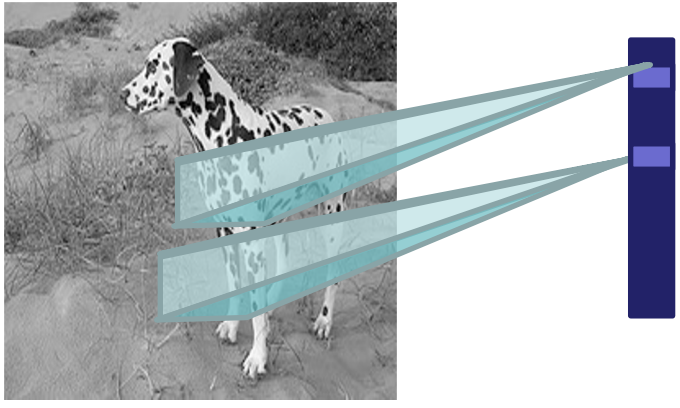


Reducing parameter count

- Idea 1: local connectivity - Pixels only related to nearby pixels



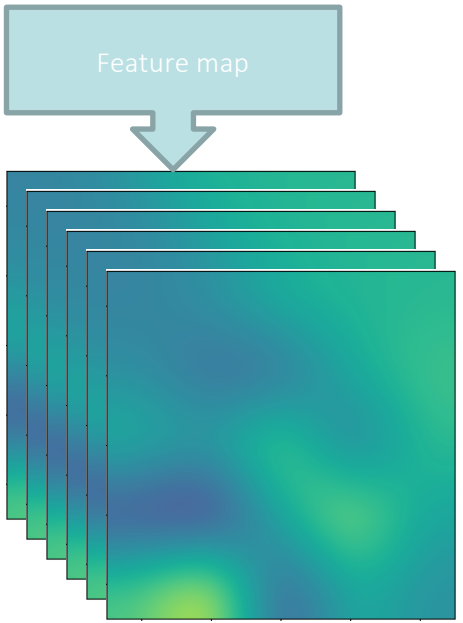
- Idea 2: Translation invariance - Pixels share the same weights
  - Weights should not depend on the location of the neighborhood



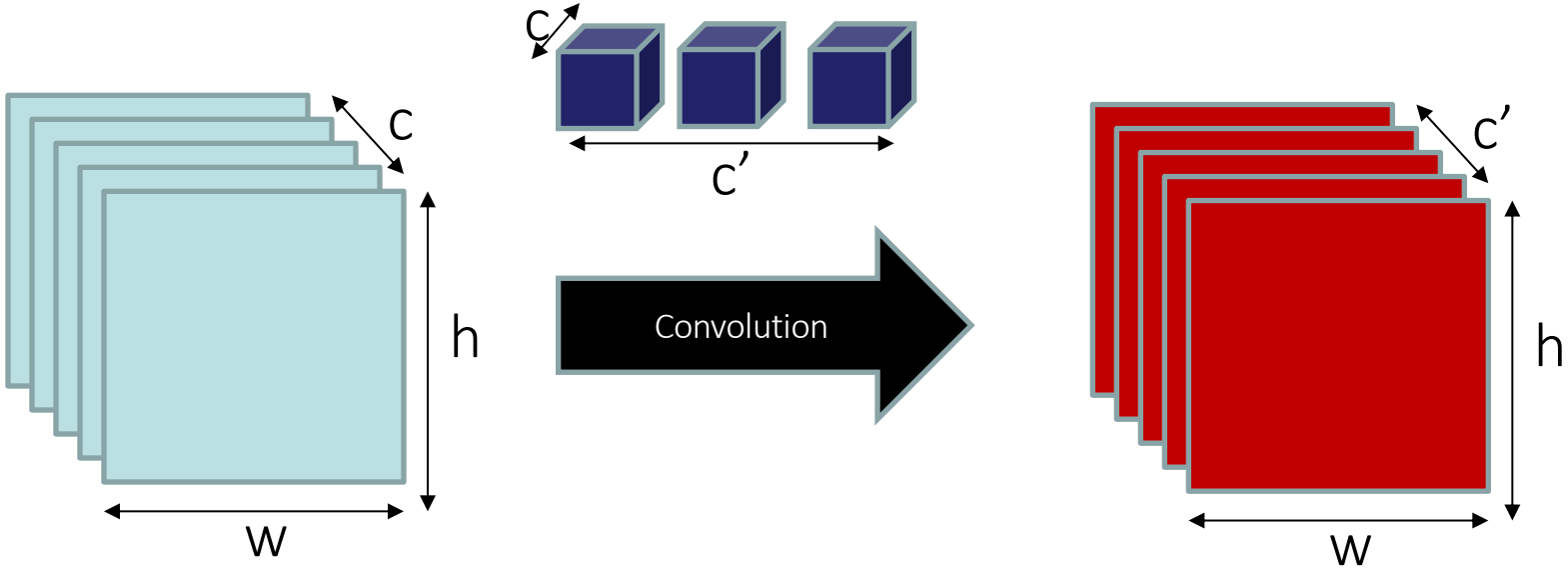
Local connectivity + translation invariance = convolution

- Local connectivity determines the kernel size
- Running multiple filters gives *multiple feature maps*
- Each feature map is a *channel* of the output

5.4	0.1	3.6
1.8	2.3	4.5
1.1	3.4	7.2



Convolution as a primitive



Convolution

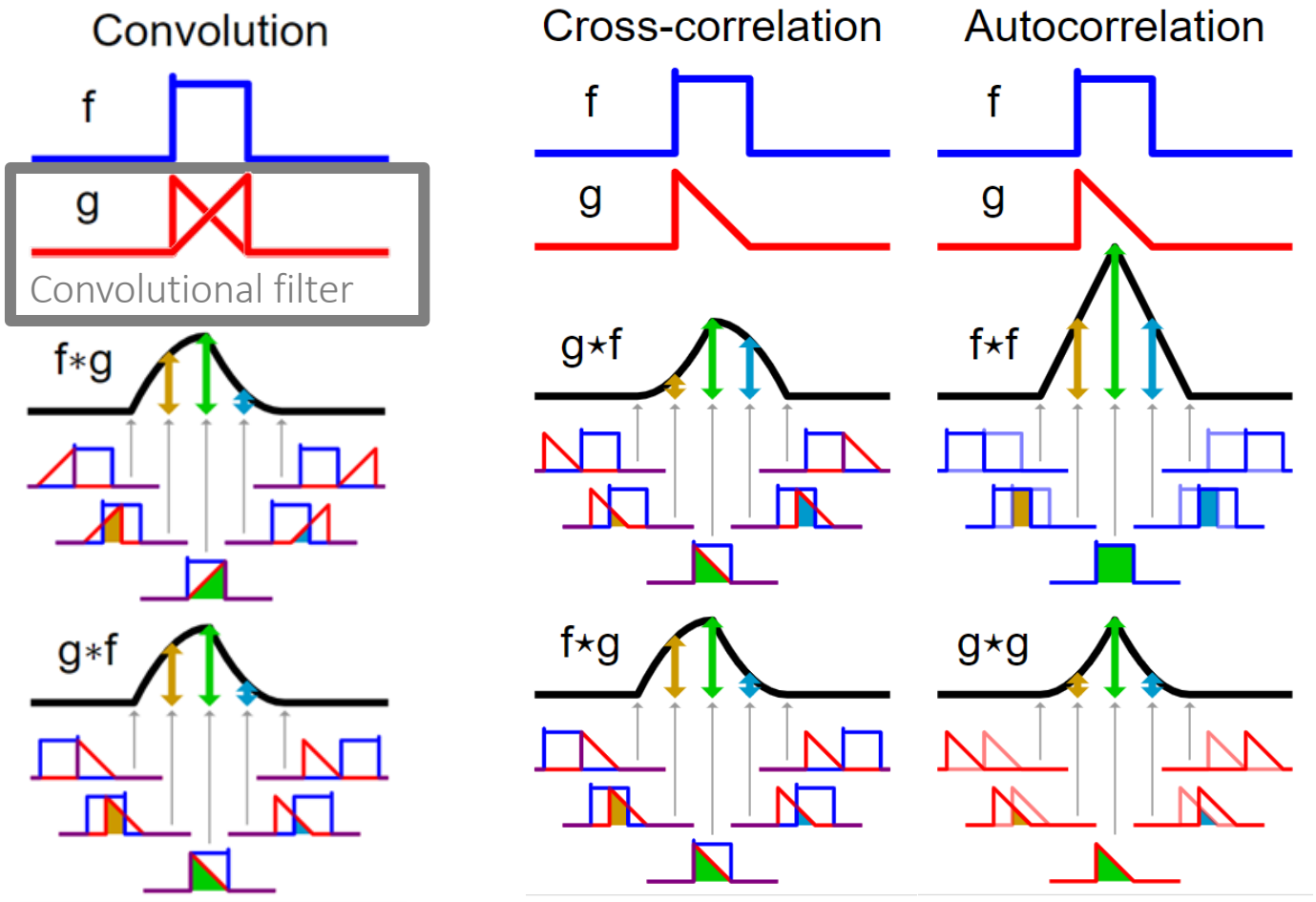
Convolution

- Convolve 감다; 감기다; 휘감다; 둘둘 말다[감다]; 빙빙 돌다
- Convolution
  - 대단히 복잡한[난해한] 것 ex) the bizarre convolutions of the story
  - (나선형의) 주름[구불구불한 것] the convolutions of the brain

$$(f * g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau = \int_{-\infty}^{\infty} f(t-\tau)\, g(\tau)\, d\tau$$

- A mathematical operation on two functions (f and g) to produce a third function that expresses how the shape of one is modified by the other

Convolution



[Wikipedia]

Convolution for 1D continuous signals

Definition of filtering as convolution:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy$$

notice the flip

filtered signal

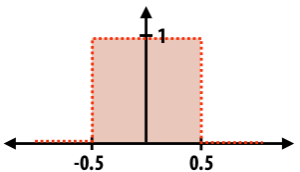
filter

input signal

Consider the box filter example:

1D continuous box filter

$$f(x) = \begin{cases} 1 & |x| \leq 0.5 \\ 0 & otherwise \end{cases}$$



filtering output is a blurred version of g

$$(f * g)(x) = \int_{-0.5}^{0.5} g(x-y)dy$$

Convolution for 2D discrete signals

Definition of filtering as convolution:

notice the flip

$$(f * g)(x, y) = \sum_{i,j=-\infty}^{\infty} f(i, j) I(x - i, y - j)$$

filtered image — filter — input image

If the filter $f(i, j)$ is non-zero only within $-1 \leq i, j \leq 1$, then

$$(f * g)(x, y) = \sum_{i,j=-1}^{1} f(i, j) I(x - i, y - j)$$

The kernel we saw earlier is the 3x3 matrix representation of $f(i, j)$.

Convolution vs correlation

Definition of discrete 2D convolution:

$$(f * g)(x, y) = \sum_{i,j=-\infty}^{\infty} f(i, j) I(x - i, y - j)$$

notice the flip

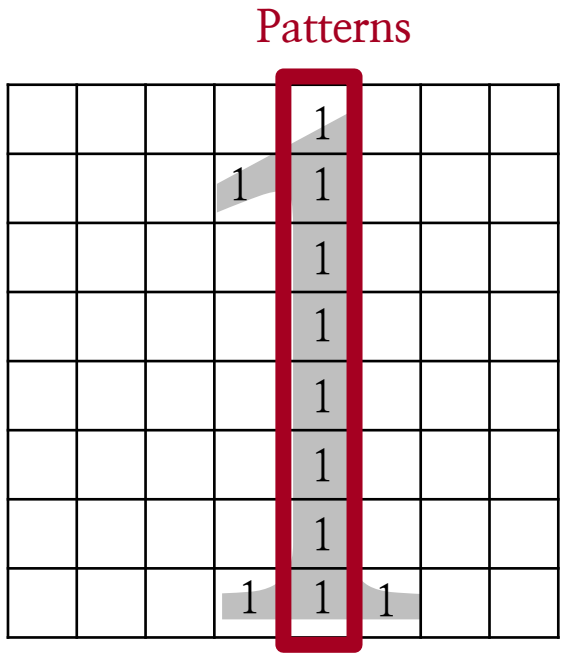
Definition of discrete 2D correlation:

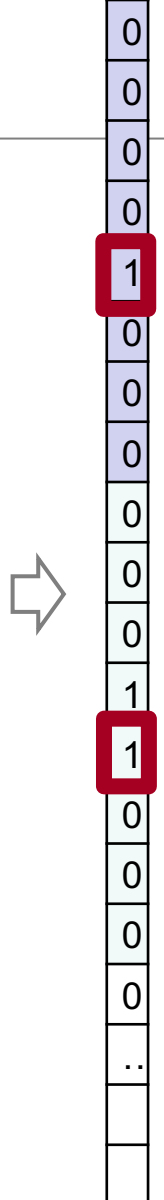
$$(f * g)(x, y) = \sum_{i,j=-\infty}^{\infty} f(i, j) I(x + i, y + j)$$

notice the lack of a flip

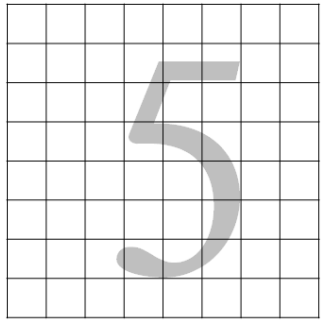
- Most of the time won't matter, because our kernels will be symmetric.

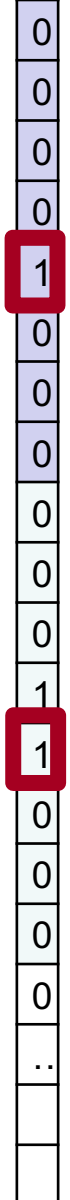
Patterns of an image

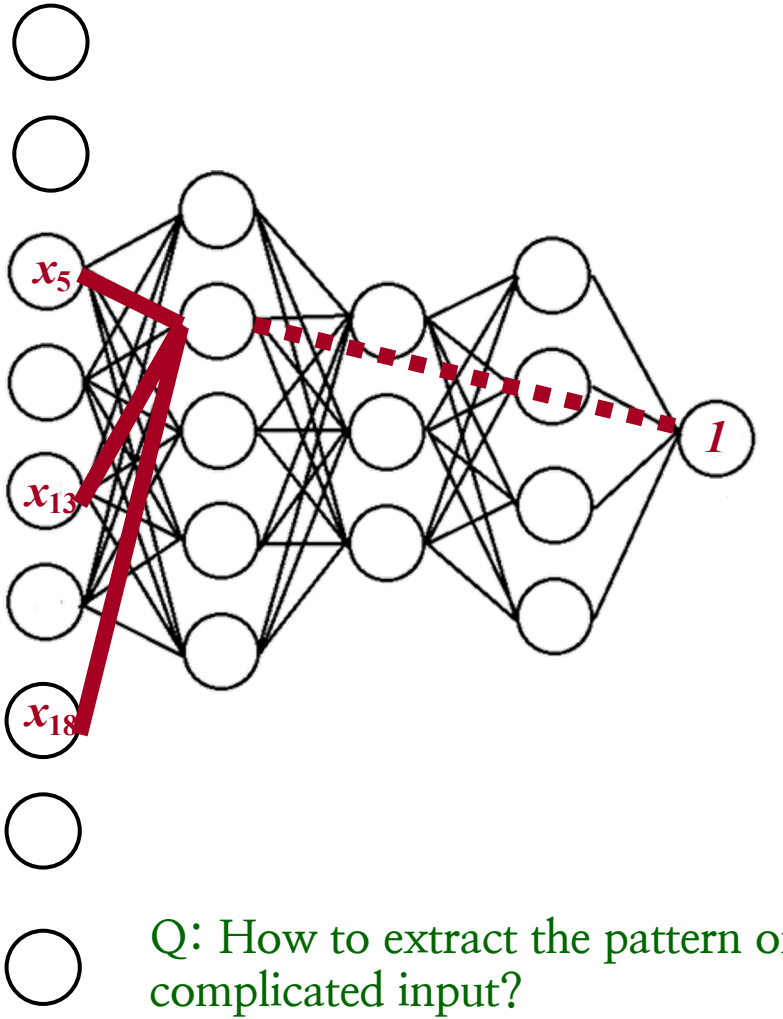




Patterns



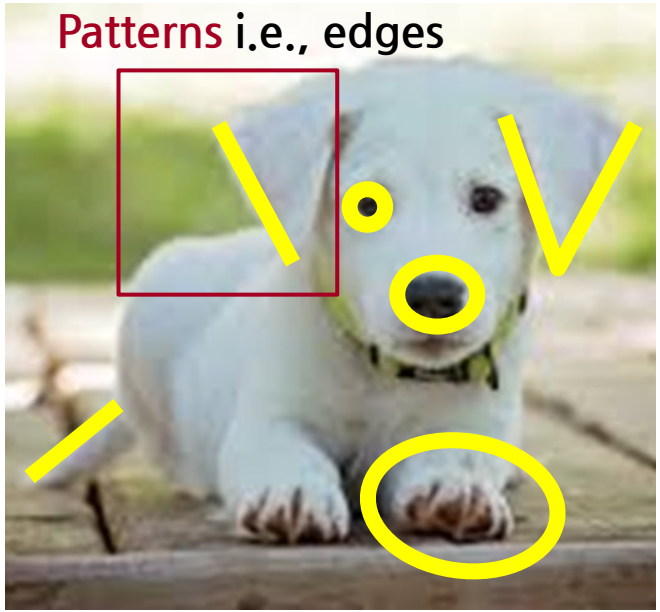




Q: How to extract the pattern of complicated input?

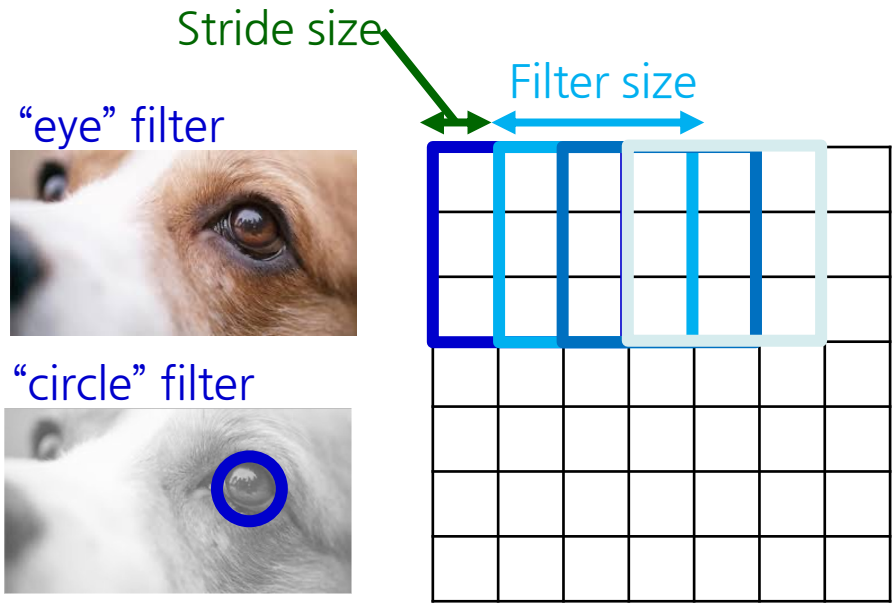
Patterns of the image

Apply multiple convolutional filters to extract feature map like edges

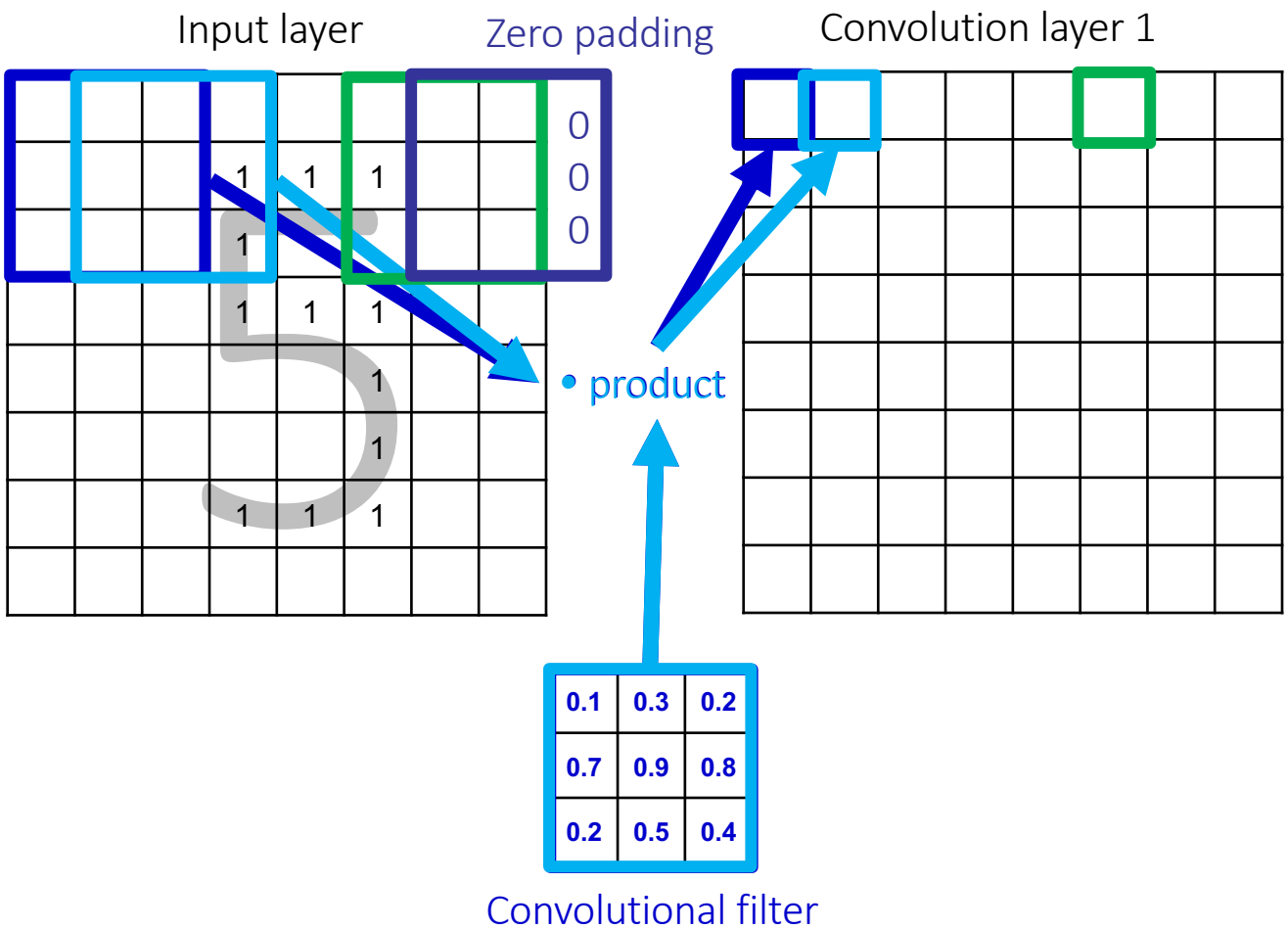


Convolution filter

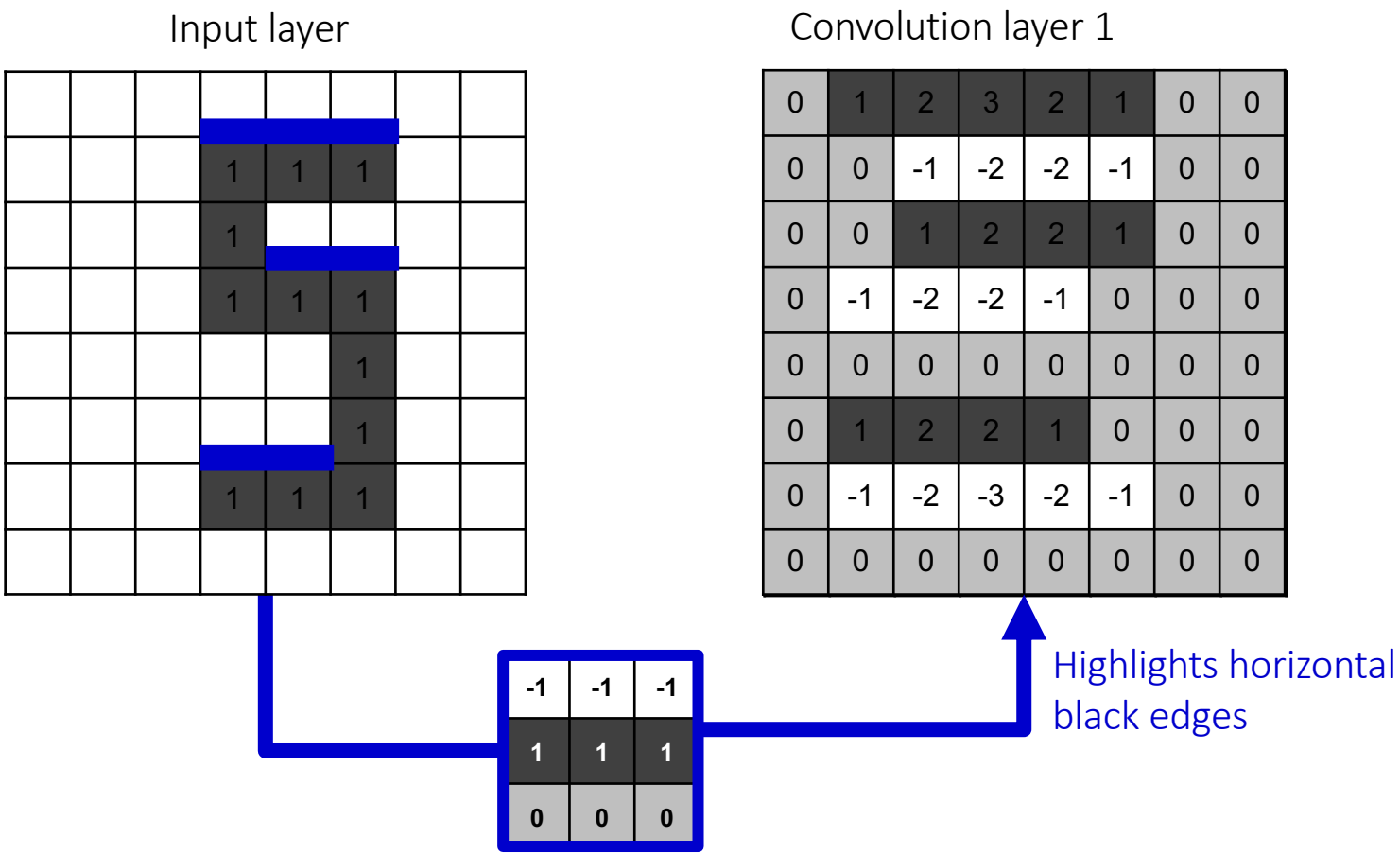
- Apply multiple convolutional filters to extract feature map like edges



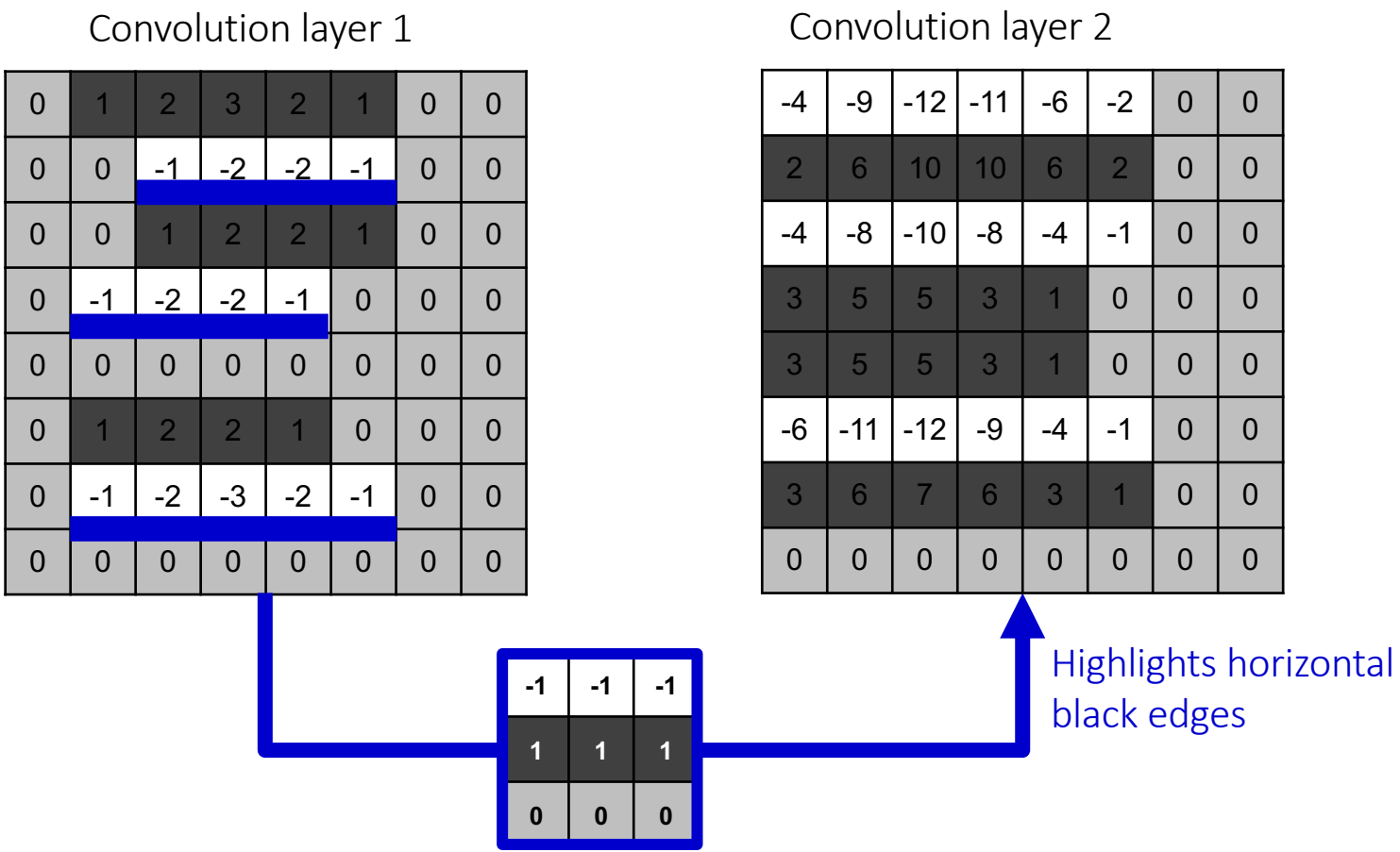
Convolution filter



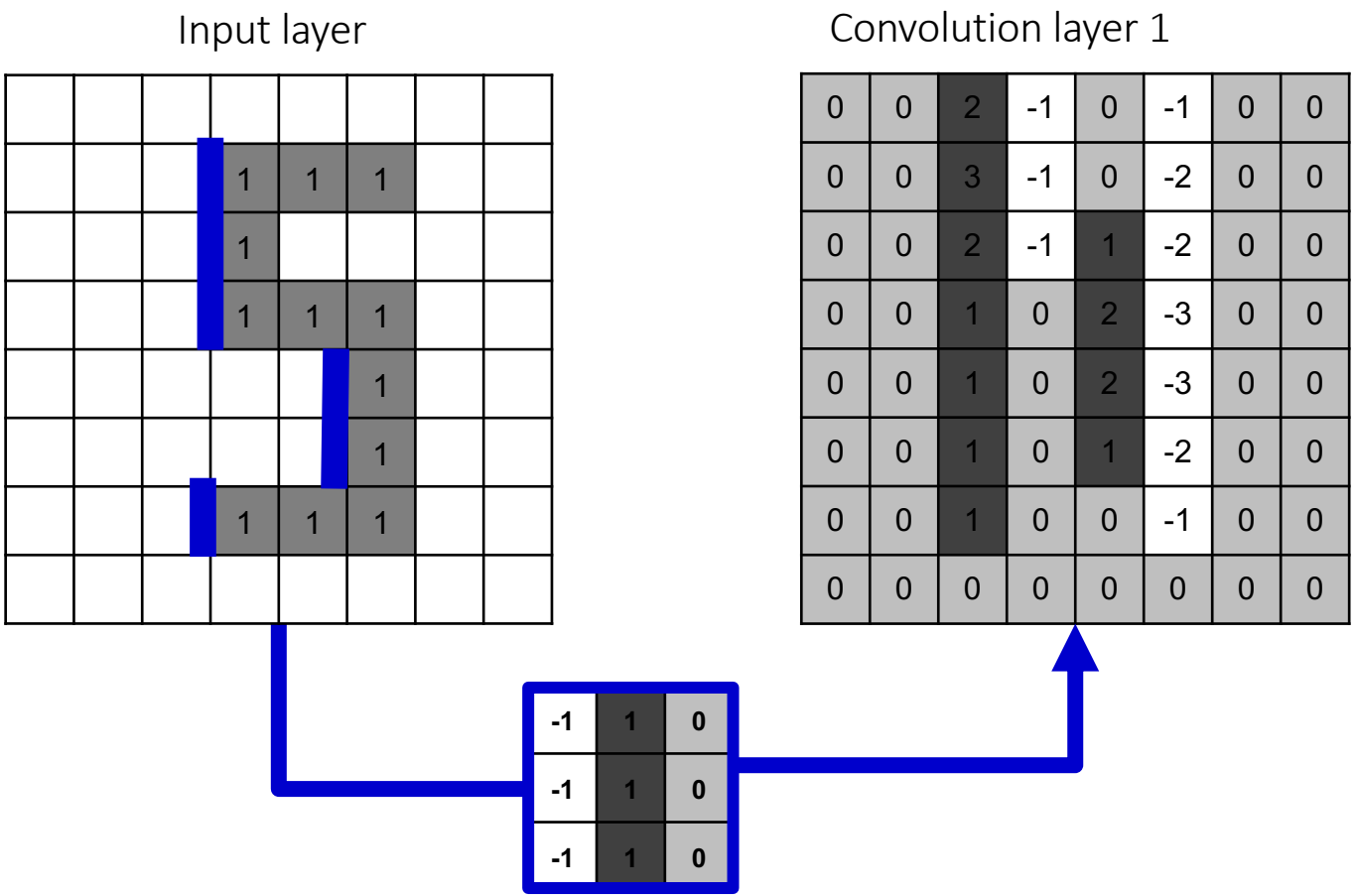
Ex) Convolution filter detecting edges



Ex) Convolution filter detecting edges

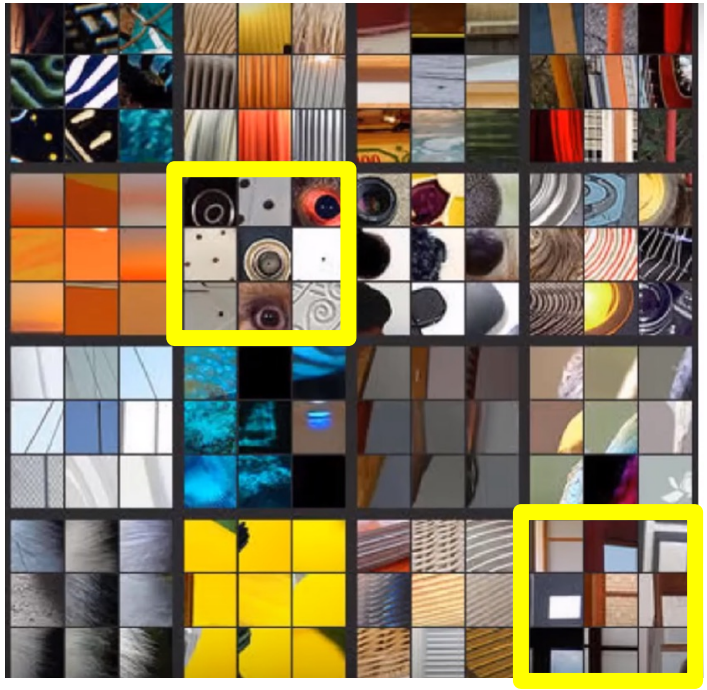


Ex) Convolution filter detecting edges

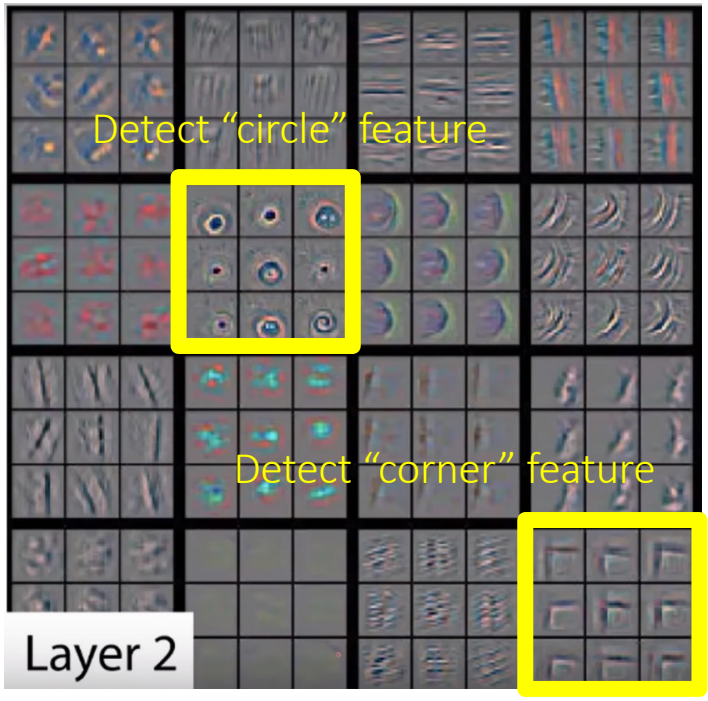


Ex) Convolution filter detecting features

Input layer

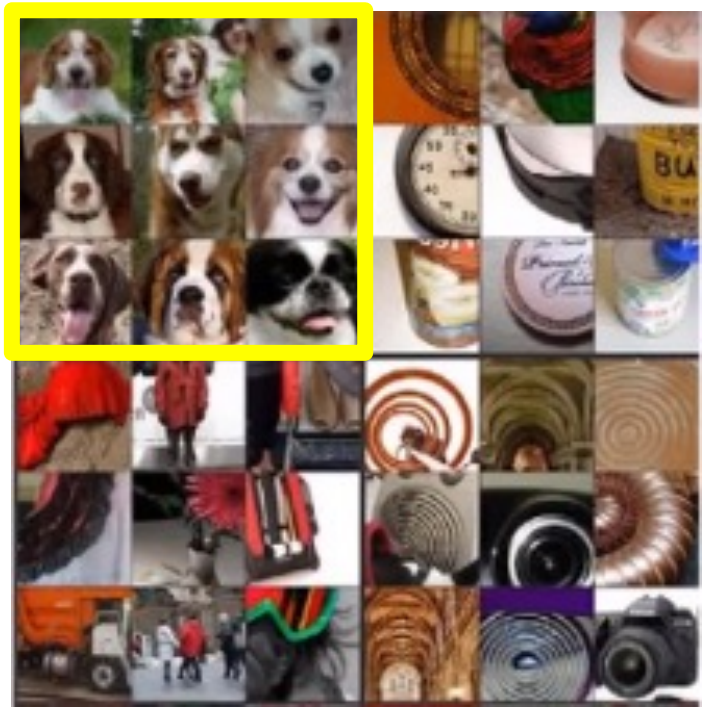


Convolution layer 2

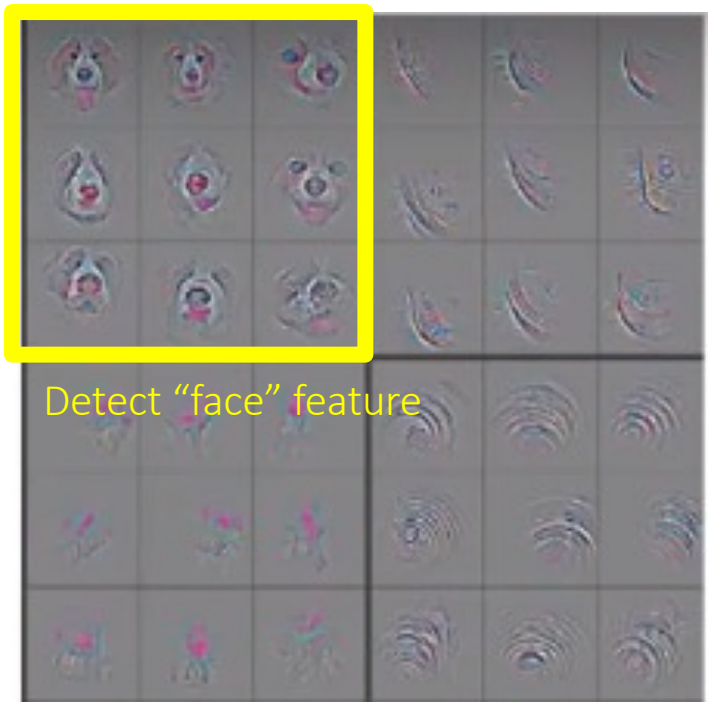


Ex) Convolution filter detecting features

Input layer

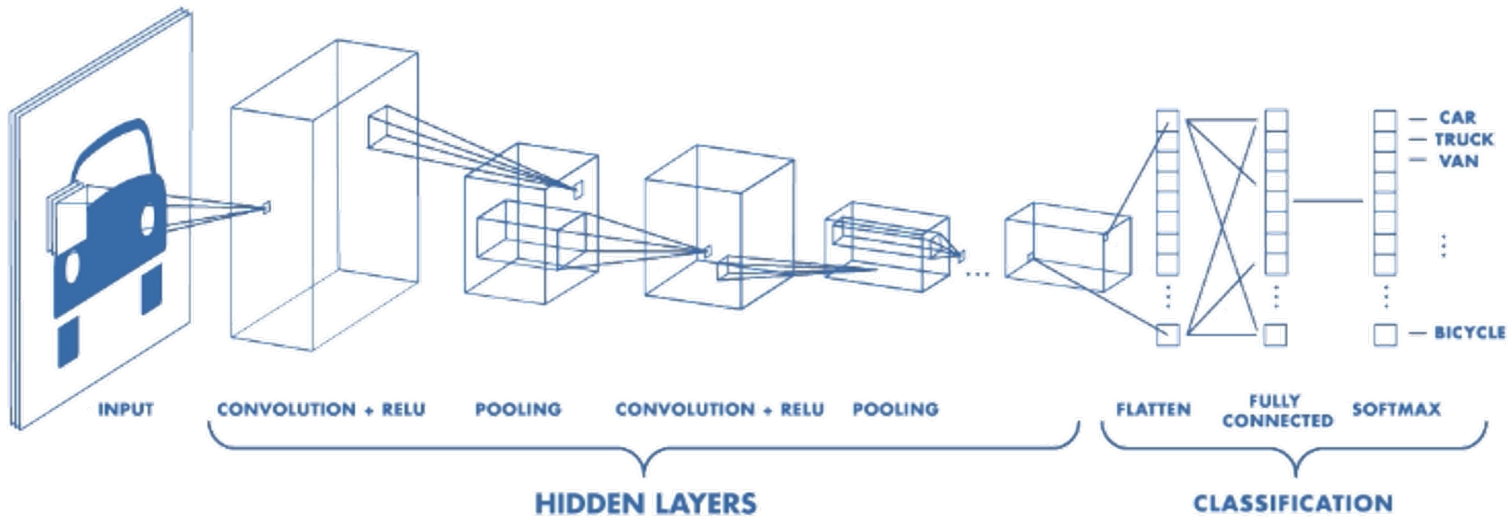


Convolution layer 4



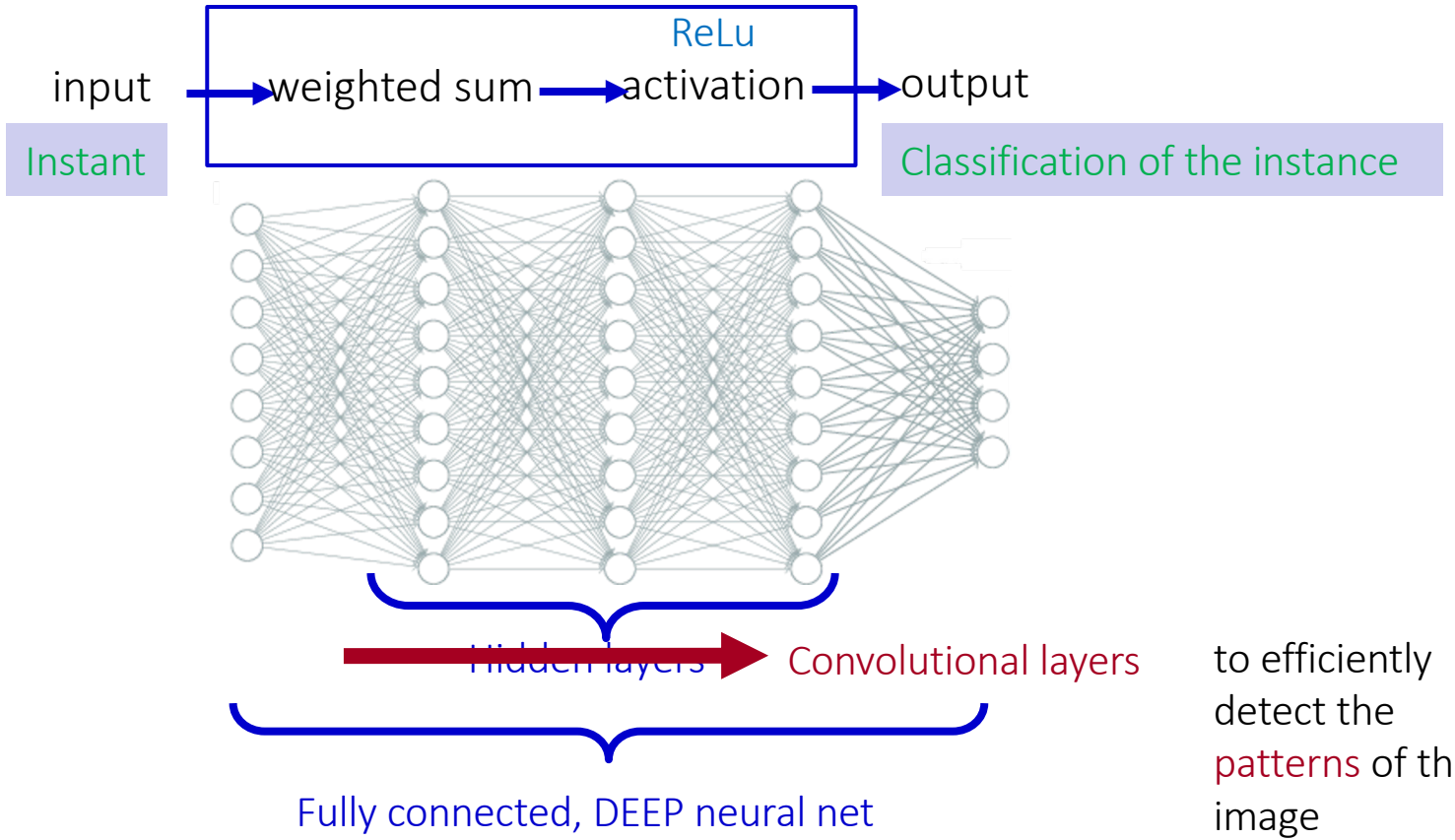
Convolutional Neural Network (CNN)

- A neural network method that scans input data using a convolutional filter that extracts feature points from the input values to efficiently calculate the data amount in the case of image input.



Images from google

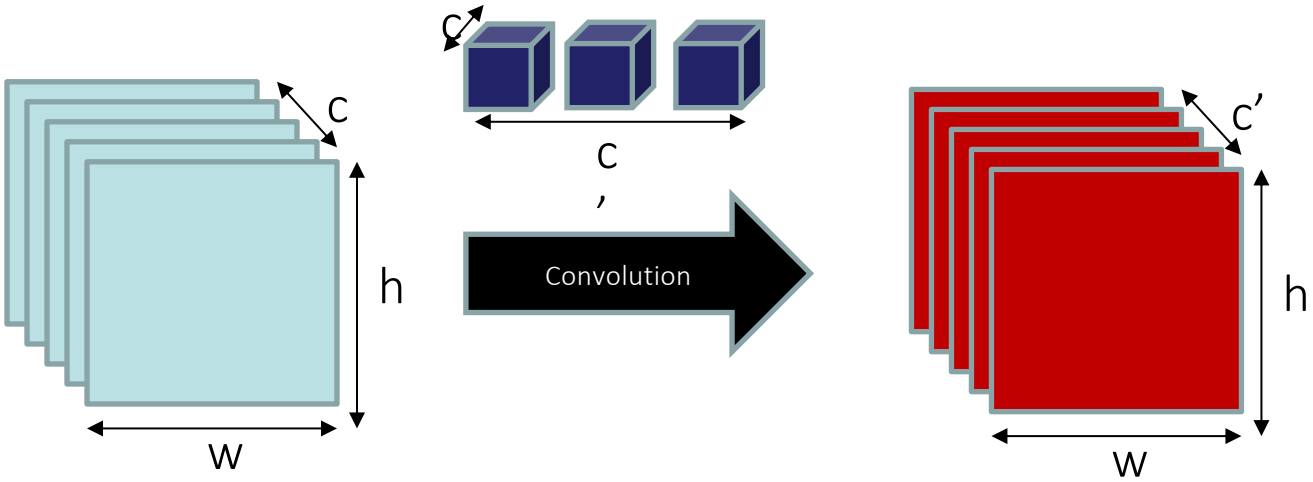
Artificial NN, i.e., fully connected NN



to efficiently detect the patterns of the image

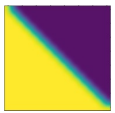
Images from google

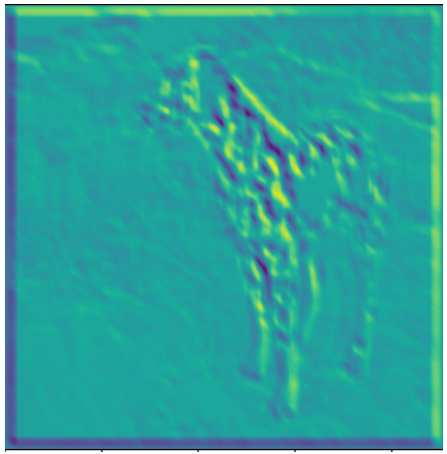
Convolution as a primitive



Convolution as a feature detector

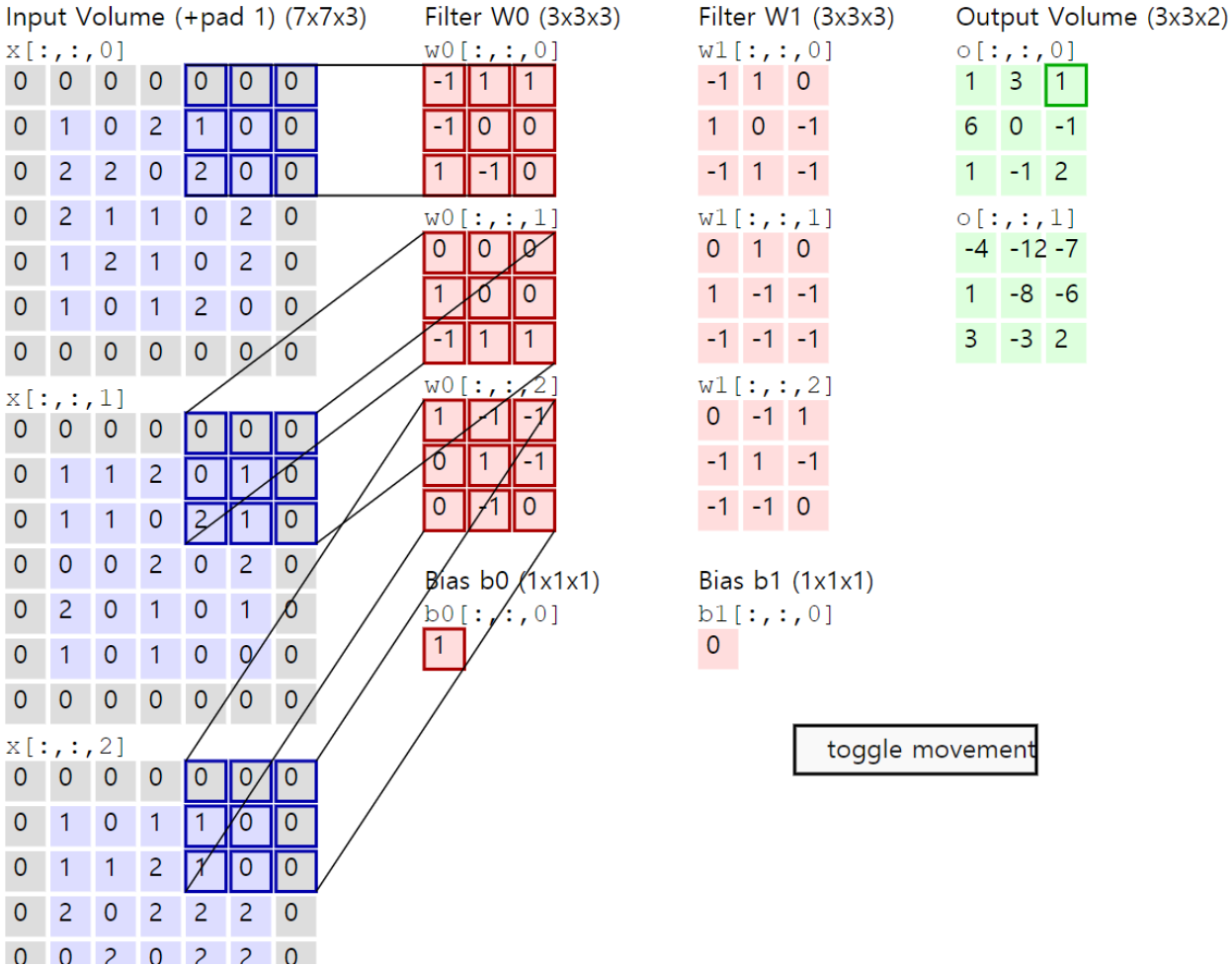
- score at (x,y) = dot product (filter, image patch at (x,y))
- Response represents similarity between filter and image patch



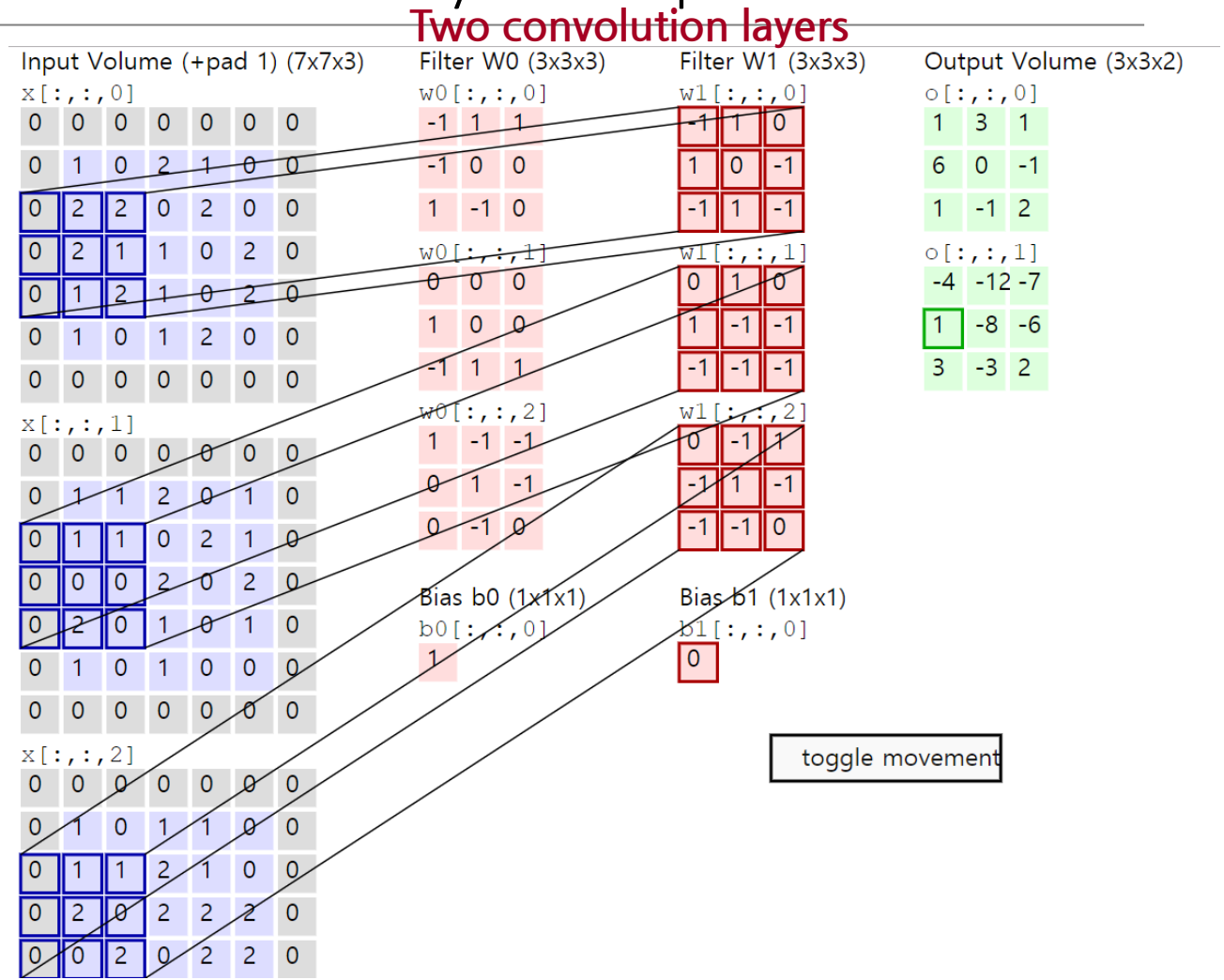


Ex) Convolution with layered input (RGB)

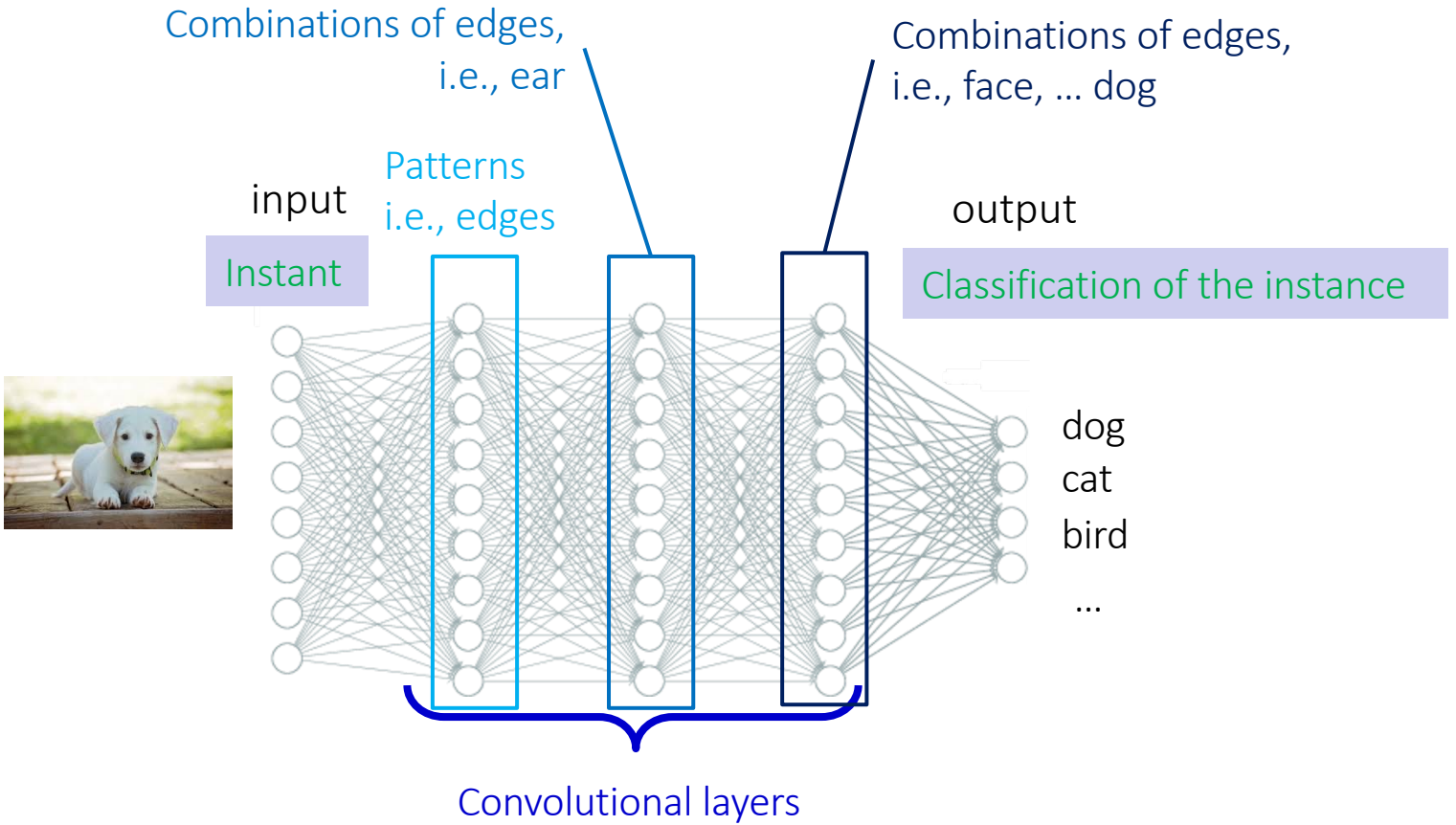
Two convolution layers



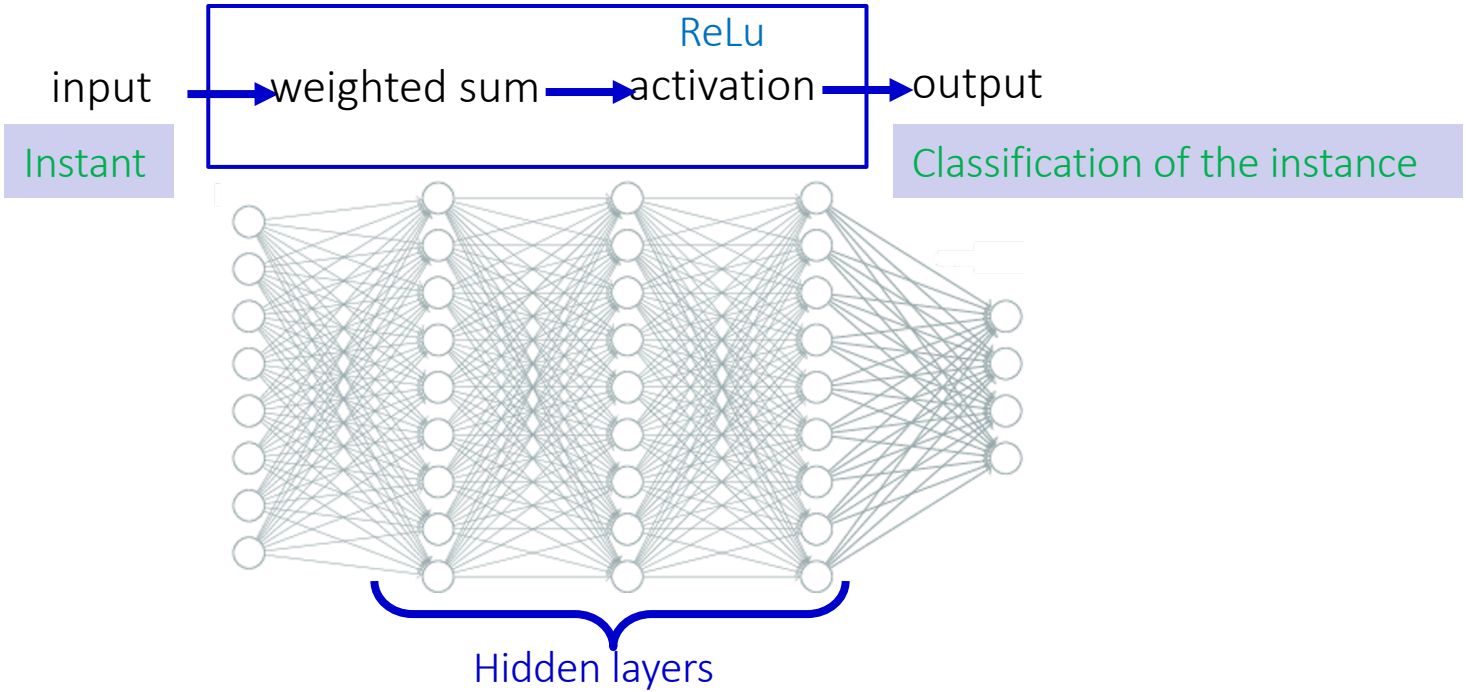
Ex) Convolution with layered input



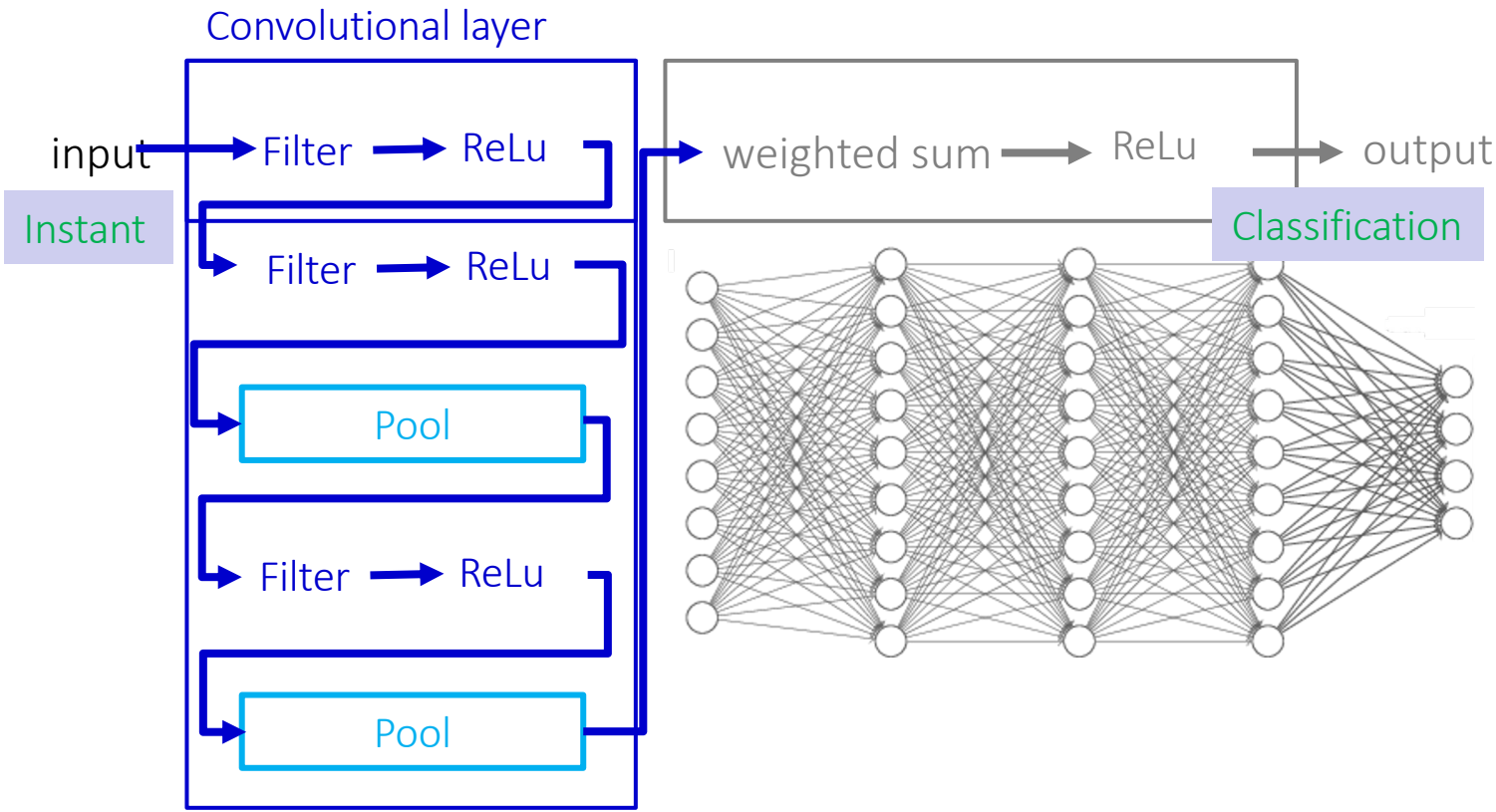
Convolutional neural network



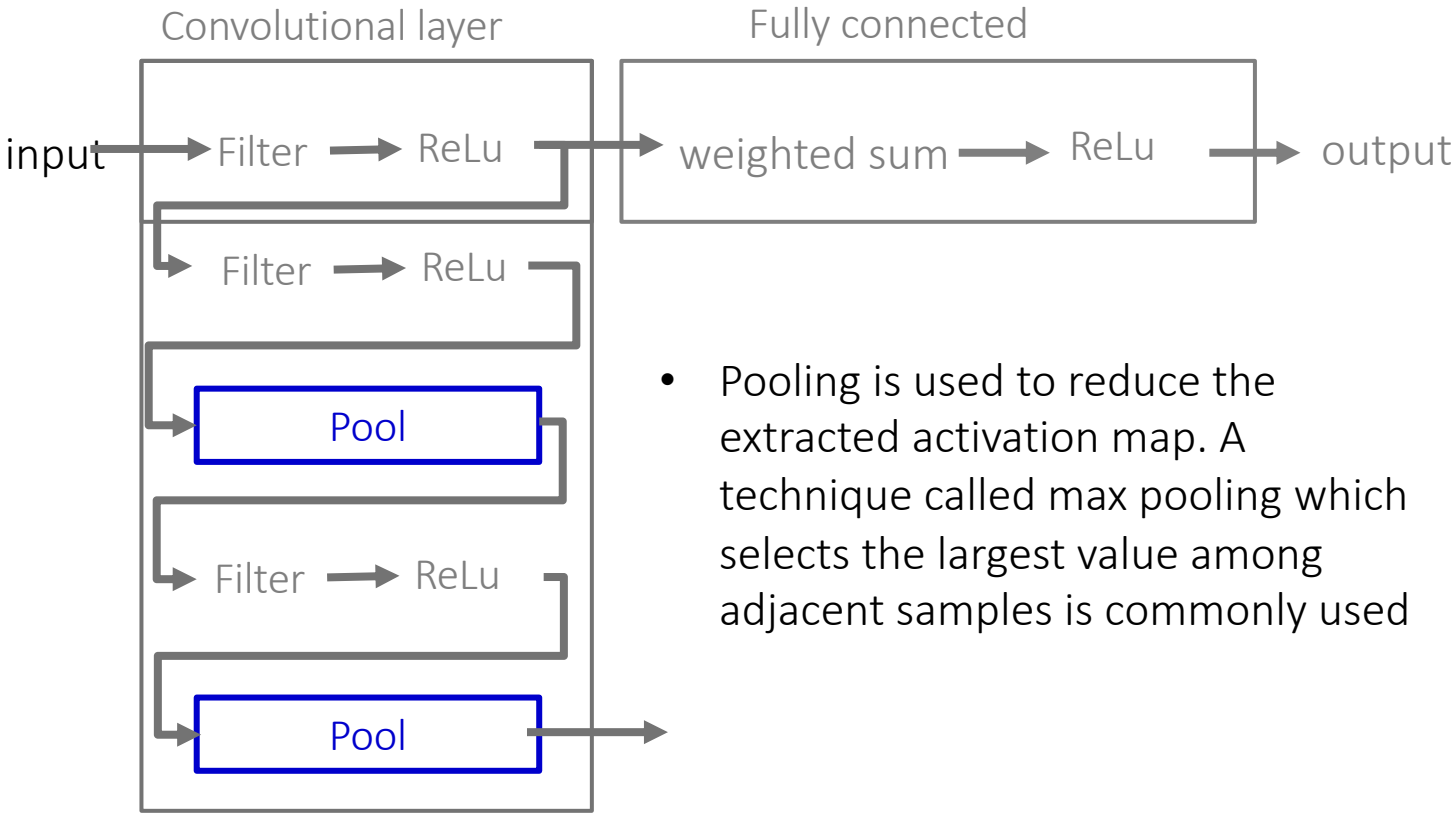
Recall, Artificial NN



Convolutional neural network



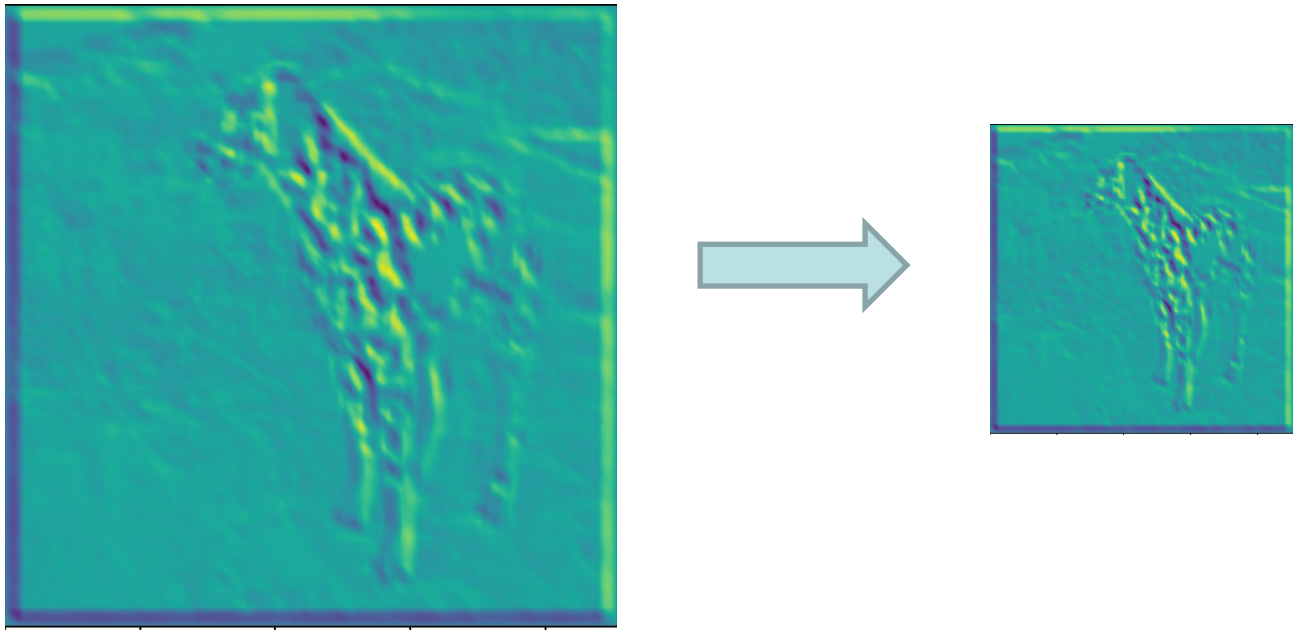
Convolutional neural network



- Pooling is used to reduce the extracted activation map. A technique called max pooling which selects the largest value among adjacent samples is commonly used

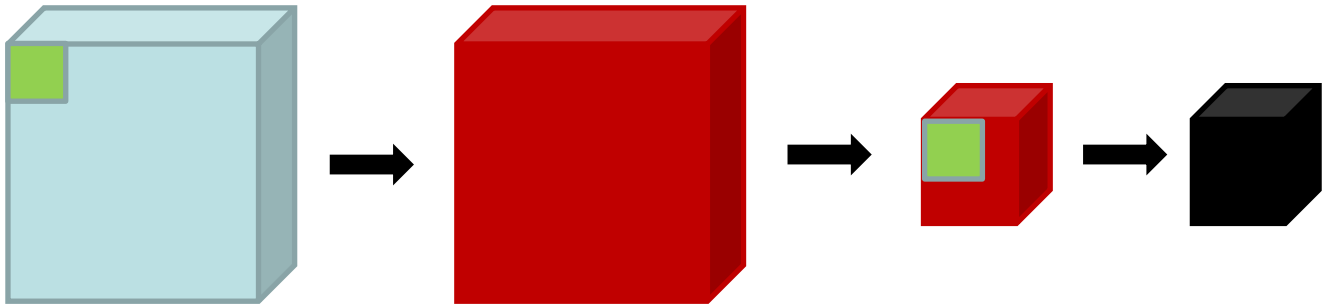
Invariance to distortions: Subsampling

- Convolution by itself doesn't grant invariance
- But by subsampling, large distortions become smaller, so more invariance



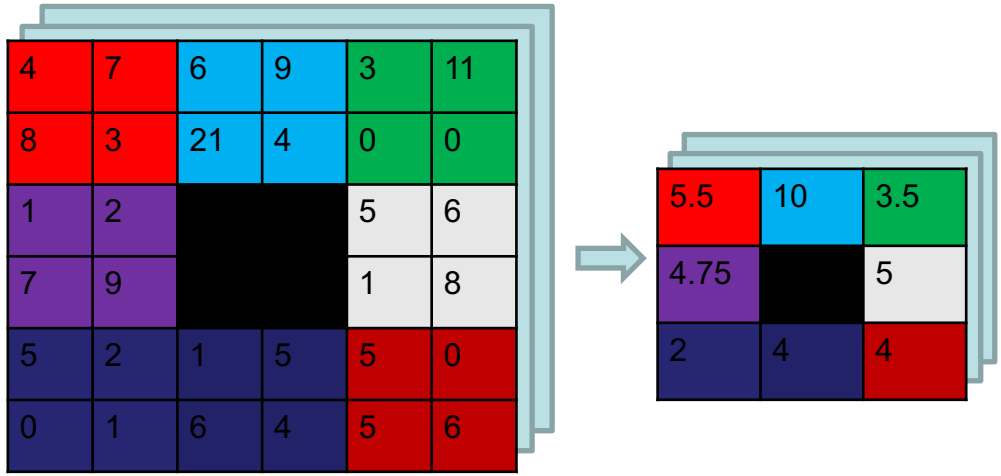
Convolution-subsampling-convolution

- Interleaving convolutions and subsamplings causes later convolutions to capture a *larger fraction of the image* with the same kernel size
- Set of image pixels that an intermediate output pixel depends on = *receptive field*
- Convolutions after subsamplings increase the receptive feild

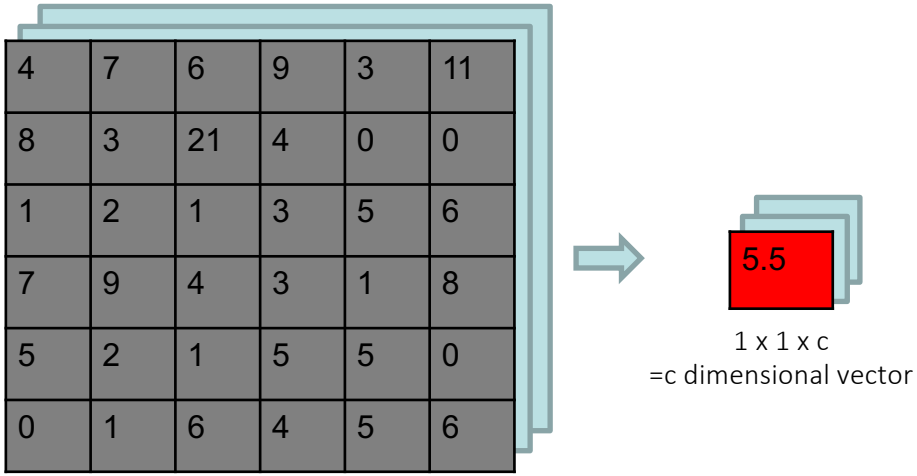


  - Convolution in earlier steps detects *more local* patterns *less resilient* to distortion
  - Convolution in later steps detects *more global* patterns *more resilient* to distortion
  - Subsampling allows capture of *larger, more invariant* patterns

Invariance to distortions: Average Pooling

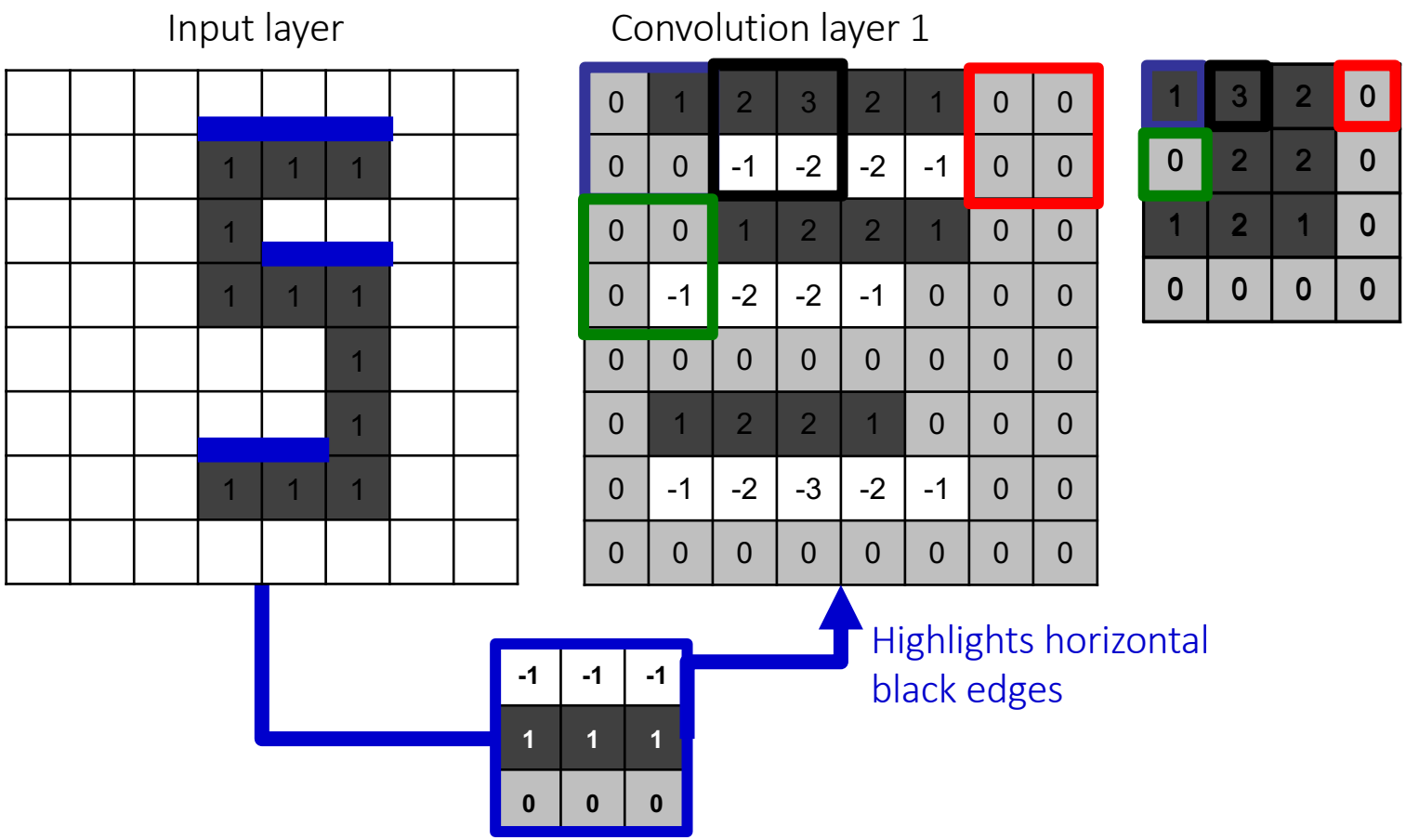


Global average pooling

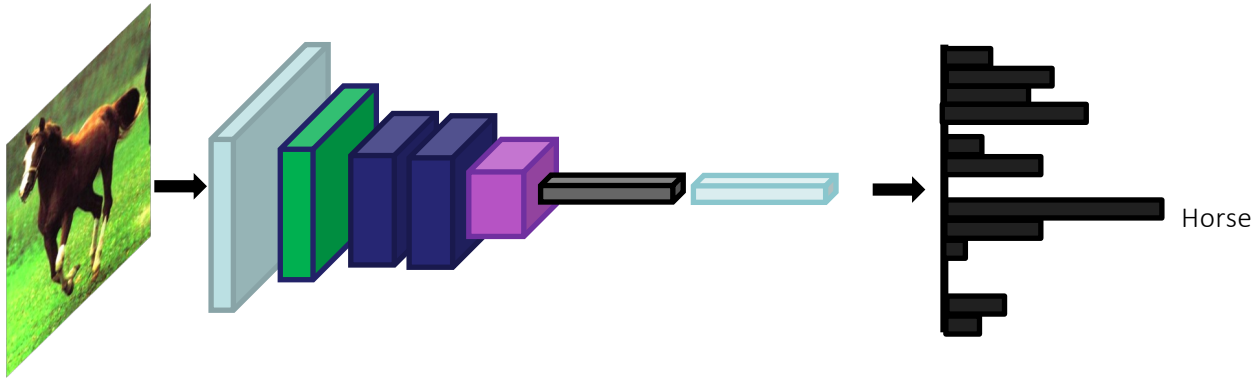


Max pooling example

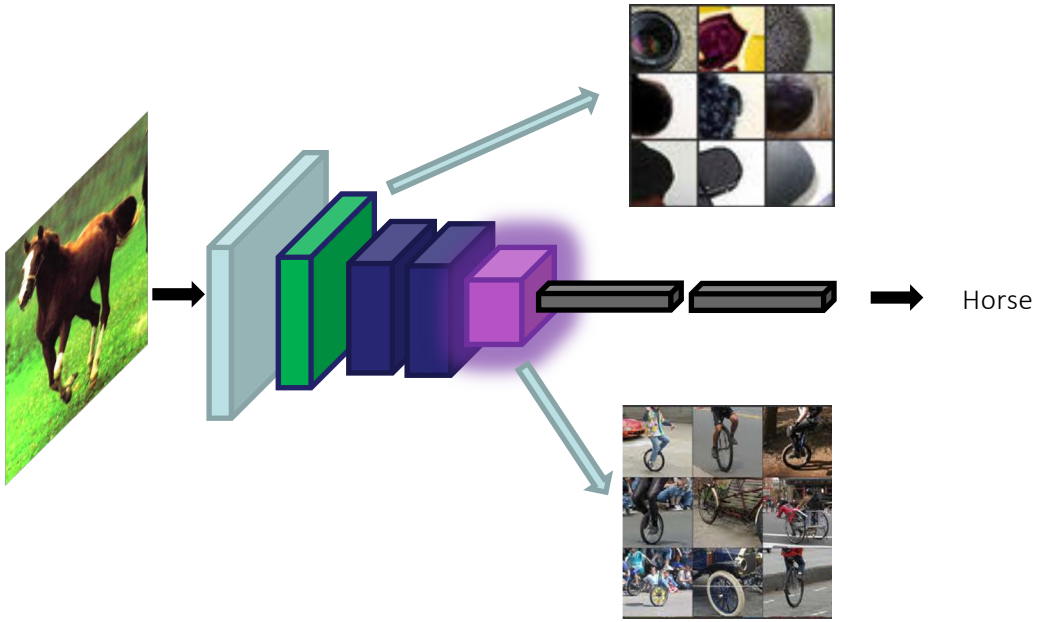
Filter size = 2
Stride size = 2



Convolutional networks



From Classifiers to Segmenters



Visualizations from : M. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. In ECCV 2014.

Convolutional networks

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE 86.11 (1998): 2278-2324.

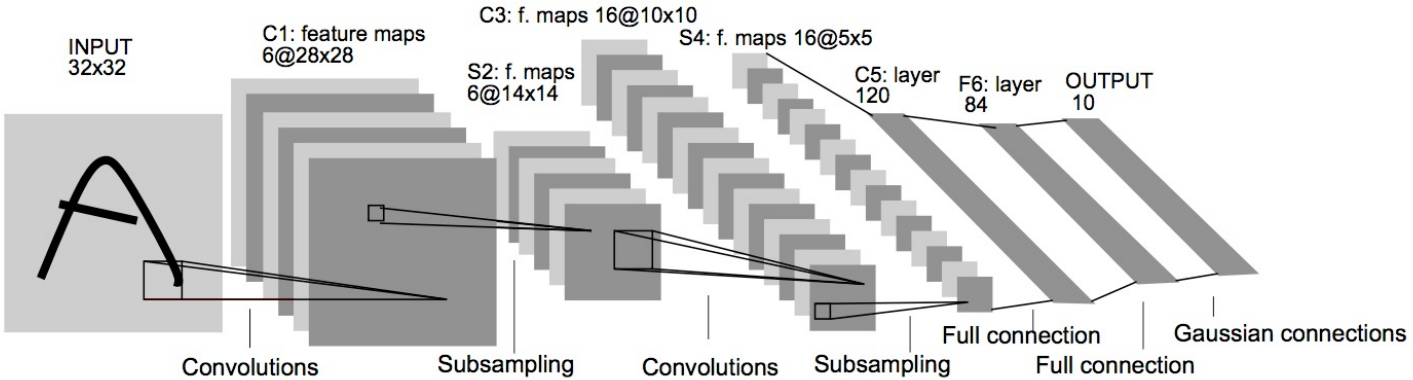
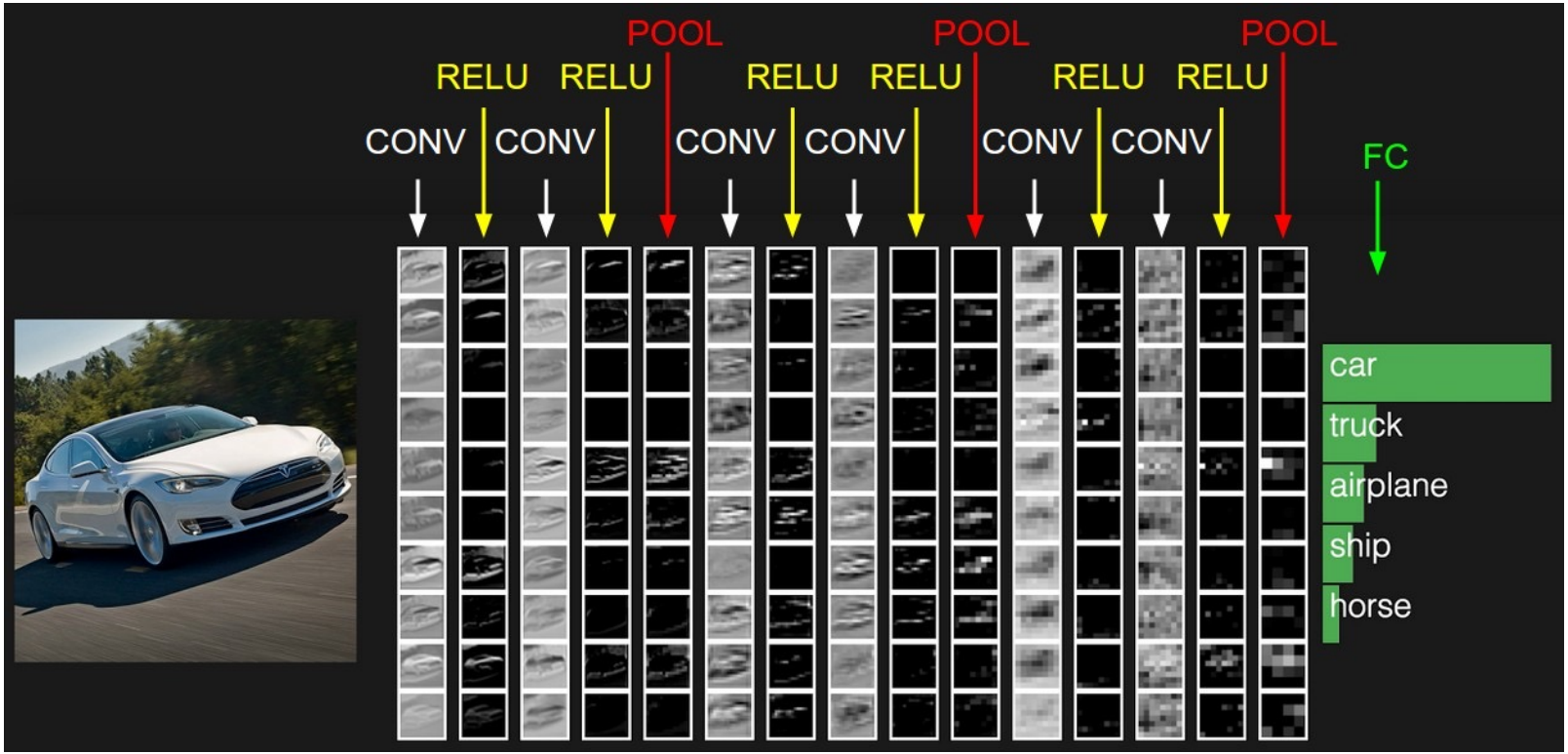


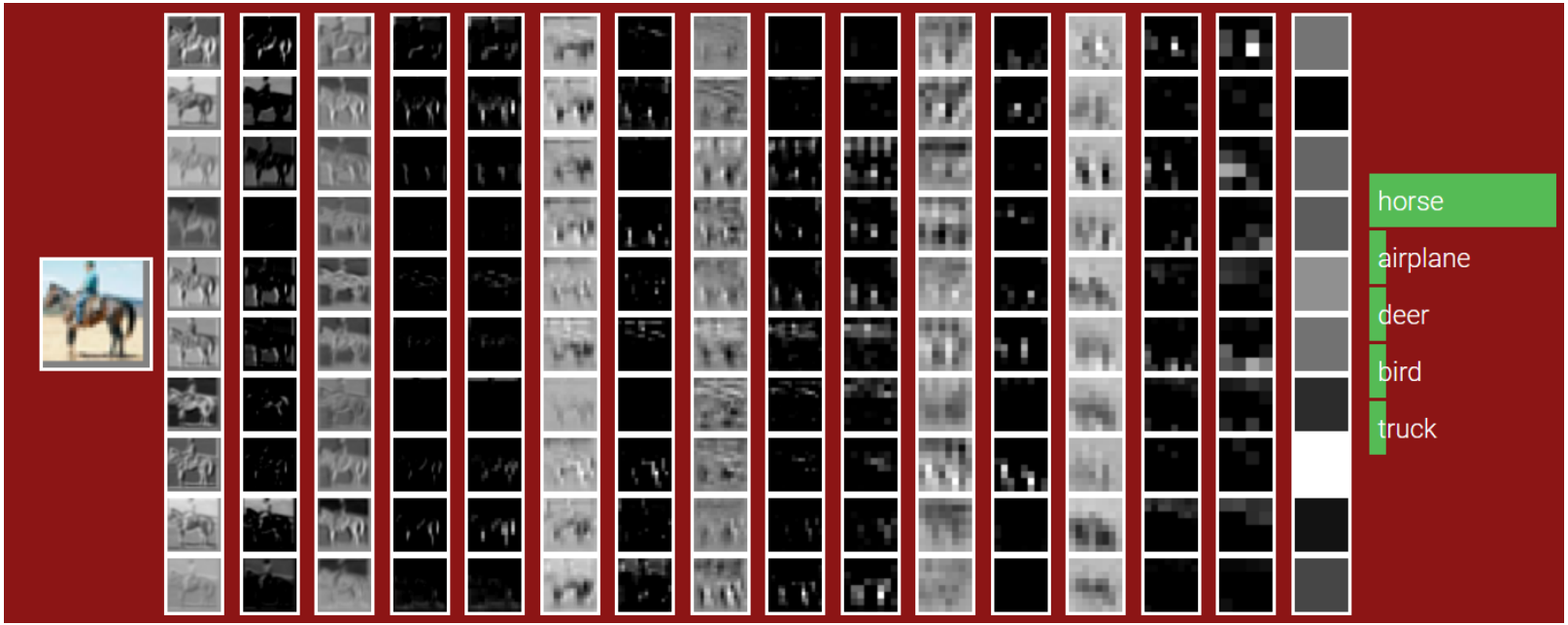
Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

CNN example


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

CNN example

http://cs231n.stanford.edu/



CNN example

http://cs231n.stanford.edu/