SPATIAL ECOLOGY

Neural Nets & Convolutional Neural Networks

Antonio Fonseca

Agenda

1) Feedforward Neural Networks

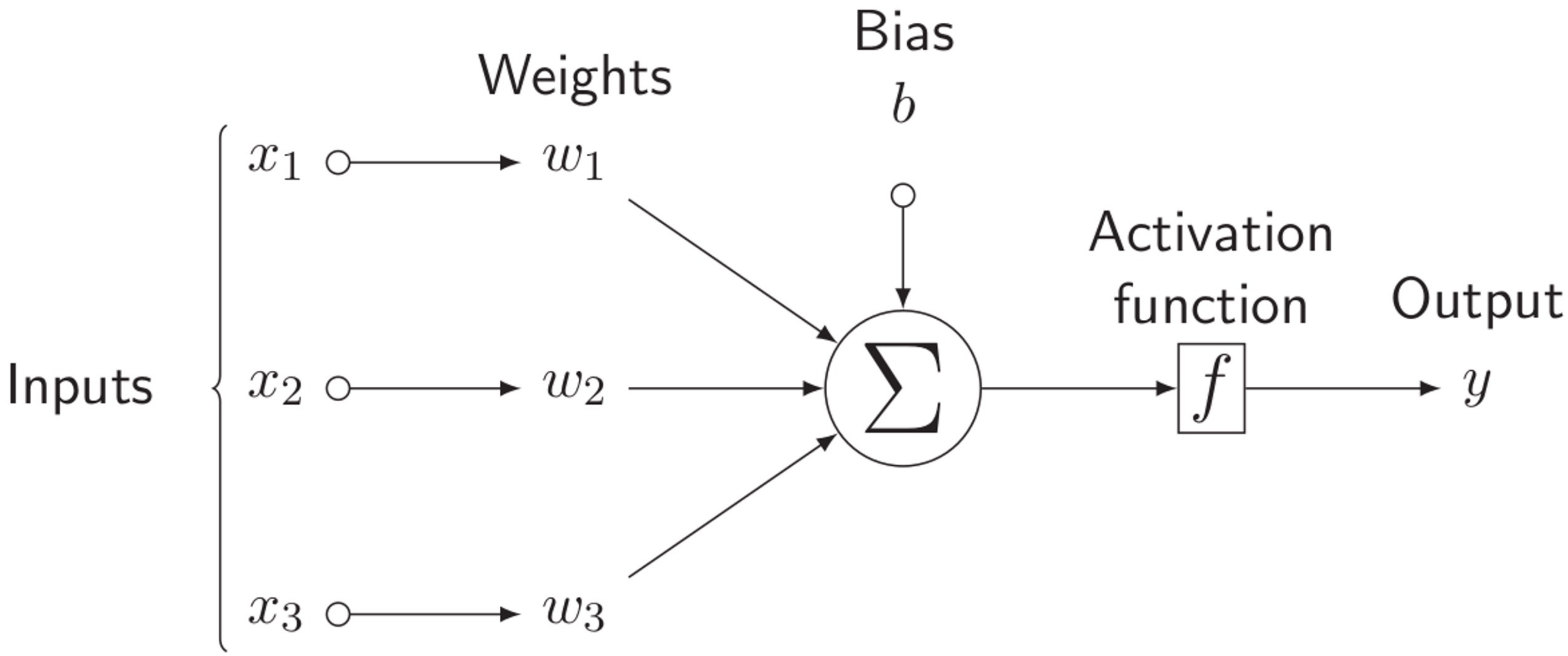
- The limitations of Perceptrons
- Multi-layer Perceptron
- Training: the forward and back-propagation
- Debugging tips
- Tutorial: Neural Nets for the tree height dataset

2) Convolutional Neural Networks

- Spatial locality structure
- Kernels, padding, pooling
- Classification tasks
- Saliency Analysis
- Tutorial: data batching, classification of satellite images, WandB

Perceptron: Threshold Logic

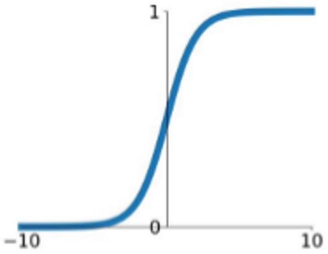
$$\mathcal{L}_{\text{perc}}(\mathbf{x}, y) = \begin{cases} 0 & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) > 0 \\ -y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) \leq 0 \end{cases}$$



Activation functions

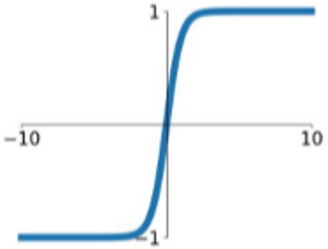
Sigmoid

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



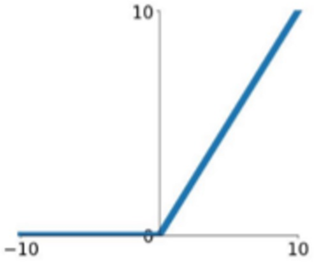
tanh

$\tanh(x)$



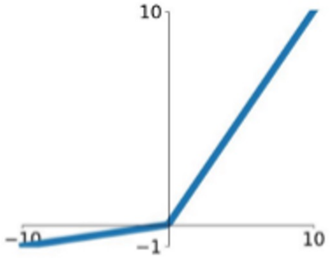
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

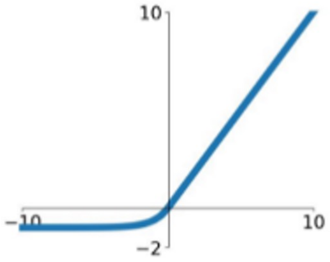


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Optimizers (pt1)

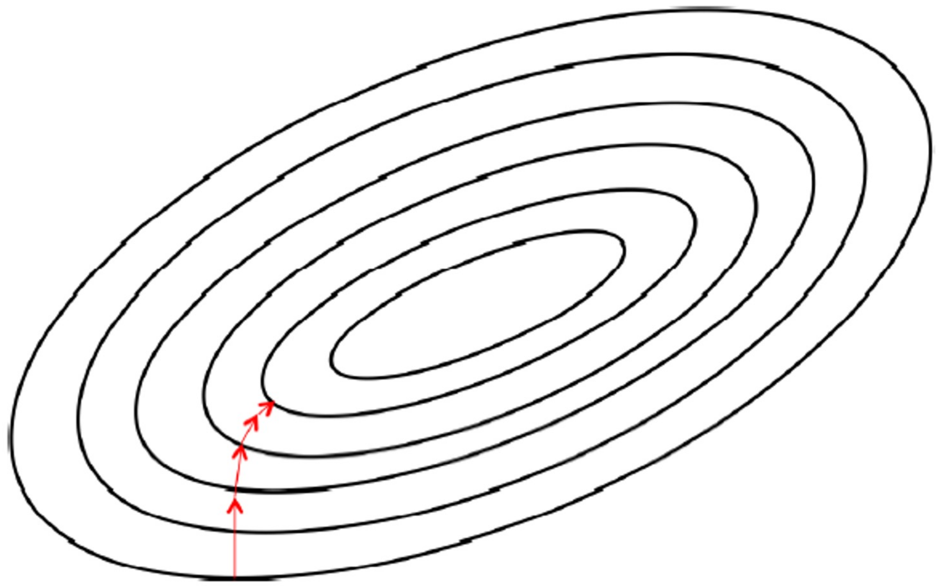
Gradient

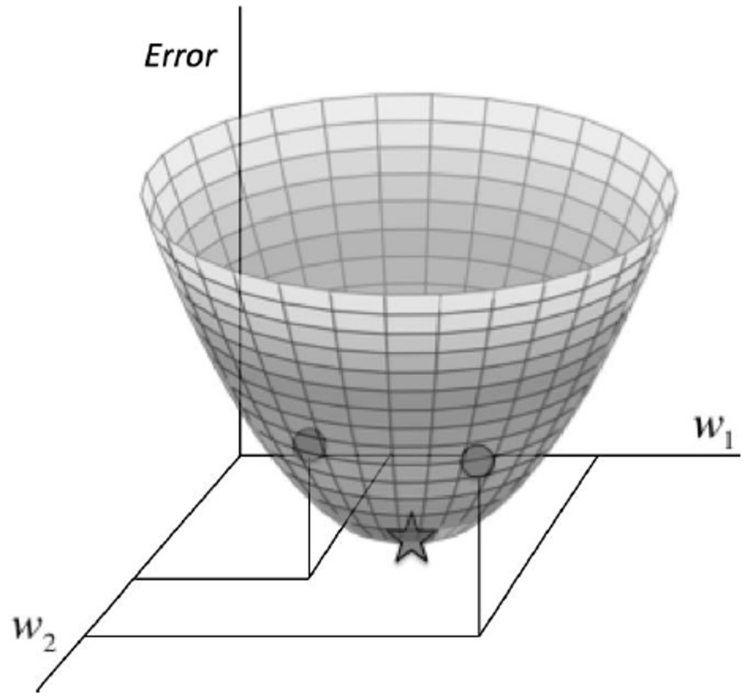
$$\Delta w_k = -\frac{\partial E}{\partial w_k}$$

$$= -\frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + \Delta w_k$$

Stochastic gradient descent (**SGD**)





Optimizers

Watch out for local minimal areas

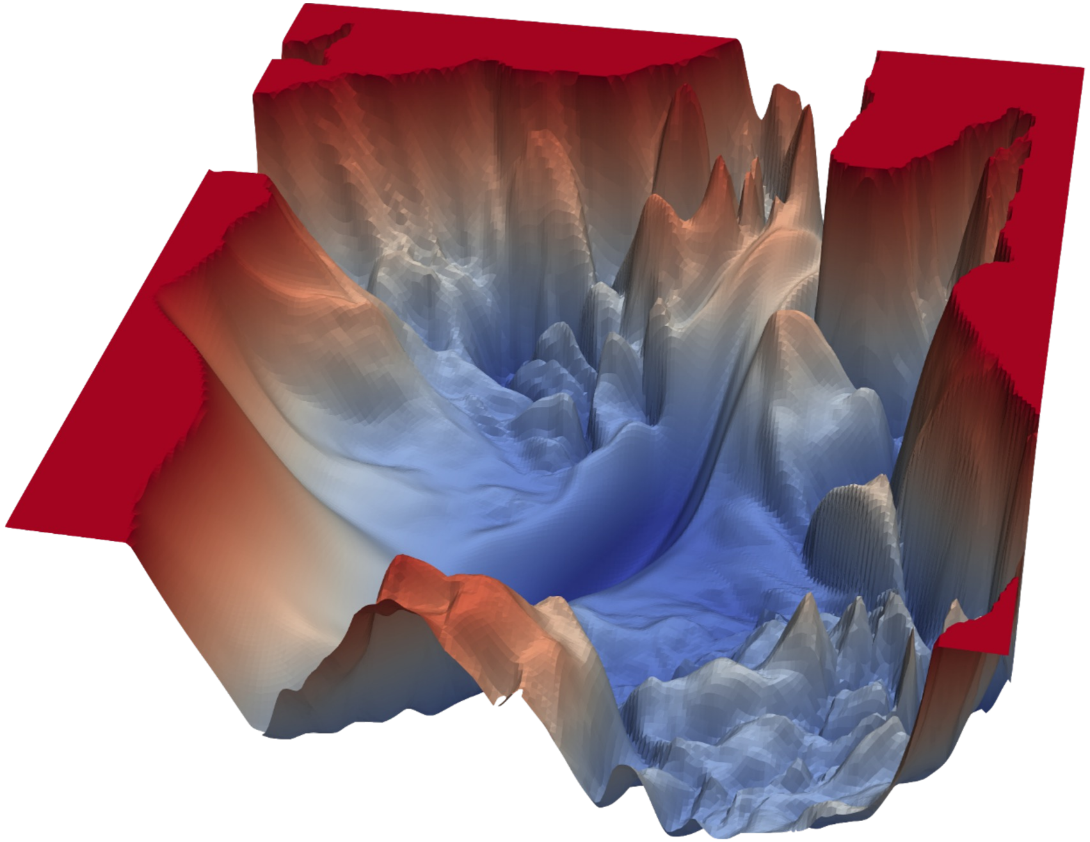
Hyperparameters

- Learning rate (α)

$$\Delta w_k = -\alpha \frac{\partial E}{\partial w_k}$$

$$= -\alpha \frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + \Delta w_k$$



Stochastic gradient descent (**SGD**)

Optimizers

Optimizers

Hyperparameters

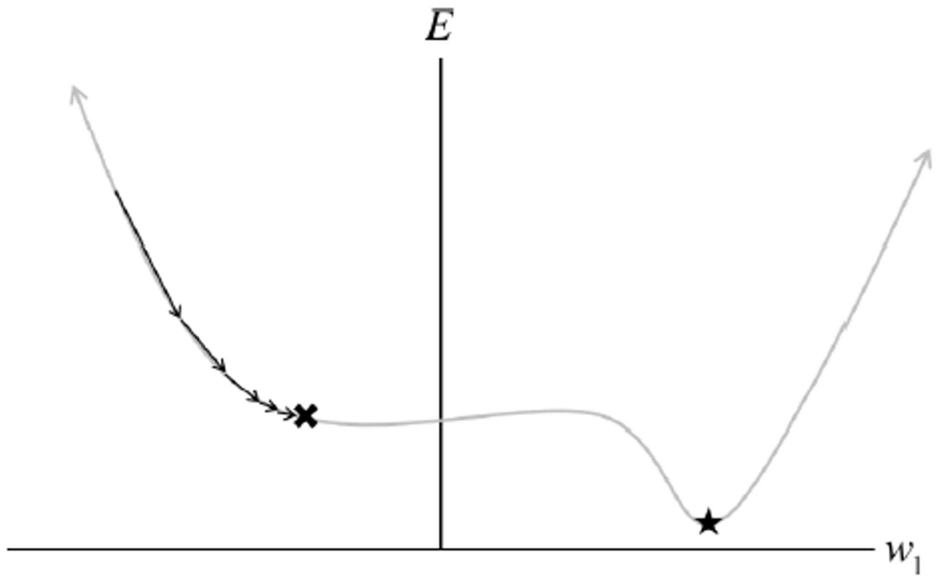
- Learning rate (α)

$$\Delta w_k = -\alpha \frac{\partial E}{\partial w_k}$$

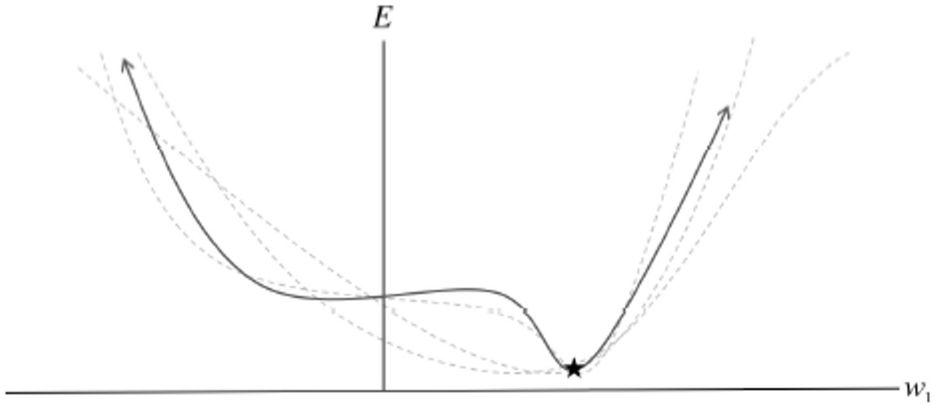
$$= -\alpha \frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + \Delta w_k$$

Stochastic gradient descent (**SGD**)



Local Minima



Multiple samples

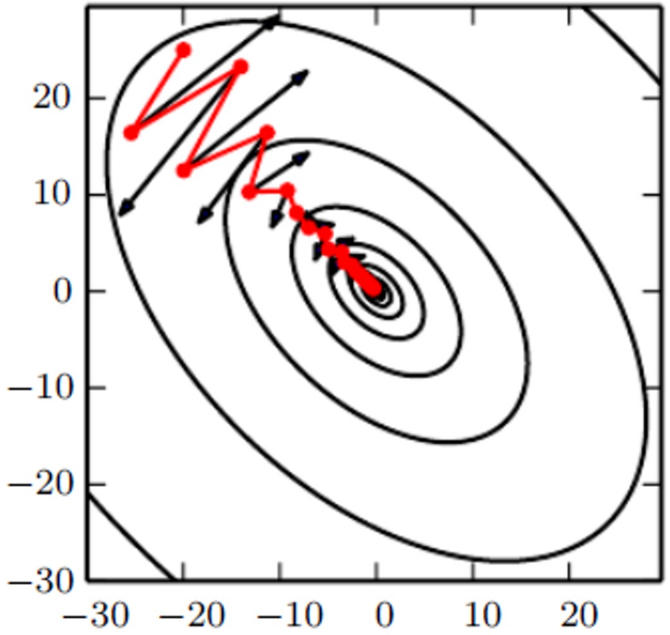
Optimizers

Hyperparameters

- Learning rate (α)
- Momentum (β)

$$v_{i+1} = v\beta - \alpha \frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + v$$



SGD

SGD+Momentum

Stochastic gradient descent with momentum (**SGD+Momentum**)

Optimizers

Hard to pick right hyperparameters

- Small learning rate: long convergence time
- Large learning rate: convergence problems

Adagrad: adapts learning rate to each parameter

$$\Delta w_{k,t} = -\alpha \frac{\partial E_t}{\partial w_{k,t}} = -\alpha \nabla_w E(w_t)$$

- Learning rate might decrease too fast
- Might not converge

$$g_{t,i} = \nabla_w E(w_{t,i})$$

$$G_{t+1,i} = G_{t,i} + g_{t,i} \odot g_{t,i}$$

$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

Optimizers

RMSprop: decaying average of the past squared gradients

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Exponentially decaying average

Adadelta

$$E[\Delta_w^2]_t = \gamma E[\Delta_w^2]_{t-1} + (1-\gamma)\Delta_w^2$$

$$\Delta w_t = \frac{\sqrt{E[\Delta_w^2]_t + \epsilon}}{\sqrt{G_{t,i} + \epsilon}} g_t$$

$$\Delta w_{k,t} = -\alpha \frac{\partial E_t}{\partial w_{k,t}} = -\alpha \nabla_w E(w_t) = -\alpha g_{t,i}$$

$$g_{t,i} = \nabla_w E(w_{t,i})$$

$$G_{t+1,i} = \gamma G_{t,i} + (1-\gamma) g_{t,i} \odot g_{t,i}$$

$$\Longrightarrow \quad w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

Optimizers

ADAM: decaying average of the past gradients and its square

RMSprop / Adadelta

$$g_{t,i} = \nabla_w E(w_{t,i})$$

$$G_{t+1,i} = \gamma G_{t,i} + (1-\gamma) g_{t,i} \odot g_{t,i}$$

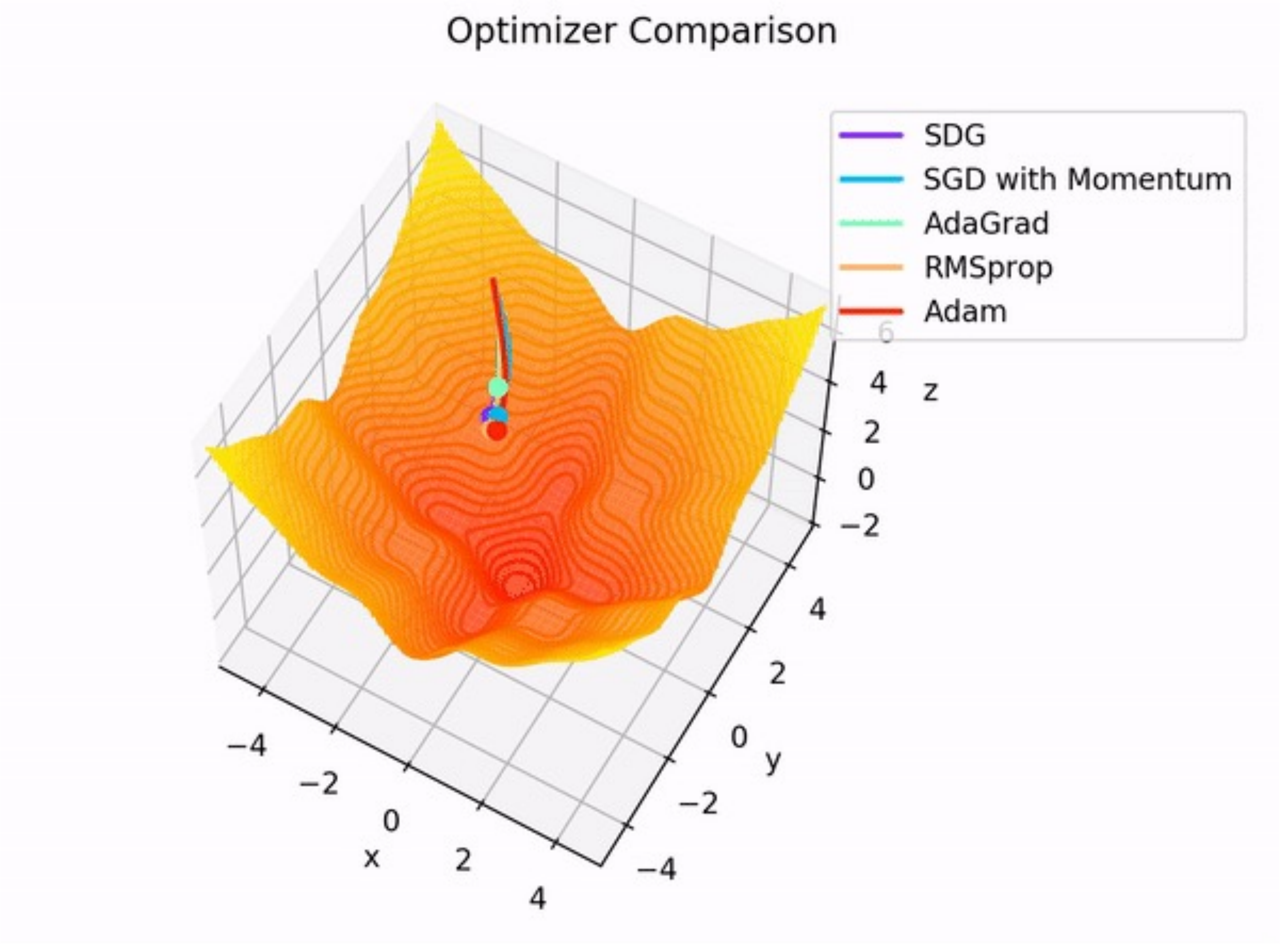
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

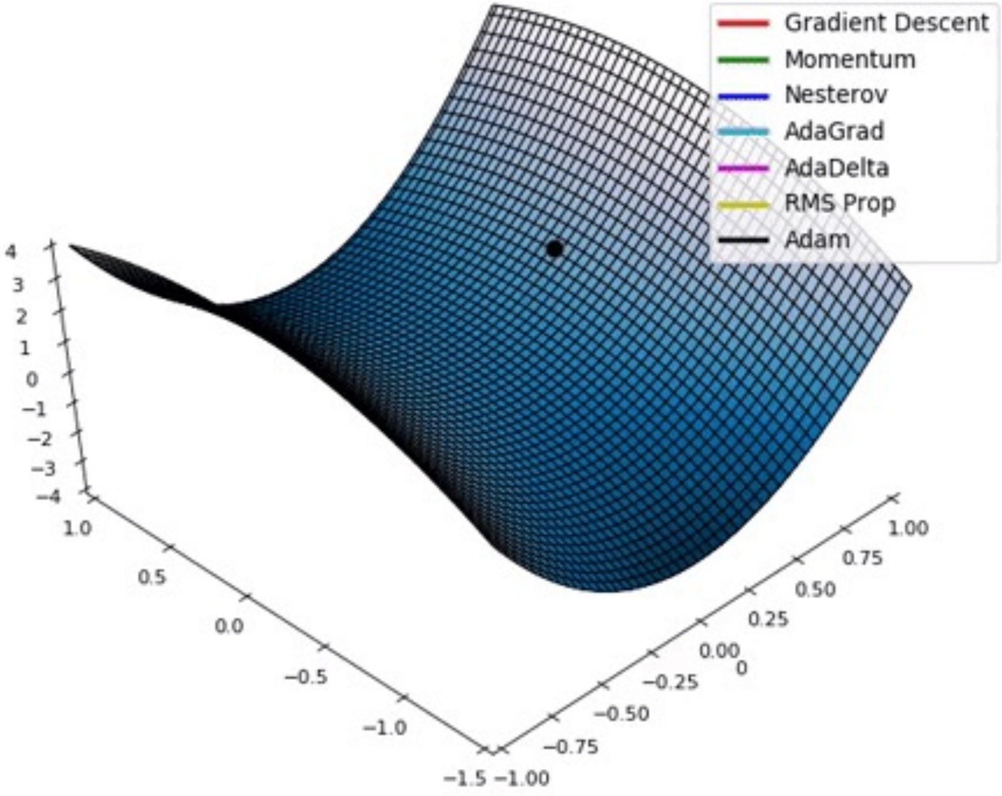
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$



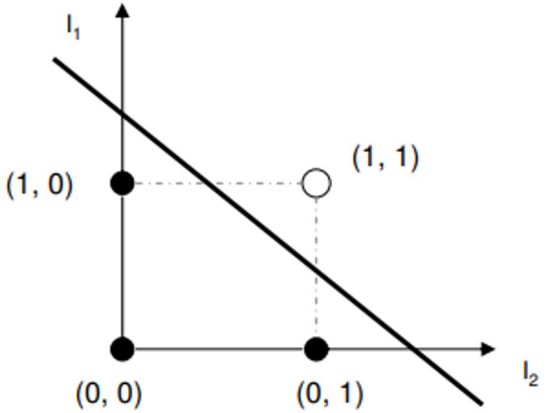


Which optimizer is the best?

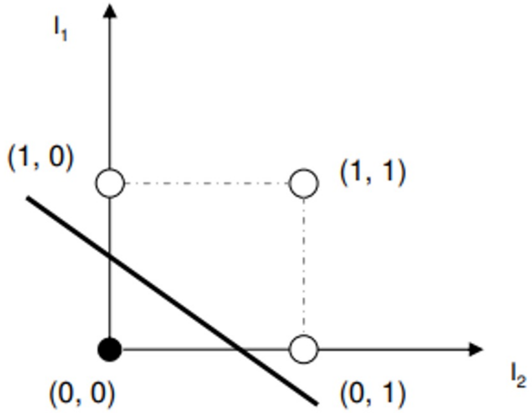
Multi-layer Perceptron

Limitations of the Perceptron

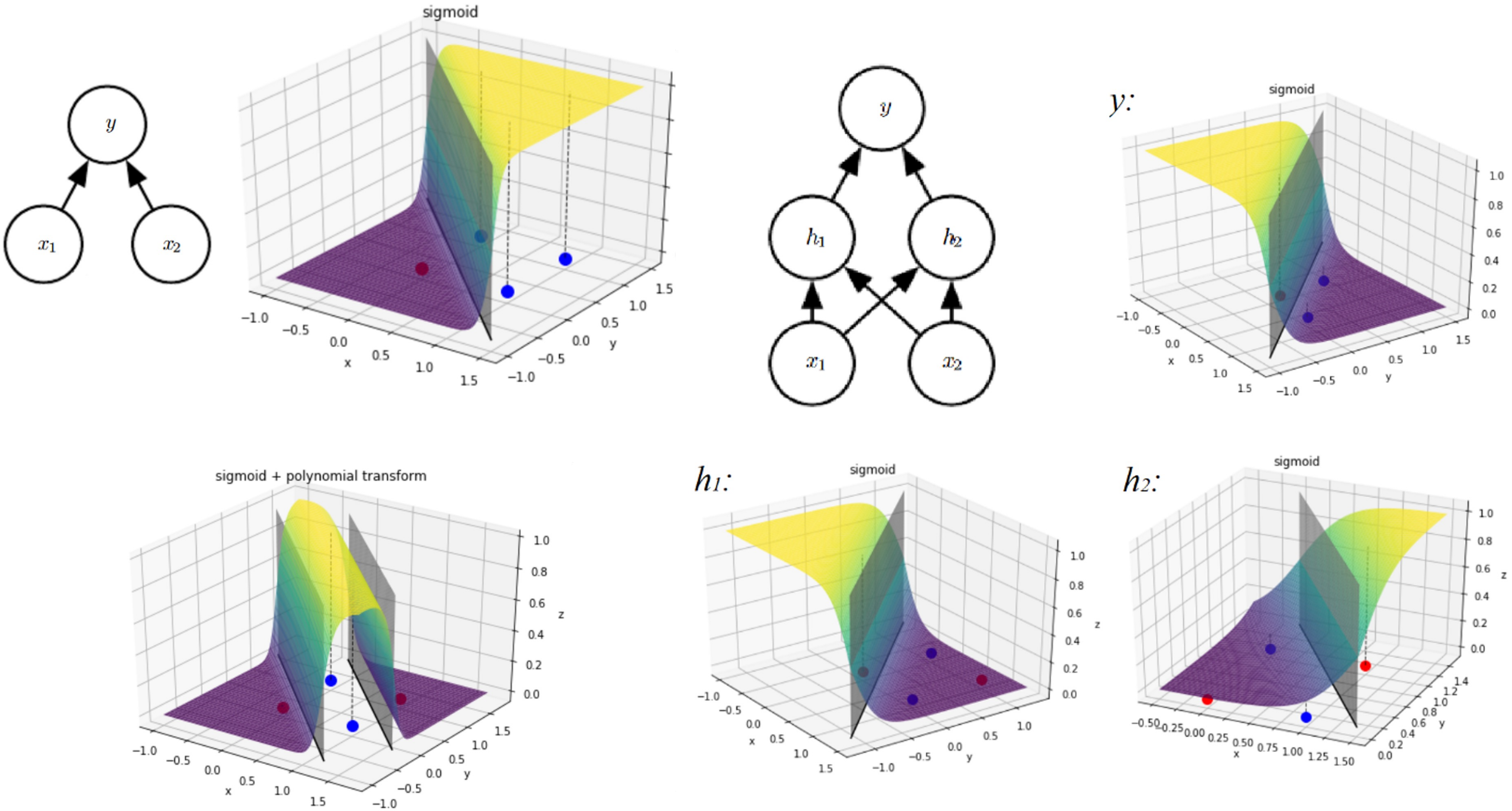
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

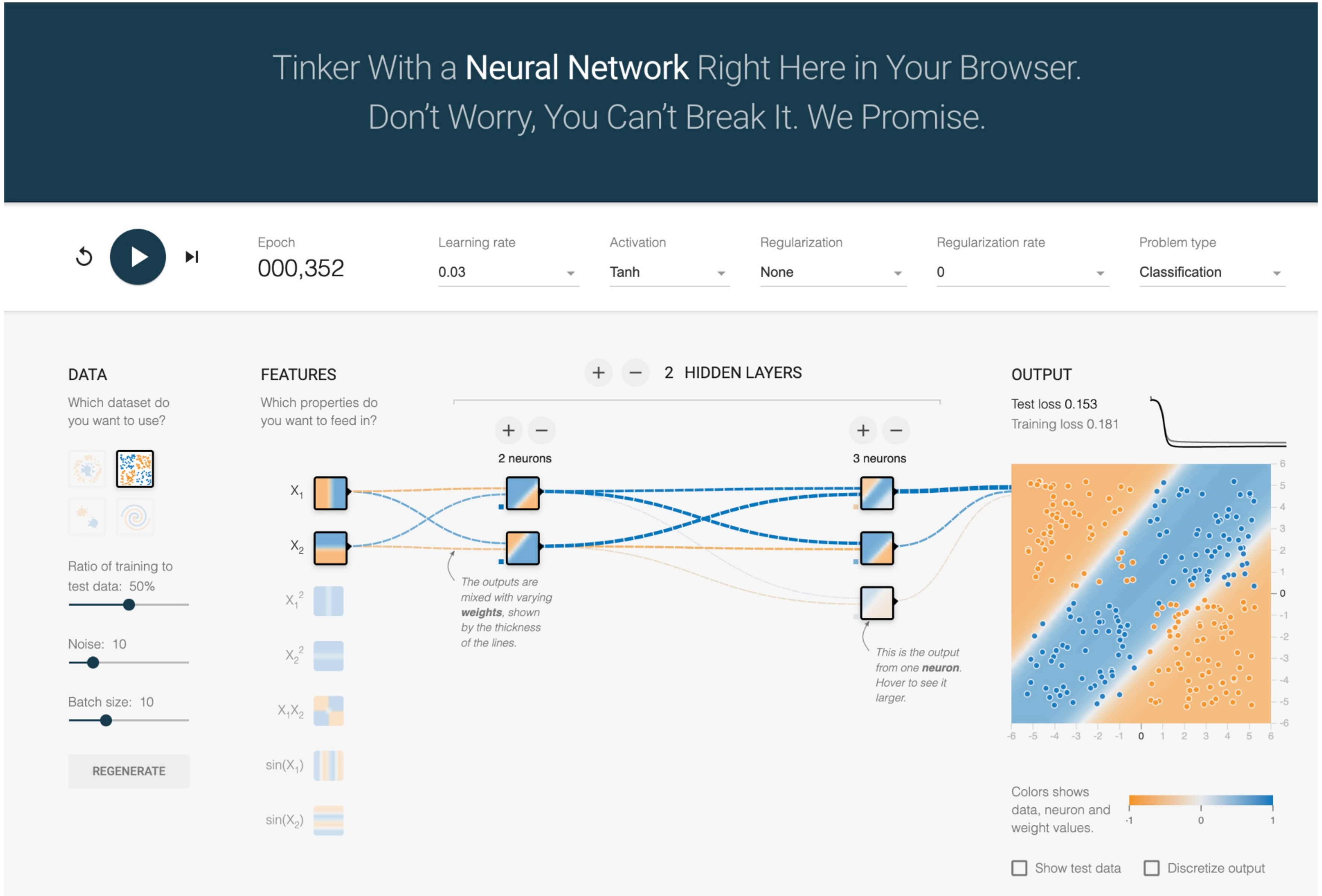


Perceptron



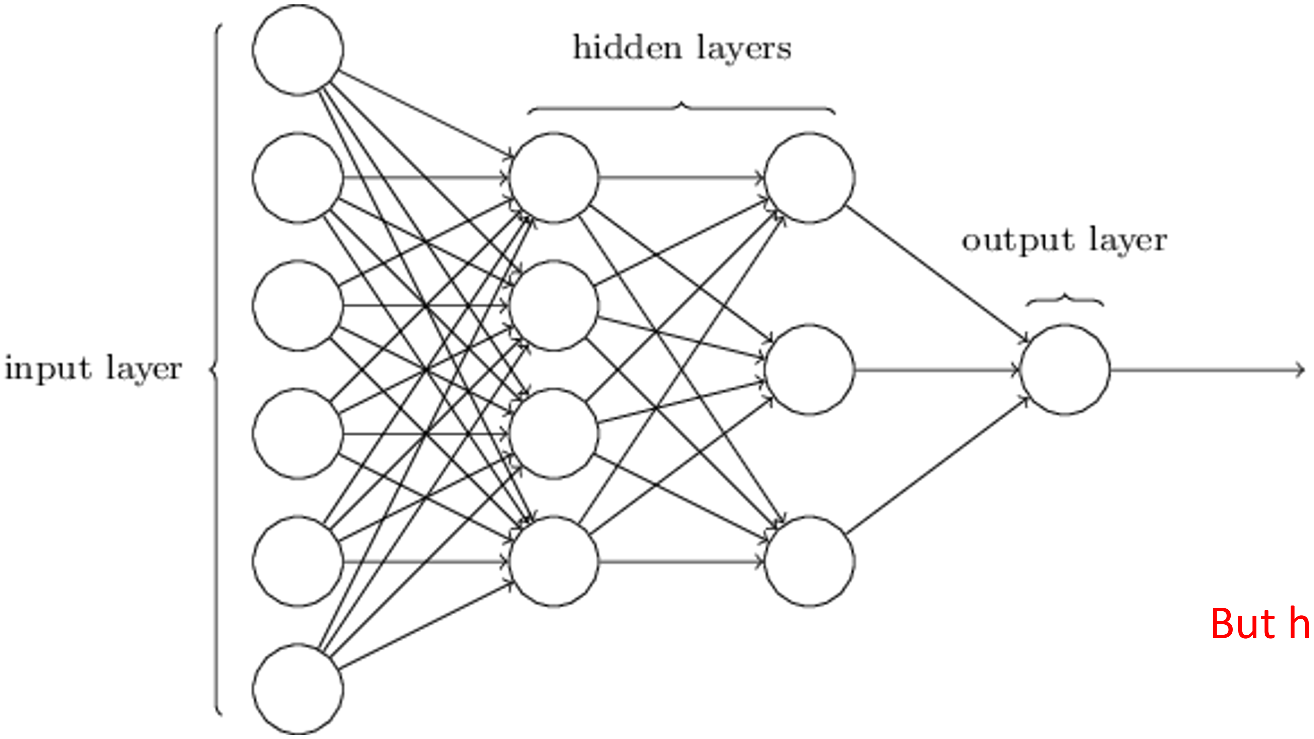
sigmoid
y
x_1
x_2
h_1
h_2
y:
sigmoid + polynomial transform
h1:
sigmoid
h2:
sigmoid
x
y
z

Let's play with it!



Try it [here]()

Architecture of Neural Networks



But how do we train it?

- Sometimes called multi-layer perceptron (MLP)
- Output from one layer is used as input for the next (Feedforward network)

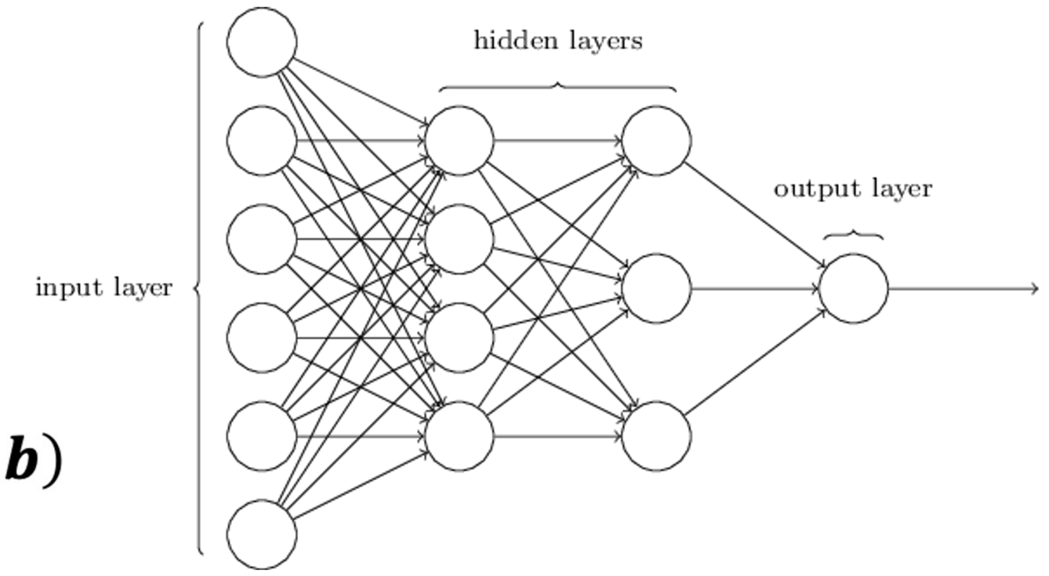
Forward Propagation

- Store weights and biases as matrices
- Suppose we are considering the weights from the second (hidden) layer to the third (output) layer
  - w is the weight matrix with w_{ij} the weight for the connection between the ith neuron in the second layer and the j^{th} neuron in the third layer
  - b is the vector of biases in the third layer
  - a is the vector of activations (output) of

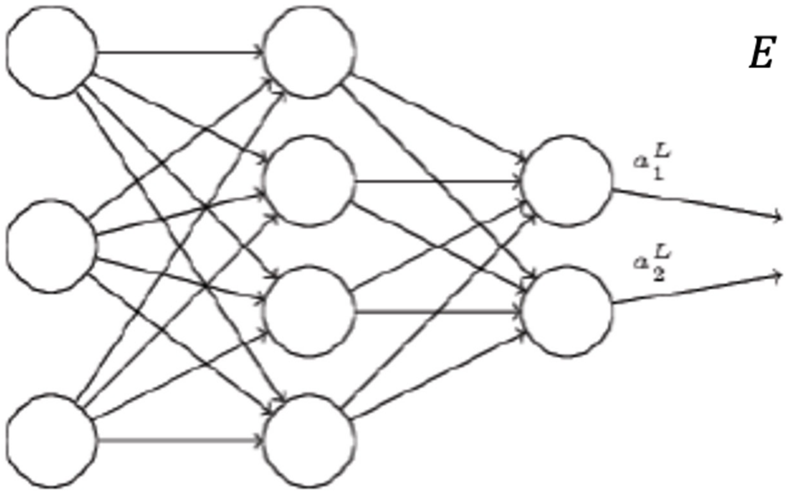
  the 2nd layer
  - a' is the vector of activations (output) of

  the third layer

$$\boldsymbol{a'} = \boldsymbol{\sigma}(\boldsymbol{wa} + \boldsymbol{b})$$



Backpropagation



$$E = \frac{1}{2}\sum_i \left(a_i^L - y_i\right)^2$$

$$cost\ E = E(a^L)$$

1. **Input x:** Set the corresponding activation a^1 for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

$$z_j^l = \sum_i w_{ji}^l a_i^{l-1} + b_j^l \qquad a_j^l = \sigma\left(\sum_i w_{ji}^l a_i^{l-1} + b_j^l\right) = \sigma(z_j^l)$$

$$\delta_j^L \equiv \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L}\sigma'(z_j^L) \qquad (1)$$

$$\delta_j^l \equiv \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial z_i^{l+1}}\frac{\partial z_i^{l+1}}{\partial z_j^l} = \frac{\partial z_i^{l+1}}{\partial z_j^l}\delta_i^{l+1}$$

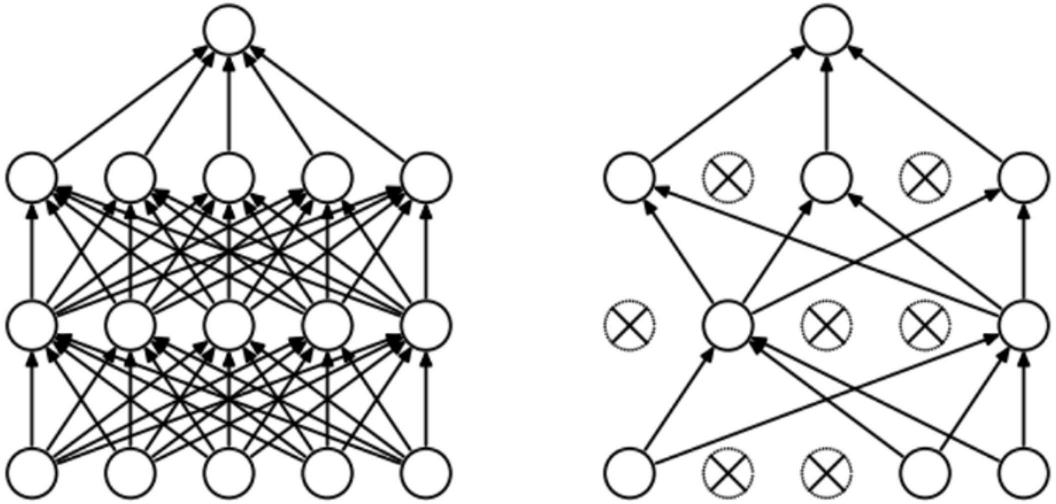
$$= \frac{\partial(\sum_i w_{ij}^{l+1} a_j^l + b_i^{l+1})}{\partial z_j^l}\delta_j^{l+1} = \sum_i w_{ij}^{l+1}\delta_i^{l+1}\sigma'(z_j^l) \qquad (2)$$

$$\frac{\partial E}{\partial w_{ji}^l} = \frac{\partial E}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_j^l}\frac{\partial(w_{ji}^l a_i^{l-1})}{\partial w_{ji}^l}$$

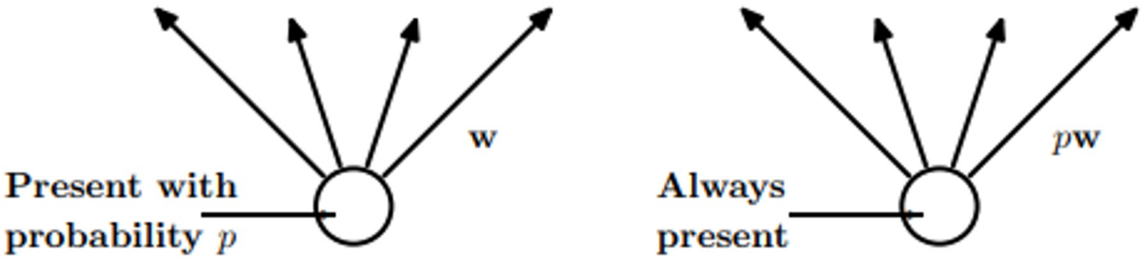
Extra Regularization for Neural Nets

Dropout: accuracy in the absence of certain information

- Prevent dependence on any one (or any small combination) of neurons



(a) Standard Neural Net (b) After applying dropout.



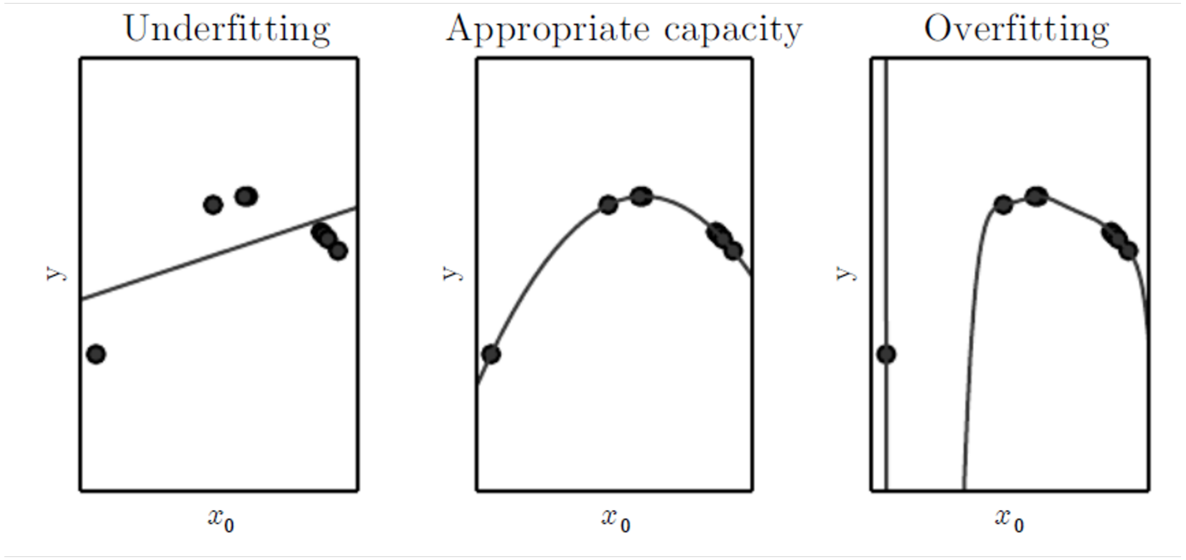
(a) At training time (b) At test time

Capacity, Overfitting and Underfitting

1) Make training error small
2) Make the gap between training and test error small

Capacity (fitting ability)

Low capacity ← $\hat{y} = \sum_{i}^{1} w_i x^i$ $\qquad$ $\hat{y} = \sum_{i}^{2} w_i x^i$ $\qquad$ $\hat{y} = \sum_{i}^{9} w_i x^i$ → High capacity



Time for a quiz and tutorial!

https://tinyurl.com/GeoComp2024

Back to the code

Open: -
FeedForward_Networks_Class4.ipynb

When people want to use Machine Learning without math

How training works

1. In each ***epoch***, randomly shuffle the training data
2. Partition the shuffled training data into ***mini-batches***
3. For each mini-batch, apply a single step of **gradient descent**
   - **Gradients** are calculated via ***backpropagation*** (the next topic)
4. Train for multiple epochs

Debugging a neural network

- What can we do?
  - Should we change the learning rate?
  - Should we initialize differently?
  - Do we need more training data?
  - Should we change the architecture?
  - Should we run for more epochs?
  - Are the features relevant for the problem?
- Debugging is an art
  - We’ll develop good heuristics for choosing good architectures and hyper parameters

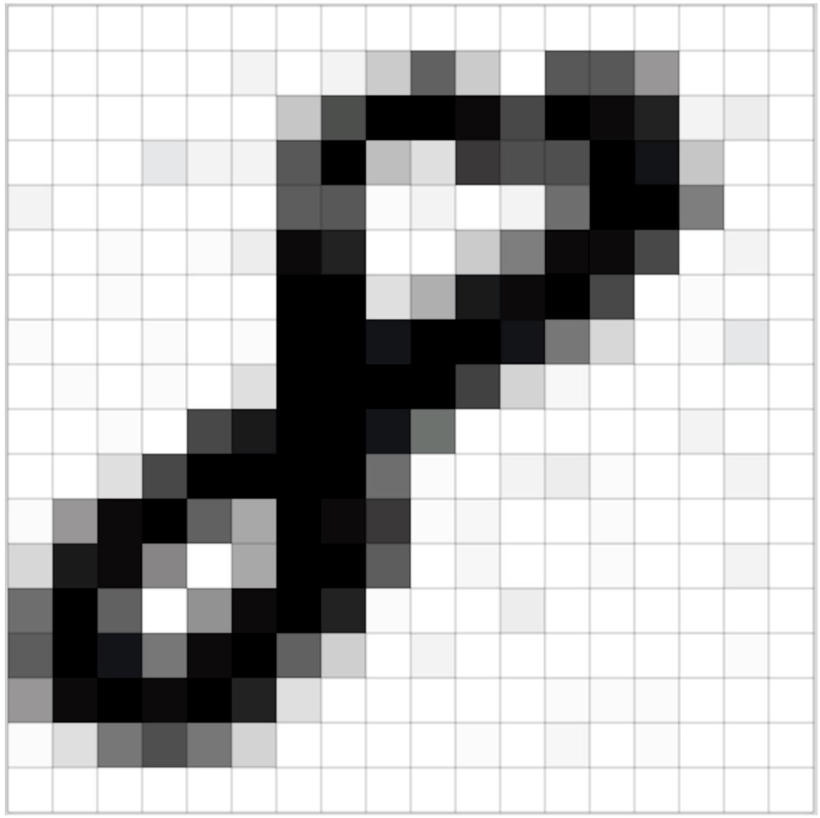
Extra readings

Deep Learning book:

- Chapter 5.9: Intro to Stochastic Gradient Descent (SGD)
- Chapter 6: Multilayer perceptrons
- Chapter 6.2.2: Output Units (Activation functions)
- Chapter 6.5: Back-Propagation
- Chapter 8.3: Basic Algorithms (Optimizers)

Convolutional Neural Networks

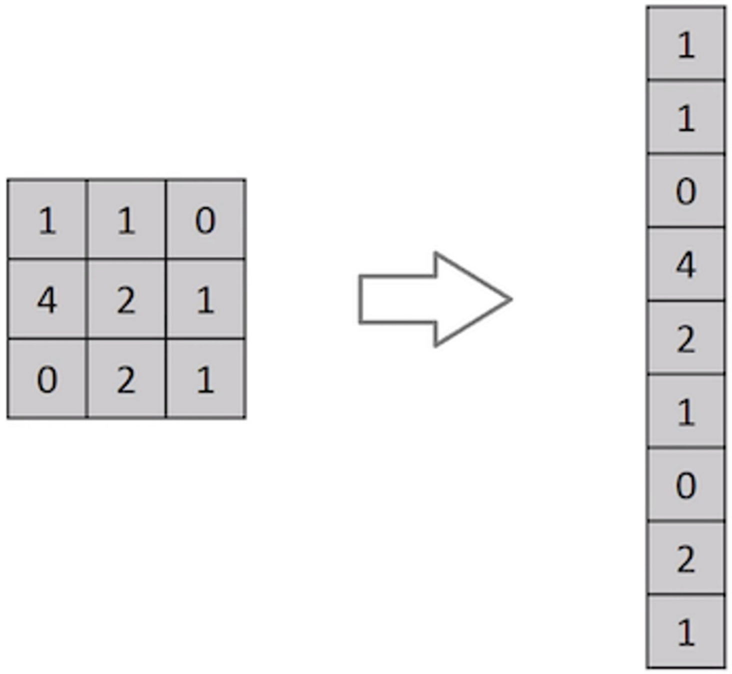
Images are a series of Pixel Values

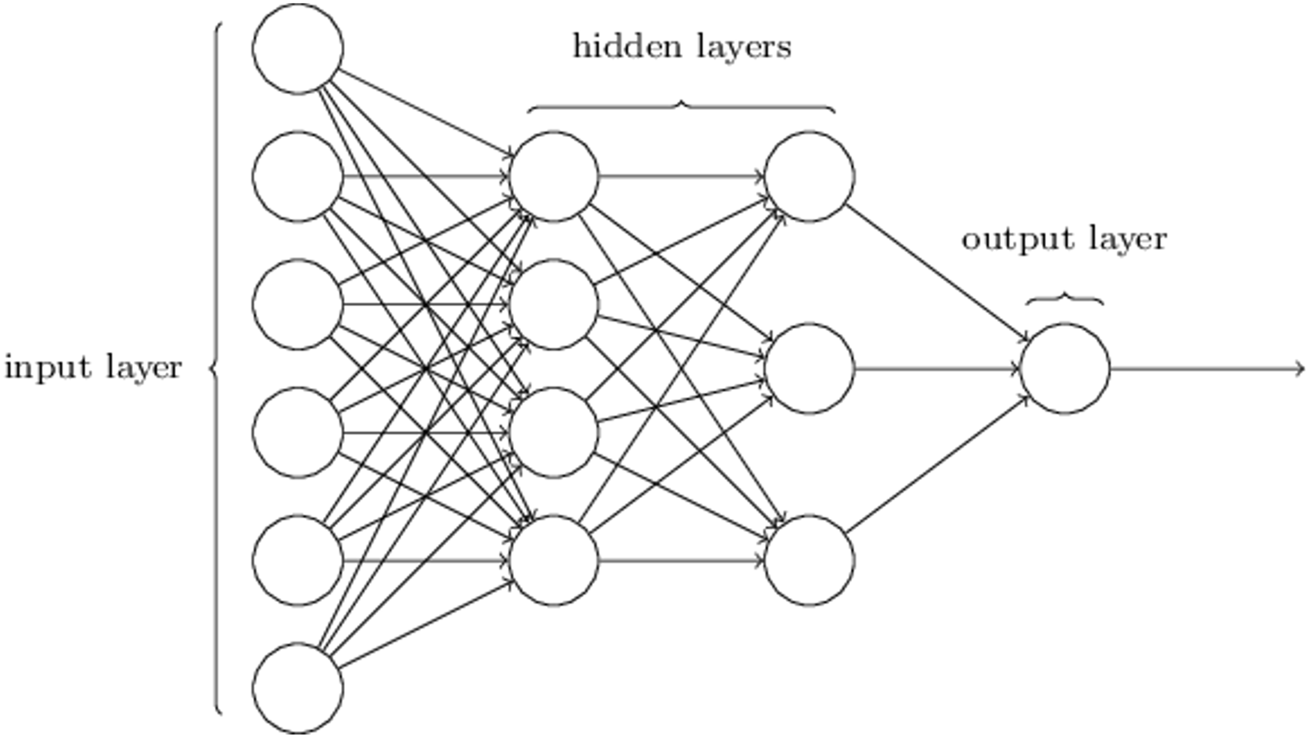


Grayscale images:
0=Black
255 = White

Spatial locality structure

Handling images with Neural Networks





Works well for simple images, but fails when there are more complex patterns in the image

Local receptive fields

Make connections in small, localized regions of the input image

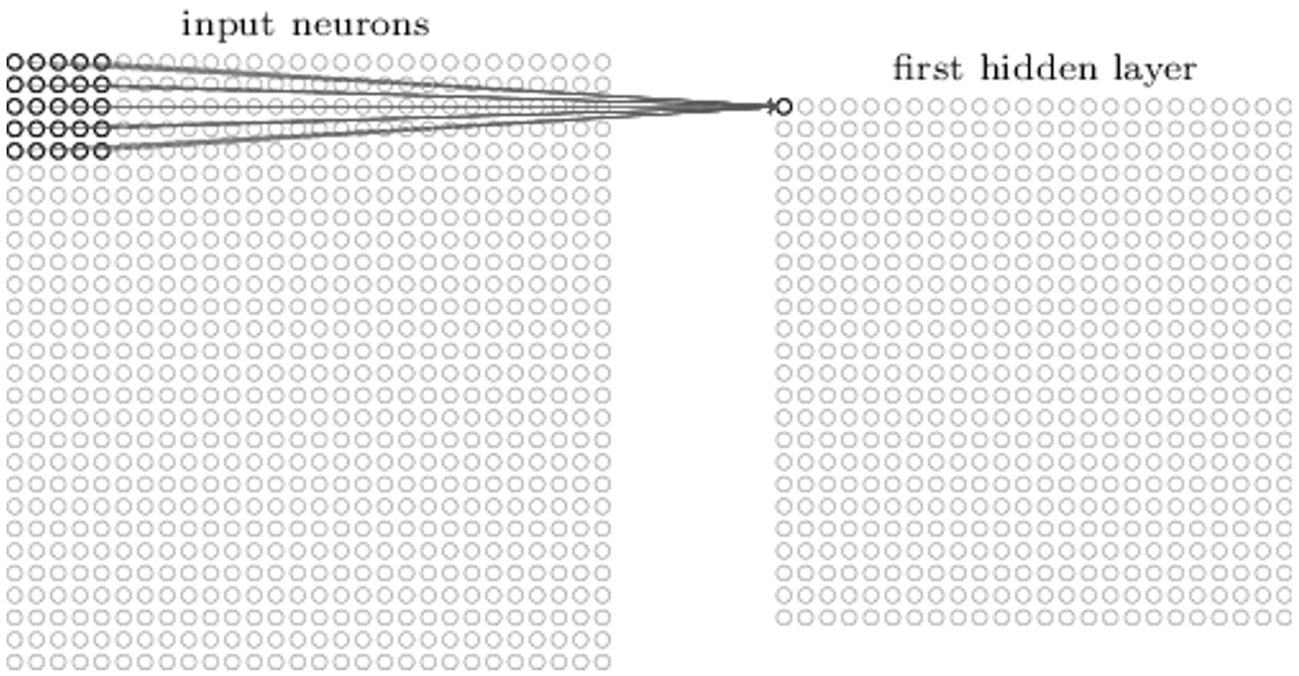


Image taken from Michael Neilsen's book "Neural Networks and Deep Learning"

Local receptive fields

Slide the local receptive field over by one (or more) pixel and repeat

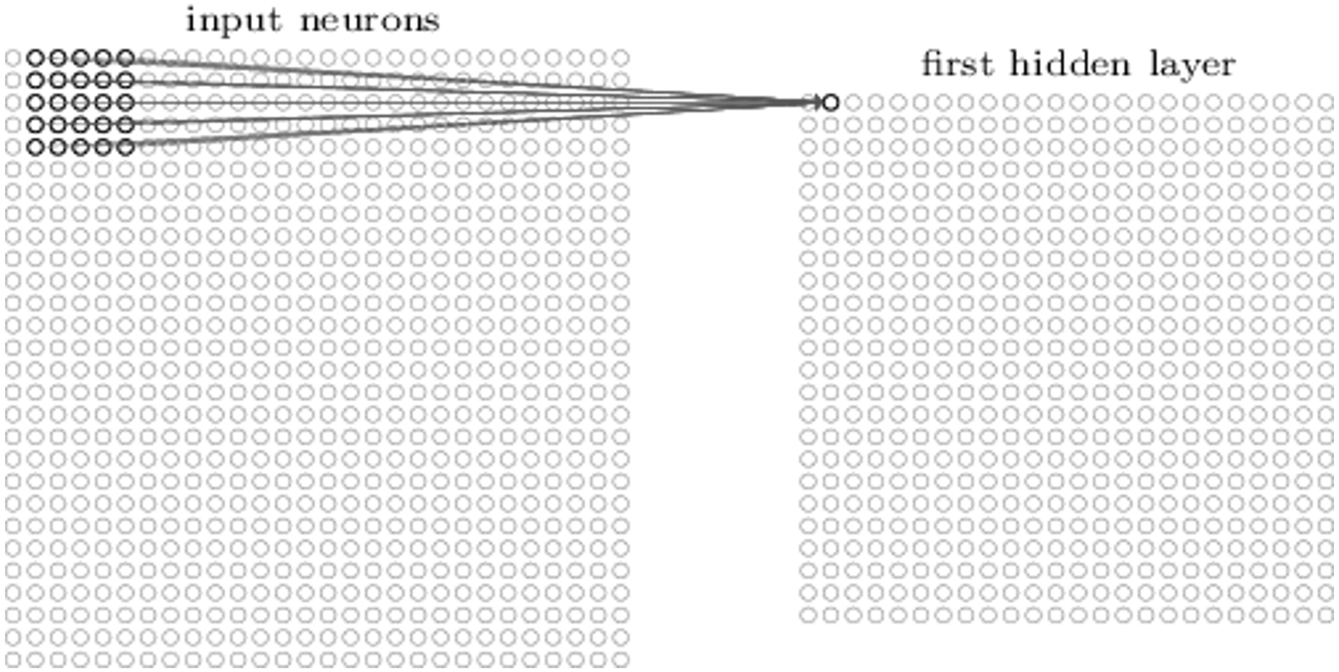


Image taken from Michael Neilsen's book "Neural Networks and Deep Learning"

The convolution operation

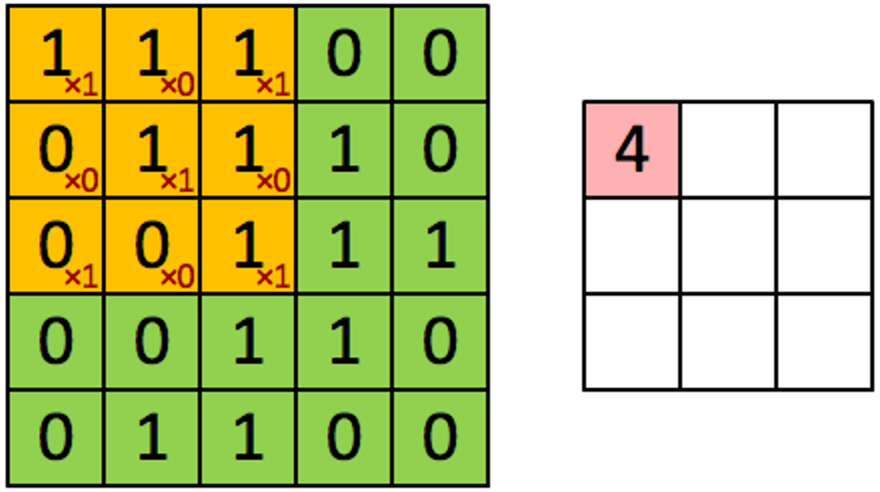
Image

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

1	0	1
0	1	0
1	0	1

Filter/
Feature detector

1. Pointwise multiply
2. Add results
3. Translate filter



Image

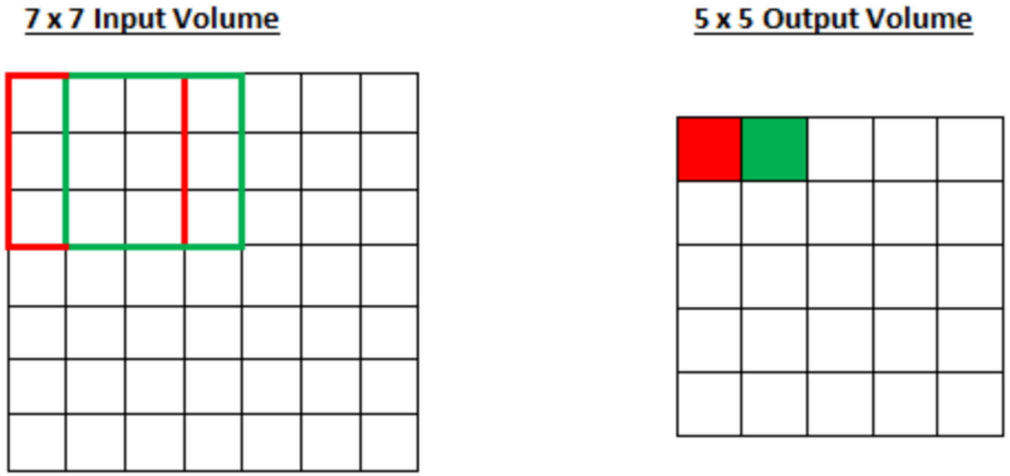
Convolved
Feature

Filters

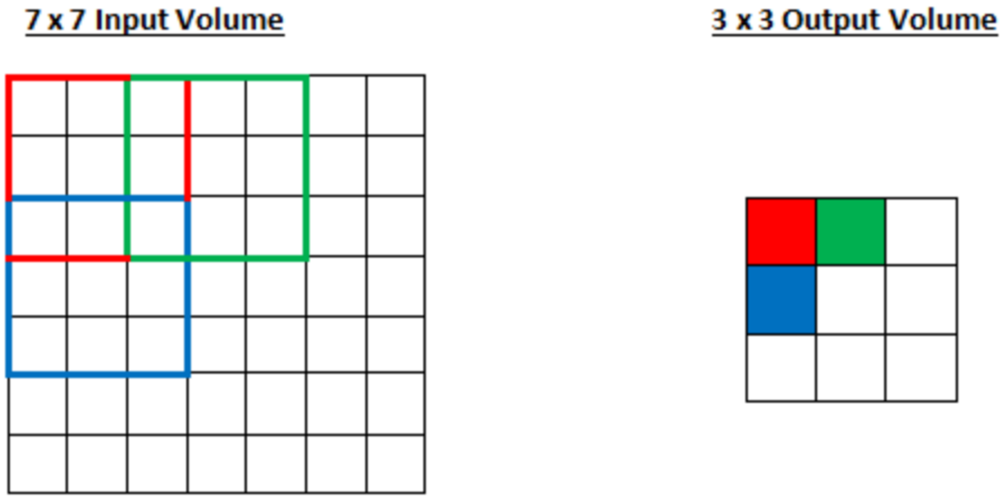
Original Image

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Stride



Stride 1



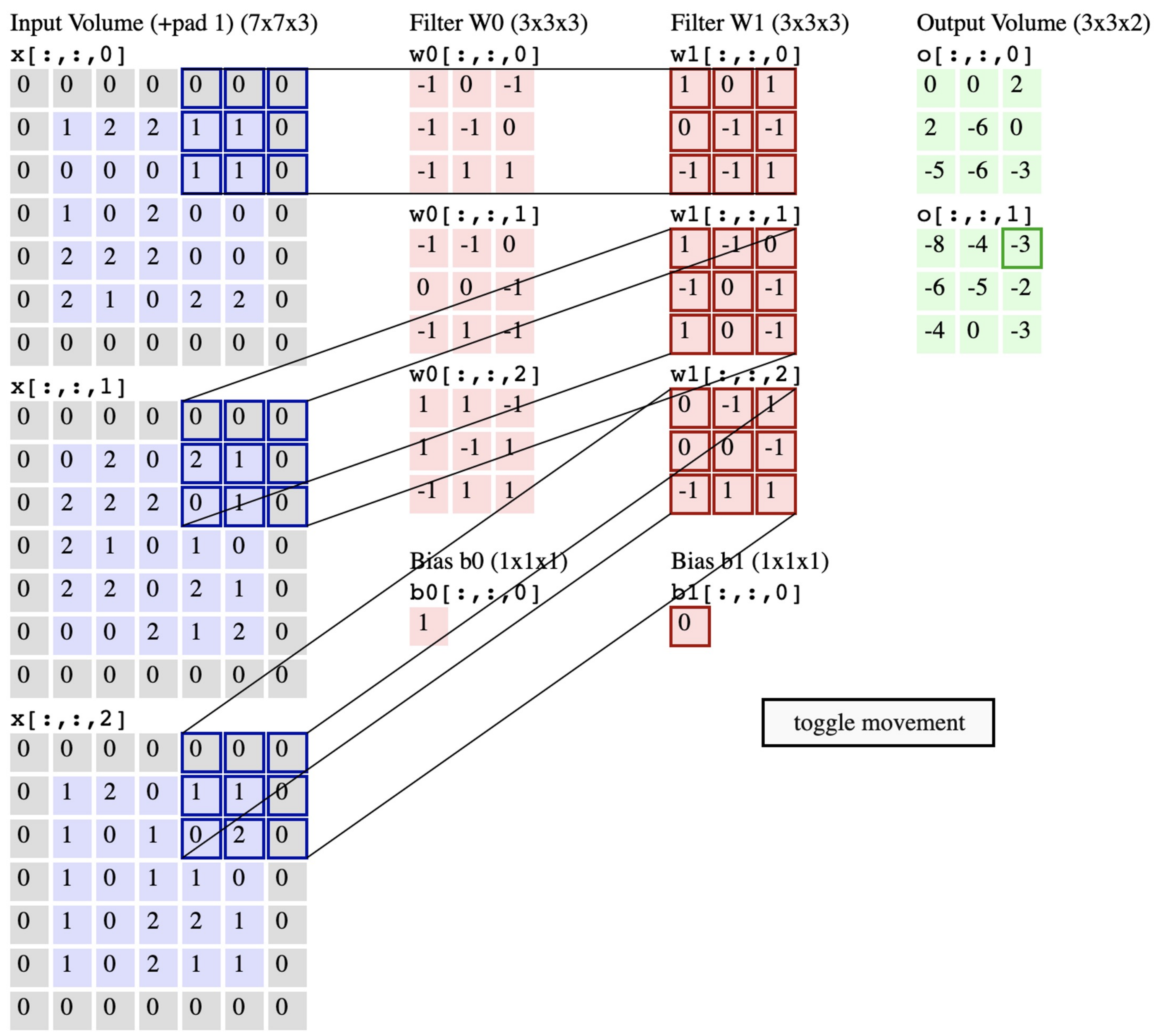
Stride 2

CNN over the image channels

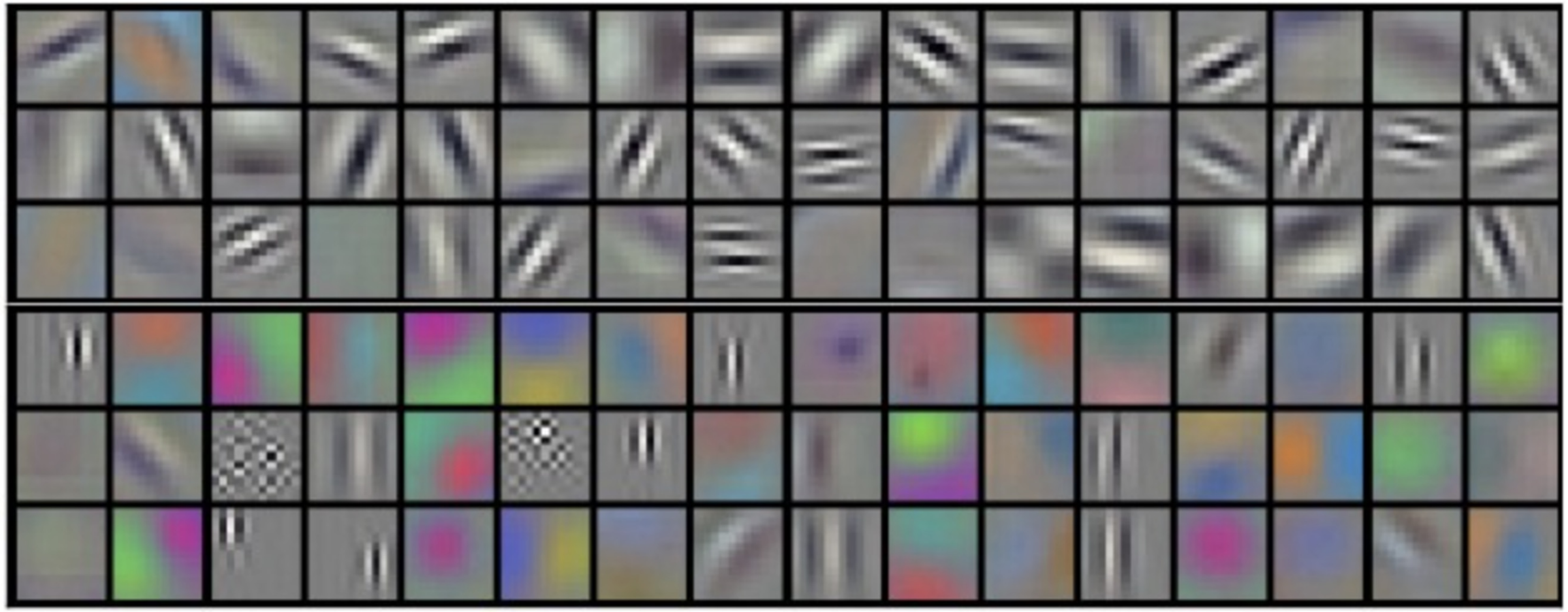
- Input: $W \times H \times D$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P
- Output: $W_2 \times H_2 \times D_2$

where:

- $W_2 = (W - F + 2P)/S + 1$
- $H_2 = (H - F + 2P)/S + 1$
- $D_2 = K$

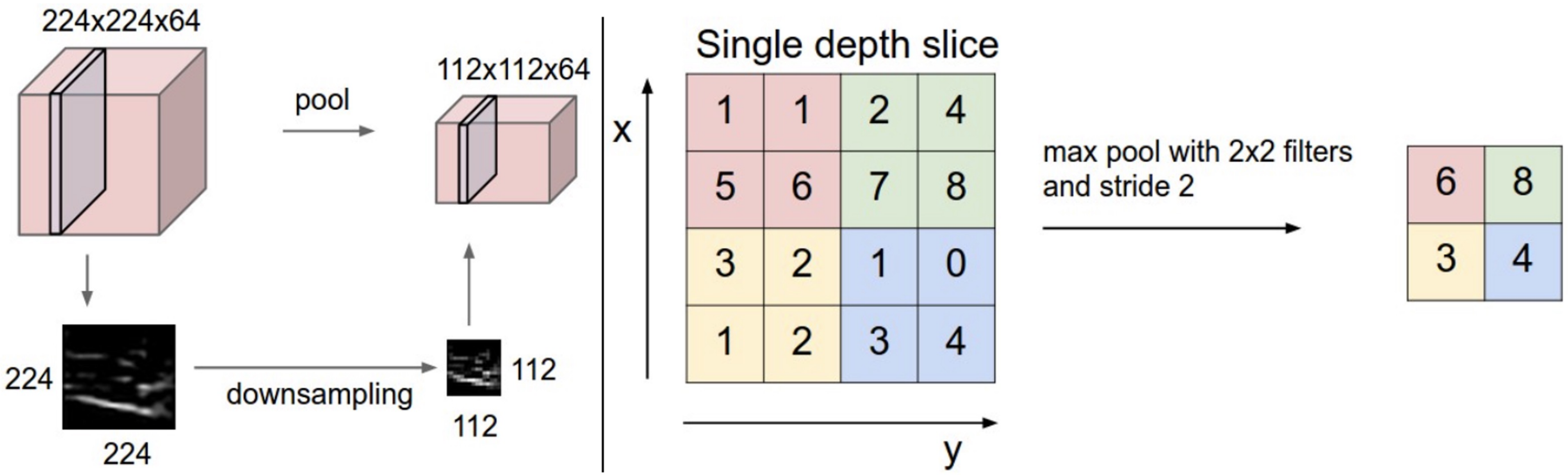


Kernels



Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

Pooling



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Pooling layers

- Intuition: the exact location of a feature isn't as important as its rough location
  - Helps prevent overfitting
- Reduces the number of parameters needed in later layers
- L_2 pooling is also common (L_2 norm)

Fully connected layer to combine

- Convolutional layers detected features
- Pooling layers reduced complexity
- Now we have a set of feature maps

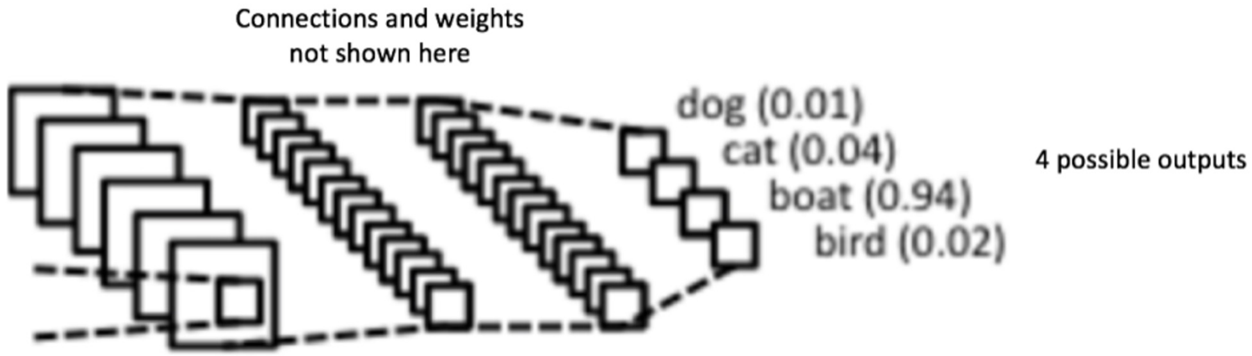
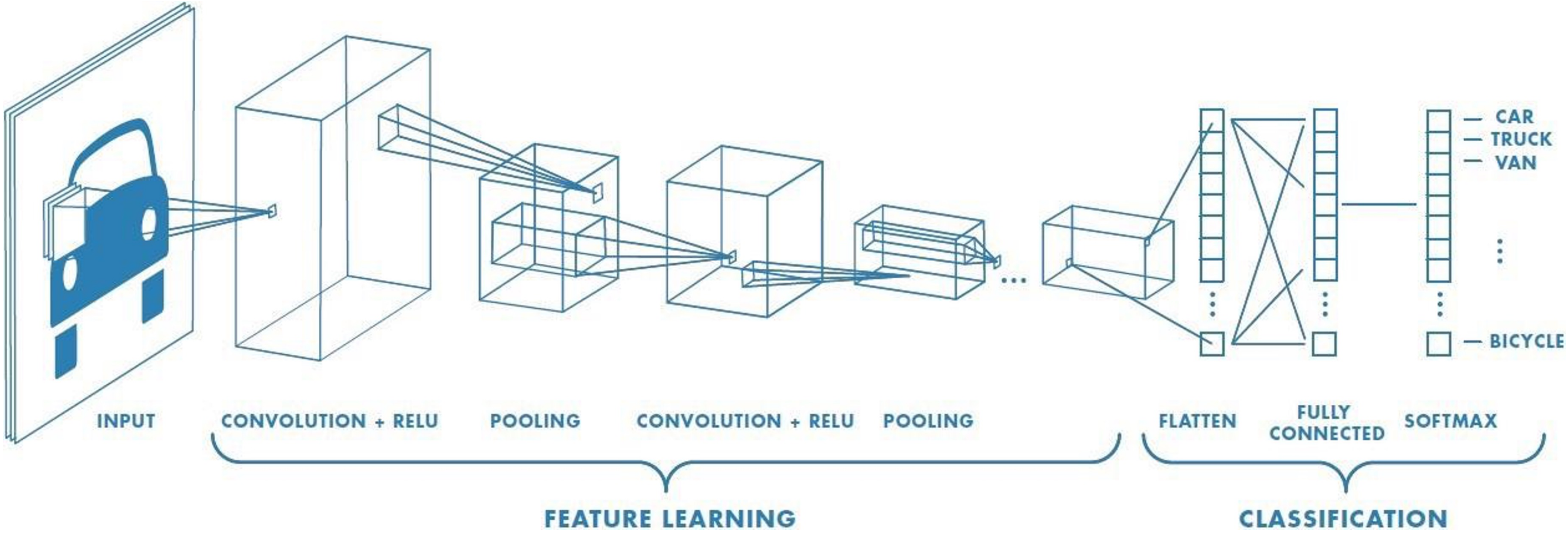


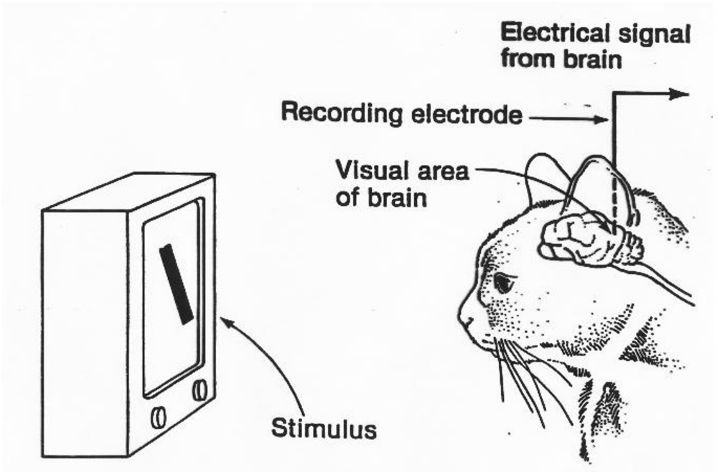
Image Classification with CNN



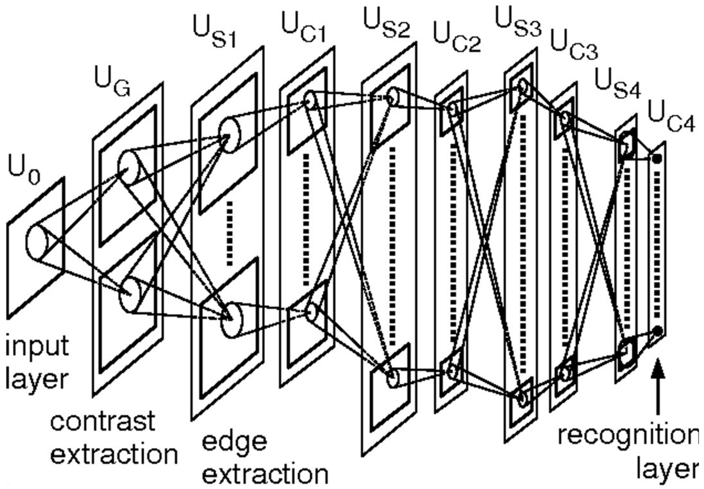
$$\mathrm{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as probability of image belonging to a particular class

CNN and brain architecture



Hubel and Wiesel, 1959-1968

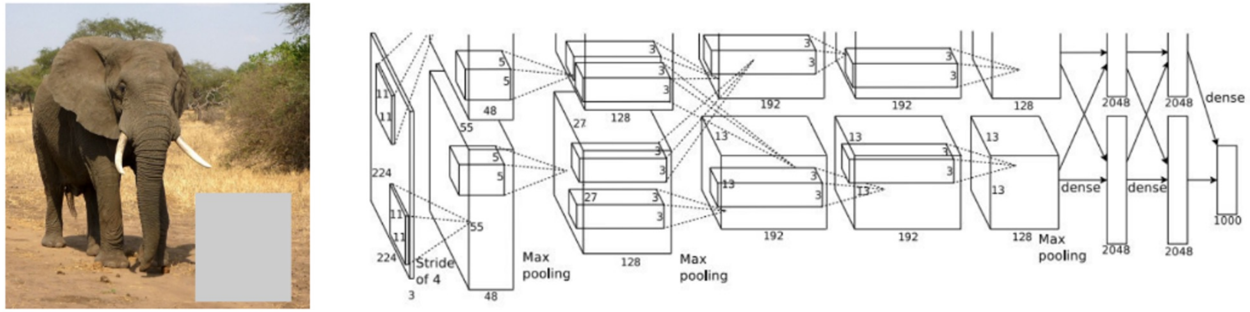


Fukushima, 1980

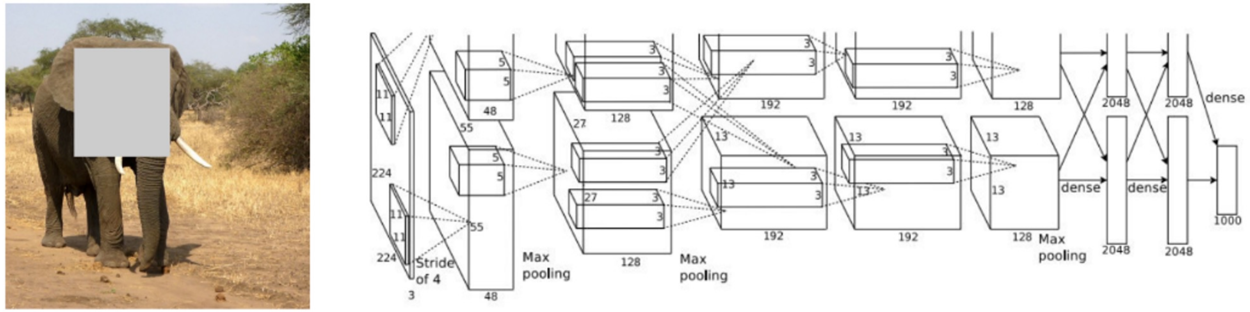
Brain “inspired” model

Which pixels matter: Saliency via Occlusion

Mask part of the image before feeding to CNN, check how much predicted probabilities change



P(elephant) = 0.95



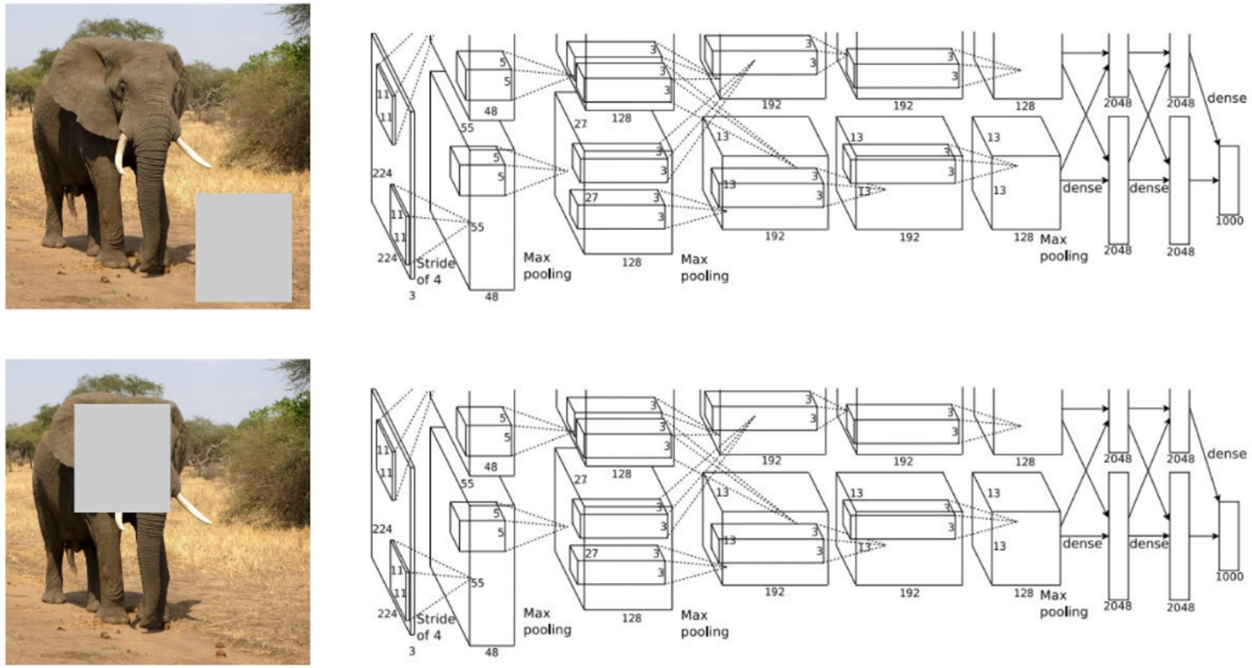
P(elephant) = 0.75

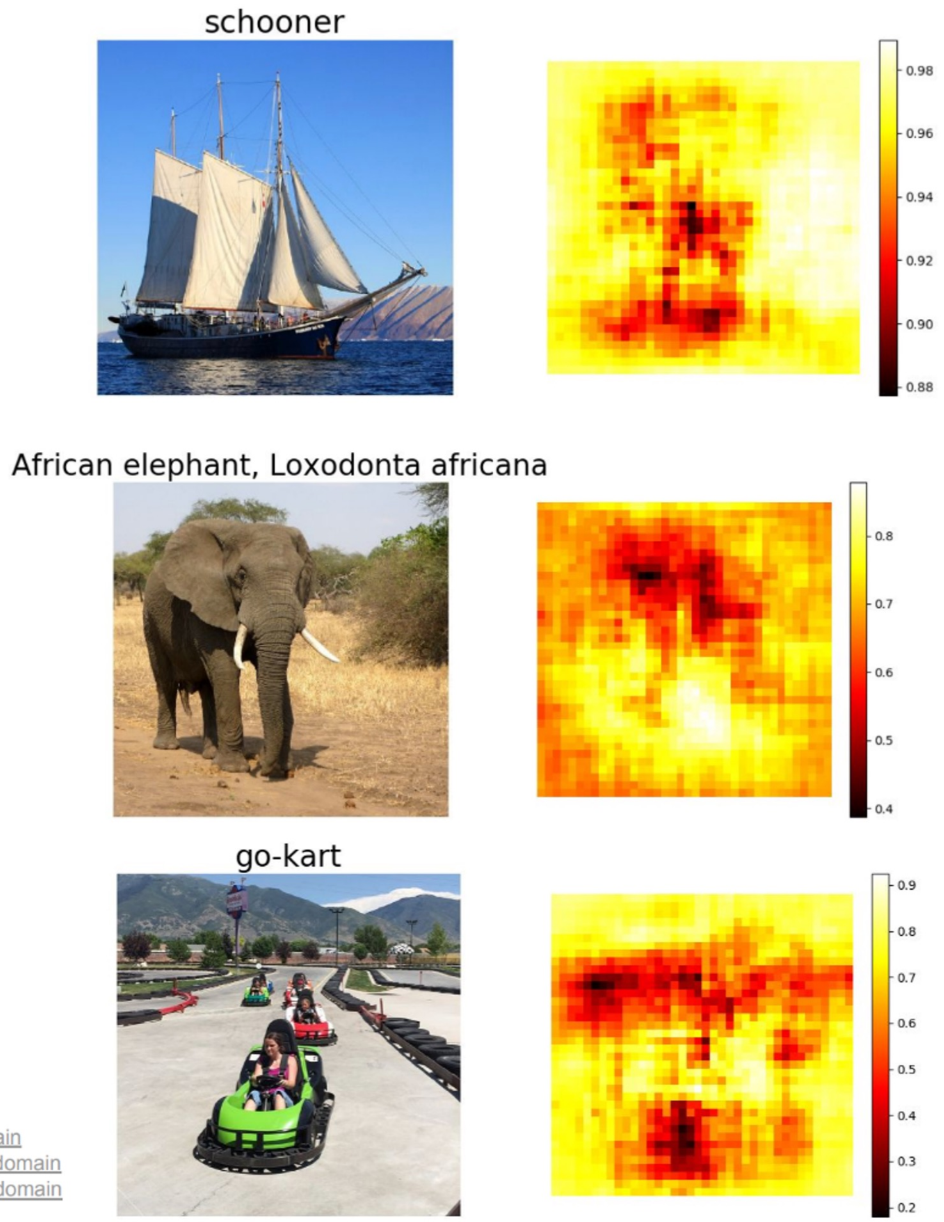
Zeiler and Fergus, “Visualizing and Understanding Convolutional Networks”, ECCV 2014

Slide from Fei-Fei Li, Standford lecture

Which pixels matter: Saliency via Occlusion

Mask part of the image before feeding to CNN, check how much predicted probabilities change

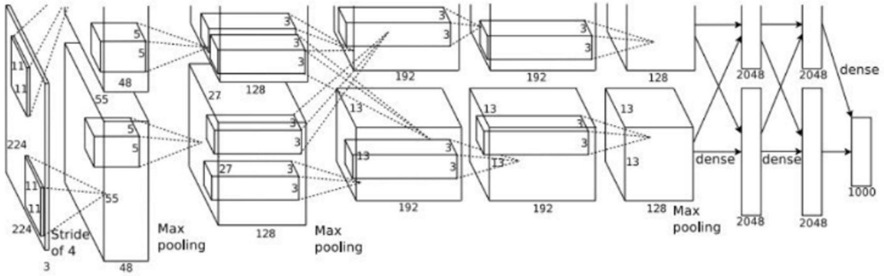




Zeiler and Fergus, “Visualizing and Understanding Convolutional Networks”, ECCV 2014

Which pixels matter: Saliency via Backprop

Forward pass: Compute probabilities

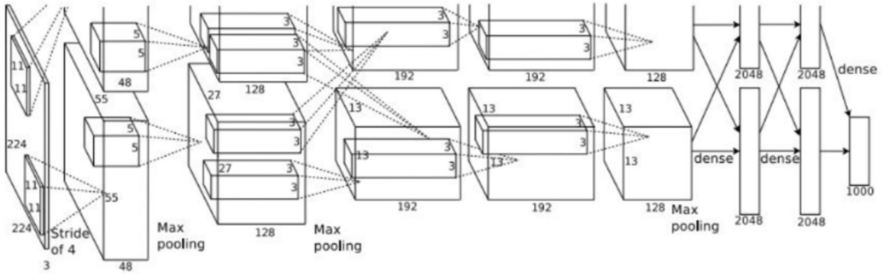


Dog

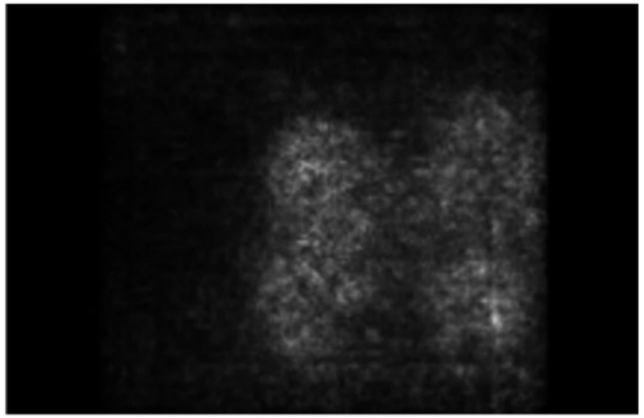
Simonyan, Vedaldi, and Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

Which pixels matter: Saliency via Backprop

Forward pass: Compute probabilities



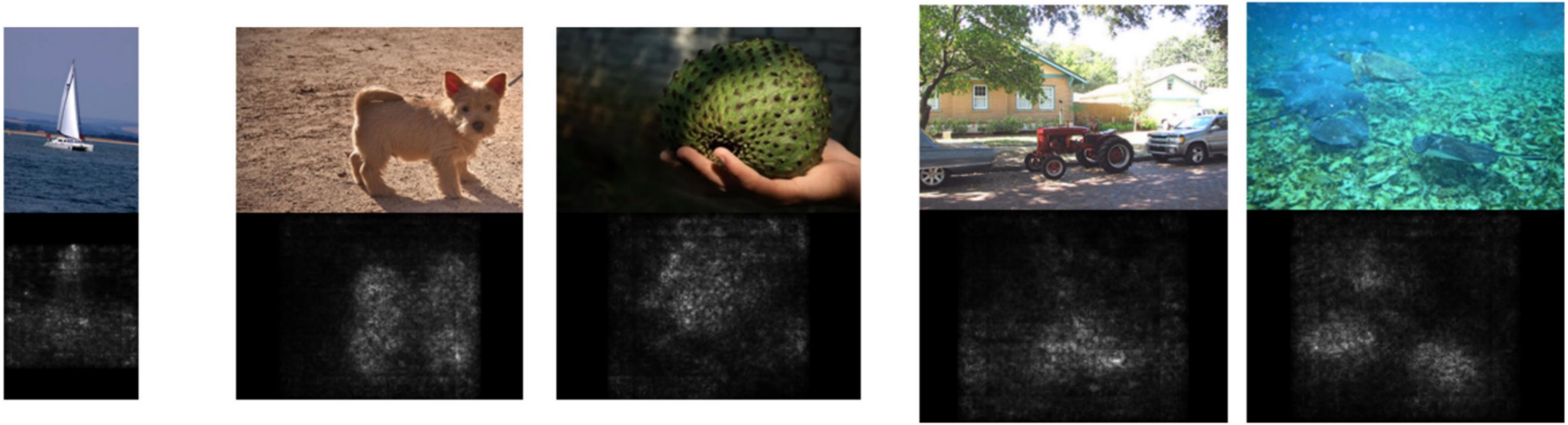
Dog



Compute gradient of (unnormalized) class score with respect to image pixels, take absolute value and max over RGB channels

Simonyan, Vedaldi, and Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

Saliency Maps



Simonyan, Vedaldi, and Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

Time for a quiz and tutorial!

https://tinyurl.com/GeoComp2024