Principles of Deep Learning I

Itthi Chatnuntawech
Nanoinformatics and Artificial Intelligence (NAI)

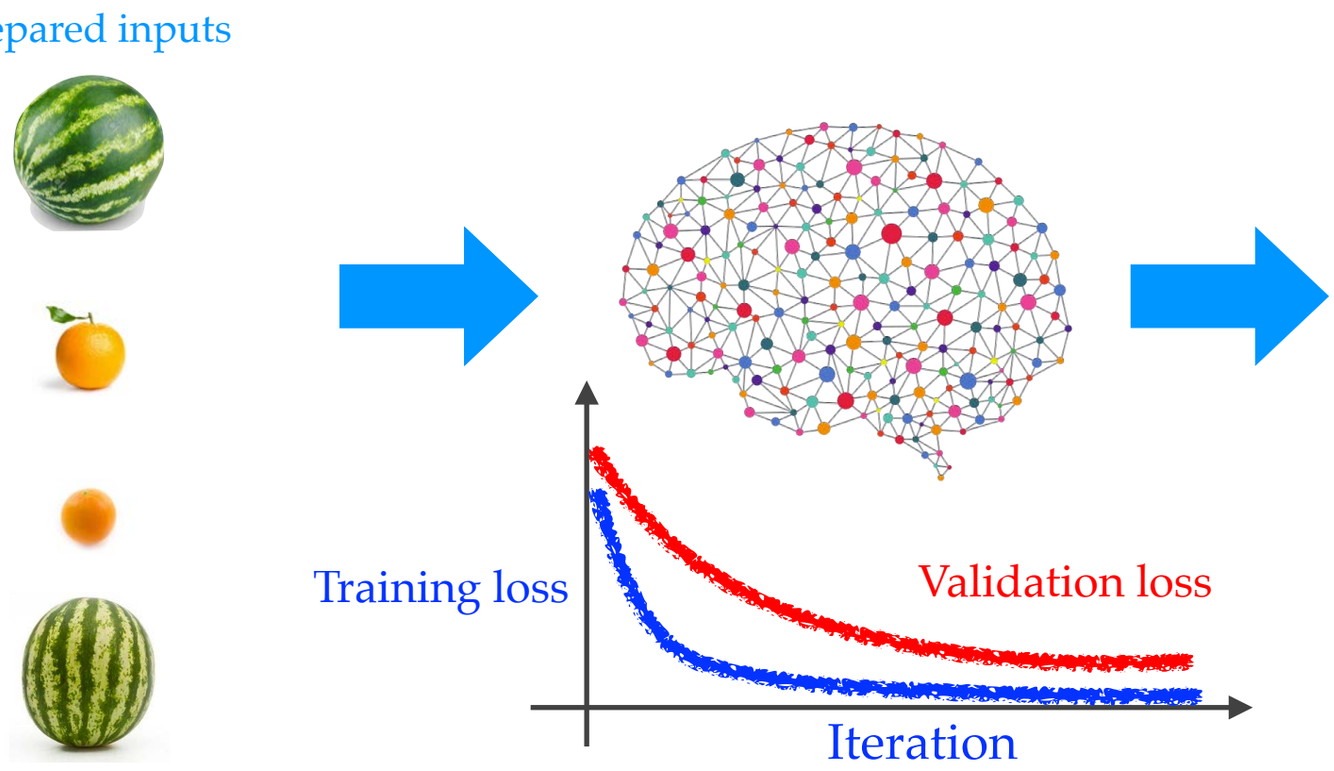
November 14, 2022

Outline

- Deep Learning Components
  - Fully connected layer
  - Computation graph
  - Activation functions
  - Loss function
  - Model optimization
    - Gradient descent
    - Backpropagation
    - Stochastic gradient descent (SGD)
  - Regularization
    - ℓ_2 and ℓ_1 regularization
    - Dropout
    - Batch normalization
  - Convolutional layer
  - Pooling layer
- Convolutional Neural Network (CNN)

From Last Time: Image Classification Using Deep Learning

- Training phase: Optimize the weights of a deep neural network



- Test phase: Perform prediction using the trained neural network

Fully connected layer
(Dense)

Optimizer

SGD
Adam
RMSprop

Convolutional layer

Conv1D, 2D, 3D, …
separable Conv

Deep Learning

Evaluation metric

accuracy
F1-score
AUC
confusion matrix

Pooling layer

max-pooling
average-pooling

Activation function

sigmoid
softmax
ESP (swish)
ReLU

Loss function

categorical crossentropy
binary crossentropy
mean squared error
mean absolute error

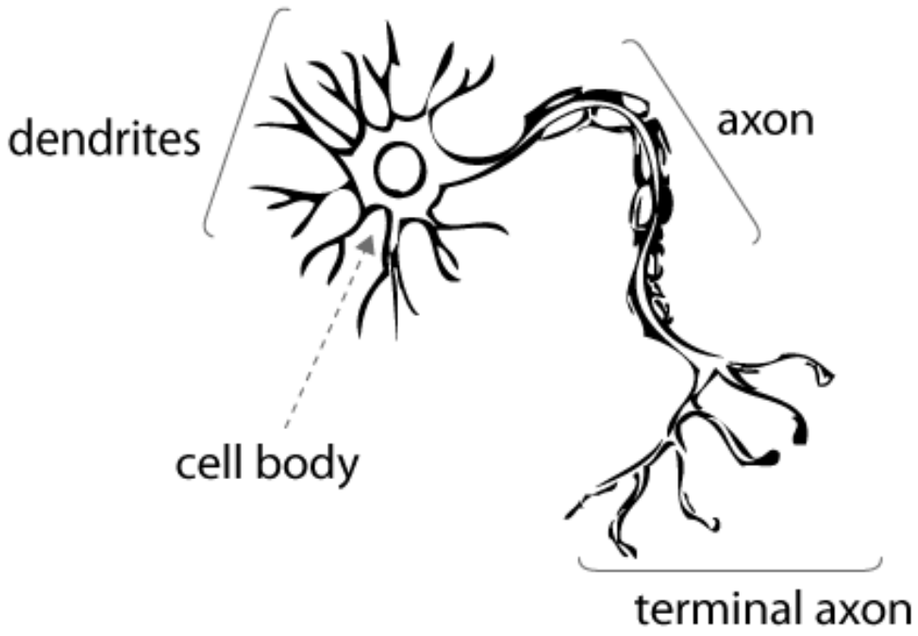
Regularization

Dropout
Data augmentation
l_1, l_2 regularizations

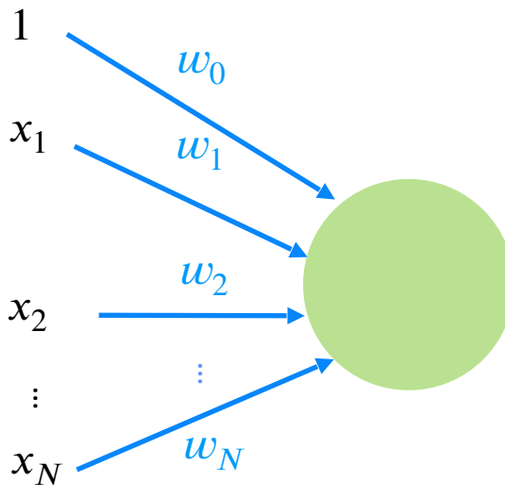
- Combine basic components to build a neural network
  - More components → "More" representative power

Artificial Neuron

Single neuron

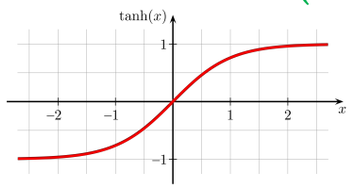


Artificial neuron



$$y = a\left(\sum_{n=1}^{N} x_n w_n + w_0\right)$$

non-linear function (activation)

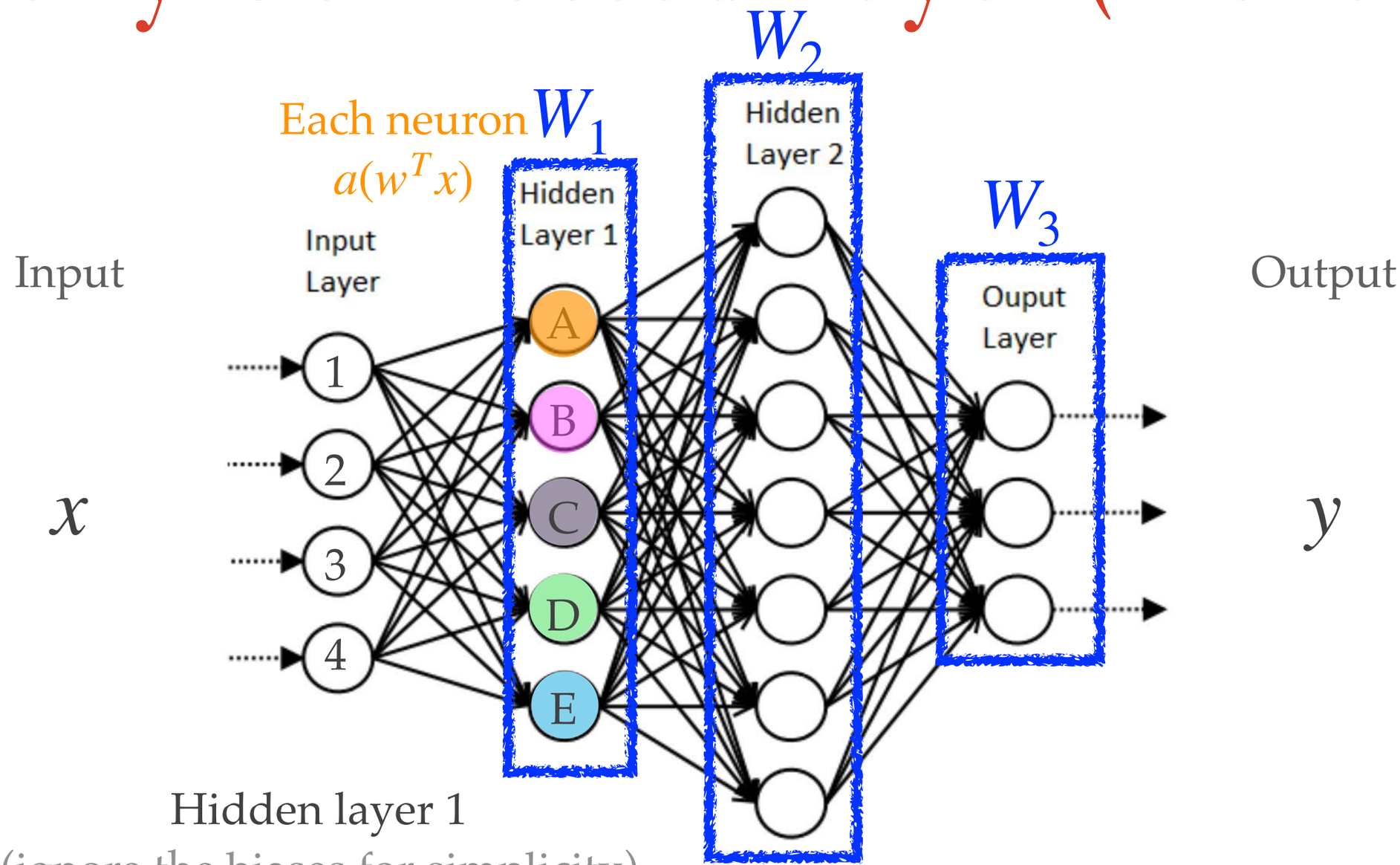


If we set all the weights to 0, then $y = 0$.
If we set $w_1 = 1$ and the rest to 0, then $y = a(x_1)$.
If we set $w_0 = 0$ and the rest to 1, then $y = a(x_1 + x_2 + \ldots + x_N)$.

Different weights give rise to different behavior

The Essense of Artificial Neural Networks by Ivan Liljeqvist (Medium)

Fully Connected Layer (Dense)



x y

Hidden layer 1

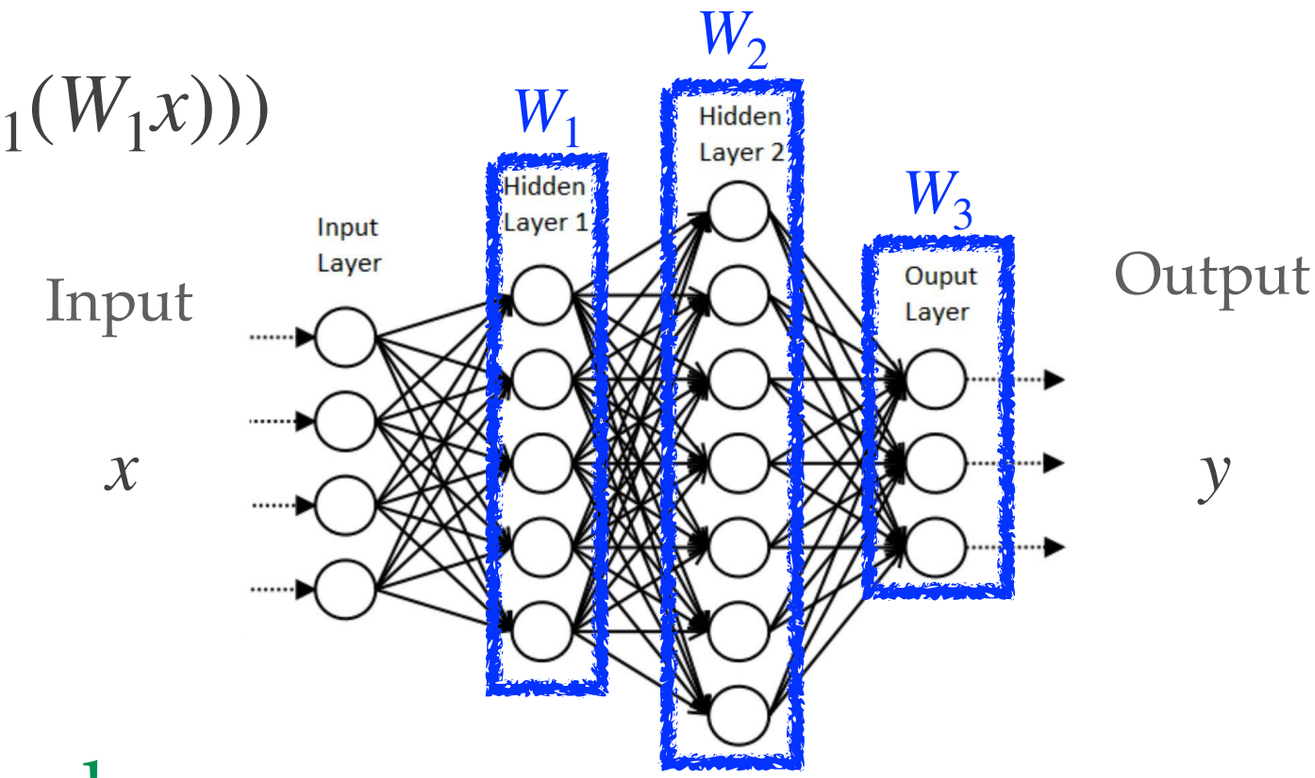
(ignore the biases for simplicity)

$$\begin{bmatrix} out_A \\ out_B \\ out_C \\ out_D \\ out_E \end{bmatrix} = a_1(W_1 x) = a_1\left(\begin{bmatrix} w_{A,1} & w_{A,2} & w_{A,3} & w_{A,4} \\ w_{B,1} & w_{B,2} & w_{B,3} & w_{B,4} \\ w_{C,1} & w_{C,2} & w_{C,3} & w_{C,4} \\ w_{D,1} & w_{D,2} & w_{D,3} & w_{D,4} \\ w_{E,1} & w_{E,2} & w_{E,3} & w_{E,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \right)$$

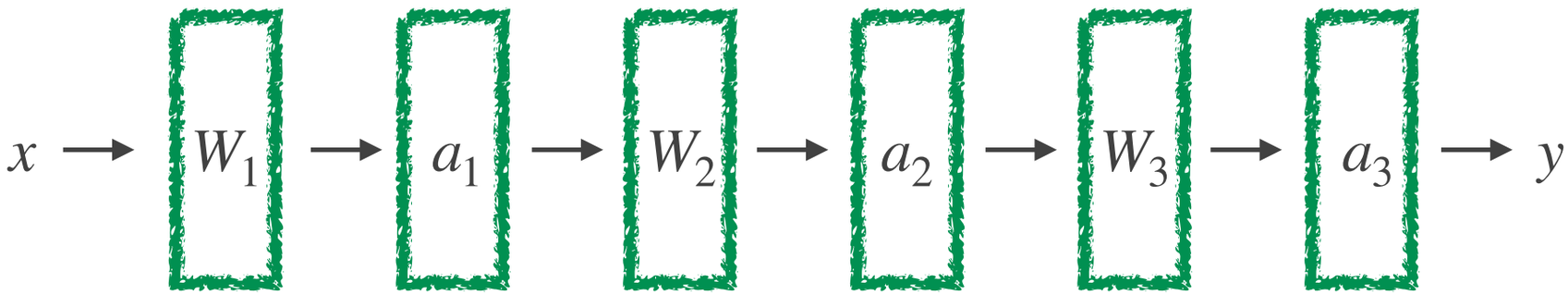
$$y = a_3(W_3 a_2(W_2 a_1(W_1 x)))$$

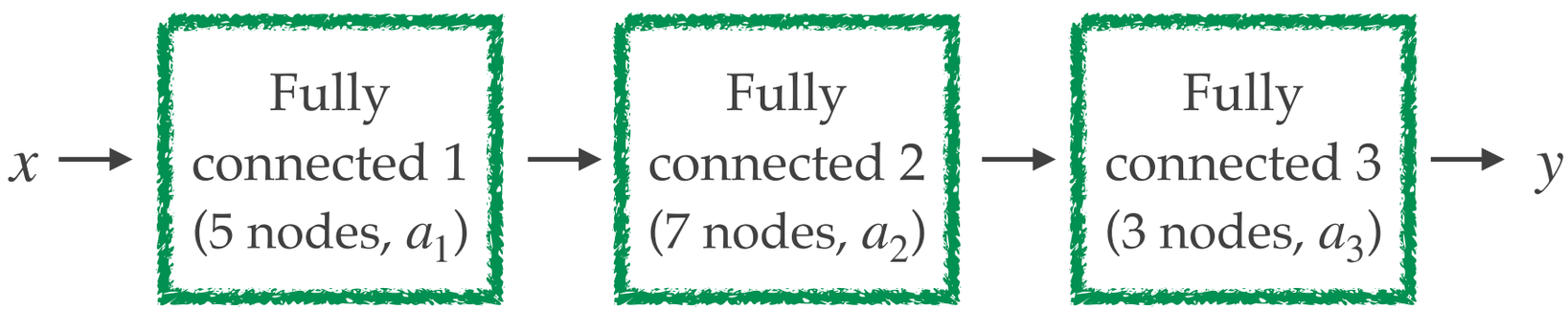
Representing Neural Network

$$y = a_3(W_3 a_2(W_2 a_1(W_1 x)))$$



Computation graph

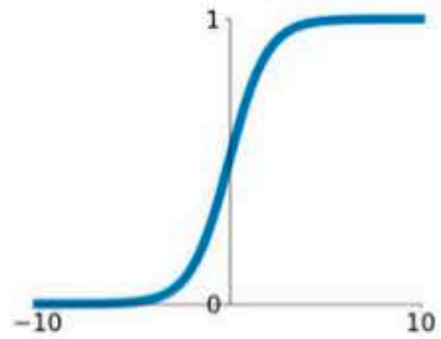




Activation Functions

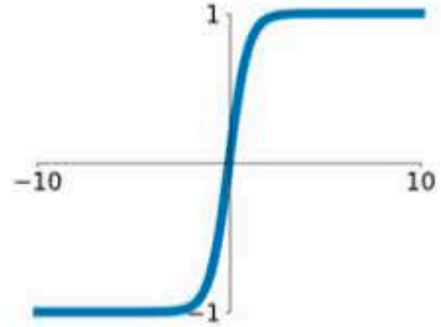
Sigmoid

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



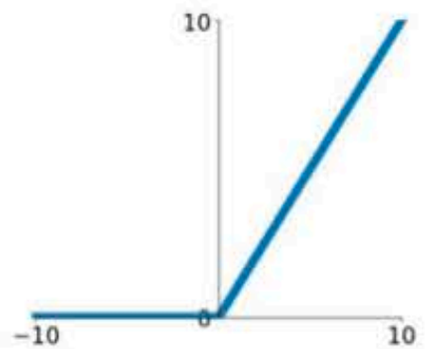
tanh

$$\tanh(x)$$



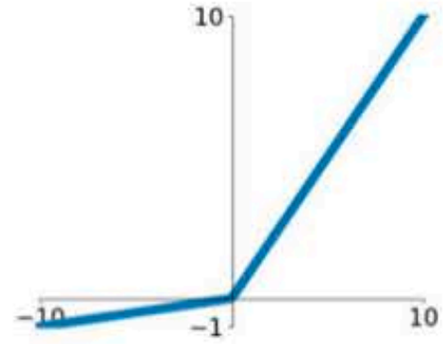
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

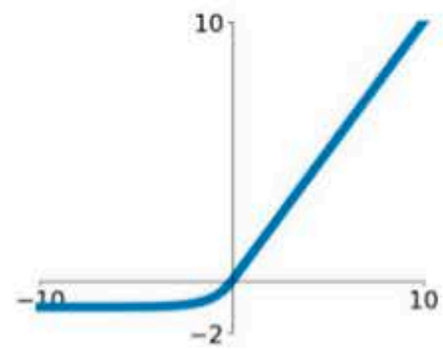


Maxout

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

ELU

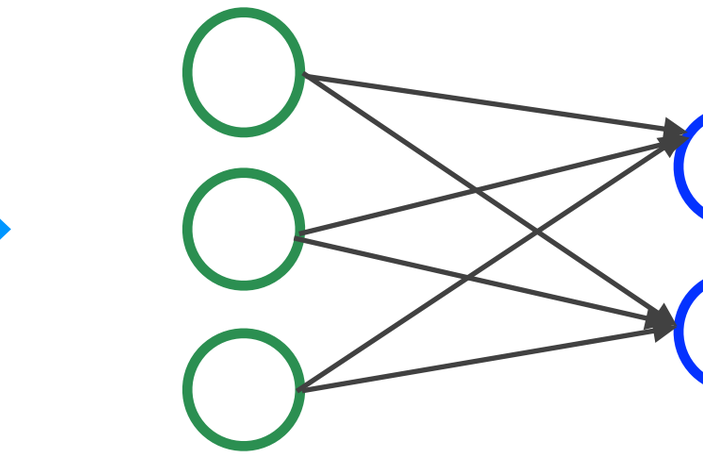
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Introduction to Exponential Linear Unit

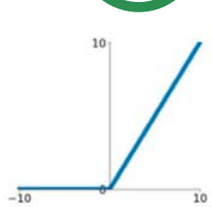
class 1 class 2

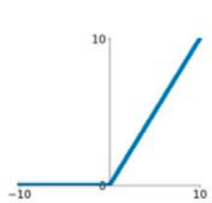
Two-class Classification

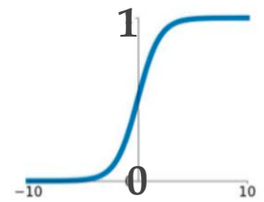


probability of being class 1

activation functions





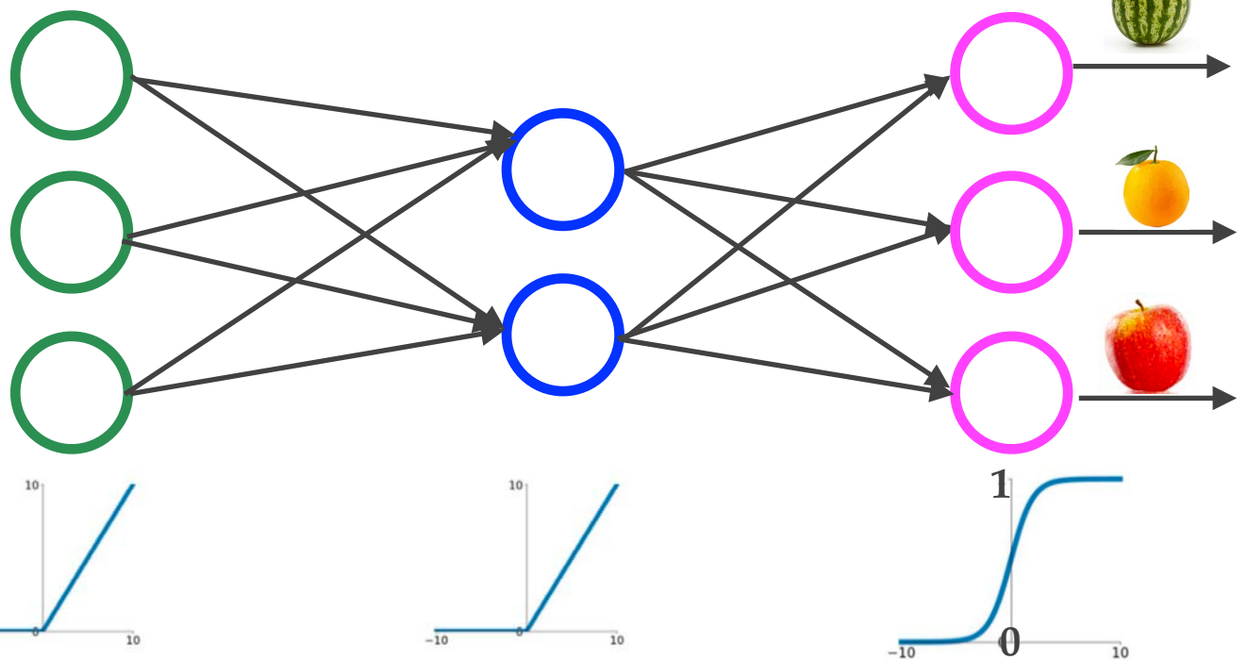


Fully connected 1 (3 nodes, ReLU) → Fully connected 2 (2 nodes, ReLU) → Fully connected 3 (1 nodes, sigmoid) → 0.99

```
model.add(tf.keras.layers.Dense(3, activation='relu'))
model.add(tf.keras.layers.Dense(2, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

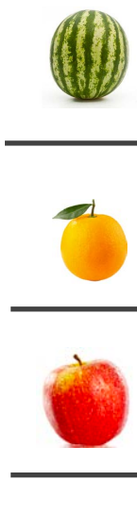
class 1 class 2 class 3

Three-class Classification



activation functions

Fully connected 1 (3 nodes, ReLU) → Fully connected 2 (2 nodes, ReLU) → Fully connected 3 (**3 nodes**, sigmoid)



0.99

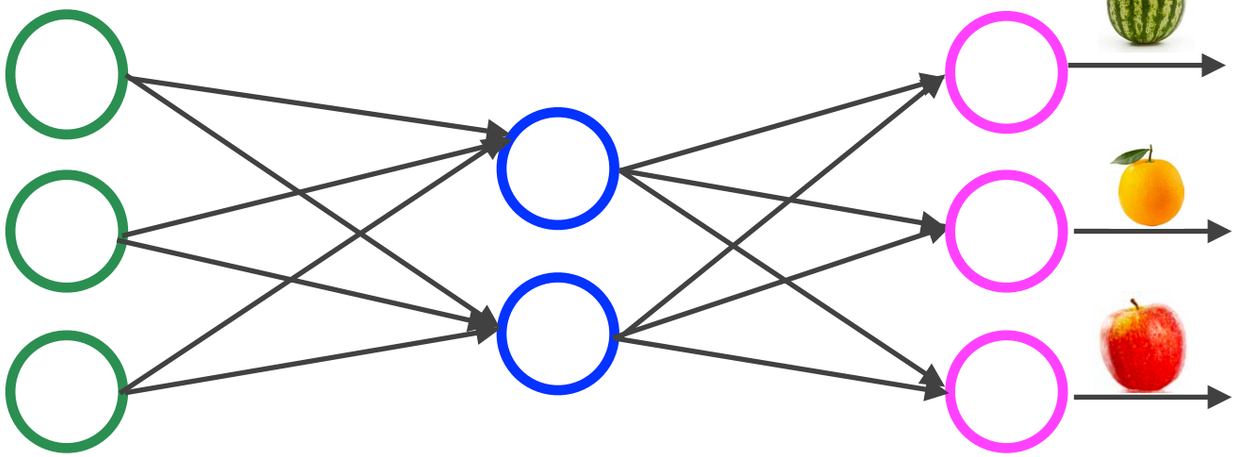
0.1

0.02

What if our model output something like 0.99 0.99 0.99 ? Not good

class 1 class 2 class 3

Three-class Classification

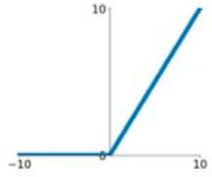


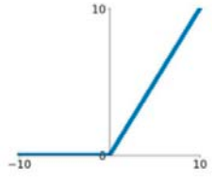
probability of being class 1

probability of being class 2

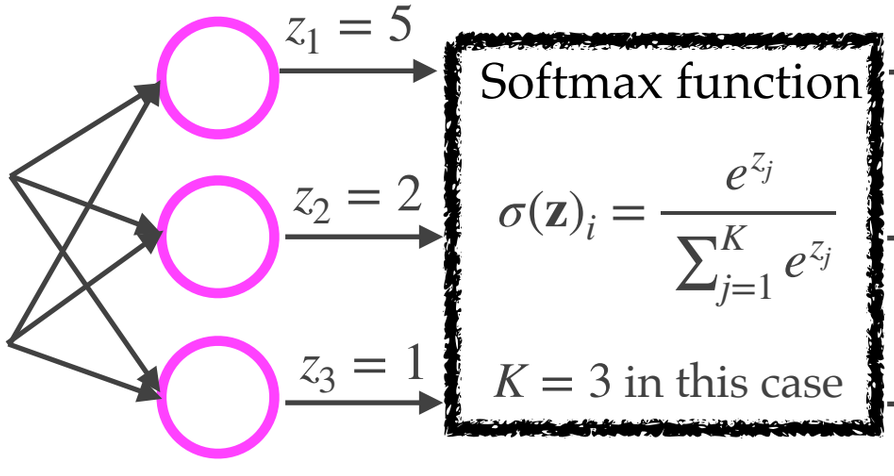
probability of being class 3

activation functions





Softmax function



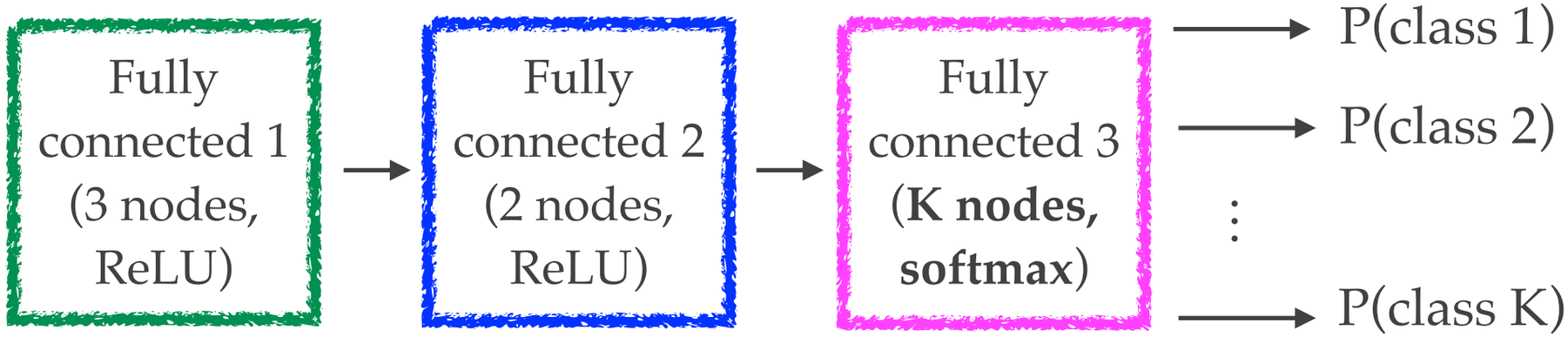
$$\frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}} = \frac{e^{5}}{e^{5}+e^{2}+e^{1}} = 0.936$$

$$\frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}} = \frac{e^{2}}{e^{5}+e^{2}+e^{1}} = 0.047$$

$$\frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}} = \frac{e^{1}}{e^{5}+e^{2}+e^{1}} = 0.017$$

$$0.936 + 0.047 + 0.017 = 1$$

Extend the model to K-class Classification




```
model.add(tf.keras.layers.Dense(3, activation='relu'))
model.add(tf.keras.layers.Dense(2, activation='relu'))
model.add(tf.keras.layers.Dense(K, activation='softmax'))
```

Fully connected layer
(Dense)

Convolutional layer

Conv1D, 2D, 3D, …
separable Conv

Optimizer

SGD
Adam
RMSprop

Pooling layer

max-pooling
average-pooling

Evaluation metric

accuracy
F1-score
AUC
confusion matrix

Activation function

sigmoid
softmax
ESP (swish)
ReLU

Loss function

categorical crossentropy
binary crossentropy
mean squared error
mean absolute error

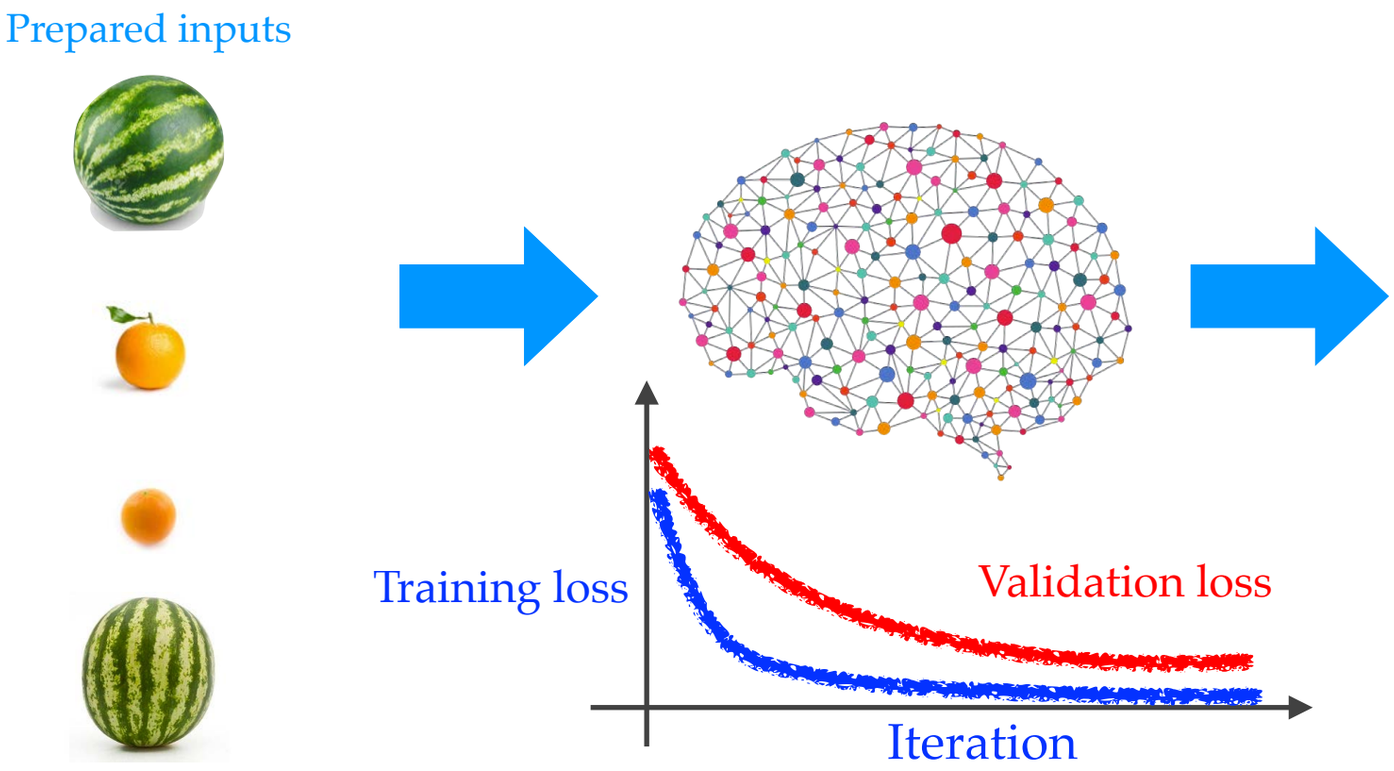
Regularization

Dropout
Data augmentation
l_1, l_2 regularizations

- Combine basic components to build a neural network
  - More components → "More" representative power

Image Classification Using Deep Learning

- Training phase: Optimize the weights of a deep neural network



- Test phase: Perform prediction using the trained neural network

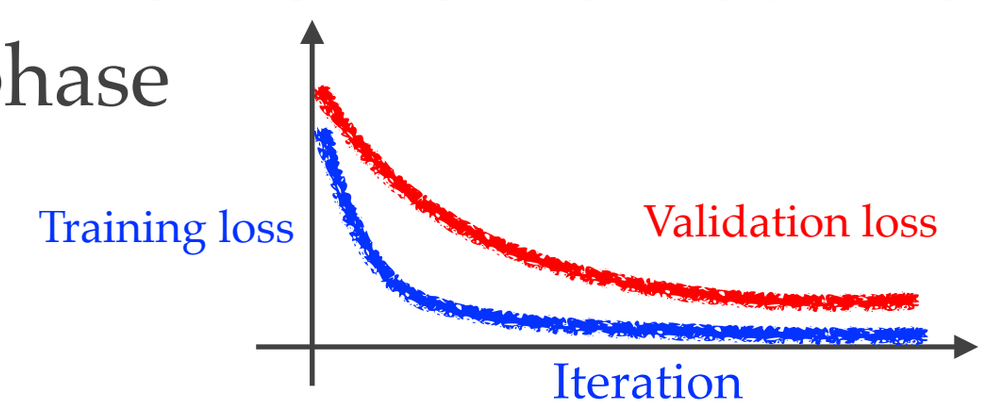
Loss function

(cost function and error function)

- A function L that maps values of variable(s) onto a real number

  For example, $L(y, \hat{y})$ gives us a number that tell us how "different" y and $\hat{y}$ are

- We have used it to monitor how our model performs during the training phase



$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

- Mean squared error (MSE) and mean absolute error (MAE) are two popular choices for regression

- Cross-entropy is the most common choice for a classification task

Cross-entropy

- Well suited to comparing probability distributions

3-class classification

deep learning layers → Fully connected (**3 nodes, softmax**)

	predicted prob. dist. $\hat{y}$	true prob. dist. y
	0.7	1
	0.1	0
	0.2	0

One-hot encoding

$$L(y,\hat{y}) = -\sum_{k=1}^{K} y_k log\hat{y}_k \quad \text{by definition } (K=3)$$

$$= -(1 \times log(0.7) + 0 \times log(0.1) + 0 \times log(0.2)) = -1 \times log(0.7) = 0.15$$

Assume that we have updated the weights of our model and got $\hat{y} = [0.9, 0.1, 0]$, then $L(y,\hat{y}) = -1 \times log(0.9) + 0 + 0 = 0.046$

Lower loss. $\hat{y}$ has become more similar to y than the previous case.
Our model performs better!

Optimizing Our Model

- To train our model f_W, we adjust its weights W in a way that decreases your loss function
- Particularly, we are minimizing our loss function with respect to W

prepared output/ labels/targets — prepared input

$$L(y, \hat{y}) = L(y, f_W(x))$$

predicted output — Weights

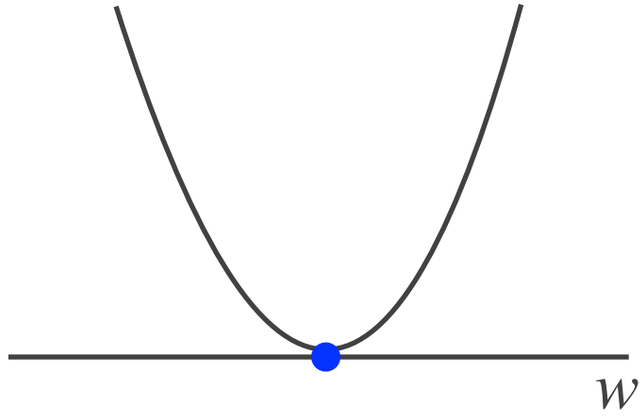
Calculus!

Compute the derivative of $L(y, f_W(x))$ with respect to W and set it to 0.

Example: Optimization

- Goal: Find w that minimizes the following loss function

$$L(w) = w^2$$



Method 1: Compute the derivative and set it to 0

Gradient $\dfrac{dL(w)}{dw} = \dfrac{dw^2}{dw} = 2w \quad = 0 \quad \Rightarrow \quad w = 0$

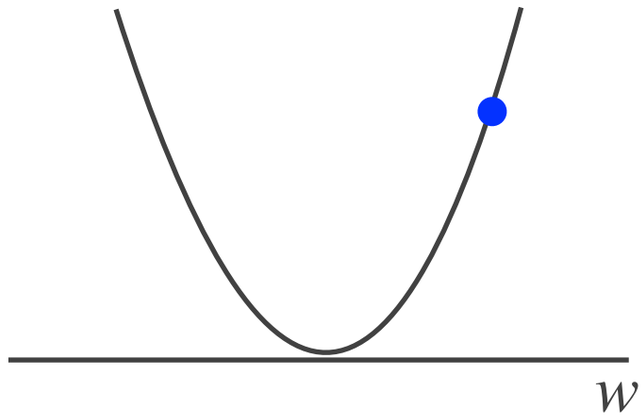
In our case, we have $L(y, f_W(x))$. Not simple to compute the gradient with respect to W.

Even when we have a way to compute the gradient and manage to set it to zero, we might not be able to get a closed-form solution for it.

Example: Optimization

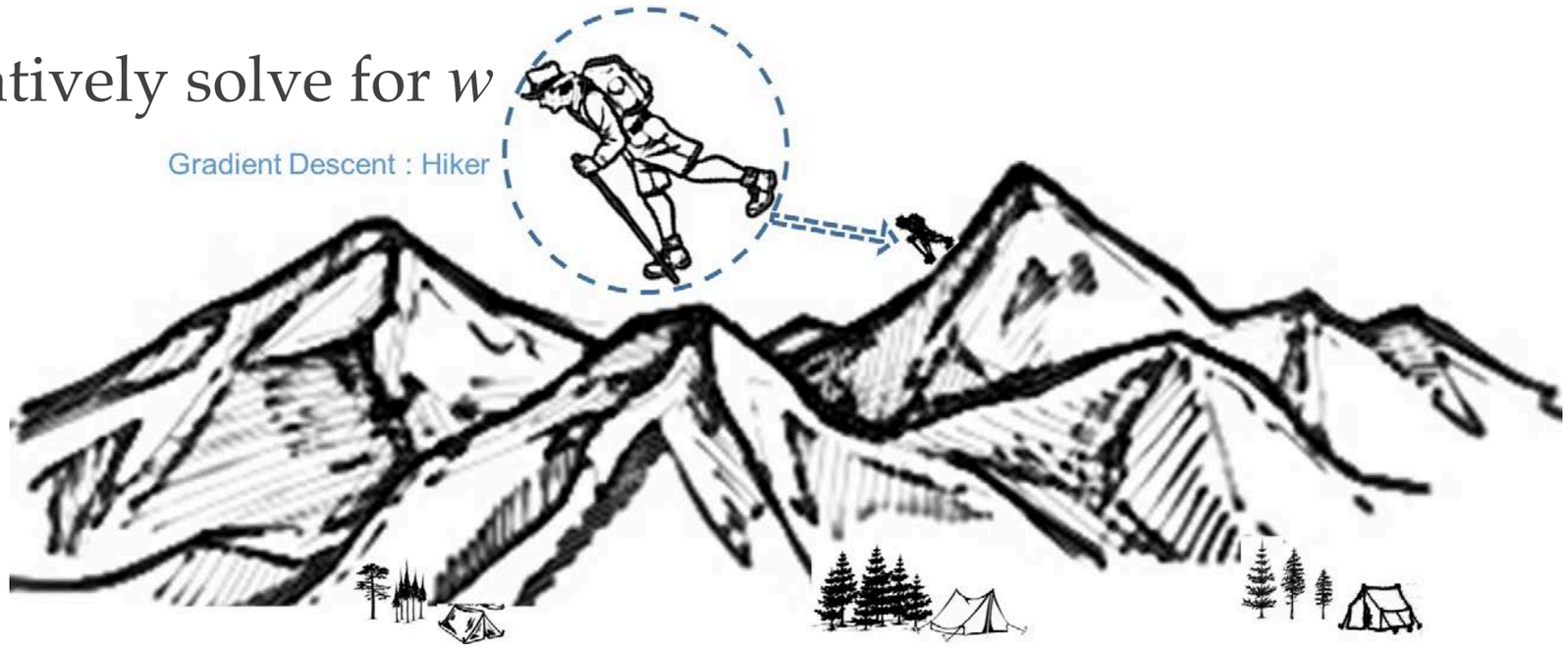
- Goal: Find w that minimizes the following loss function

$$L(w) = w^2$$



- Randomly start somewhere
- Gradually move to the minimum of the function

Method 2: Iteratively solve for w

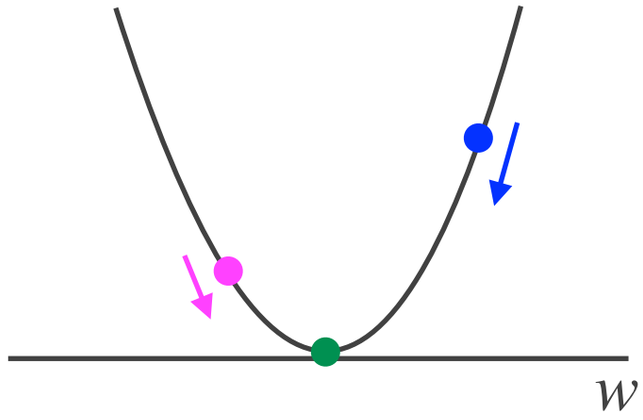


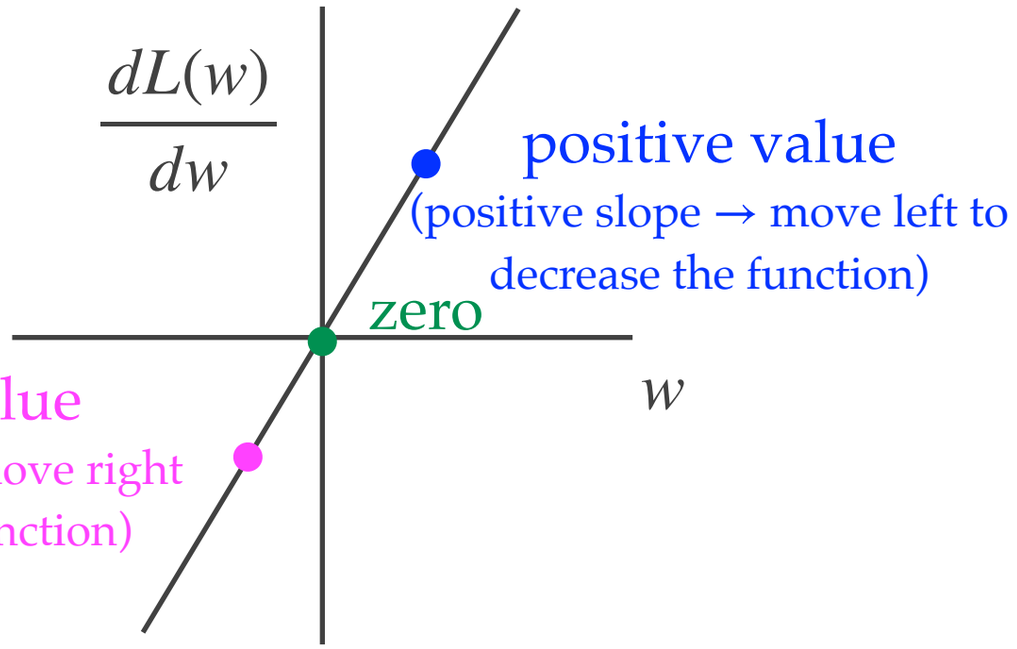
Gradient Descent for Logistic Regression Simplified – Step by Step Visual Guide

Example: Optimization

- Goal: Find w that minimizes the following loss function

$$L(w) = w^2$$





Method 2: Iteratively solve for w

1. Guess an initial solution w_0 and pick α

For $k = 0,1,2,...$

2. Compute $\frac{dL(w)}{dw}|_{w=w_{(k)}} = 2w|_{w=w_{(k)}} = 2w_{(k)}$

Iteration k

The negative of the gradient $-\frac{dL(w)}{dw}$ tells us the direction we should move to decrease the function

3. Compute a new solution $w_{(k+1)} := w_{(k)} - \alpha \frac{dL(w)}{dw}|_{w=w_{(k)}} = w_{(k)} - \alpha * 2w_{(k)}$

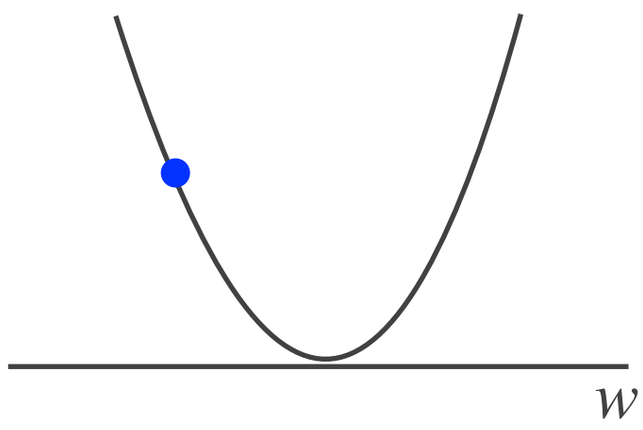
step size/learning rate α indicates how far we want to move

4. Repeat step 2 and 3 until converge

Example: Optimization

- Goal: Find w that minimizes the following loss function

$$L(w) = w^2$$



$$w_{(0)} = -2, \alpha = 0.5$$

$$w_{(1)} = w_{(0)} - \alpha * 2w_{(0)} = -2 - 0.5 * 2 * -2 = 0$$

$$w_{(2)} = w_{(1)} - \alpha * 2w_{(1)} = 0$$

$$w_{(3)} = w_{(2)} - \alpha * 2w_{(2)} = 0$$

Method 2: Iteratively solve for w

1. Guess an initial solution w_0 and pick α

For $k = 0,1,2,...$

2. Compute $\frac{dL(w)}{dw}|_{w=w_{(k)}} = 2w|_{w=w_{(k)}} = 2w_{(k)}$

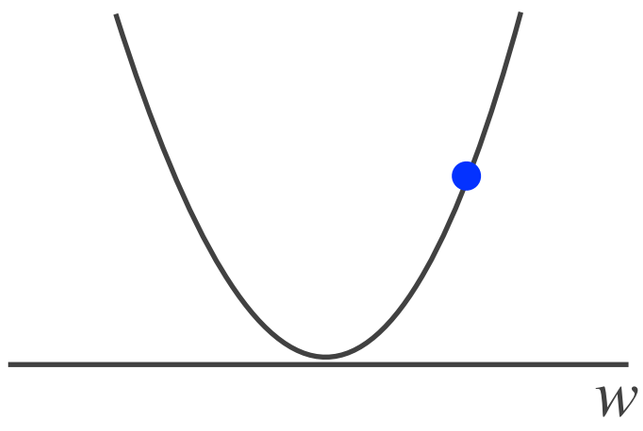
3. Compute a new solution $w_{(k+1)} := w_{(k)} - \alpha \frac{dL(w)}{dw}|_{w=w_{(k)}} = w_{(k)} - \alpha * 2w_{(k)}$

4. Repeat step 2 and 3 until converge

Example: Optimization

- Goal: Find w that minimizes the following loss function

$$L(w) = w^2$$



$w_{(0)} = 2, \alpha = 0.5$

$w_{(1)} = w_{(0)} - \alpha * 2w_{(0)} = 2 - 0.5 * 2 * 2 = 0$

$w_{(2)} = w_{(1)} - \alpha * 2w_{(1)} = 0$

$w_{(3)} = w_{(2)} - \alpha * 2w_{(2)} = 0$

Method 2: Iteratively solve for w

1. Guess an initial solution w_0 and pick α

For $k = 0,1,2,...$

2. Compute $\frac{dL(w)}{dw}|_{w=w_{(k)}} = 2w|_{w=w_{(k)}} = 2w_{(k)}$

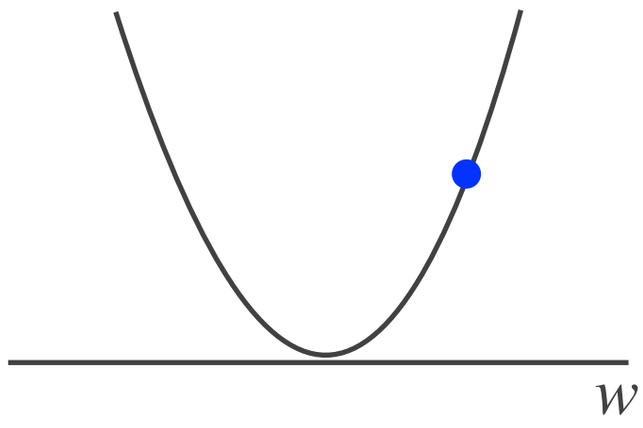
3. Compute a new solution $w_{(k+1)} := w_{(k)} - \alpha \frac{dL(w)}{dw}|_{w=w_{(k)}} = w_{(k)} - \alpha * 2w_{(k)}$

4. Repeat step 2 and 3 until converge

Example: Optimization

- Goal: Find w that minimizes the following loss function

$$L(w) = w^2$$



$$w_{(0)} = 2, \alpha = 0.25$$

$$w_{(1)} = w_{(0)} - \alpha * 2w_{(0)} = 2 - 0.25 * 2 * 2 = 1$$

$$w_{(2)} = w_{(1)} - \alpha * 2w_{(1)} = 1 - 0.25 * 2 * 1 = 0.5$$

$$w_{(3)} = w_{(2)} - \alpha * 2w_{(2)} = 0.5 - 0.25 * 2 * 0.5 = 0.25$$

$$w_{(4)} = w_{(3)} - \alpha * 2w_{(3)} = 0.125$$

$$w_{(5)} = w_{(4)} - \alpha * 2w_{(4)} = 0.0625$$

Method 2: Iteratively solve for w

1. Guess an initial solution w_0 and pick α

For $k = 0,1,2,...$

2. Compute $\frac{dL(w)}{dw}|_{w=w_{(k)}} = 2w|_{w=w_{(k)}} = 2w_{(k)}$

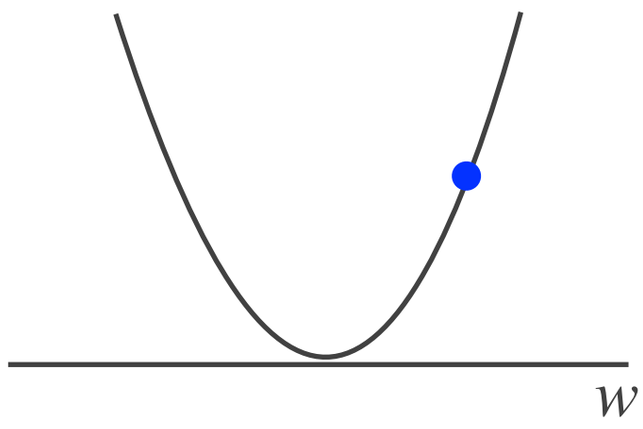
3. Compute a new solution $w_{(k+1)} := w_{(k)} - \alpha \frac{dL(w)}{dw}|_{w=w_{(k)}} = w_{(k)} - \alpha * 2w_{(k)}$

4. Repeat step 2 and 3 until converge

Example: Optimization

- Goal: Find w that minimizes the following loss function

$$L(w) = w^2$$



$$w_{(0)} = 2, \alpha = 1$$

$$w_{(1)} = w_{(0)} - \alpha * 2w_{(0)} = 2 - 1 * 2 * 2 = -2$$

$$w_{(2)} = w_{(1)} - \alpha * 2w_{(1)} = -2 - 1 * 2 * -2 = 2$$

$$w_{(3)} = w_{(2)} - \alpha * 2w_{(2)} = -2$$

$$w_{(4)} = w_{(3)} - \alpha * 2w_{(3)} = 2$$

$$w_{(5)} = w_{(4)} - \alpha * 2w_{(4)} = -2$$

Method 2: Iteratively solve for w

1. Guess an initial solution w_0 and pick α

For $k = 0,1,2,...$

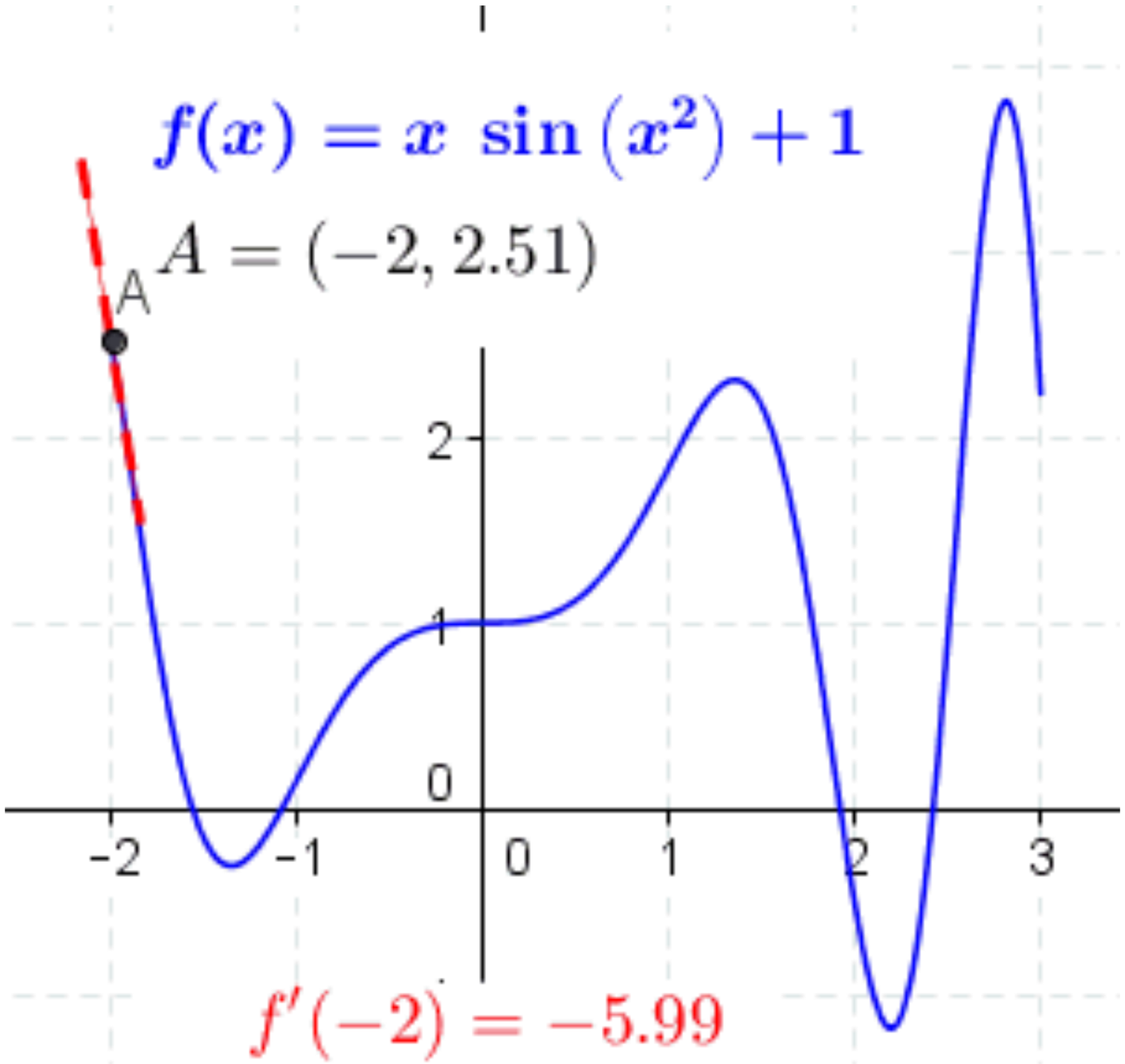
2. Compute $\frac{dL(w)}{dw}|_{w=w_{(k)}} = 2w|_{w=w_{(k)}} = 2w_{(k)}$

3. Compute a new solution $w_{(k+1)} := w_{(k)} - \alpha \frac{dL(w)}{dw}|_{w=w_{(k)}} = w_{(k)} - \alpha * 2w_{(k)}$

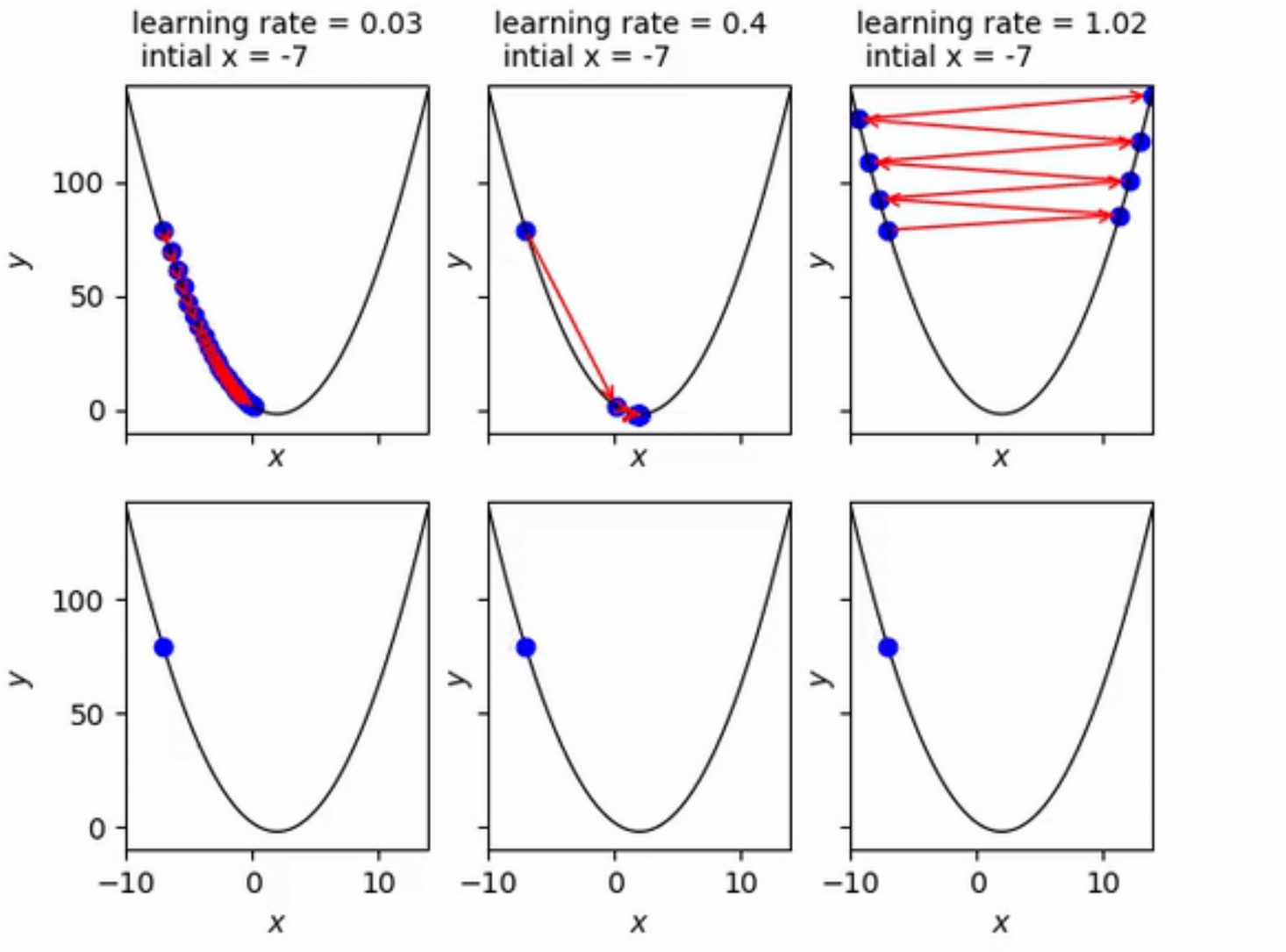
4. Repeat step 2 and 3 until converge

α: learning rate

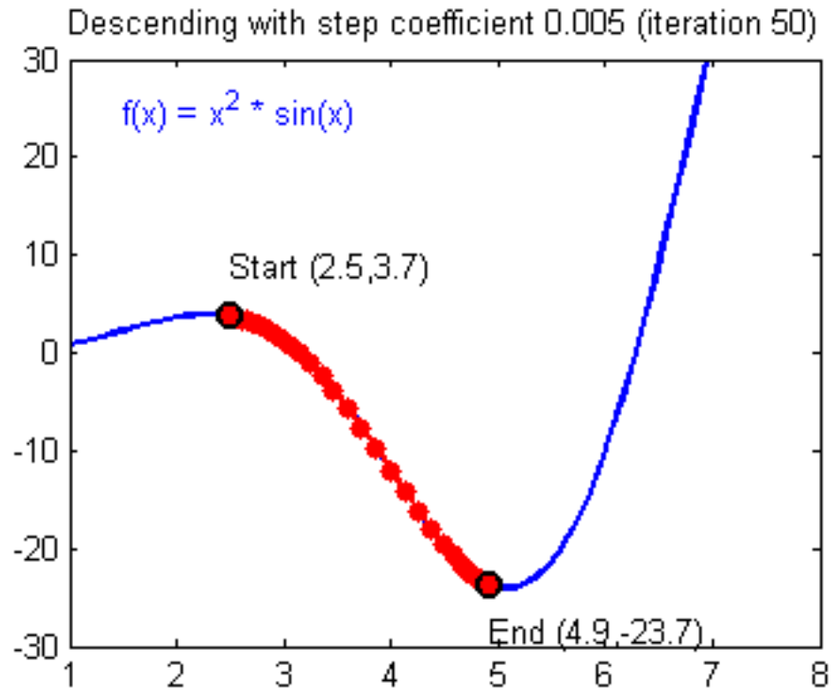
Example: Gradient



Optimization: Learning Rate



Fixed learning rate

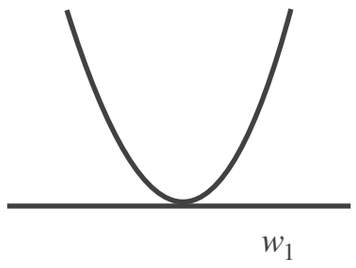


Adaptive learning rate

Gradient

1-dimensional case

$$L(w_1) = w_1^2$$



Derivative of $L(w_1)$

$$\frac{dL(w_1)}{dw_1} = \frac{dw_1^2}{dw_1} = 2w_1$$

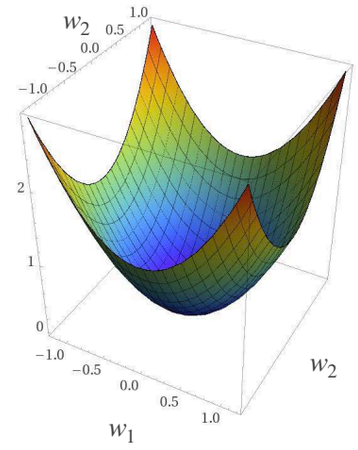
Weight update

$$w_{1,(k+1)} := w_{1,(k)} - \alpha \frac{dL(w_1)}{dw_1}\Big|_{w_1=w_{1,(k)}}$$

$$= w_{1,(k)} - 2w_{1,(k)}$$

2-dimensional case

$$L(w_1, w_2) = w_1^2 + w_2^2$$



Partial derivatives of $L(w_1, w_2)$

$$\frac{\partial L(w_1, w_2)}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1^2 + w_2^2) = 2w_1$$

$$\frac{\partial L(w_1, w_2)}{\partial w_2} = \frac{\partial}{\partial w_2}(w_1^2 + w_2^2) = 2w_2$$

Gradient $\nabla_w L(w_1, w_2)$

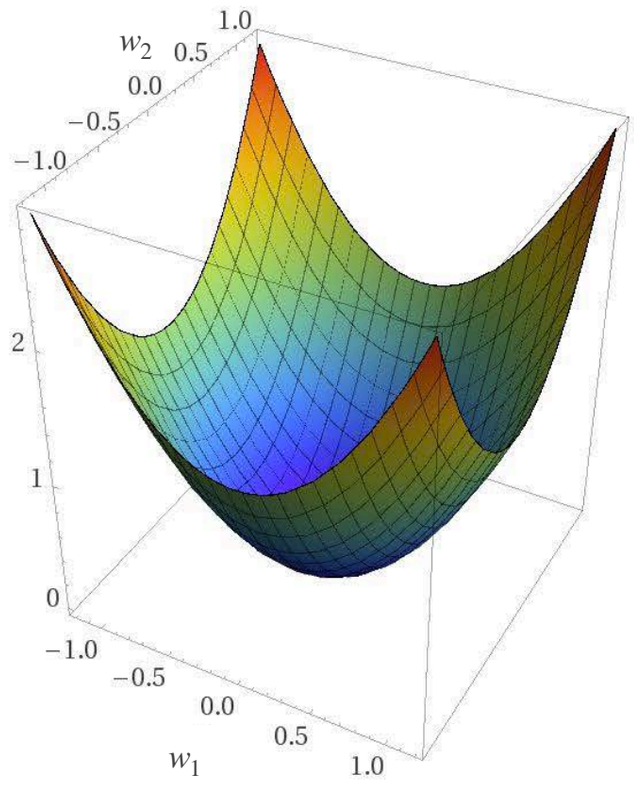
Weight update

$$\begin{bmatrix} w_{1,(k+1)} \\ w_{2,(k+1)} \end{bmatrix} = \begin{bmatrix} w_{1,(k)} \\ w_{2,(k)} \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L(w_1, w_2)}{\partial w_1}\Big|_{w_1=w_{1,(k)}} \\ \frac{\partial L(w_1, w_2)}{\partial w_2}\Big|_{w_2=w_{2,(k)}} \end{bmatrix}$$

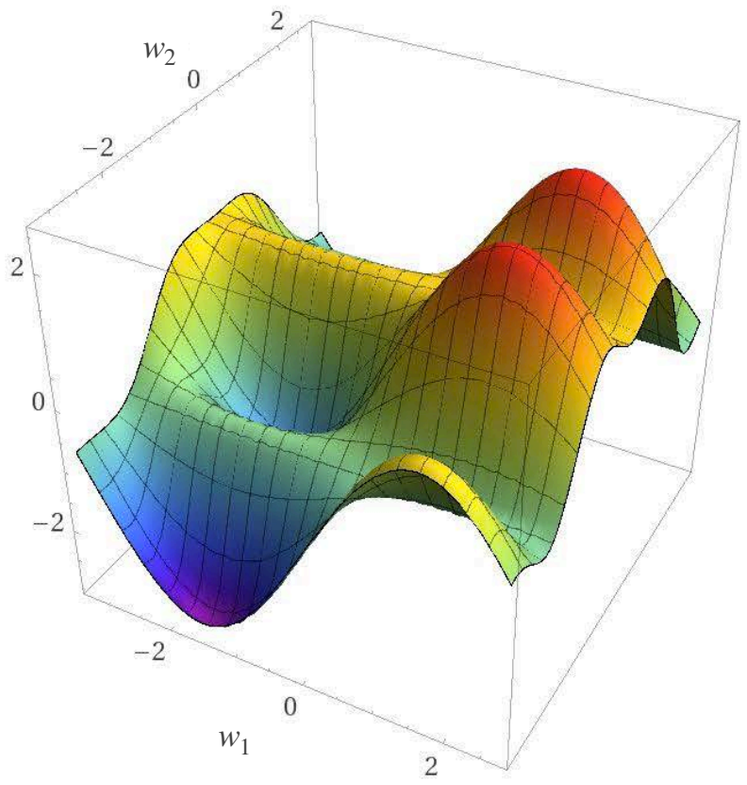
$$= \begin{bmatrix} w_{1,(k)} \\ w_{2,(k)} \end{bmatrix} - \alpha \begin{bmatrix} 2w_{1,(k)} \\ 2w_{2,(k)} \end{bmatrix}$$

Optimization

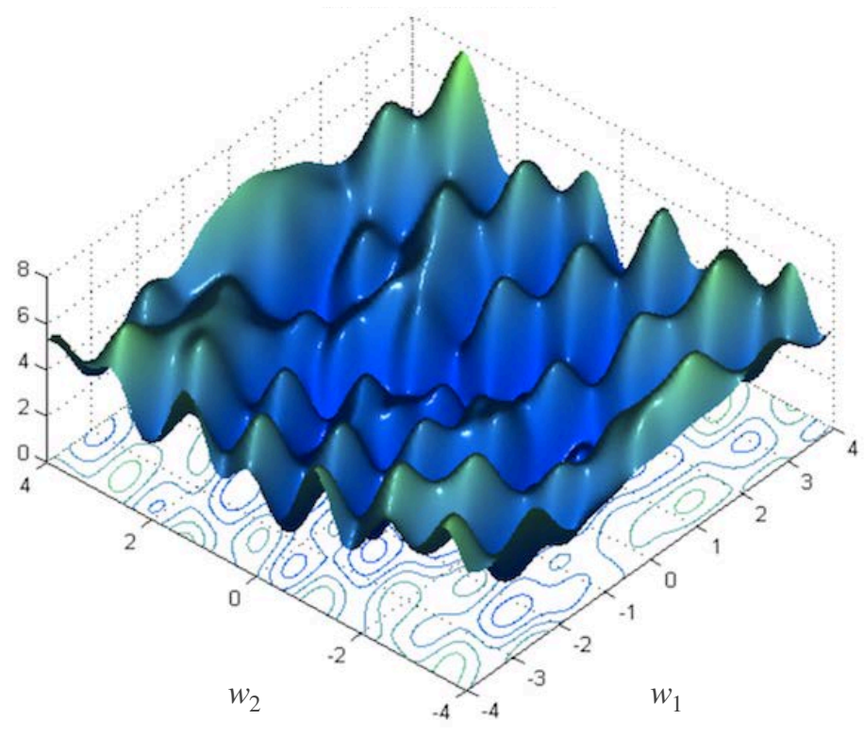
Real Life



No local minima problem (global minimum = local minimum)



Could get stuck in local minima



@#Q#@%!!!!!!

Furthermore, how do we compute a complicated loss function such as the one in our example $L(y, f_W(x))$?

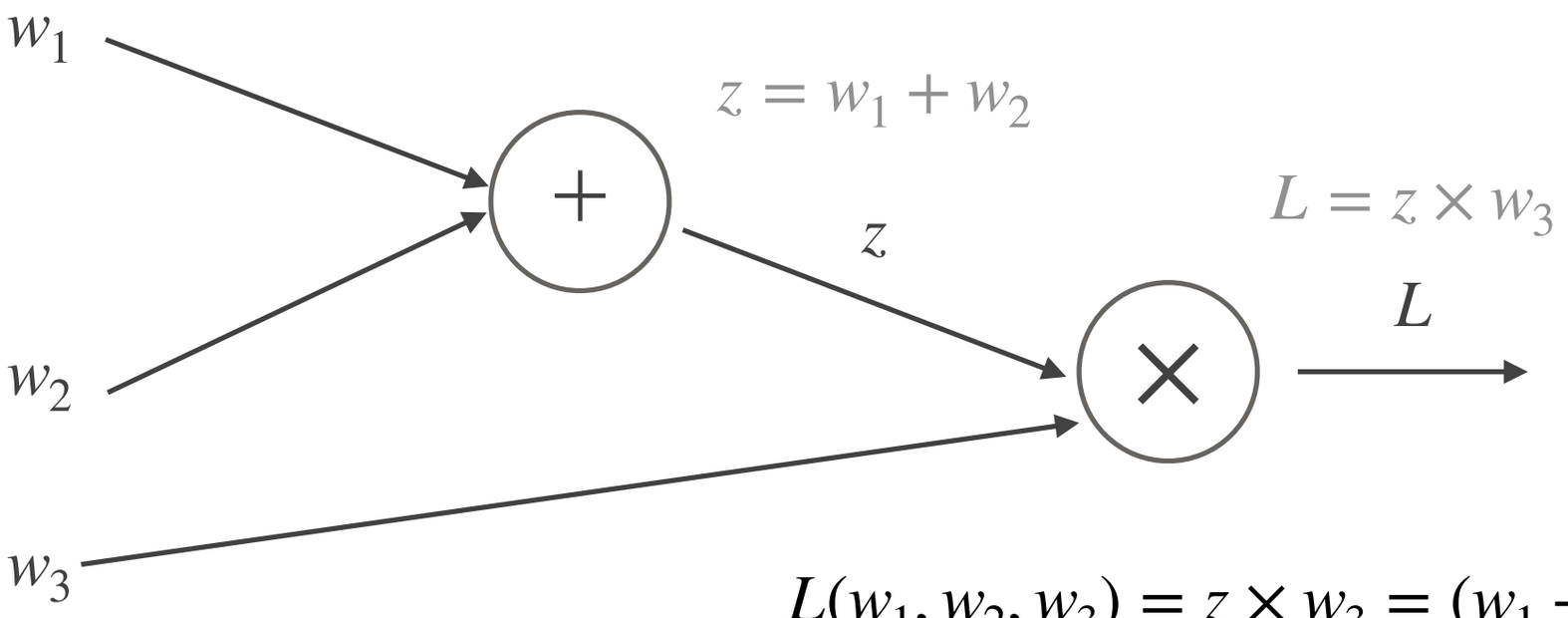
Backpropagation!

O'Reilly Media

Yoshua Bengio

Backpropagation

Computation graph



$$L(w_1, w_2, w_3) = z \times w_3 = (w_1 + w_2) \times w_3$$

Exercise: Compute the gradient vector

Use chain rule

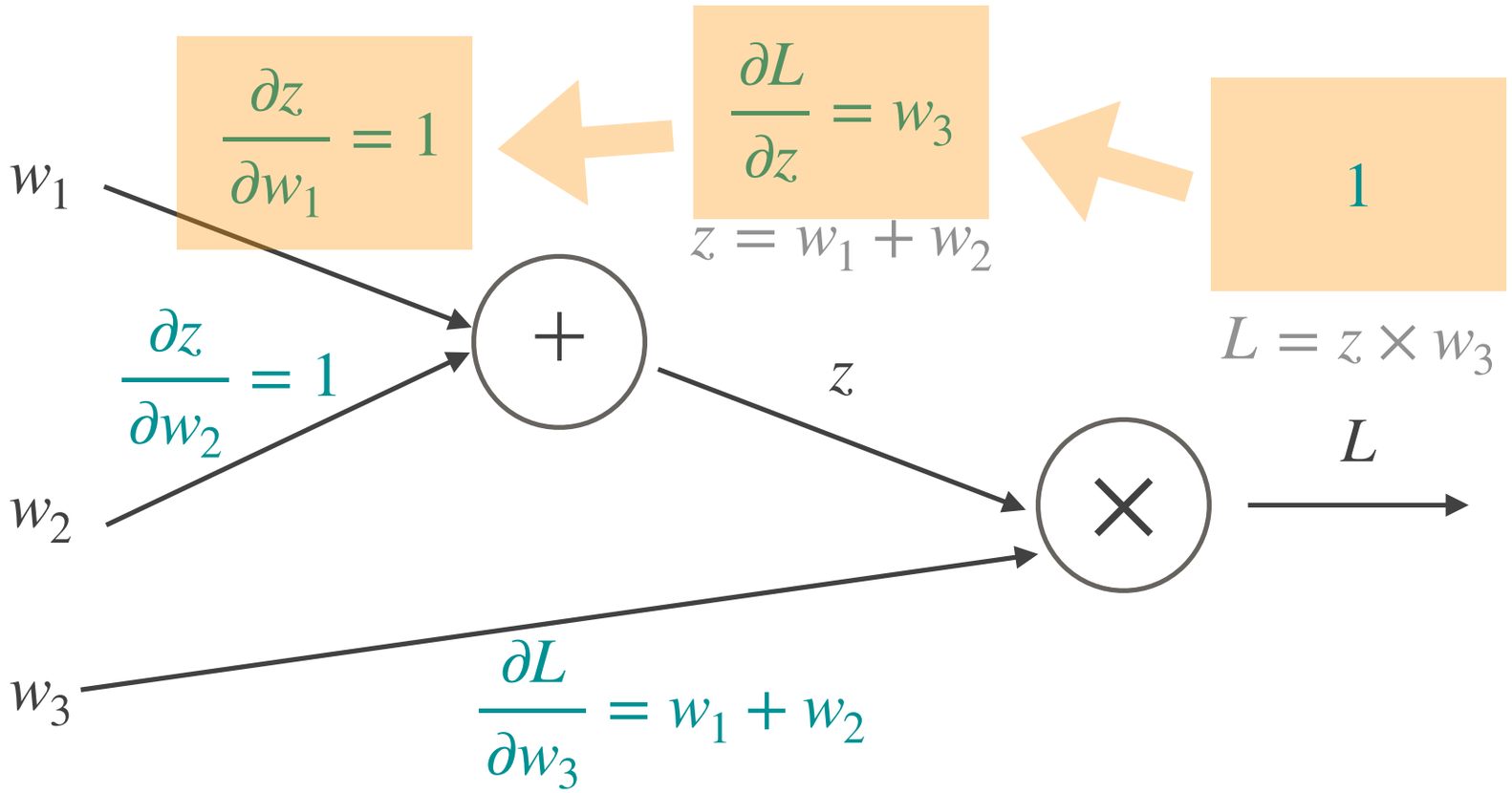
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_1} = w_3(1 + 0) = w_3$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_2} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_2} = w_3(0 + 1) = w_3$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial z w_3}{\partial w_3} = z = w_1 + w_2$$

$$\nabla_w L(w_1, w_2, w_3) = \begin{bmatrix} w_3 \\ w_3 \\ w_1 + w_2 \end{bmatrix}$$

Backpropagate



Exercise: Compute the gradient vector

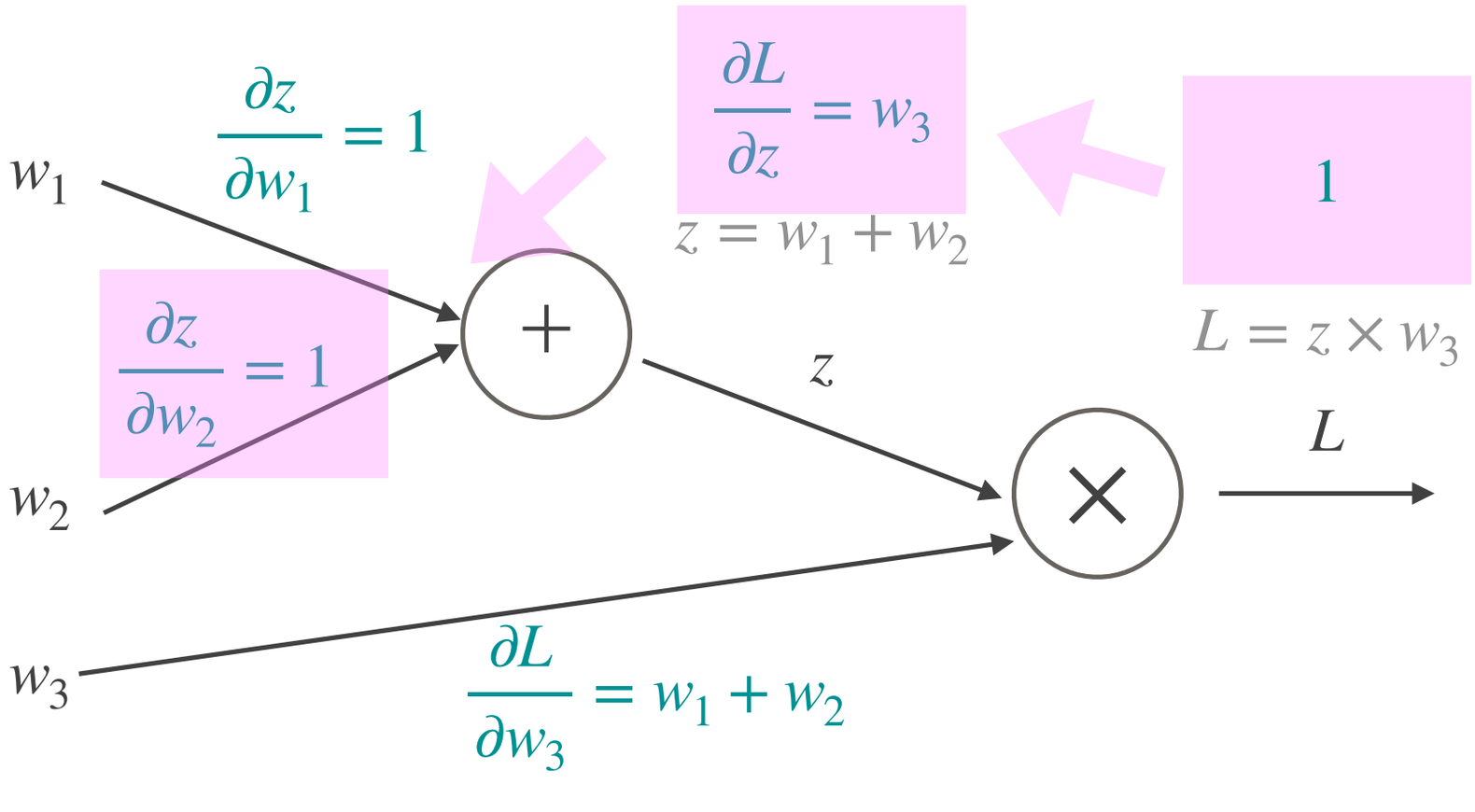
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_1} = w_3(1+0) = w_3$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_2} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_2} = w_3(0+1) = w_3$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial z w_3}{\partial w_3} = z = w_1 + w_2$$

$$\nabla_w L(w_1, w_2, w_3) = \begin{bmatrix} w_3 \\ w_3 \\ w_1 + w_2 \end{bmatrix}$$

Backpropagate



Exercise: Compute the gradient vector

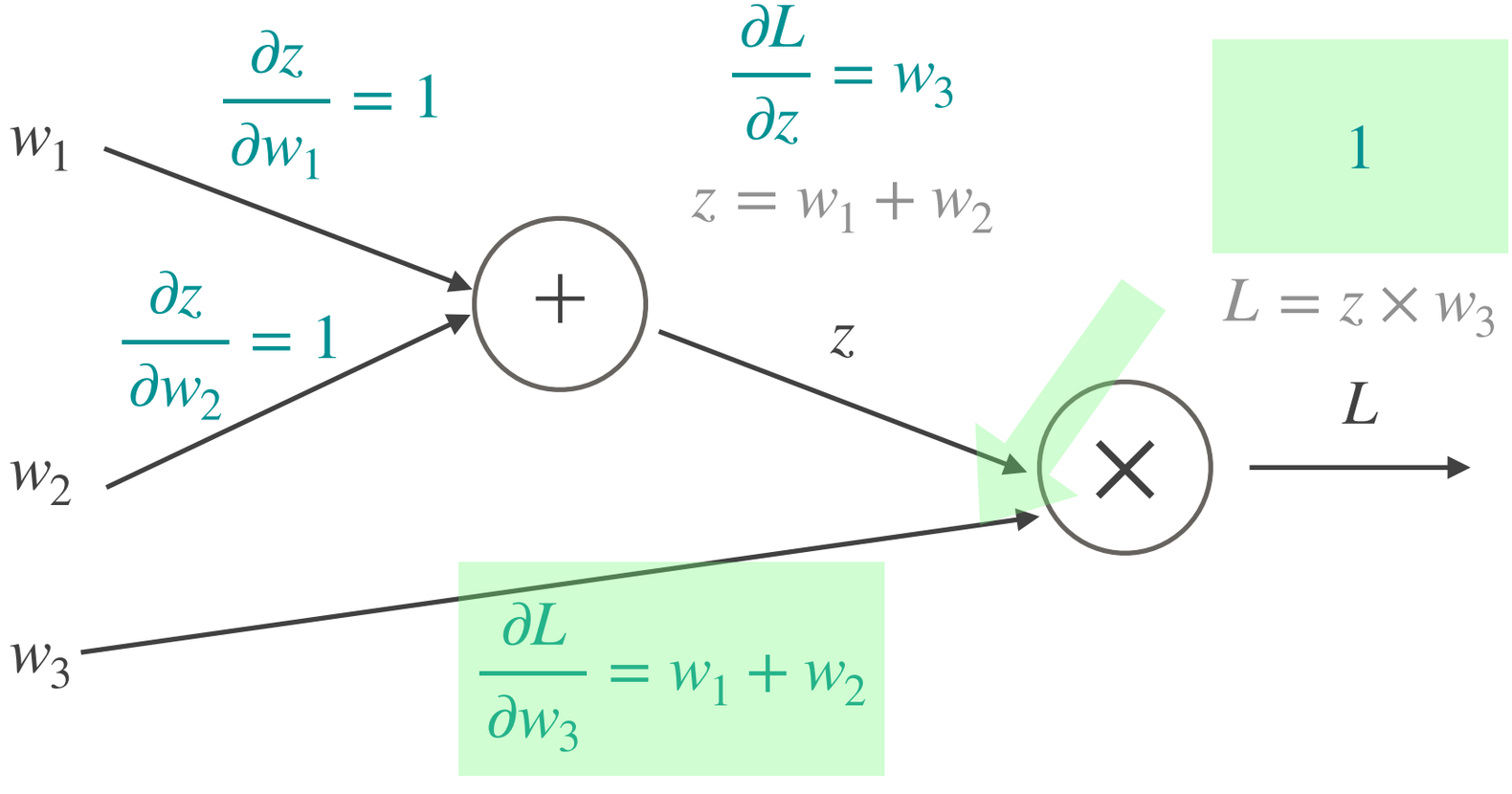
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_1} = w_3(1+0) = w_3$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_2} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_2} = w_3(0+1) = w_3$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial z w_3}{\partial w_3} = z = w_1 + w_2$$

$$\nabla_w L(w_1, w_2, w_3) = \begin{bmatrix} w_3 \\ w_3 \\ w_1 + w_2 \end{bmatrix}$$

Backpropagate



Exercise: Compute the gradient vector

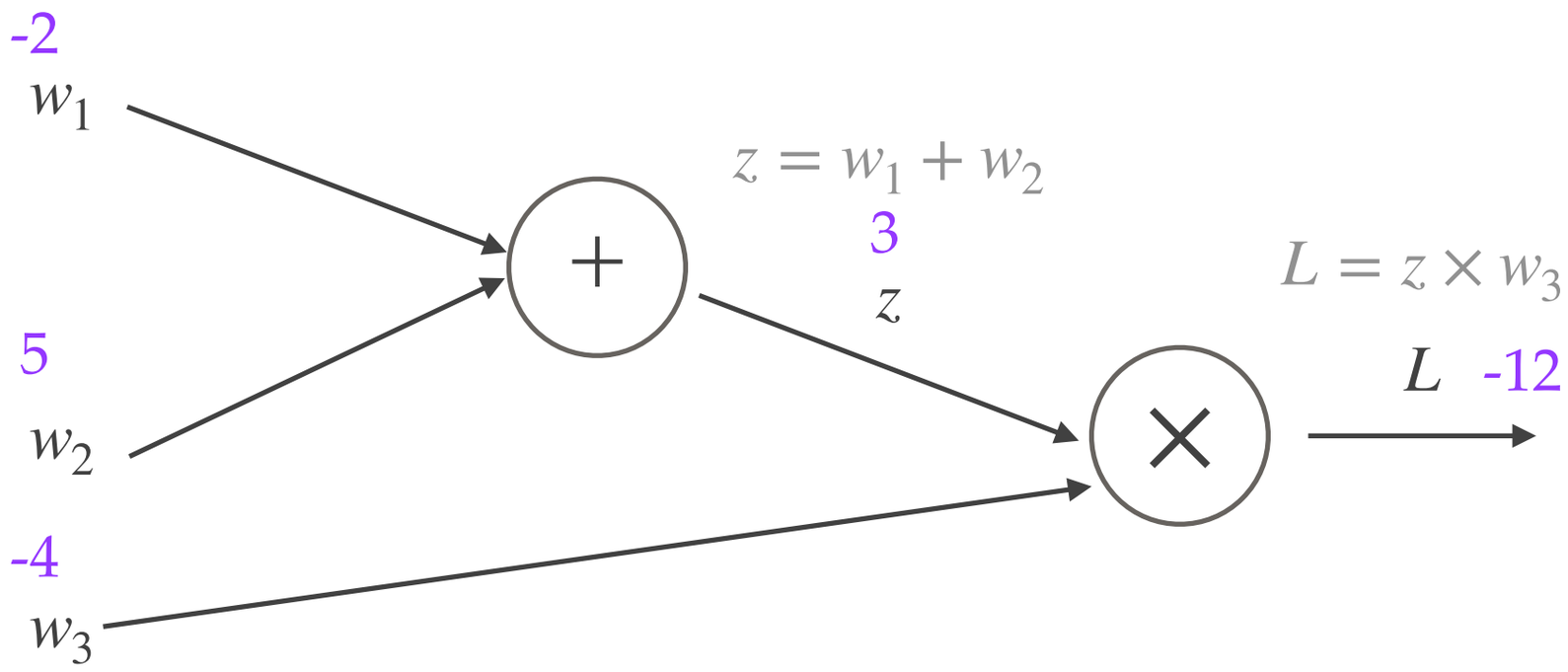
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_1} = w_3(1+0) = w_3$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_2} = \frac{\partial w_3 z}{\partial z}\frac{\partial (w_1 + w_2)}{\partial w_2} = w_3(0+1) = w_3$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial z w_3}{\partial w_3} = z = w_1 + w_2$$

$$\nabla_w L(w_1, w_2, w_3) = \begin{bmatrix} w_3 \\ w_3 \\ w_1 + w_2 \end{bmatrix}$$

With actual numbers

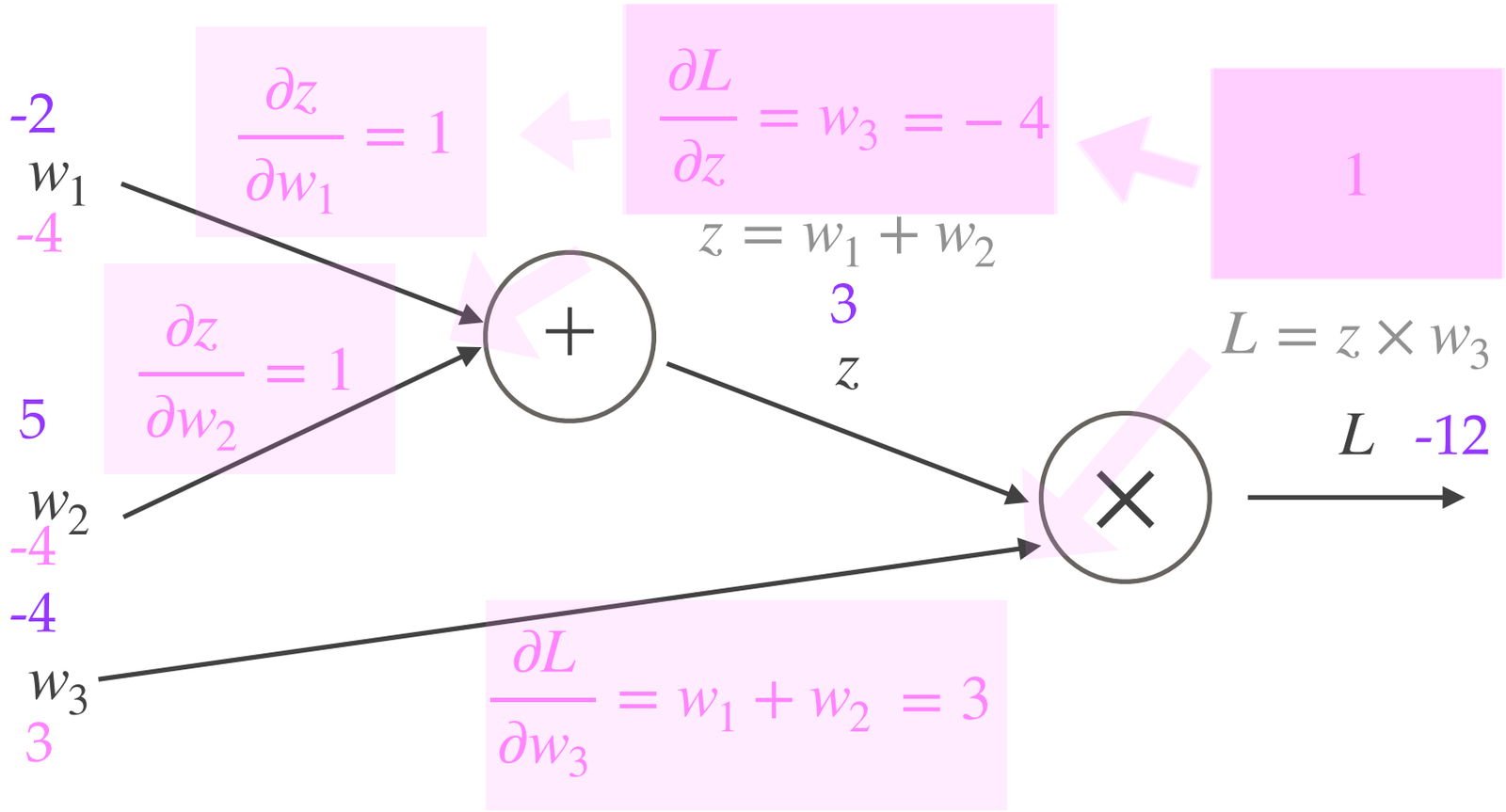


Forward pass

$z = -2 + 5 = 3$ $L = -4 \times 3 = -12$

We calculate the values of the variables in the computation graph

With actual numbers



Forward pass

$z = -2 + 5 = 3$ $L = -4 + 3 = -12$

We calculate the values of the variables in the computation graph

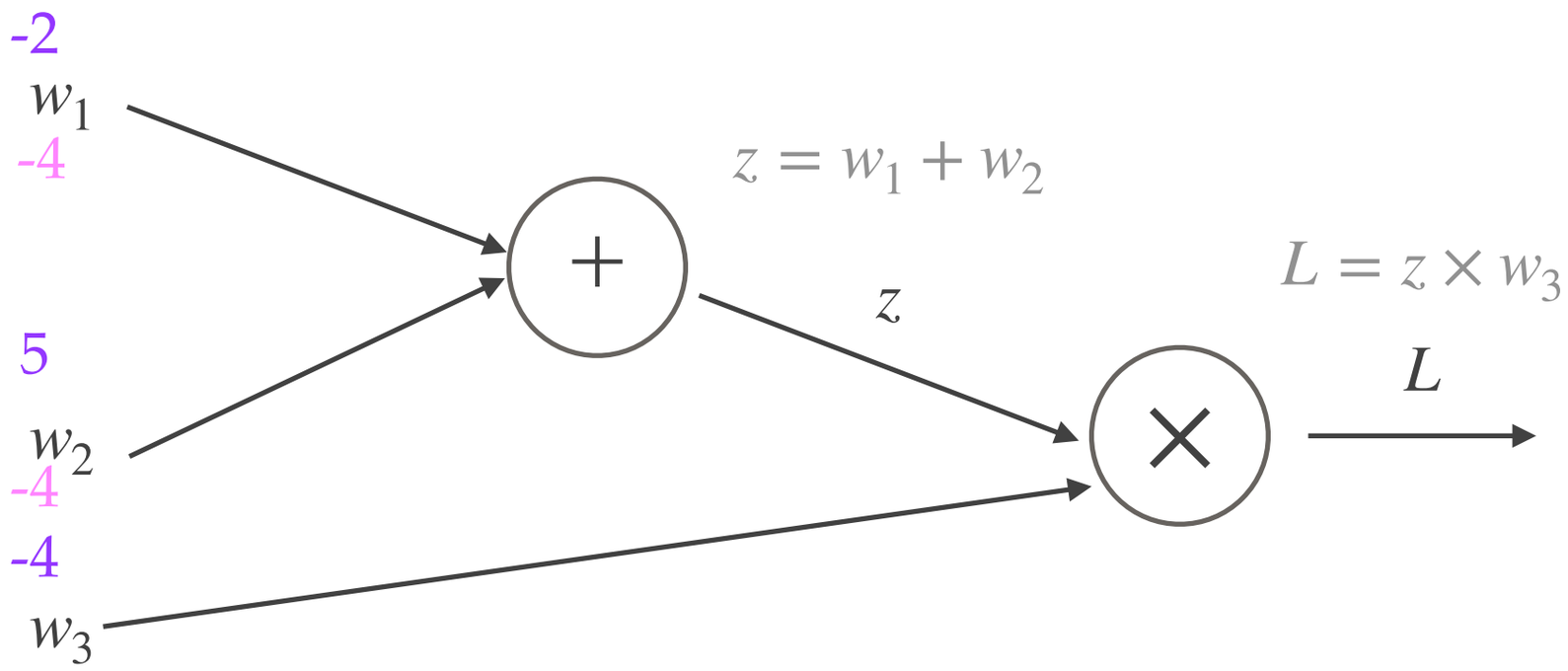
Backward pass

We calculate the gradients and evaluate them using the values from the forward pass

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = -4 \times 1 = -4 \qquad \frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_2} = -4 \times 1 = -4 \qquad \frac{\partial L}{\partial w_3} = z = 3$$

$$\nabla_w L = \begin{bmatrix} -4 \\ -4 \\ 3 \end{bmatrix}$$

Update the weights



Given

$$w_{(0)} = \begin{bmatrix} -2 \\ 5 \\ -4 \end{bmatrix}, \quad \nabla_w L = \begin{bmatrix} -4 \\ -4 \\ 3 \end{bmatrix} \quad \text{at } w_{(0)} \text{ and } \alpha = 1 \quad \Rightarrow \quad L(w_{(0)}) = -12$$

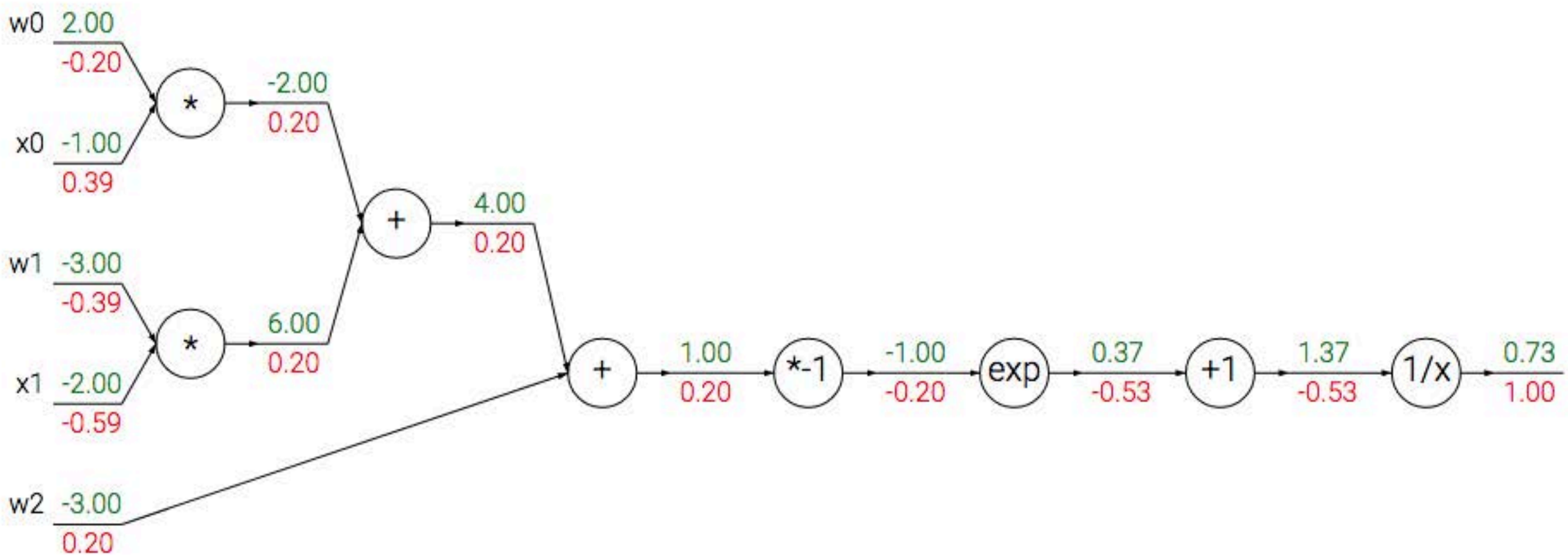
Weight update

$$w_{(1)} = w_{(0)} - \alpha \nabla_w L = \begin{bmatrix} -2 \\ 5 \\ -4 \end{bmatrix} - 1 \begin{bmatrix} -4 \\ -4 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 9 \\ -7 \end{bmatrix} \quad \Rightarrow \quad L(w_{(1)}) = -77$$

$$w_{(2)} = w_{(1)} - \alpha \nabla_w L = ? \quad \Rightarrow \quad L(w_{(2)}) = ?$$

Backpropagation

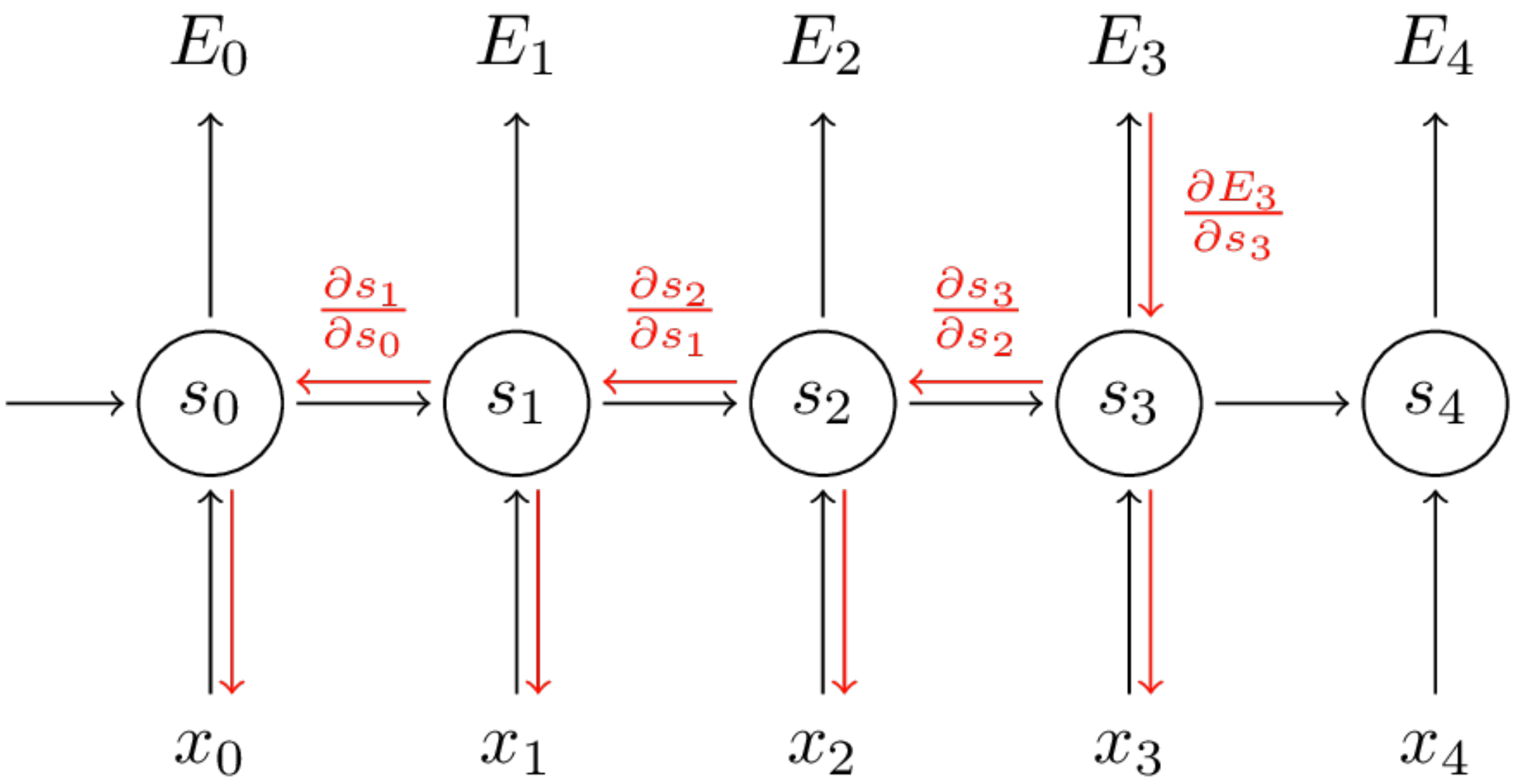
$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Example circuit for a 2D neuron with a sigmoid activation function. The inputs are [x0,x1] and the (learnable) weights of the neuron are [w0,w1,w2]. As we will see later, the neuron computes a dot product with the input and then its activation is softly squashed by the sigmoid function to be in range from 0 to 1.

Taken from CS231n Convolutional Neural Networks for Visual Recognition

Exploding/Vanishing Gradients



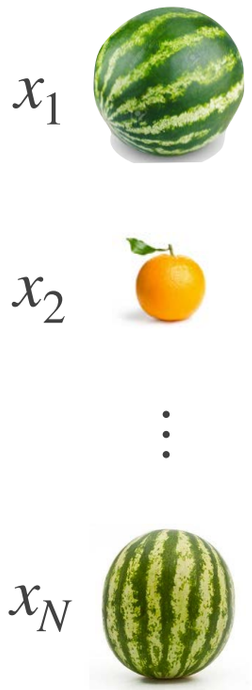
$0.05^5 = 3.125 \times 10^{-7}$ Very small

$15^5 = 759375$ Very large

Taken from Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients

Training Our Model

prepared inputs



f_W

deep learning model

predicted outputs	prepared outputs	loss values
$\hat{y}_1$	y_1	$L(y_1, \hat{y}_1)$
$\hat{y}_2$	y_2	$L(y_2, \hat{y}_2)$
$\vdots$	$\vdots$	$\vdots$
$\hat{y}_N$	y_N	$L(y_N, \hat{y}_N)$

Total loss $L(y, \hat{y}) = \sum_{i=1}^{N} L(y_i, \hat{y}_i) = \sum_{i=1}^{N} L(y_i, f_W(x_i))$

Update the weights

Compute the gradient of the total loss w.r.t. to W using backpropagation

$$\nabla_W L = \sum_{i=1}^{N} \nabla_W L(y_i, f_W(x_i))$$

Assign the new weights $W \leftarrow W - \alpha \nabla_W L$

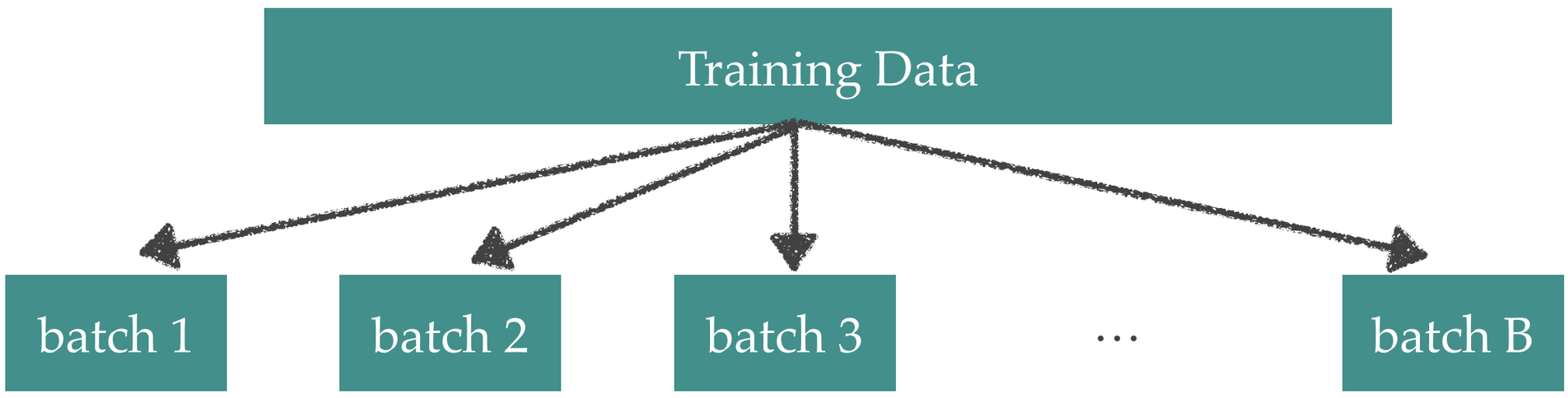
Stochastic Gradient Descent

- Every time we want to update W, we need to compute the loss functions and their gradients of all samples

$$\nabla_W L = \sum_{i=1}^{N} \nabla_W L(y_i, f_W(x_i))$$

  - For very large N, we might not be able to do it due to both the time and memory constraints 14,000,000 images?

  - For stochastic gradient descent (SGD), we update W based on subsets of samples called a mini-batch



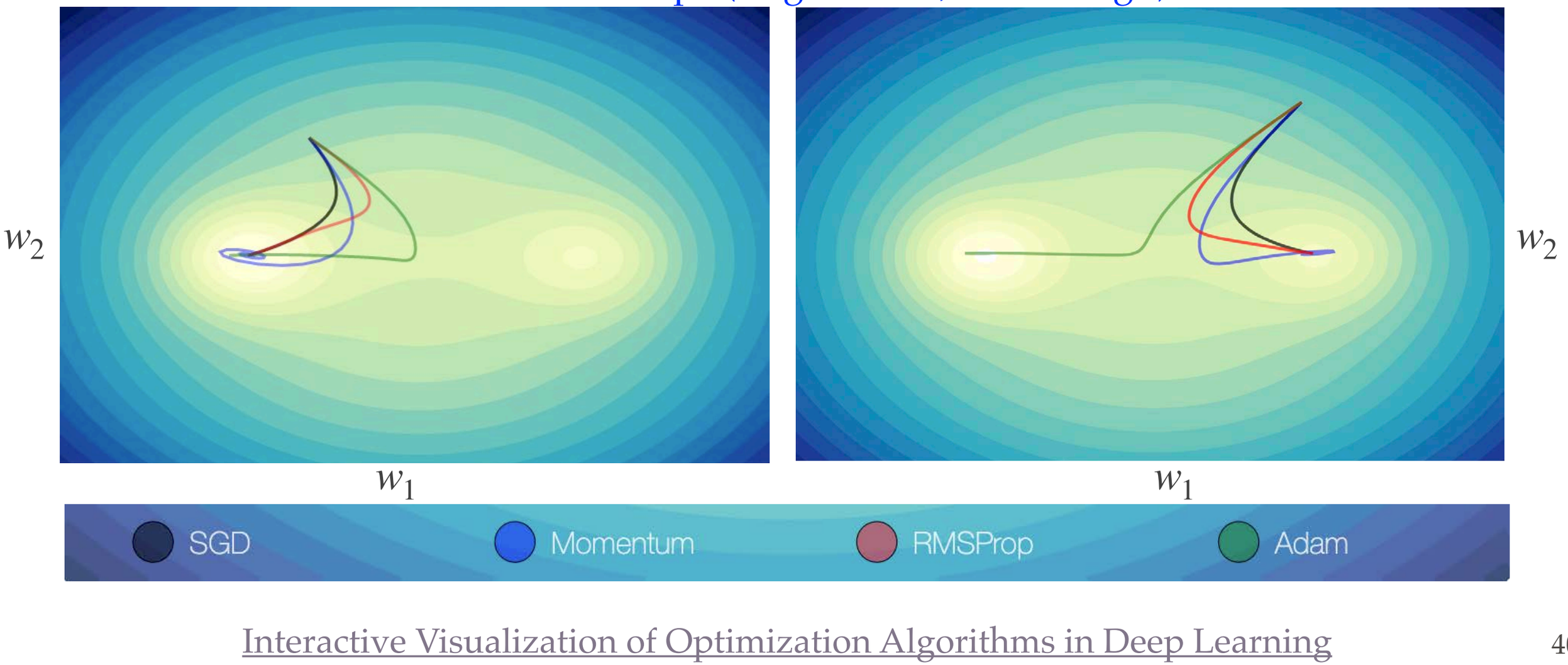
1 epoch = the model has seen the whole training data once (i.e., B batches in this case)

In this example, the weight matrix W is updated B times per epoch.

Optimizers

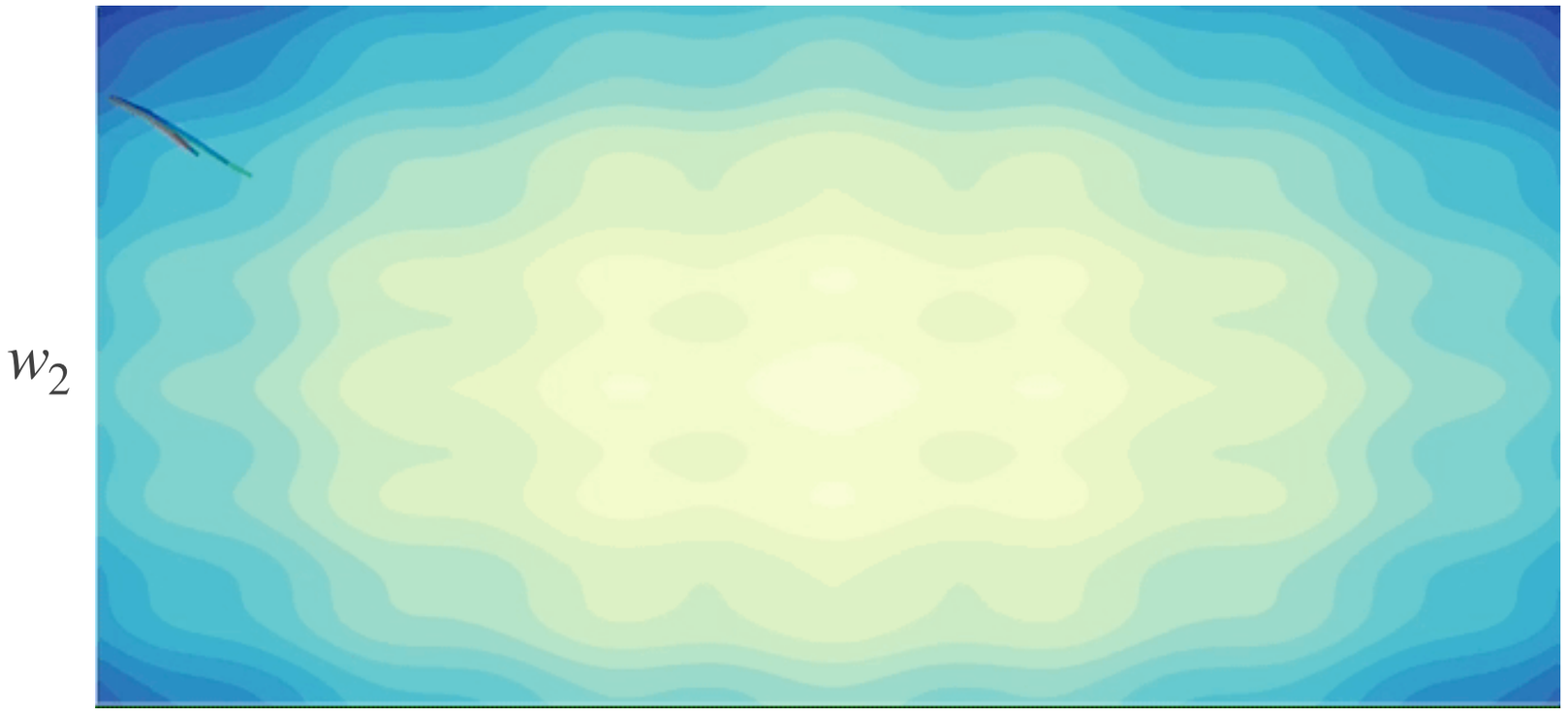
- Popular choices are
  - Stochastic gradient descent (SGD)
  - SGD with momentum
  - RMSprop
  - Adam
  - ADADELTA
  - AdaGrad

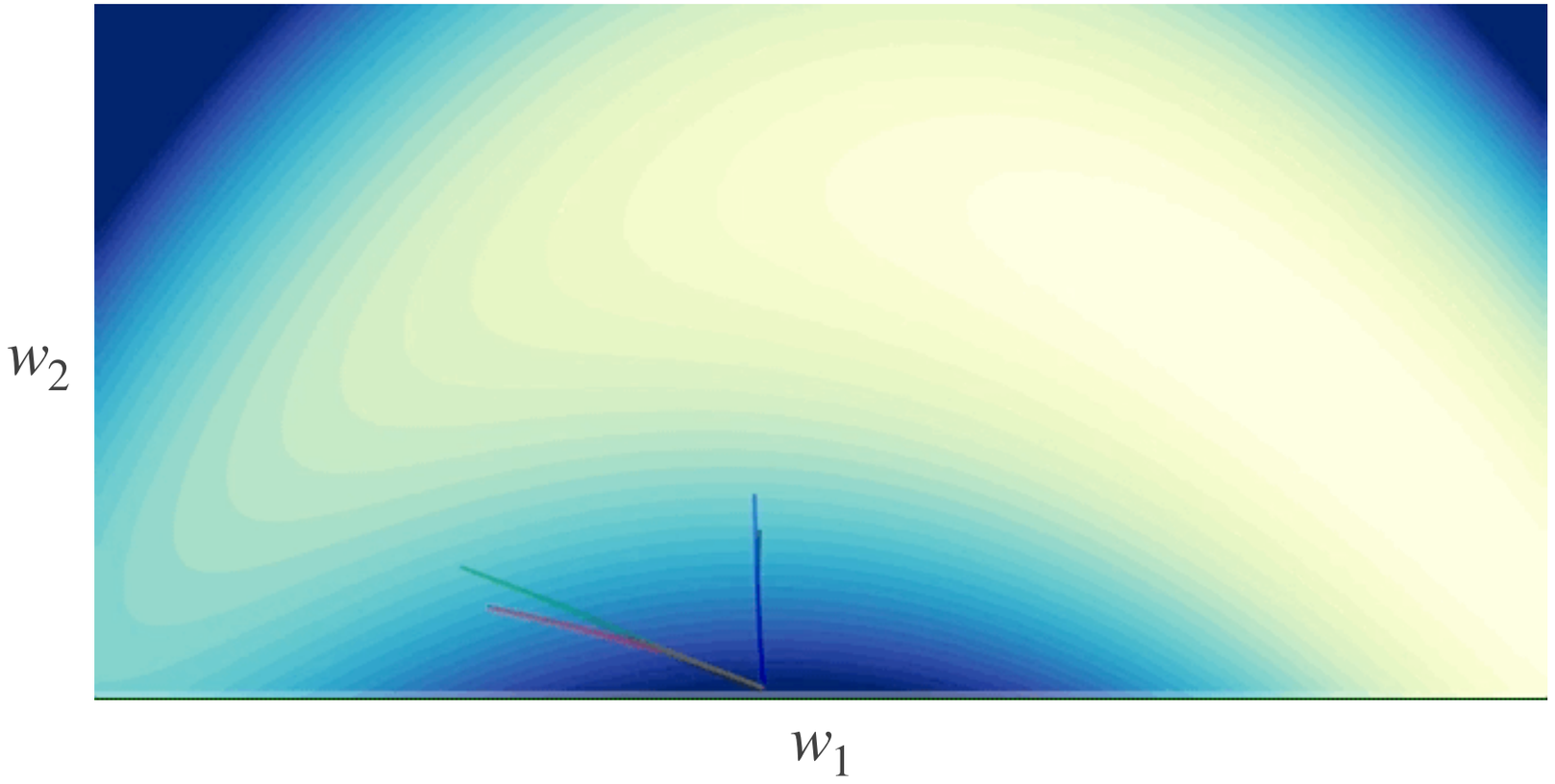
Loss landscape (bright = low, dark = high)



Interactive Visualization of Optimization Algorithms in Deep Learning

Loss landscape (bright = low, dark = high)





Interactive Visualization of Optimization Algorithms in Deep Learning

Fully connected layer
(Dense)

Convolutional layer

Conv1D, 2D, 3D, …
separable Conv

Optimizer

SGD
Adam
RMSprop

Pooling layer

max-pooling
average-pooling

Deep Learning

Evaluation metric

accuracy
F1-score
AUC
confusion matrix

Activation function

sigmoid
softmax
ESP (swish)
ReLU

Loss function

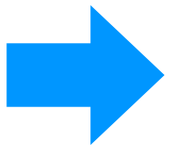
categorical crossentropy
binary crossentropy
mean squared error
mean absolute error

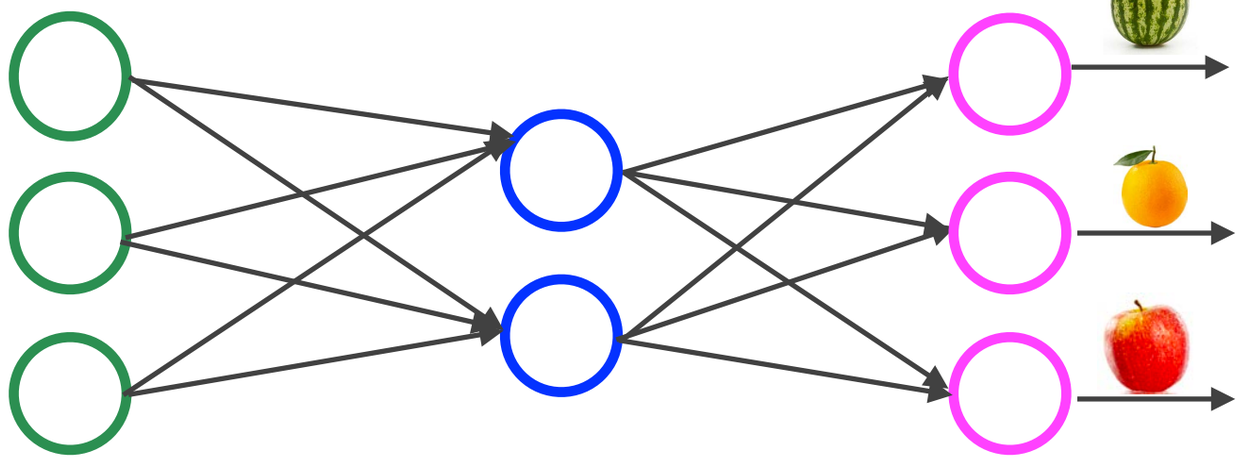
Regularization

Dropout
Data augmentation
l_1, l_2 regularizations

- Combine basic components to build a neural network
  - More components → "More" representative power

x





y

probability of being class 1

probability of being class 2

probability of being class 3

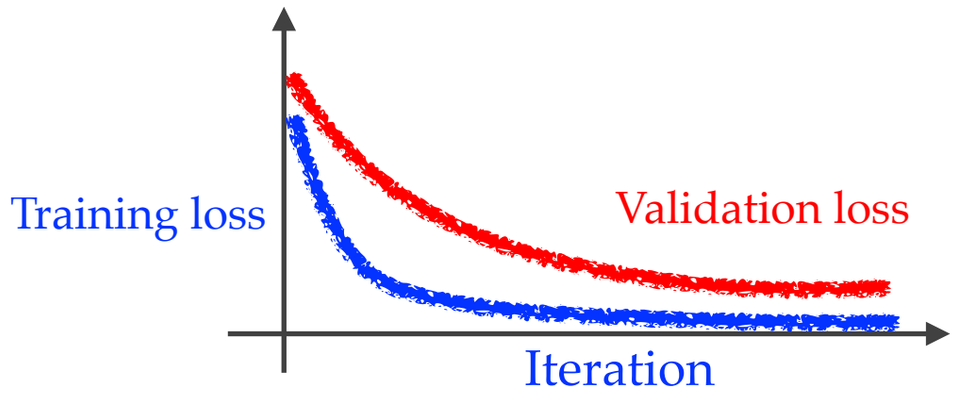
activation functions

ReLU

ReLU

Softmax

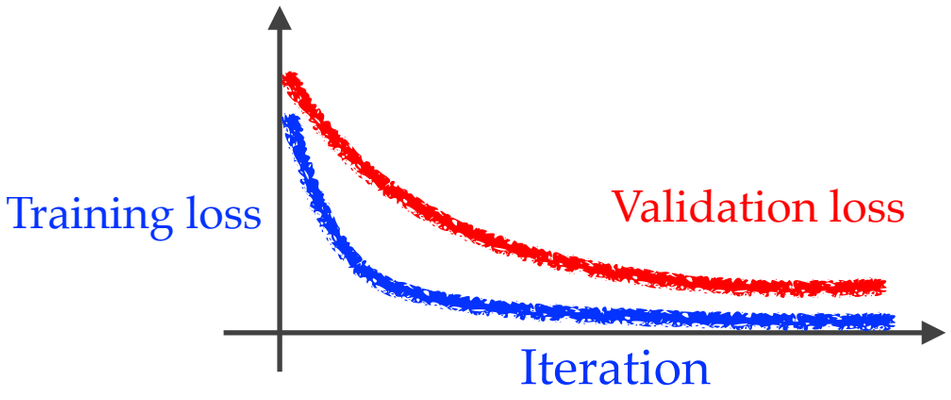
```
# Import necessary modules
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Create the model
model = keras.Sequential()
model.add(layers.Dense(3, activation="relu"))
model.add(layers.Dense(2, activation="relu"))
model.add(layers.Dense(3, activation="softmax"))

# Compile the model
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.CategoricalCrossentropy(),
)

# Train the model for 100 epochs with a batch size of 32
model.fit(x_train, y_train, batch_size=32, epochs=100, validation_data=(x_val,y_val))
```

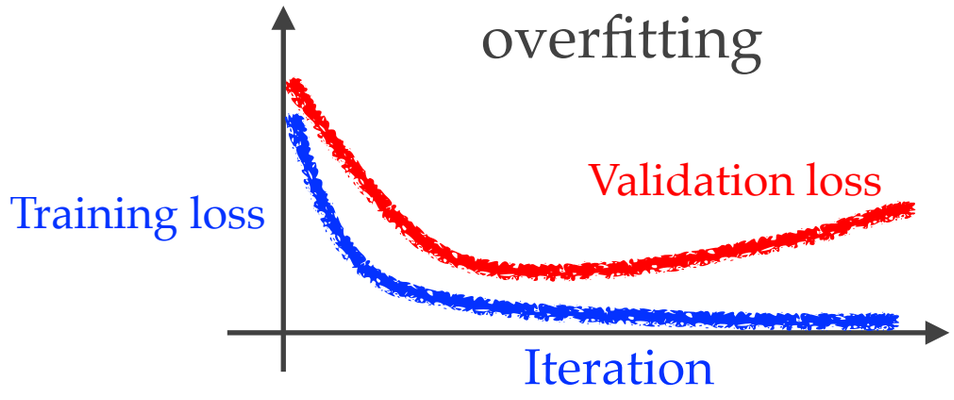


Model

Loss function and optimizer

Regularization





- Regularization is frequently used to mitigate overfitting
  - Add a regularization term to the loss function
    - ℓ_2 regularization

$$min \sum_{i=1}^{N} L(y_i, f_W(x)) \quad \rightarrow \quad min \sum_{i=1}^{N} L(y_i, f_W(x)) + \lambda \|w\|_2$$

    - ℓ_1 regularization

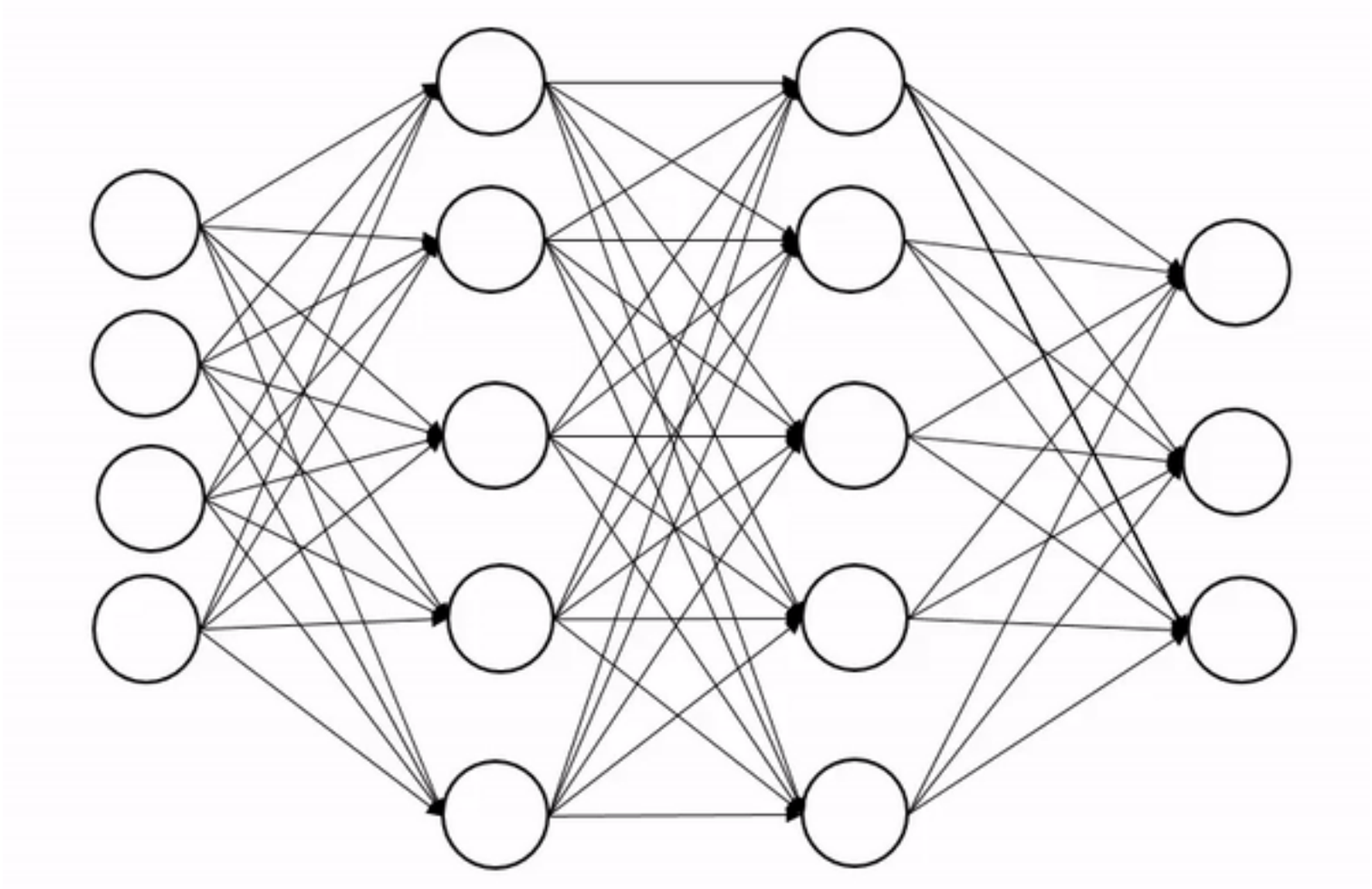
$$min \sum_{i=1}^{N} L(y_i, f_W(x)) \quad \rightarrow \quad min \sum_{i=1}^{N} L(y_i, f_W(x)) + \lambda \|w\|_1$$

  - Use dropout - much more popular

keyword: dropblock for CNN

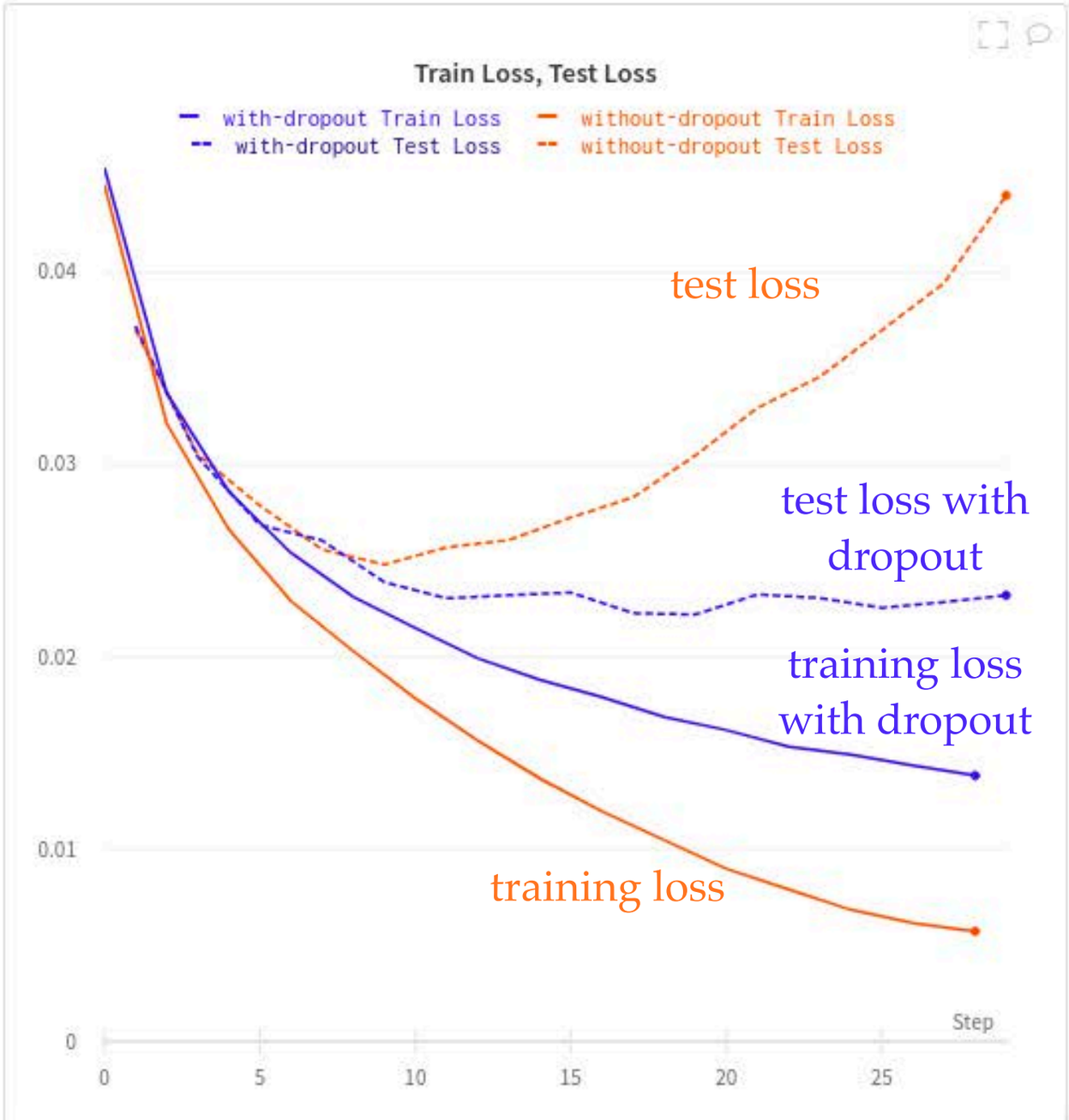
Dropout

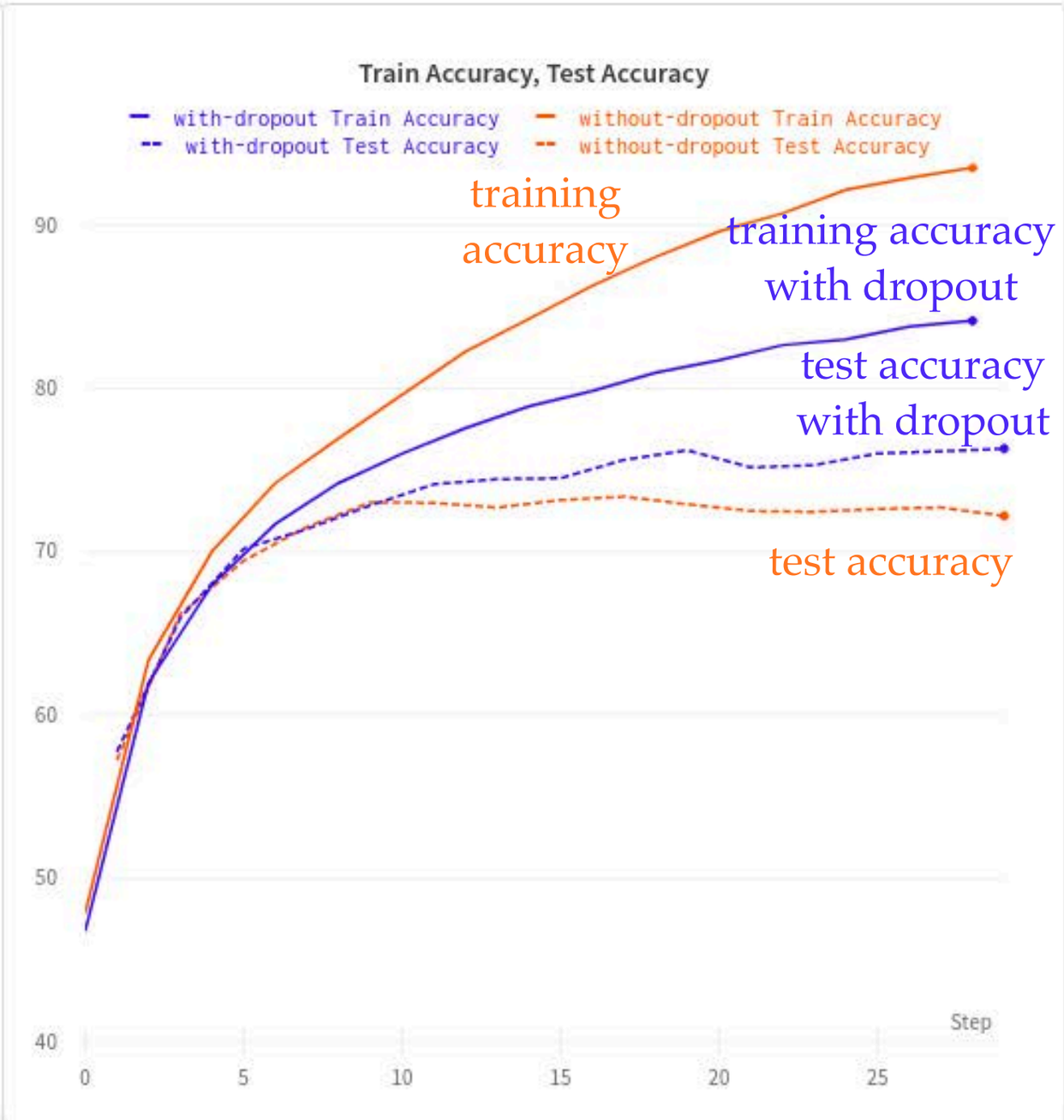
- Dropout can be used to reduce overfitting
  - Randomly omit some neurons/units



Dropout for Deep Learning Regularization, explained with Examples

Dropout

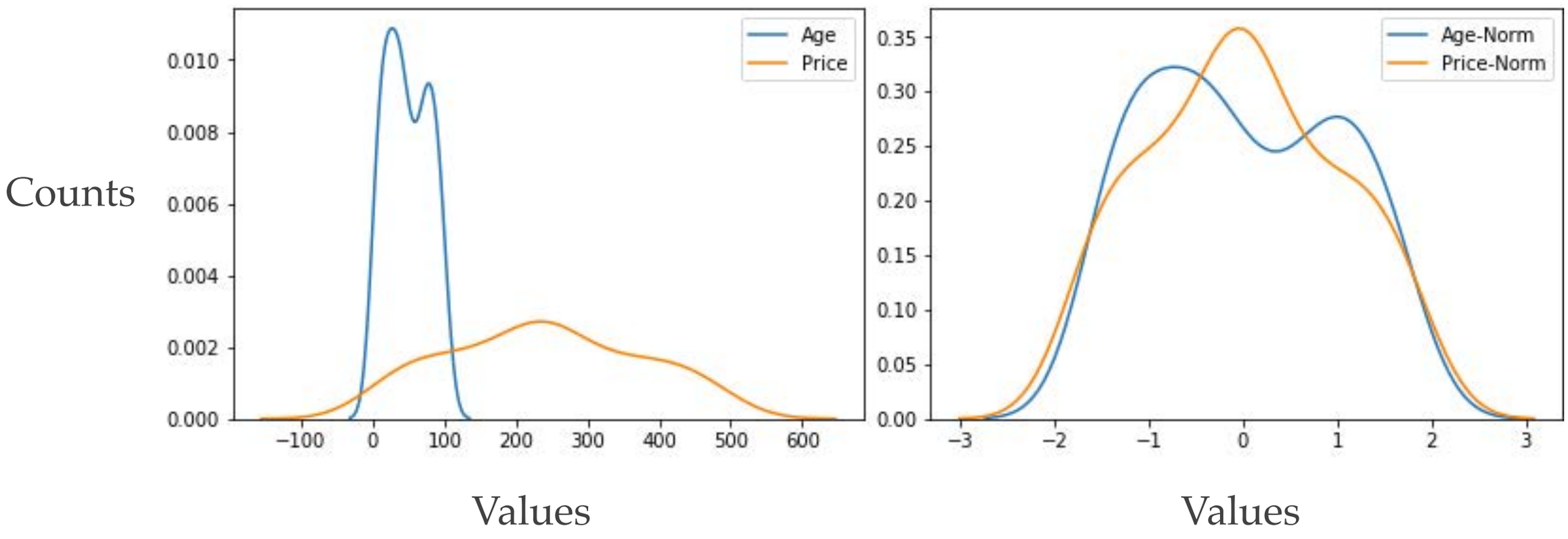




Dropout for Deep Learning Regularization, explained with Examples

Batch Normalization

- Normalize every batch
  - reduce internal covariate shift problems*
- Can be thought of as a form of implicit regularization
  - can help reduce overfitting

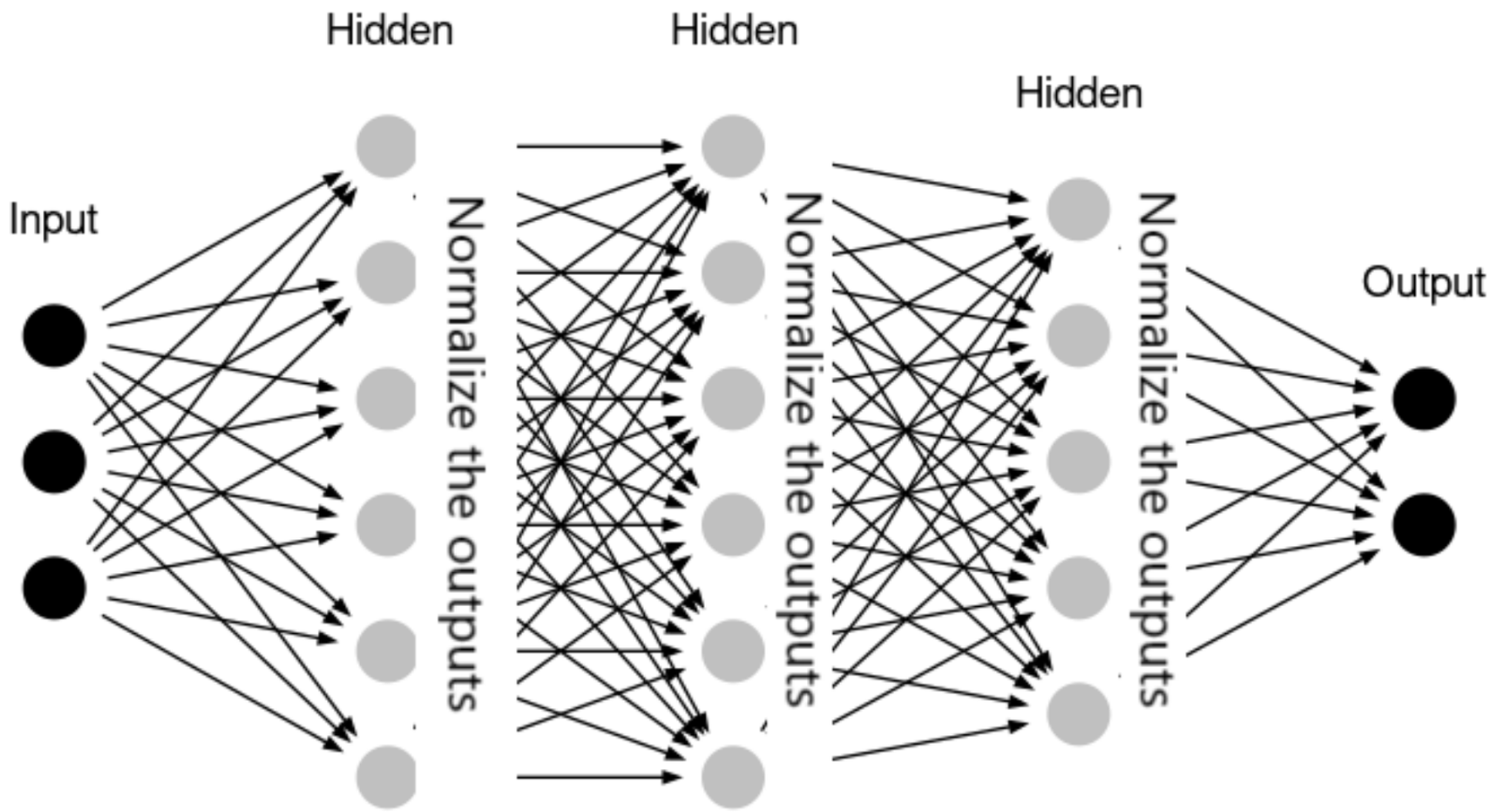


Batch Normalization — Speed up Neural Network Training

*This has been challenged by more recent work

Batch Normalization

- Normalize every batch
  - reduce internal covariate shift problems*
- Can be thought of as a form of implicit regularization
  - can help reduce overfitting



Batch Normalization — Speed up Neural Network Training

*This has been challenged by more recent work

keywords: layer normalization, instance normalization, group normalization, standardization

Batch Normalization

- Reduce internal covariate shift problems*

batch size # of features

Input: $x : N \times D$

Learnable params:
$\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$
$\hat{x} : N \times D$

Output: $y : N \times D$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

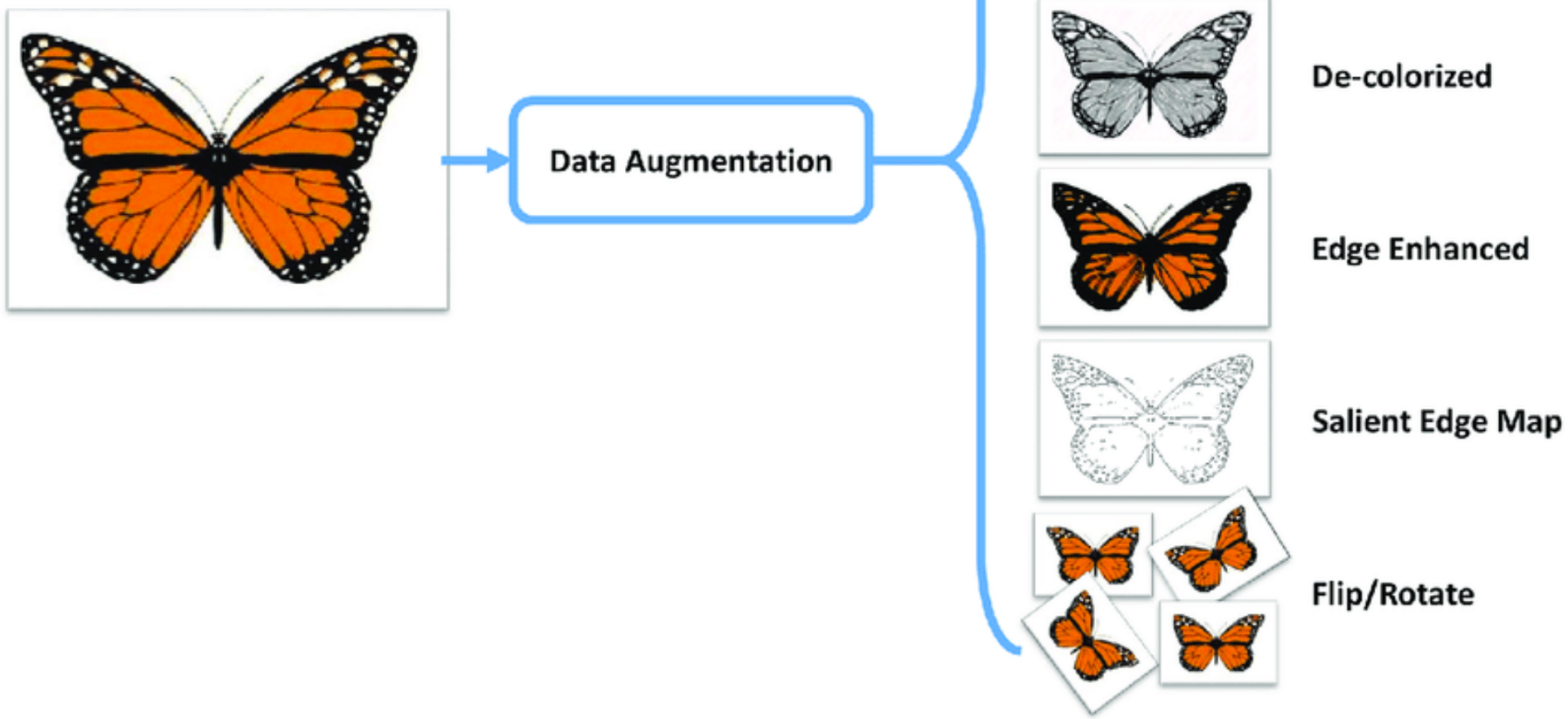
$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

- More robust to bad initialization

Lecture 7 (Stanford, CS231n, 2018)

*This has been challenged by more recent work

Data Augmentation



Data augmentation-assisted deep learning of hand-drawn partially colored sketches for visual search

Fully connected layer
(Dense)

Optimizer

SGD
Adam
RMSprop

Convolutional layer

Conv1D, 2D, 3D, …
separable Conv

Deep Learning

Pooling layer

max-pooling
average-pooling

Evaluation metric

accuracy
F1-score
AUC
confusion matrix

Activation function

sigmoid
softmax
ESP (swish)
ReLU

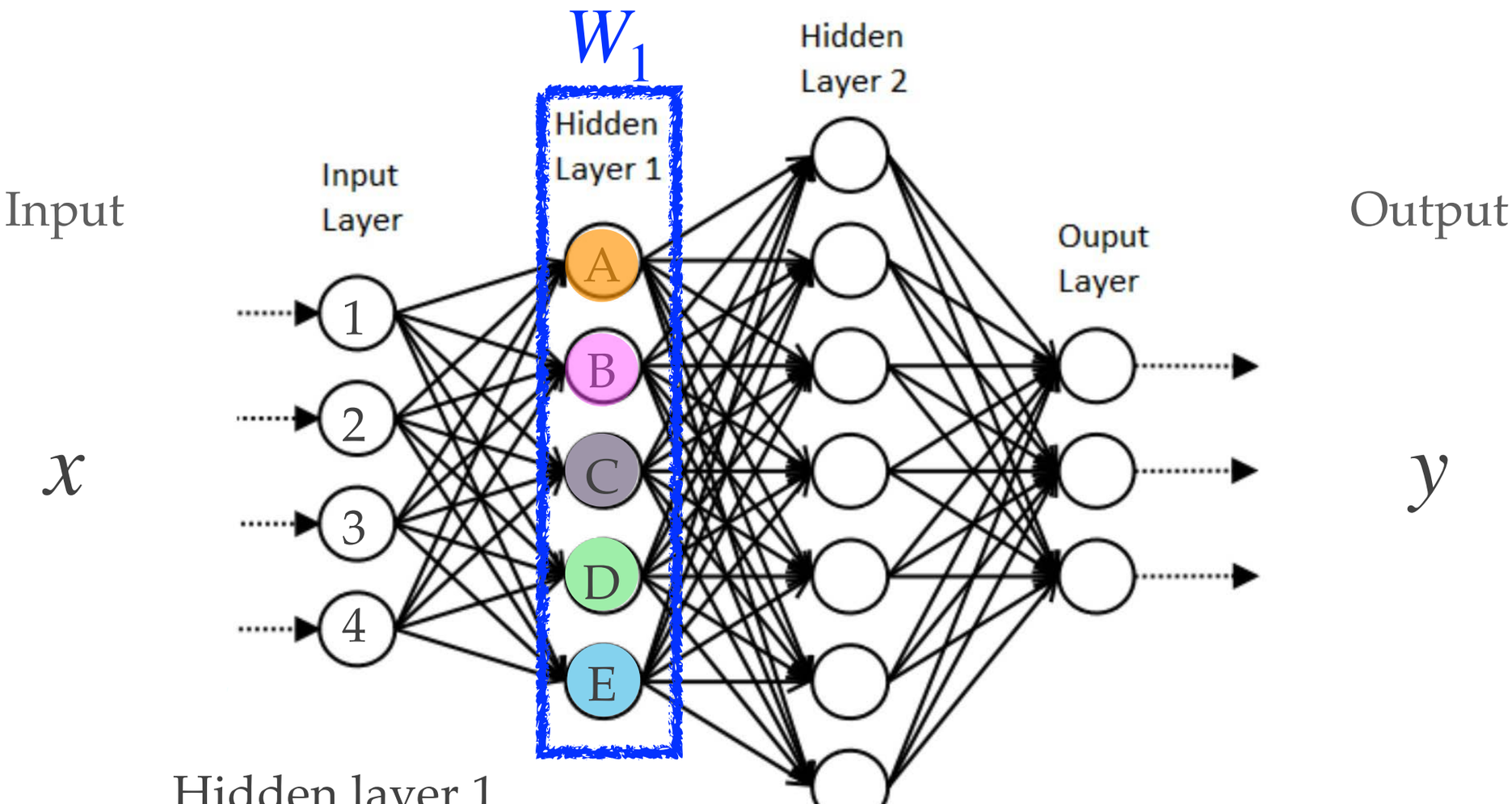
Loss function

categorical crossentropy
binary crossentropy
mean squared error
mean absolute error

Regularization

Dropout
Data augmentation
l_1, l_2 regularizations

- Combine basic components to build a neural network
  - More components → "More" representative power



Hidden layer 1

(ignore the biases for simplicity)

$$\begin{bmatrix} out_A \\ out_B \\ out_C \\ out_D \\ out_E \end{bmatrix} = a_1(W_1 x) = a_1\left(\begin{bmatrix} w_{A,1} & w_{A,2} & w_{A,3} & w_{A,4} \\ w_{B,1} & w_{B,2} & w_{B,3} & w_{B,4} \\ w_{C,1} & w_{C,2} & w_{C,3} & w_{C,4} \\ w_{D,1} & w_{D,2} & w_{D,3} & w_{D,4} \\ w_{E,1} & w_{E,2} & w_{E,3} & w_{E,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \right)$$

# of weights for hidden layer 1
= 4 x 5 = 20

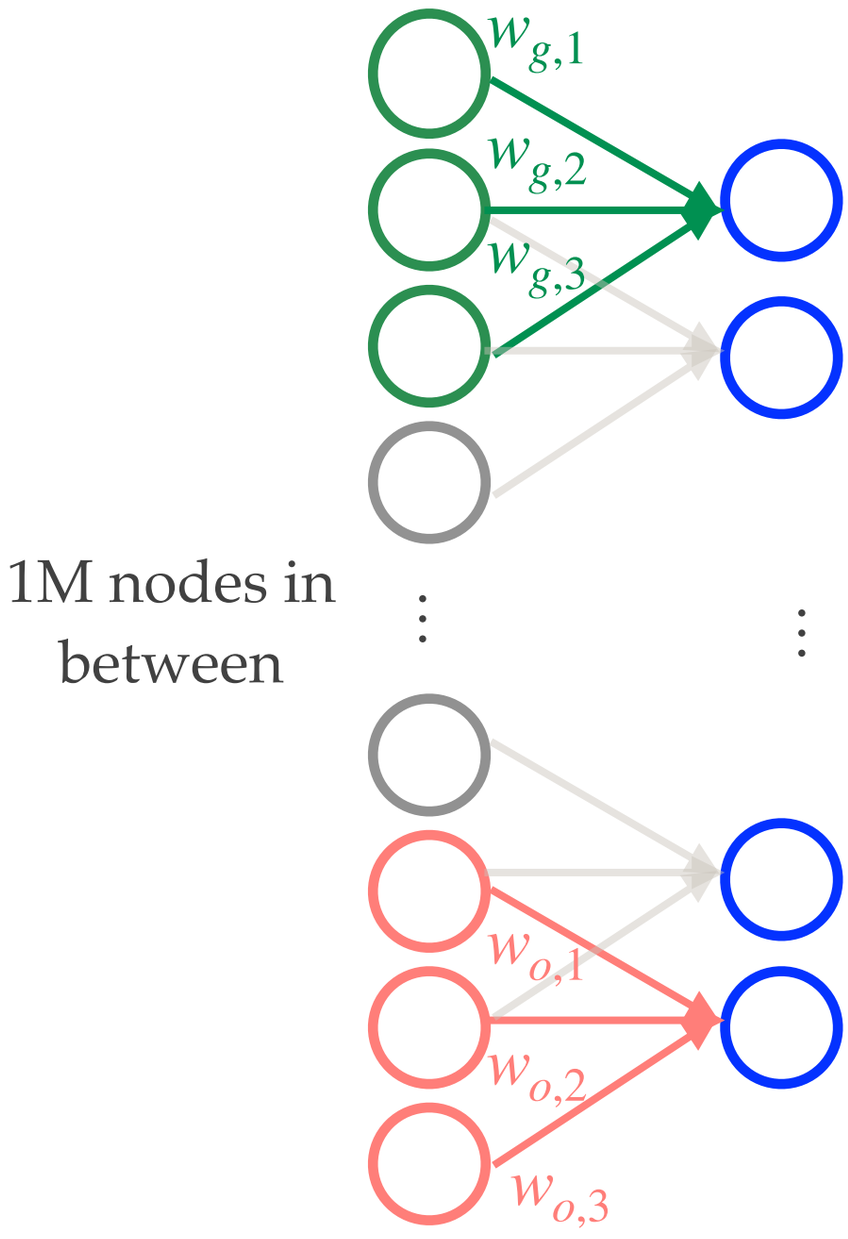
# of weights per layer = # of incoming nodes x # of hidden nodes

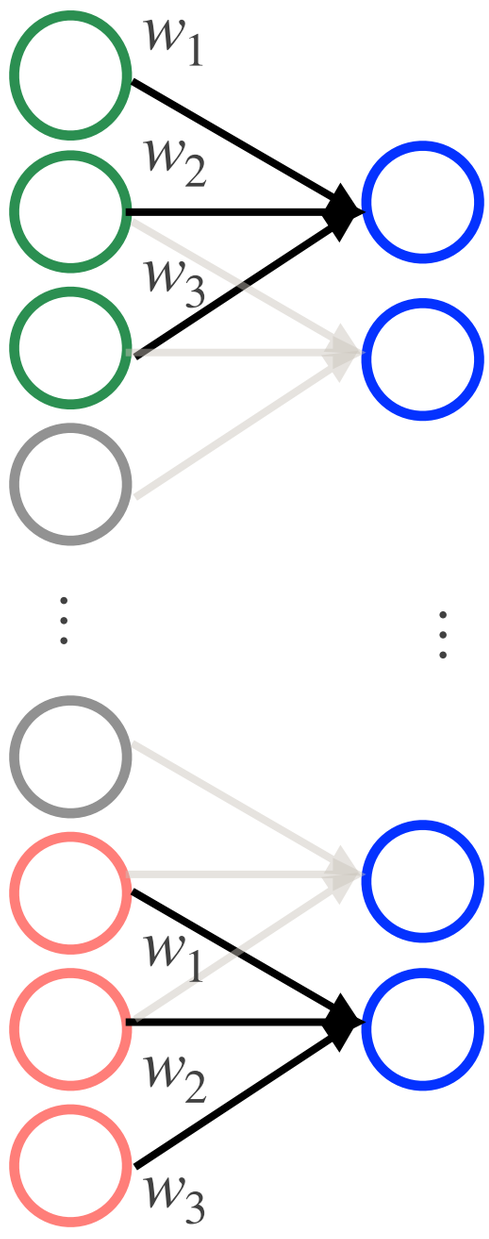
What if our input dimension is 256x256=65,536 and we use 1024 hidden nodes?

67 M trainable parameters for just one layer!

Do we need that many parameters?

- Would it be the case that
  - Only neighboring neurons talk to each other?
  - Furthermore, they talk to their neighbors in the same way?

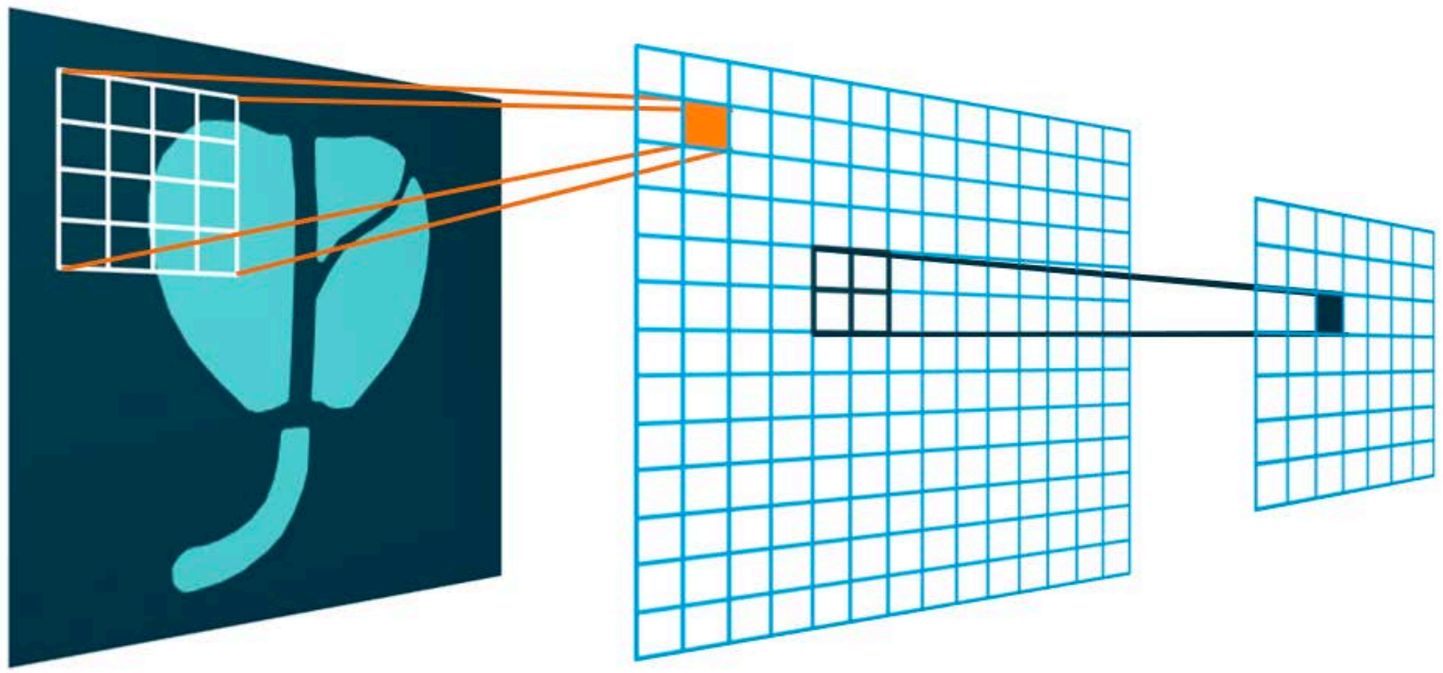




Much fewer parameters!

2D Convolution

- Similarly, for images, maybe only neighboring pixels talk to each other?



2D convolution 2D pooling

Deep learning radiology: the secret of convolutional neural networks

2D Convolution

$$y[n_1, n_2] = x[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1, k_2] h[n_1 - k_1, n_2 - k_2]$$

2D Conv

0*0 + 2*1 + 1*0 + 0*1 + 1*1 + 1*1 + -3*0 + -1*1 + 1*0

0	2	1	1	0
0	1	1	1	0
-3	-1	1	0	1
0	6	0	0	1
0	4	1	7	0

x

Input

0	1	0
1	1	1
0	1	0

h

Filter / kernel

defines the relationship between neighboring pixels

	3			

y

2D Convolution

$$y[n_1, n_2] = x[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1, k_2] h[n_1 - k_1, n_2 - k_2]$$

2D Conv

2*0 + 1*1 + 1*0 + 1*1 + 1*1 + 1*1 + -1*0 + 1*1 + 0*0

0	2	1	1	0
0	1	1	1	0
-3	-1	1	0	1
0	6	0	0	1
0	4	1	7	0

x

Input

0	1	0
1	1	1
0	1	0

h

Filter / kernel

defines the relationship between neighboring pixels

3	5	

y

2D Convolution

$$y[n_1, n_2] = x[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1, k_2] h[n_1 - k_1, n_2 - k_2]$$

2D Conv

1*0 + 1*1 + 0*0 + 1*1 + 1*1 + 0*1 + 1*0 + 0*1 + 1*0

0	2	1	1	0
0	1	1	1	0
-3	-1	1	0	1
0	6	0	0	1
0	4	1	7	0

x

Input

0	1	0
1	1	1
0	1	0

h

Filter / kernel

defines the relationship between neighboring pixels

	3	5	3	

y

2D Convolution

$$y[n_1, n_2] = x[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1, k_2] h[n_1 - k_1, n_2 - k_2]$$

2D Conv

0	2	1	1	0
0	1	1	1	0
-3	-1	1	0	1
0	6	0	0	1
0	4	1	7	0

x

Input

0	1	0
1	1	1
0	1	0

h

Filter / kernel

defines the relationship between neighboring pixels

	3	5	3	
	4	1	3	
	9	8	8	

y

smaller output

2D Convolution

Zero-padding

0	0	0	0	0	0	0
0	0	2	1	1	0	0
0	0	1	1	1	0	0
0	-3	-1	1	0	1	0
0	0	6	0	0	1	0
0	0	4	1	7	0	0
0	0	0	0	0	0	0

x

Input

0	1	0
1	1	1
0	1	0

h

Filter / kernel

defines the relationship between neighboring pixels

	3	5	3	
	4	1	3	
	9	8	8	

y

smaller output

2D Convolution

Zero-padding

0	0	0	0	0	0	0
0	0	2	1	1	0	0
0	0	1	1	1	0	0
0	-3	-1	1	0	1	0
0	0	6	0	0	1	0
0	0	4	1	7	0	0
0	0	0	0	0	0	0

x

Input

0	1	0
1	1	1
0	1	0

h

Filter / kernel

defines the relationship between neighboring pixels

2	4	5	3	1
-2	3	5	3	2
-4	4	1	3	2
3	9	8	8	2
4	11	12	8	8

y

output of the same size

2D Convolution

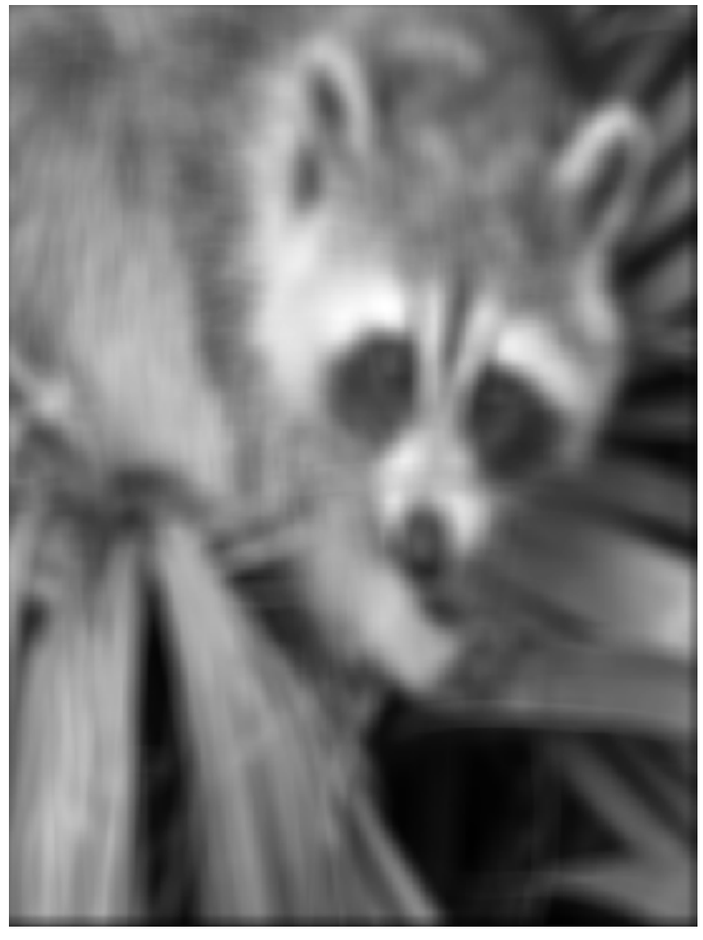
- Low-pass Filter

*

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

Filter / kernel

=



2D Convolution

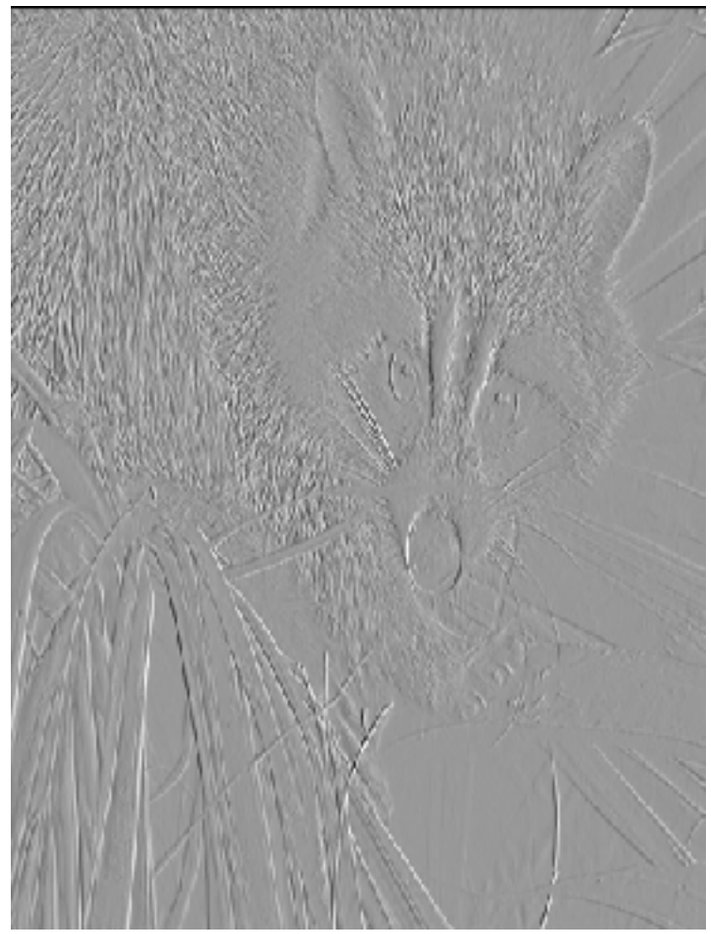
- "Gradient image"

$*$

-1	1
-1	1

Filter / kernel

$=$



2D Convolution

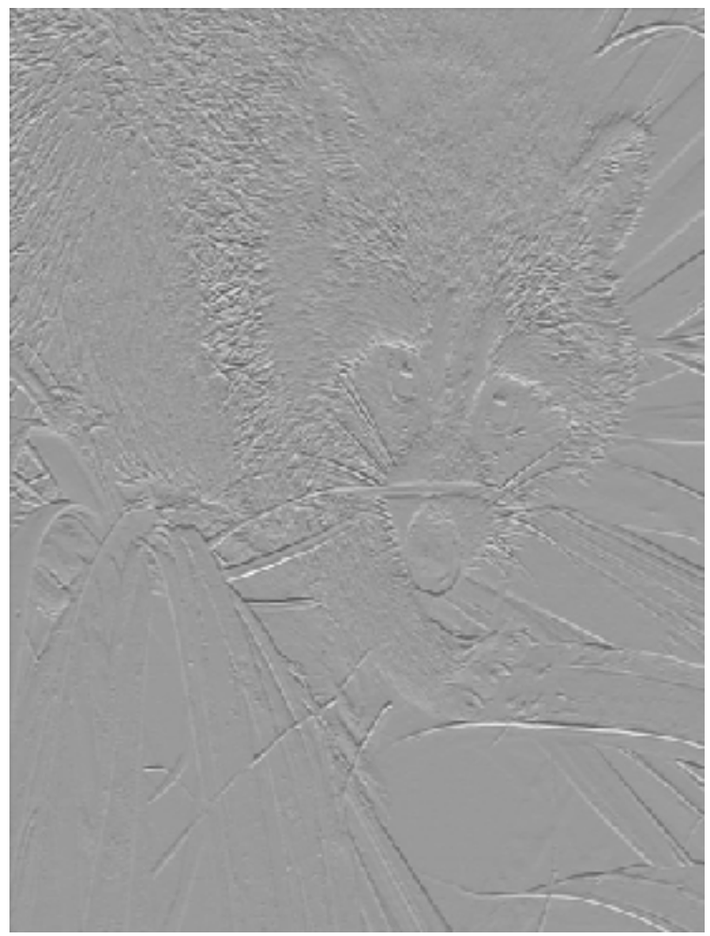
- "Gradient image"

*

-1	-1
1	1

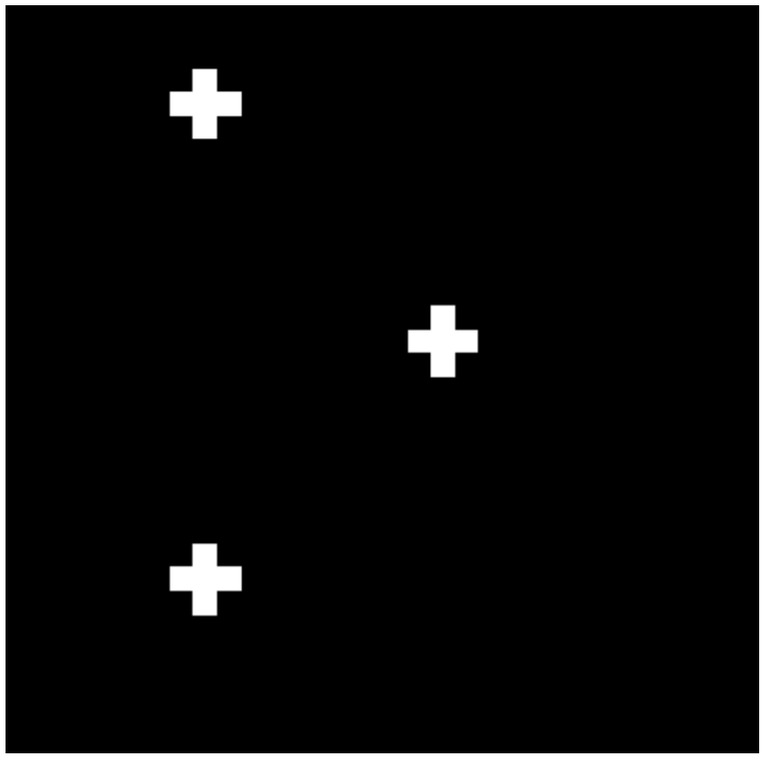
Filter / kernel

=



2D Convolution

- "Plus-sign Detector"

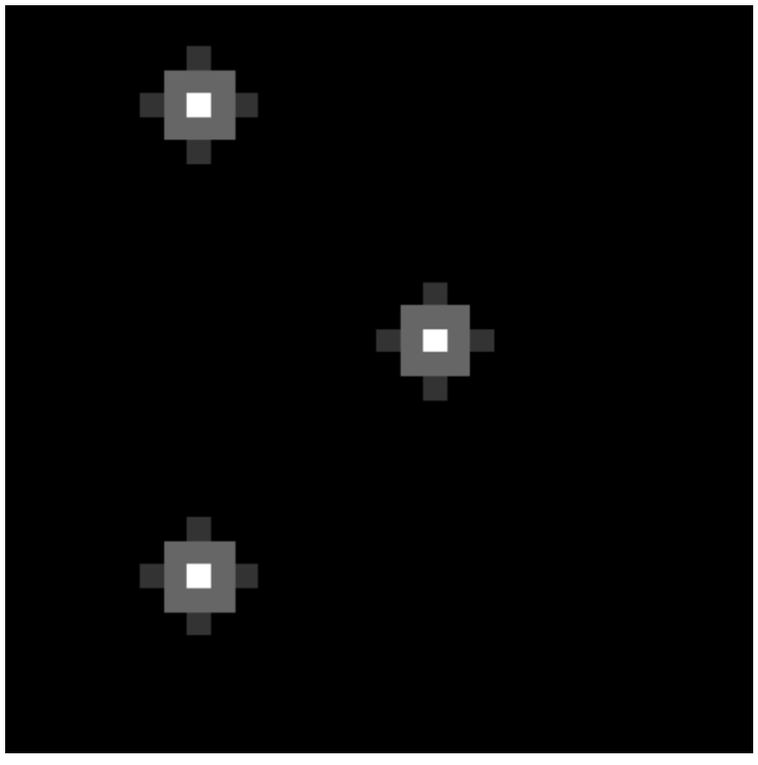


*

0	1	0
1	1	1
0	1	0

Filter / kernel

=



2D Convolution

- Different filters/kernels process images differently
  - Low-pass filter (blurring/denoising)
  - High-pass filter (sharpening)
  - Shape detector
  - Refer to Image Kernels for a good demo

complementary information

- Instead of defining the filter ourselves, we let neural network come up with good ones for us

w_{11}	w_{12}	w_{13}
w_{21}	w_{22}	w_{23}
w_{31}	w_{32}	w_{33}

Filter/kernel
defines the relationship
between neighboring pixels

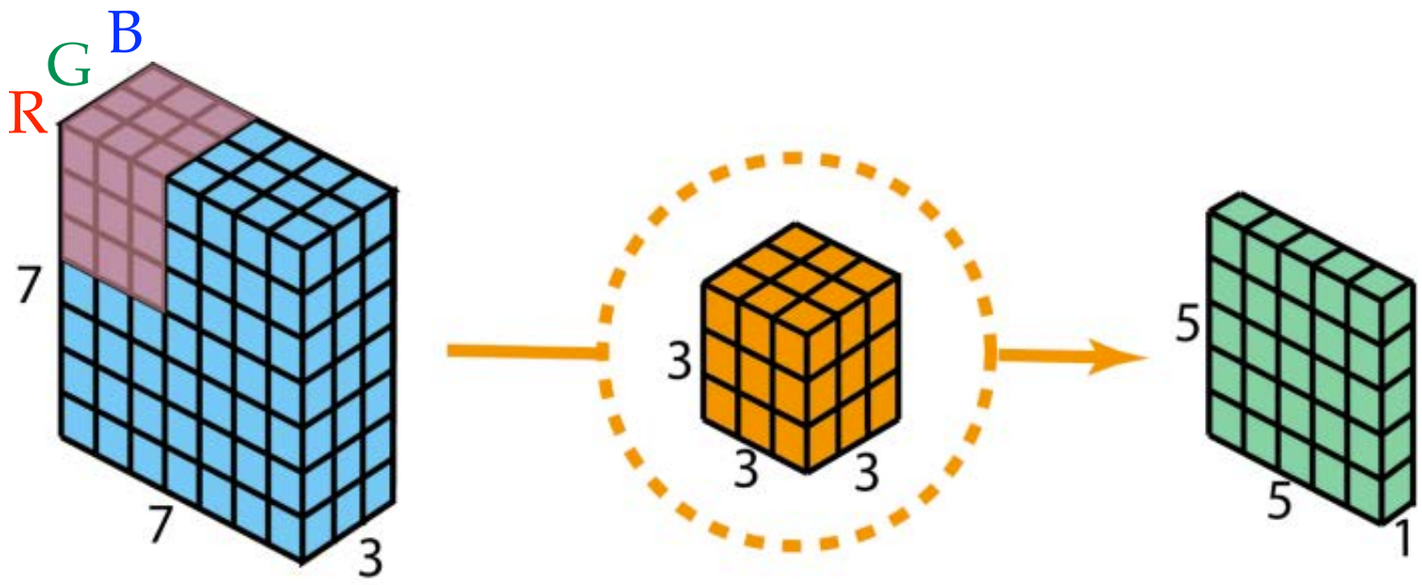
```
model.add(Conv2D(32, (3, 3), padding="same", activation="relu"))
```

# of filters filter size

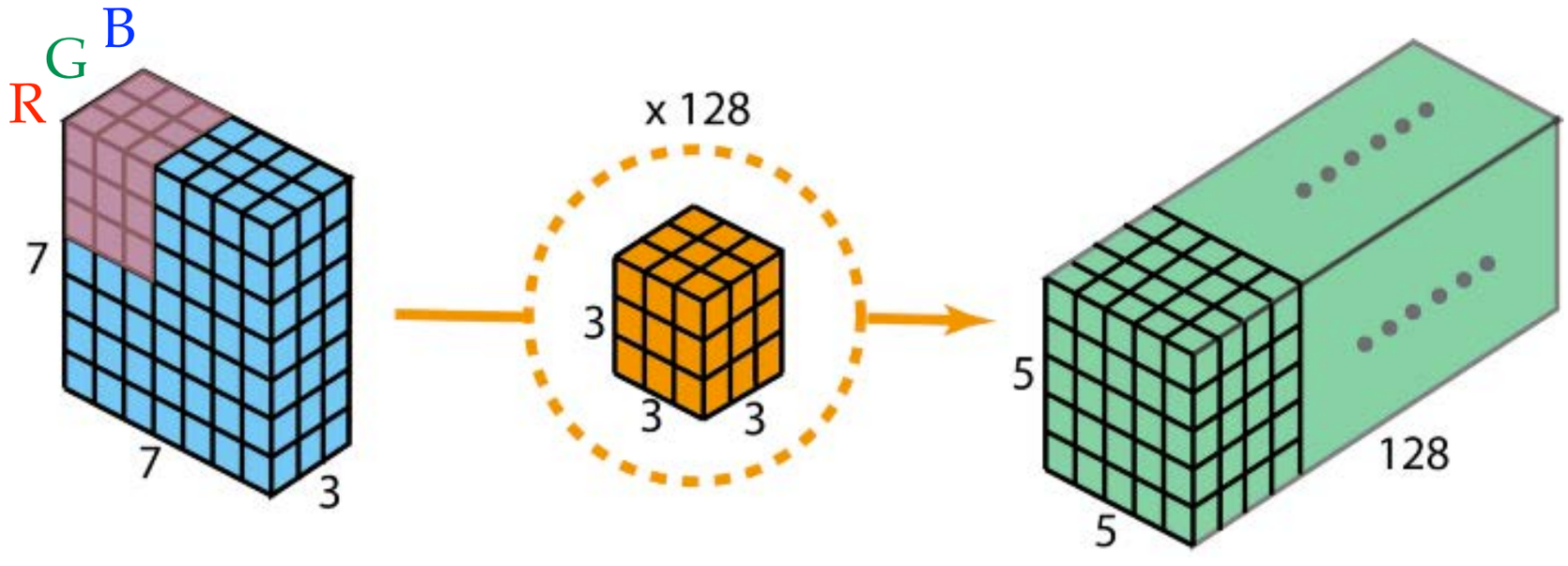
2D Convolution

- 2D convolution with more than one channels

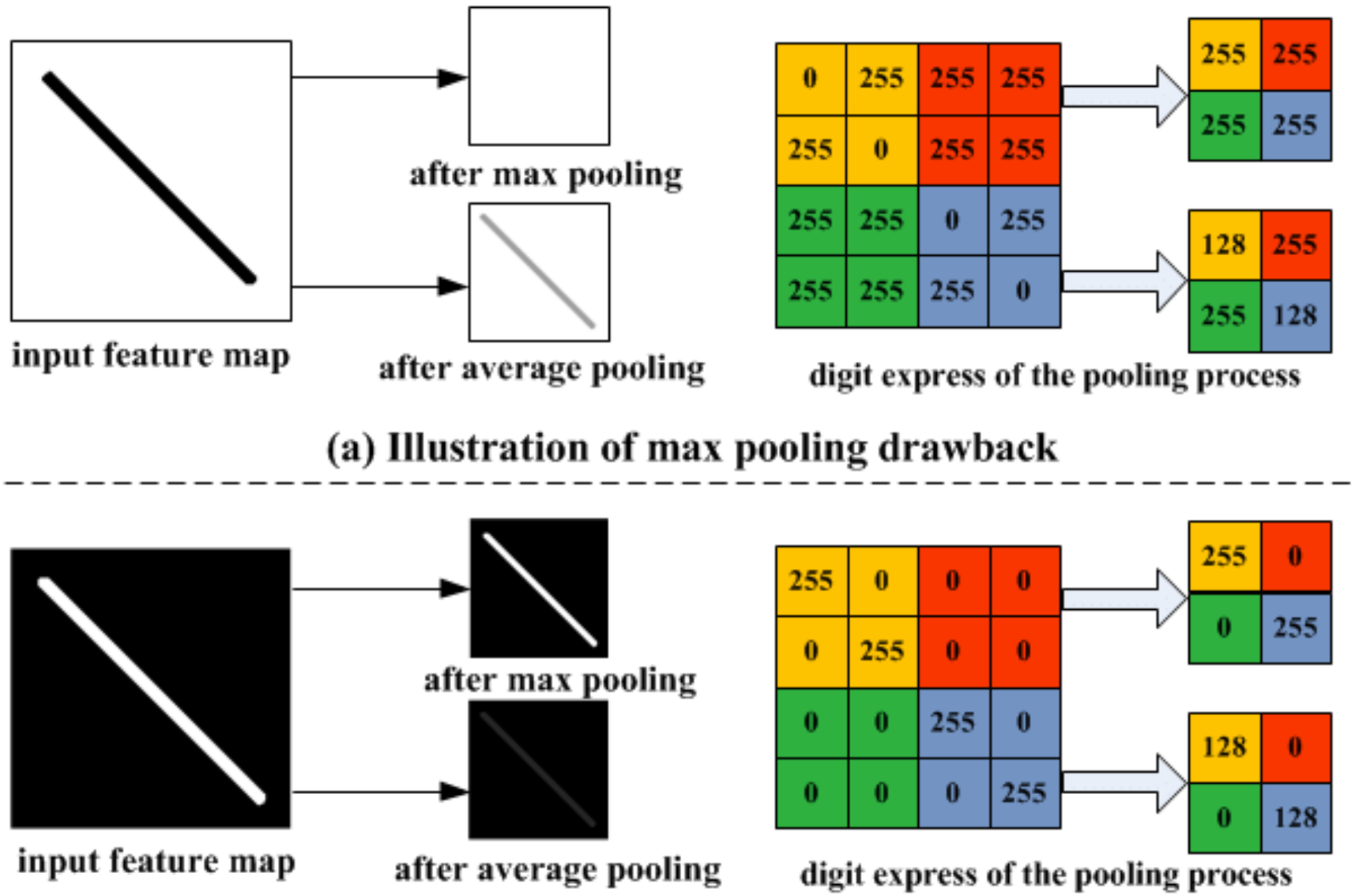
1 filter



128 filters



Pooling Layers



(a) Illustration of max pooling drawback

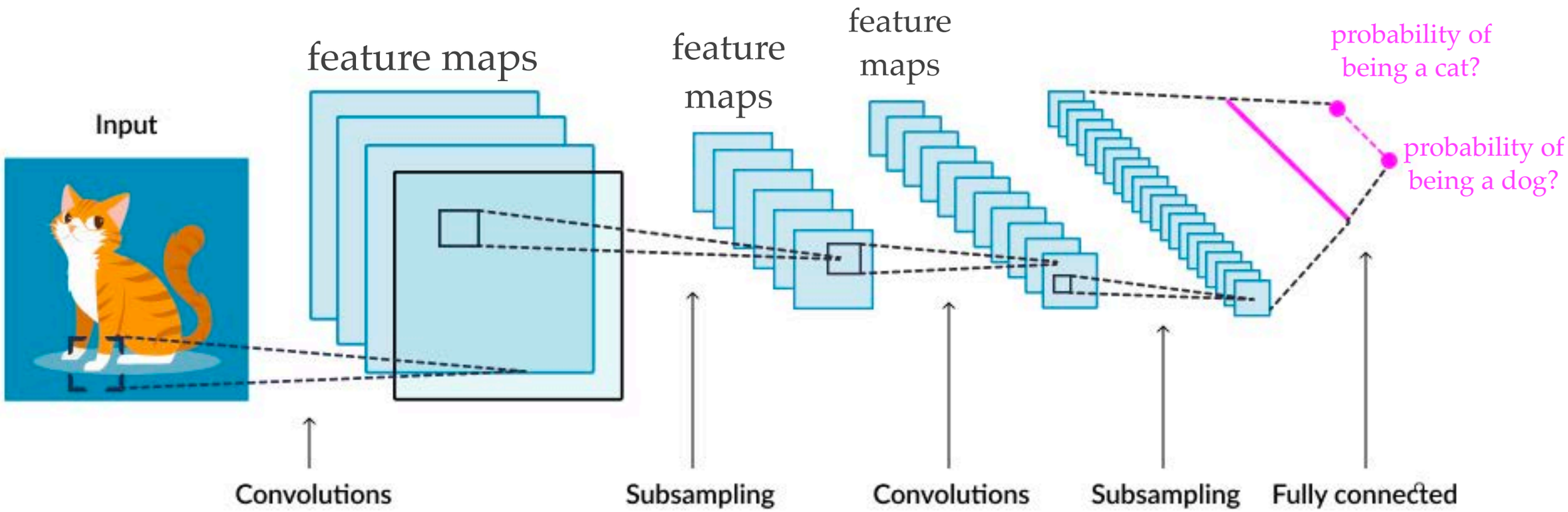
(b) Illustration of average pooling drawback

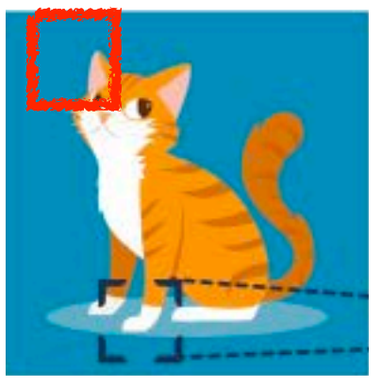
```
model.add(layers.MaxPooling2D((2,2)))
```

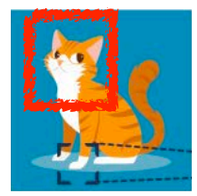
Mixed Pooling for Convolutional Neural Networks

keywords: feature maps, activation maps, receptive field, backbone

Convolutional Neural Network (CNN)



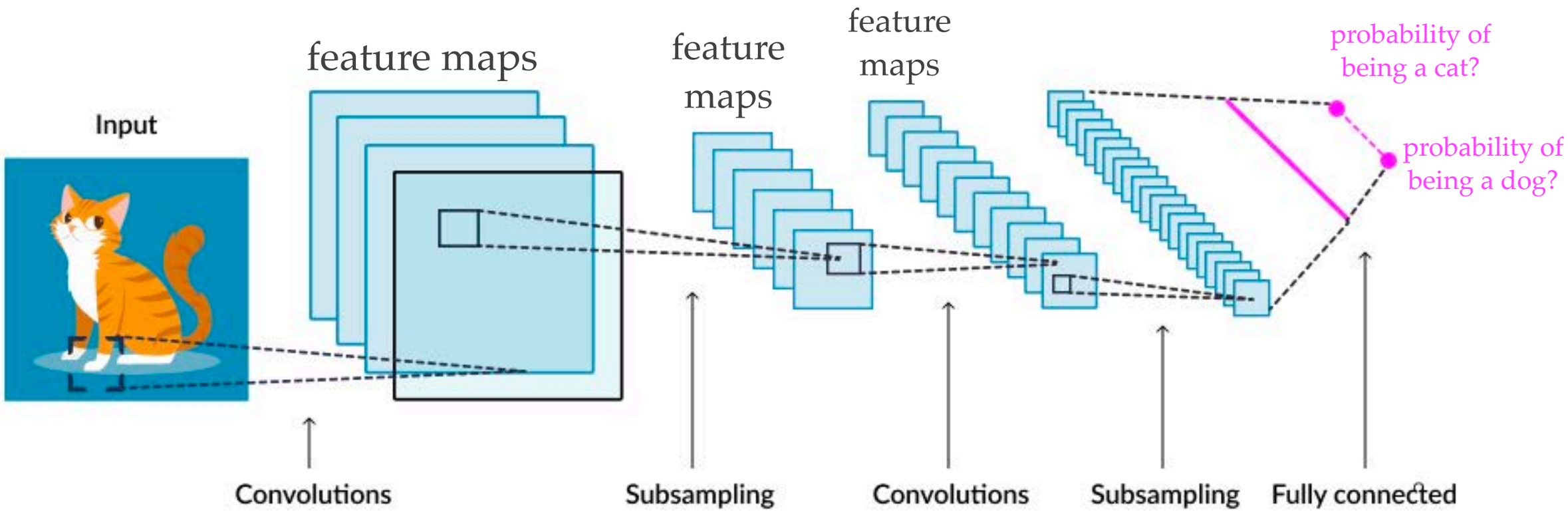




Same filter size (e.g., 3 x 3 filter), but different coverages for different "resolutions"

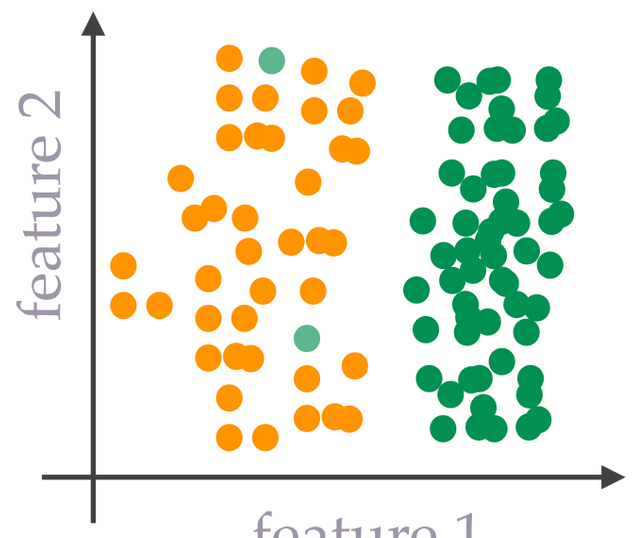
keywords: feature maps, activation maps, receptive field, backbone

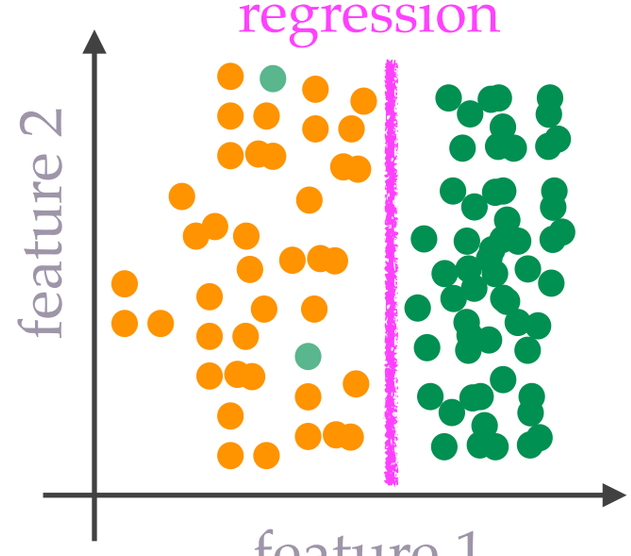
Convolutional Neural Network (CNN)



Feature extraction

Classification/ regression





Disadvantages of CNN models

Fully connected layer
(Dense)

Optimizer

SGD
Adam
RMSprop

Convolutional layer

Conv1D, 2D, 3D, …
separable Conv

Deep Learning

Evaluation metric

accuracy
F1-score
AUC
confusion matrix

Pooling layer

max-pooling
average-pooling

Activation function

sigmoid
softmax
ESP (swish)
ReLU

Loss function

categorical crossentropy
binary crossentropy
mean squared error
mean absolute error

Regularization

Dropout
Data augmentation
l_1, l_2 regularizations

- Combine basic components to build a neural network
  - More components → "More" representative power

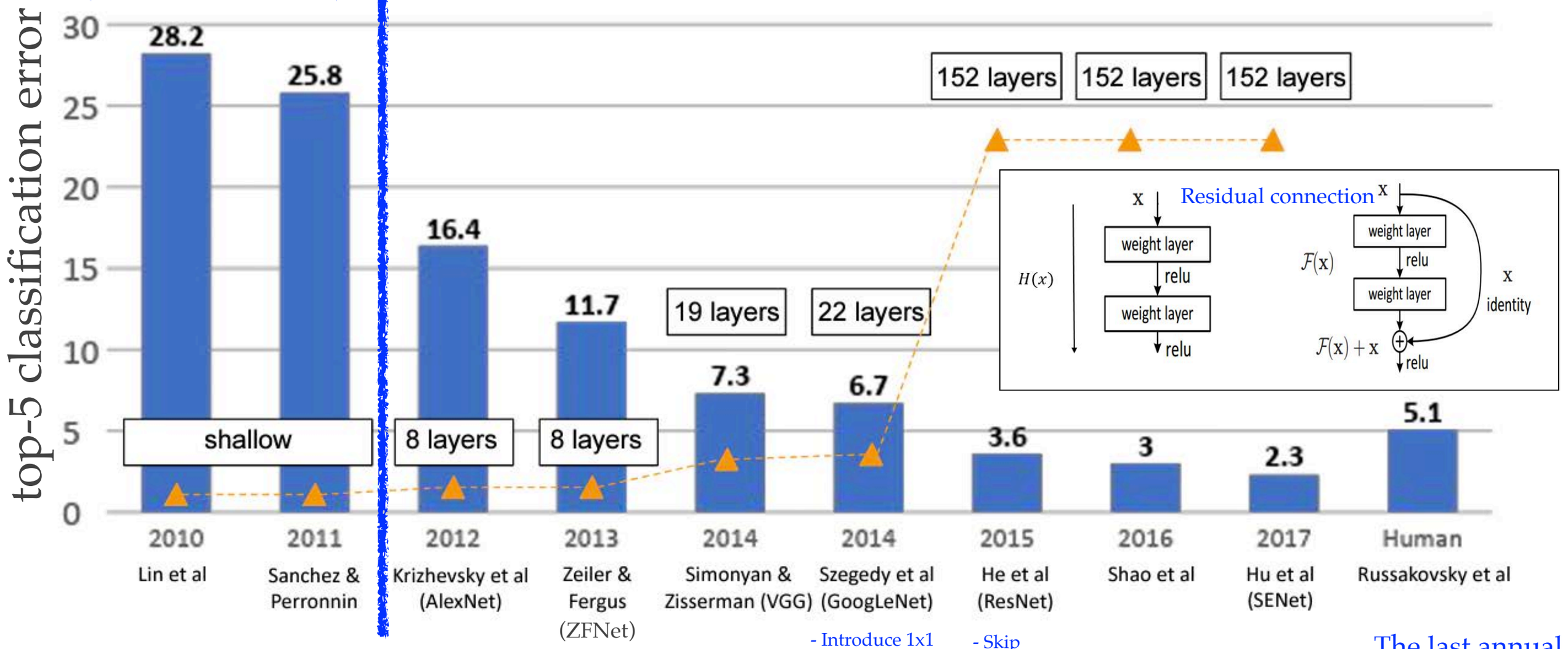
More networks: Xception, ResNeXt, DenseNet, MobileNet, NASNet, EfficientNet, SqueezeNet, Feature Pyramid Network

ImageNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

The SVM era (traditional ML)

The deep learning era



- Conv
- Pooling
- Dense
- Dropout
- ReLU
- SGD with momentum

Optimized "AlexNet" in terms of # filter, kernel sizes etc.

19 layers with filter size = 3x3

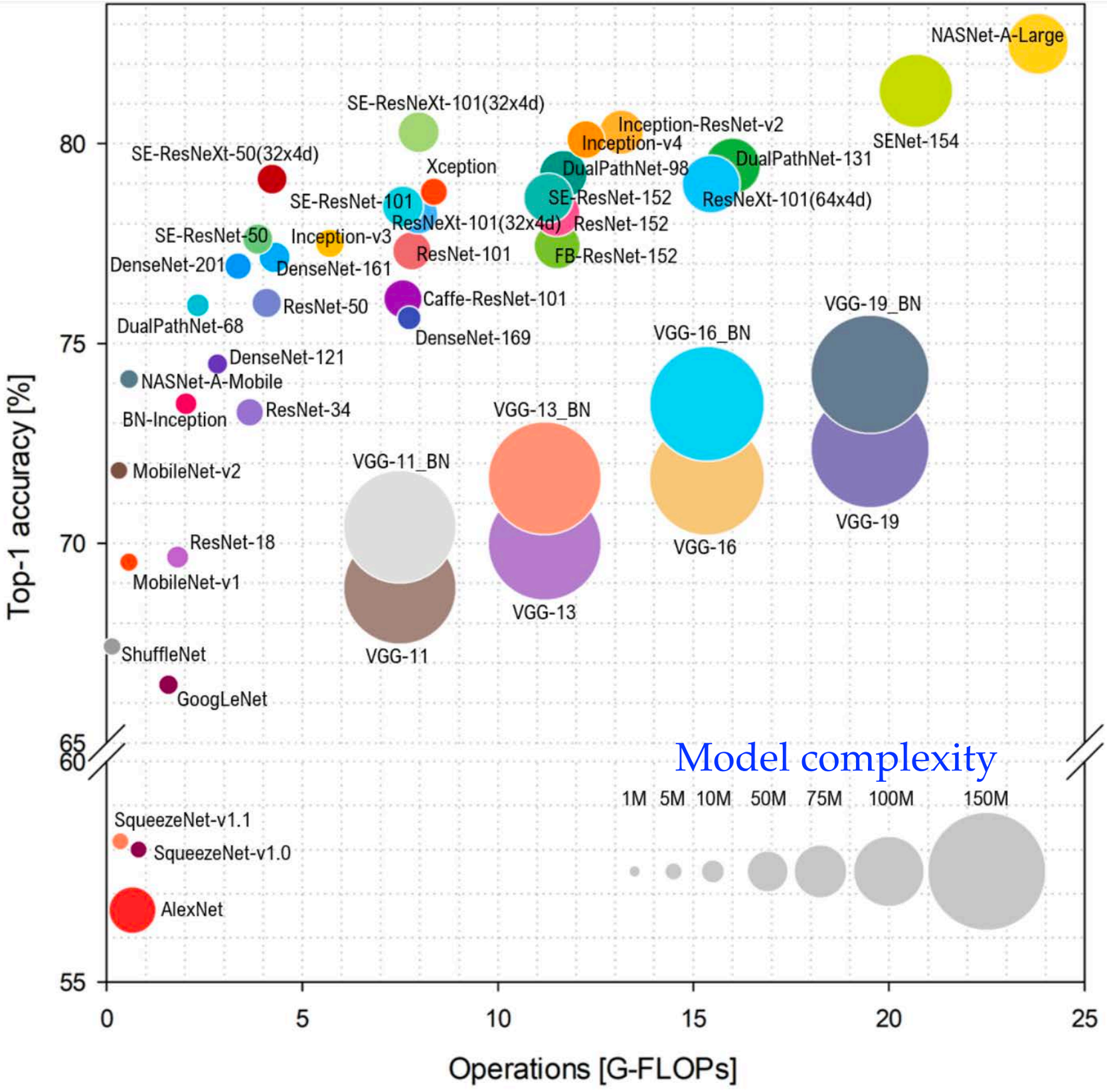
- Introduce 1x1 Conv to reduce the dimensions
- Multiple losses to help with gradients
- Inception module

- Skip connection to help with the vanishing gradient problem
- Batch normalization

- Network ensemble

- Squeeze-and-Excitation (SE) block
- Adaptive feature map reweighting

The last annual ImageNet competition was held in 2017

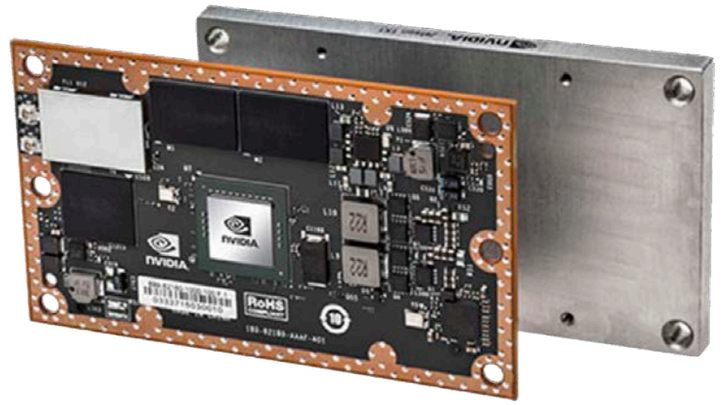


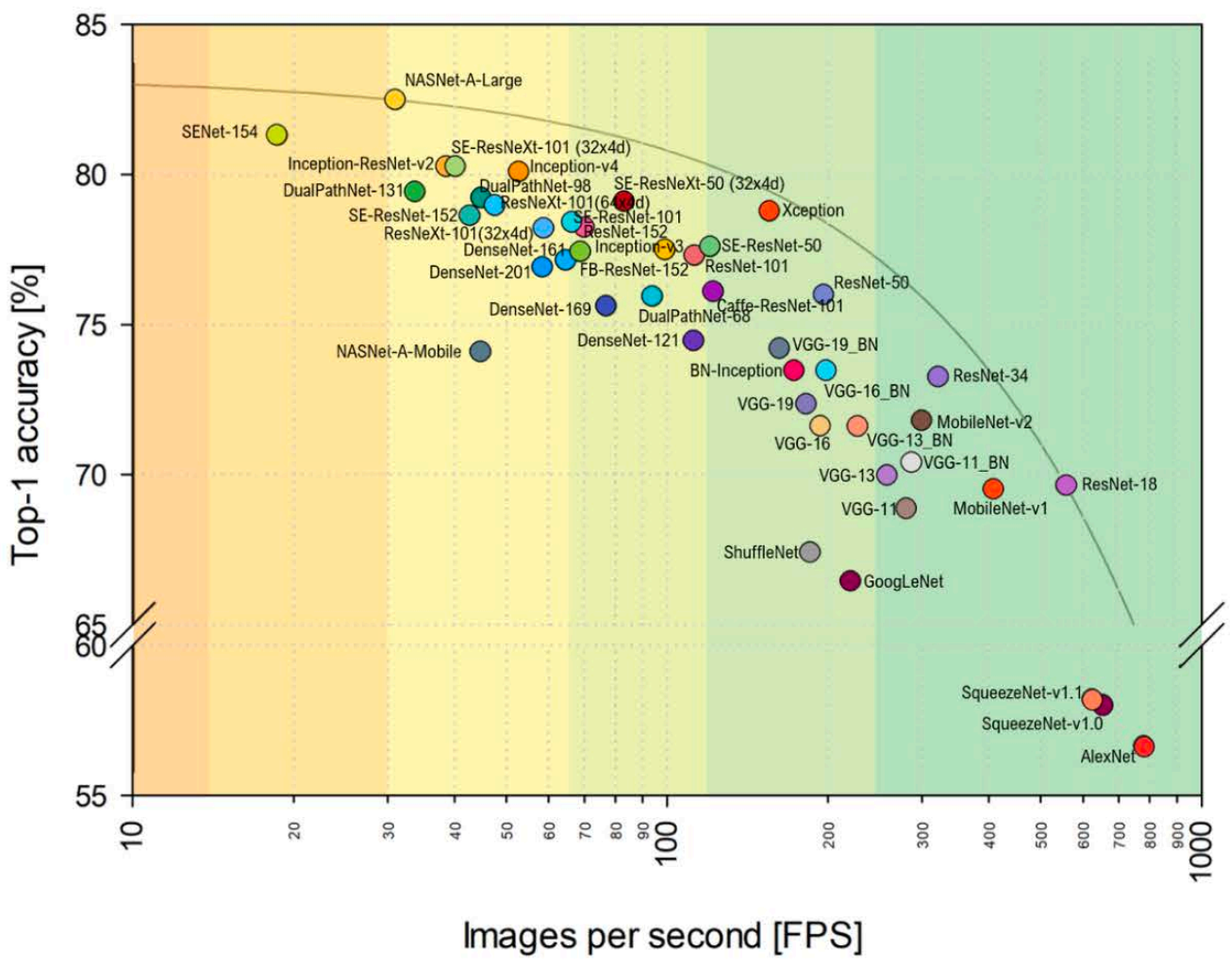
Floating-point operations (FLOPs) required for a single forward pass

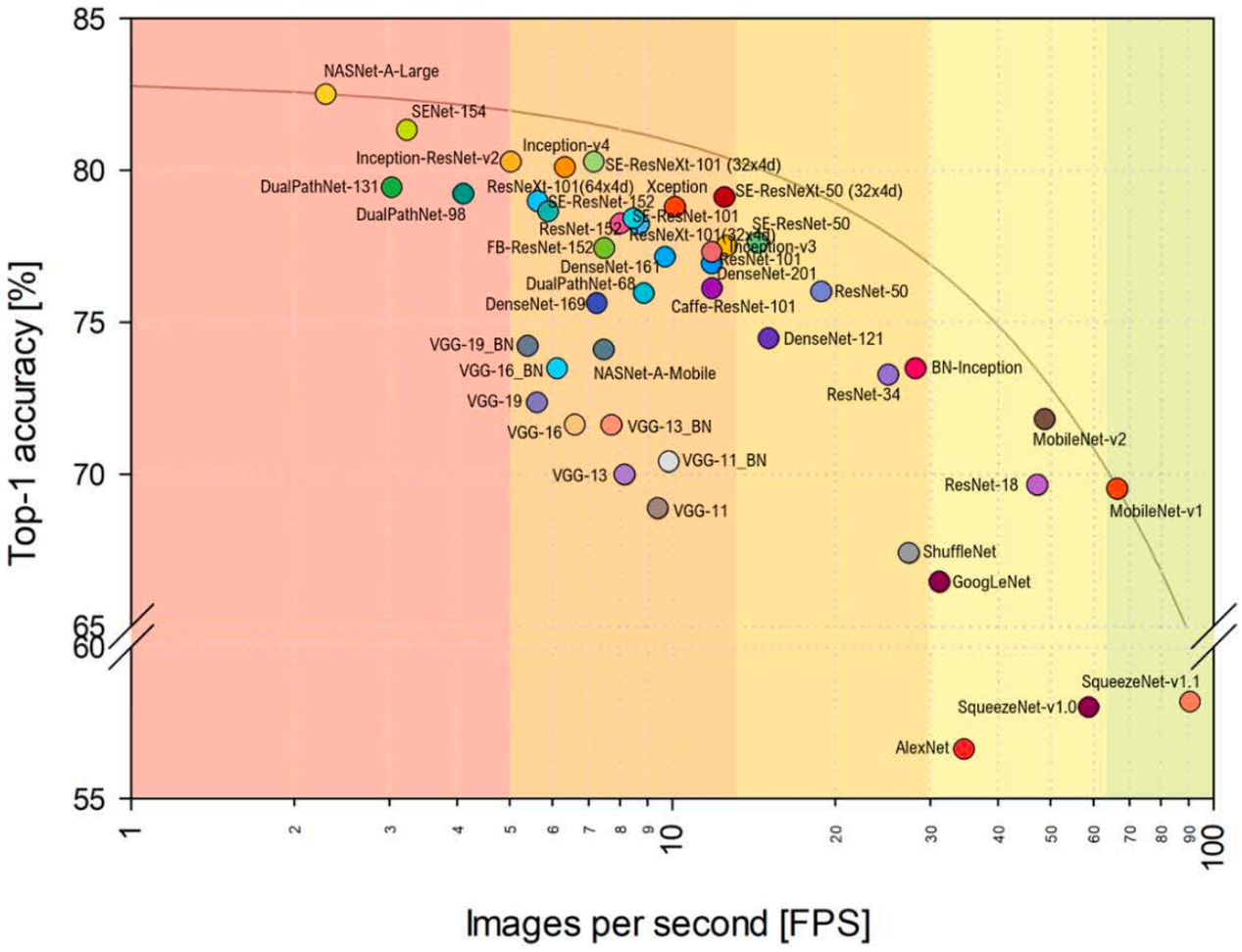
Bianco, Simone, et al. "Benchmark analysis of representative deep neural network architectures." IEEE access 6 (2018): 64270-64277.

Titan Xp

Jetson TX1



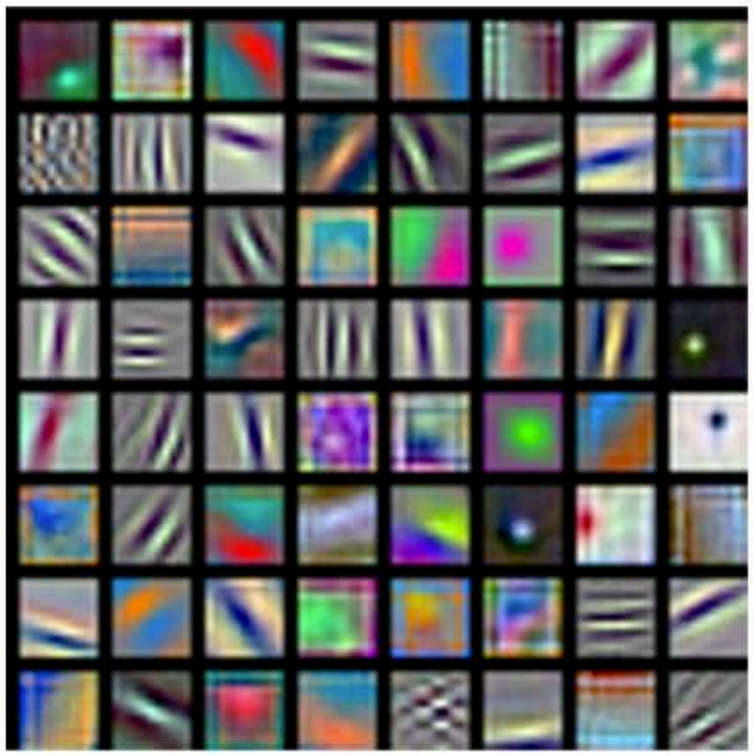




Bianco, Simone, et al. "Benchmark analysis of representative deep neural network architectures." IEEE access 6 (2018): 64270-64277.

Understanding CNN - Filter Visualization

Learned weights of the filters at the first layer

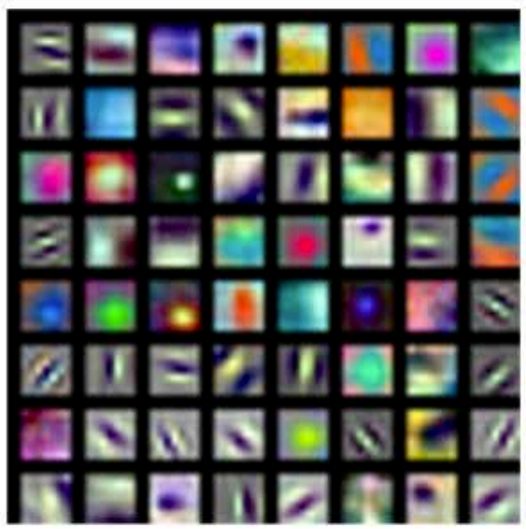


AlexNet:
64 x 3 x 11 x 11

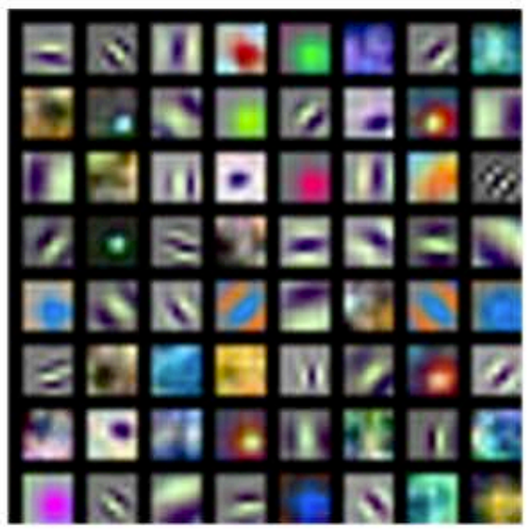
64 filters filter size = 11 x 11
with 3 channels

What these filters look for (through template matching/inner product)
- oriented edges
- opposing colors

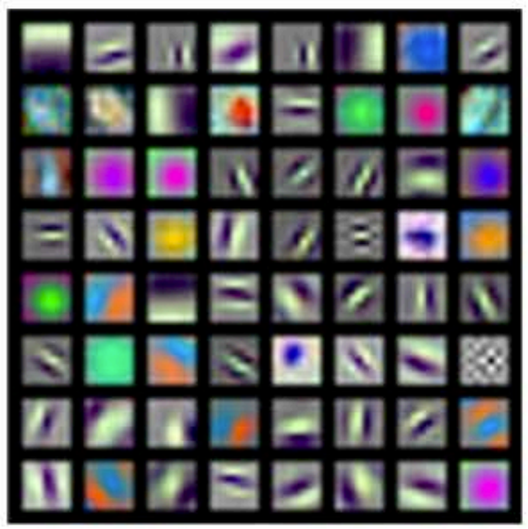
Similar to human visual system!



ResNet-18:
64 x 3 x 7 x 7

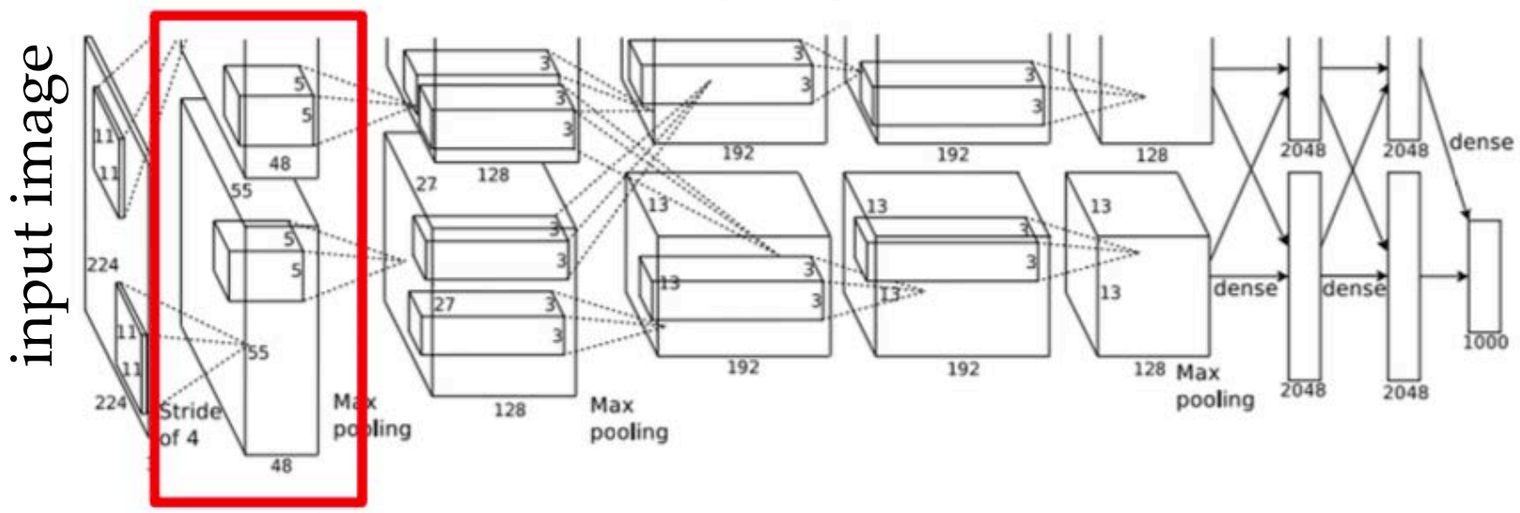


ResNet-101:
64 x 3 x 7 x 7



DenseNet-121:
64 x 3 x 7 x 7

AlexNet



Understanding CNN - Filter Visualization

Visualize the filters/kernels (raw weights)

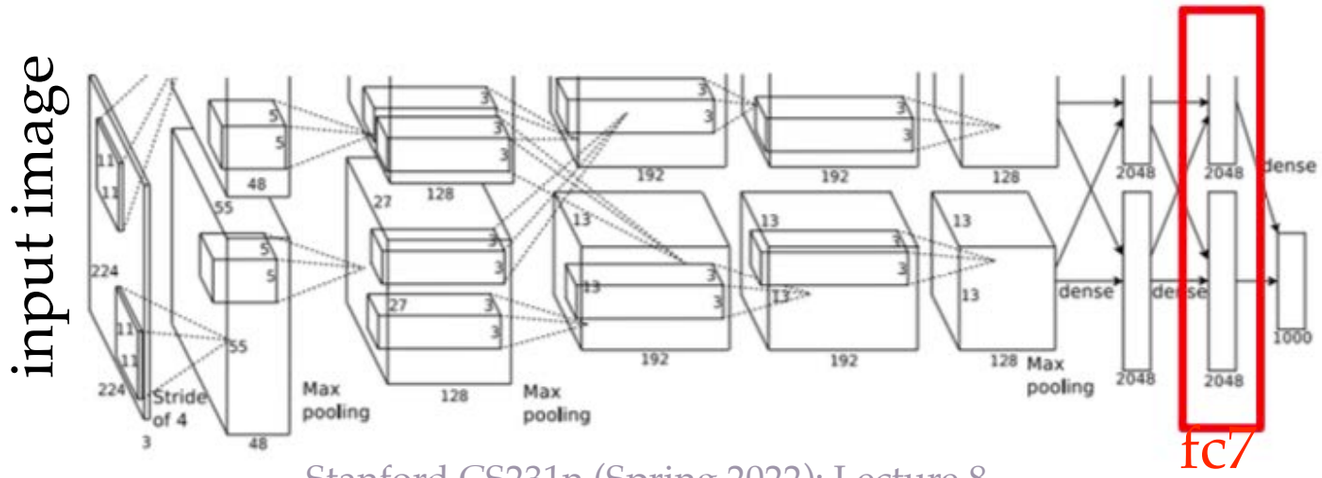
We can visualize filters at higher layers, but not that interesting

(these are taken from ConvNetJS CIFAR-10 demo)

The weights of conv2 tell us what type of the activation patterns after conv1 that would cause conv2 to maximally activate. These filters are not connected directly to the input image, so it is not directly interpretable.

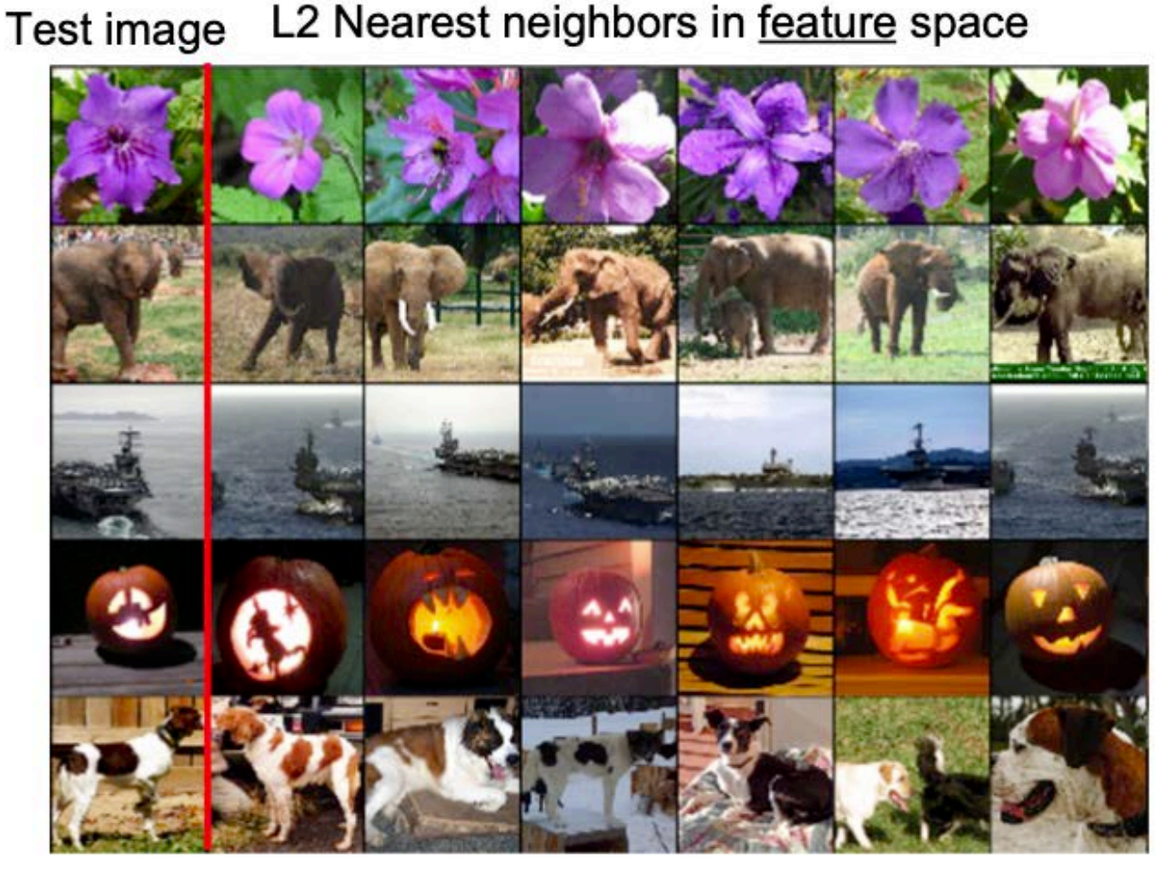
We need more complicated techniques to get a sense of what is going on.

Understanding CNN - Last Layer

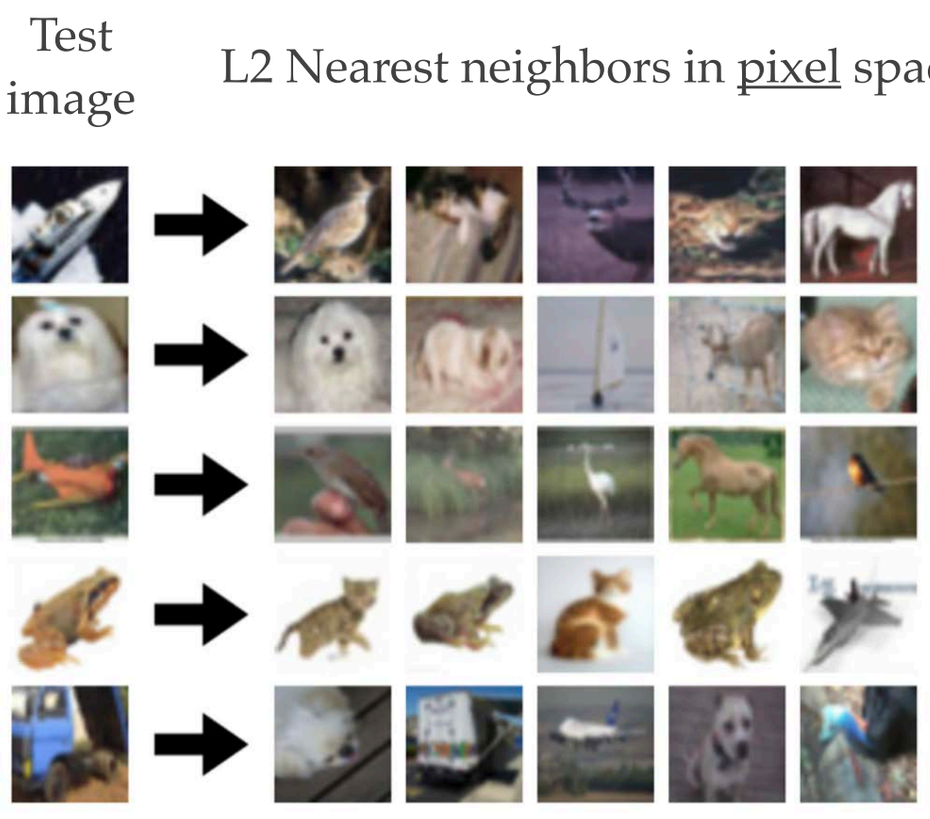


Stanford CS231n (Spring 2022): Lecture 8.

output of fc 7 = feature vector (last layer) of size 4096

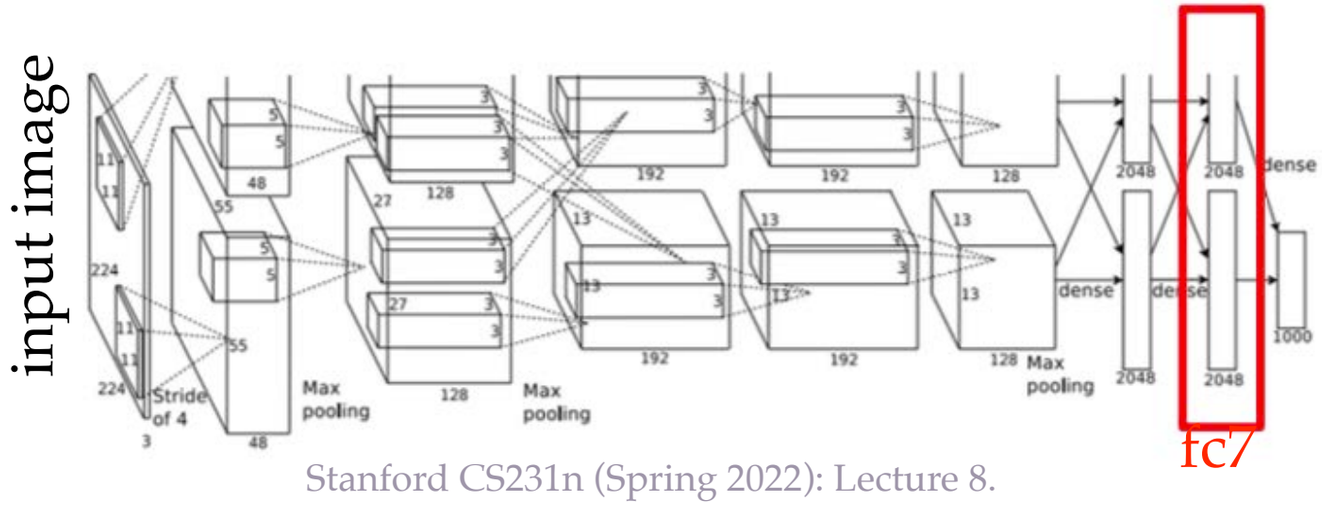


Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012).



Stanford CS231n (Spring 2022): Lecture 8.

Understanding CNN - Last Layer



output of fc 7 = feature vector (last layer) of size 4096

Stanford CS231n (Spring 2022): Lecture 8.

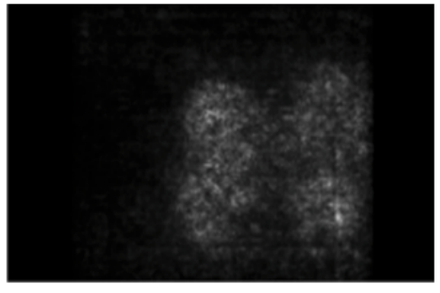
Perform a machine learning algorithm to the output of fc7

Barnes-Hut t-SNE

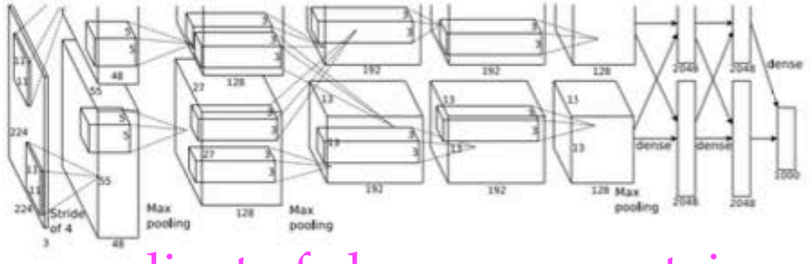
https://cs.stanford.edu/people/karpathy/cnnembed/

Understanding CNN - Other Techniques

Saliency Maps



Forward pass: Compute probabilities



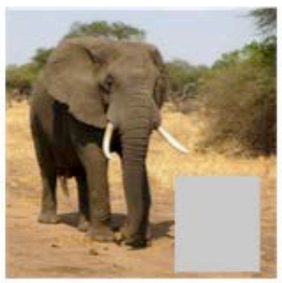
Dog

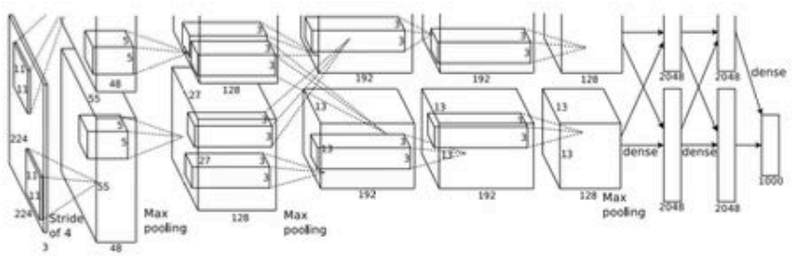
gradient of class score w.r.t. image pixels

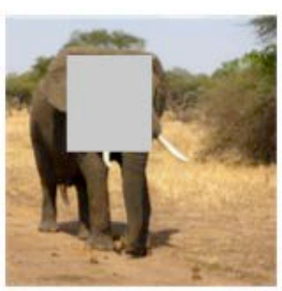
Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman. "Deep inside convolutional networks: Visualising image classification models and saliency maps." arXiv preprint arXiv:1312.6034 (2013).

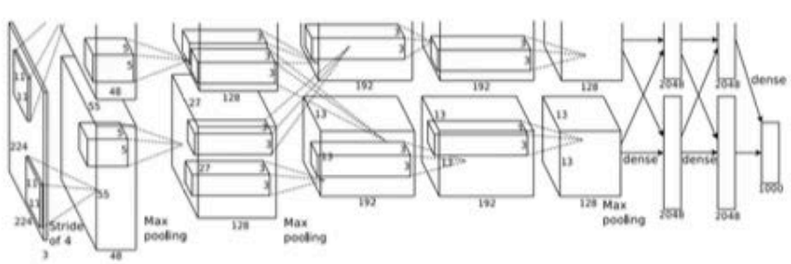
Occlusion Experiments

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



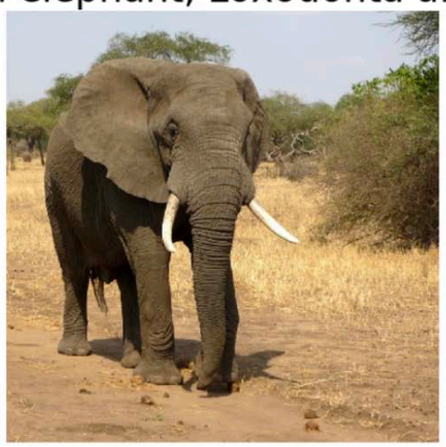


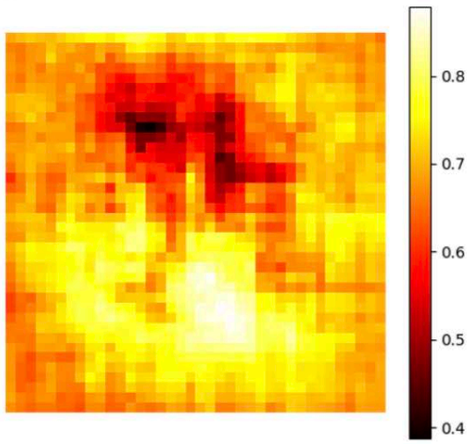




Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." European conference on computer vision. Springer, Cham, 2014.

African elephant, Loxodonta africana

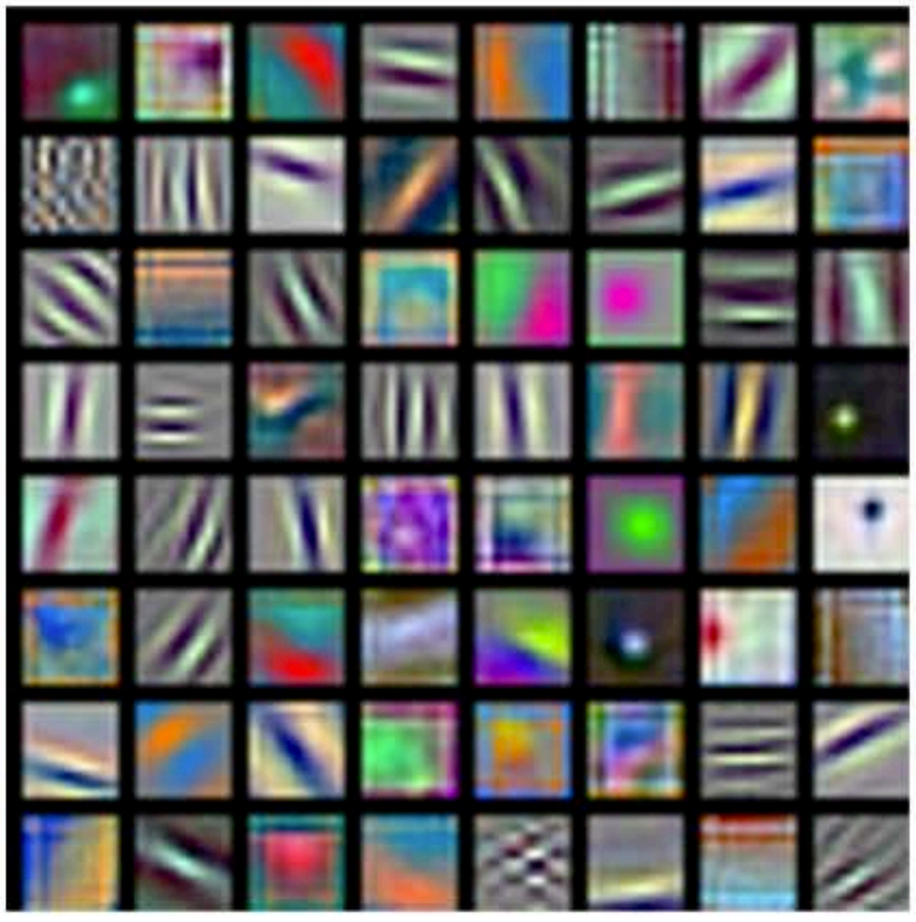




…and many more such as

- Guided Backpropagation
- Grad-CAM, Score-CAM, HiResCAM
- Attention map visualization

Trained CNN as a Feature Extractor



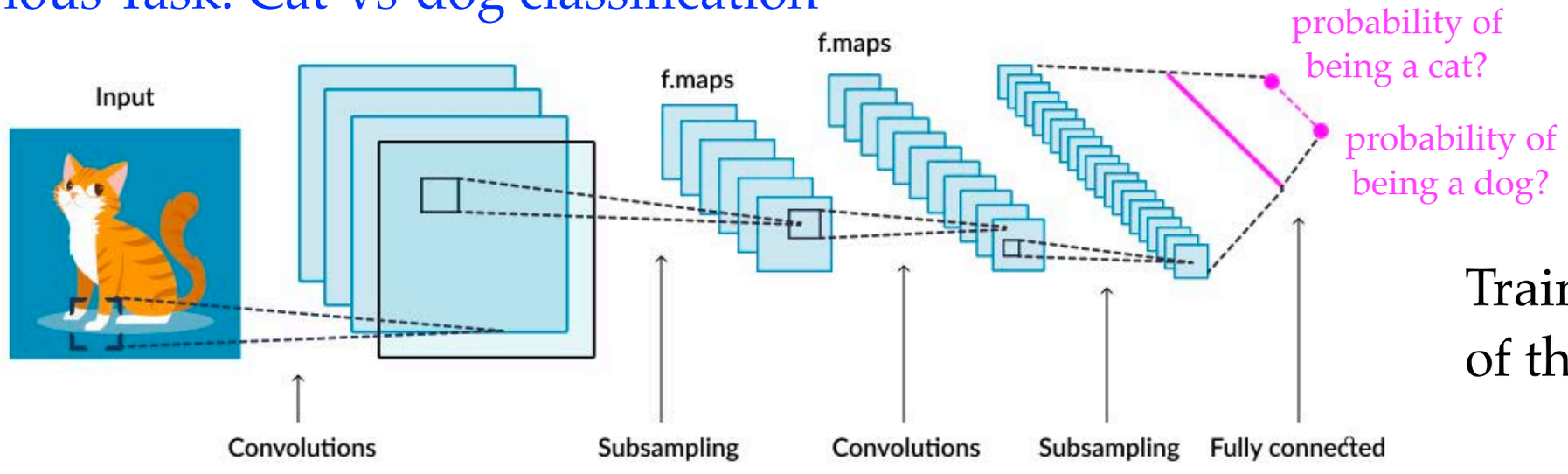
What these filters look for (through template matching/inner product)

- oriented edges
- opposing colors

These patterns are important for processing other types of images as well, so we can reuse trained weights when we encounter a related task or the same task but with a different dataset

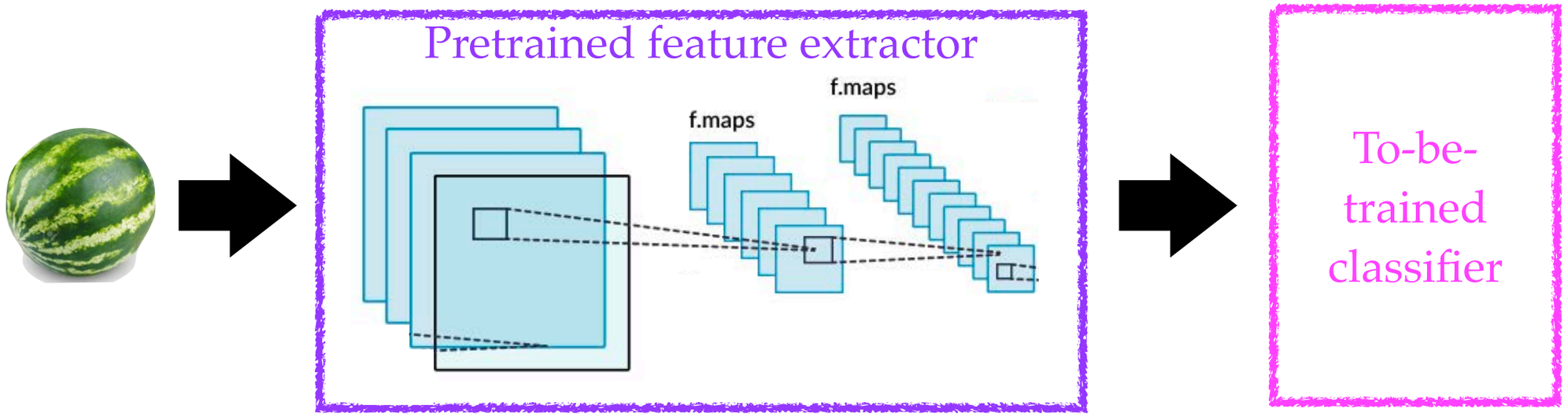
Trained CNN as a Feature Extractor

Previous Task: Cat-vs-dog classification



Train all the weights of this network

New Task: Watermelon-vs-orange classification



Take parts of the trained network and use them as a feature extractor

Only train the weights here

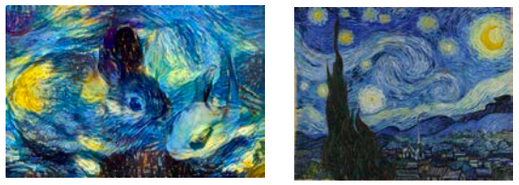
Transfer Learning and Fine-Tuning

ImageNet (1000 classes)

14M images

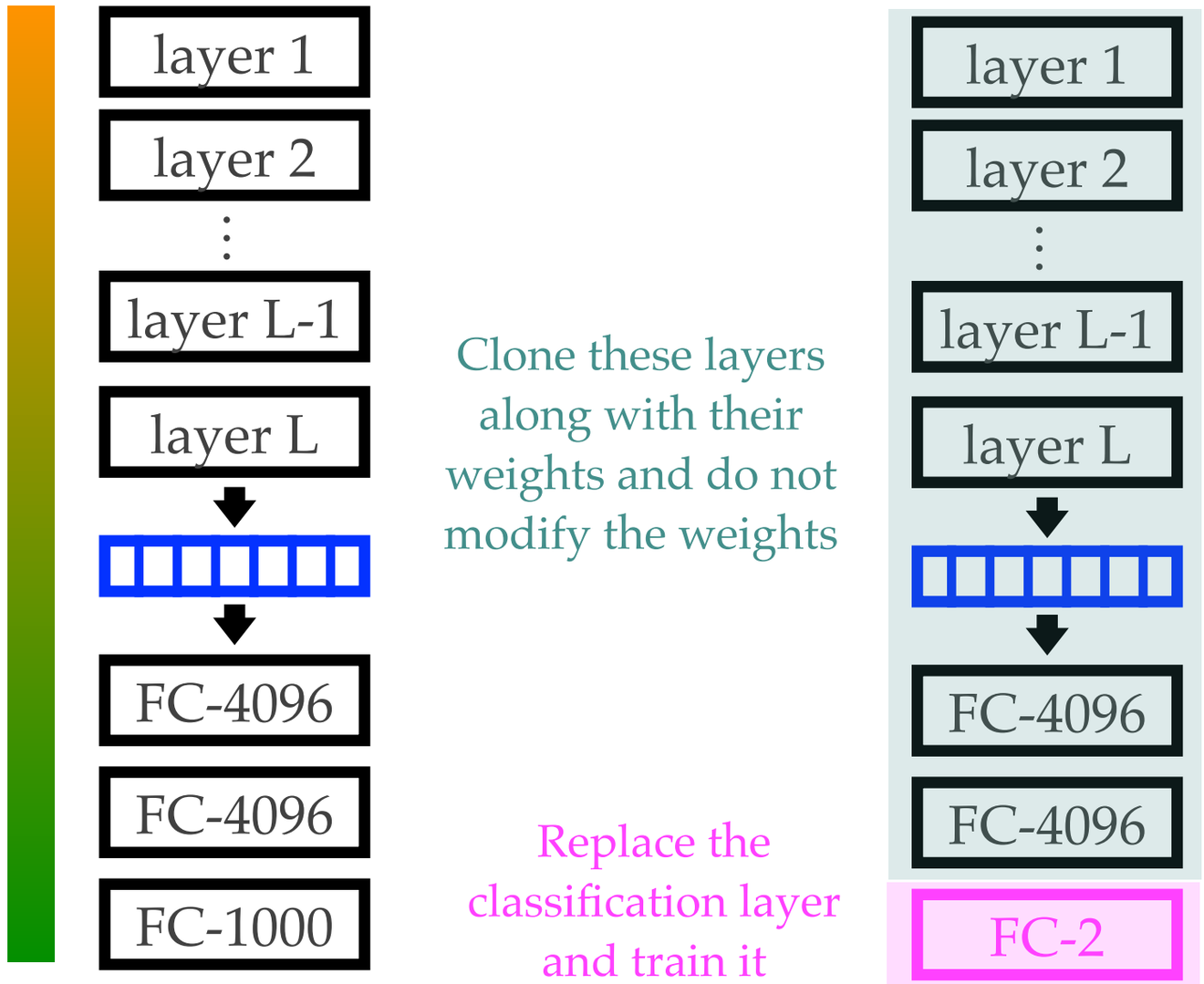
AI-vs-Artists (2 classes)

Assume small dataset



More "reusable"

Less "reusable"



Transfer Learning and Fine-Tuning

ImageNet (1000 classes)

14M images

More "reusable"

layer 1

layer 2

⋮

layer L-1

layer L

FC-4096

FC-4096

FC-1000

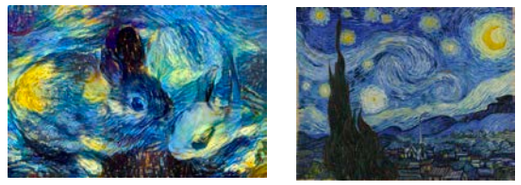
Less "reusable"

Clone these layers along with their weights and do not modify the weights

Replace the classification layer and train it

AI-vs-Artists (2 classes)

Assume small dataset



layer 1

layer 2

⋮

layer L-1

layer L

FC-4096

FC-4096

FC-2

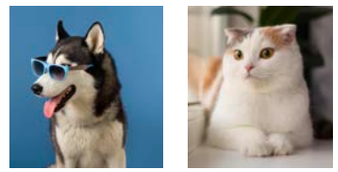
Clone these layers along with their weights and do not modify the weights

Clone these layers along with their weights and retrain them with an initially low learning rate

Replace these layers and train them

Dog-vs-cat (2 classes)

Assume dataset of moderate size



layer 1

layer 2

⋮

layer L-1

layer L

FC-4096

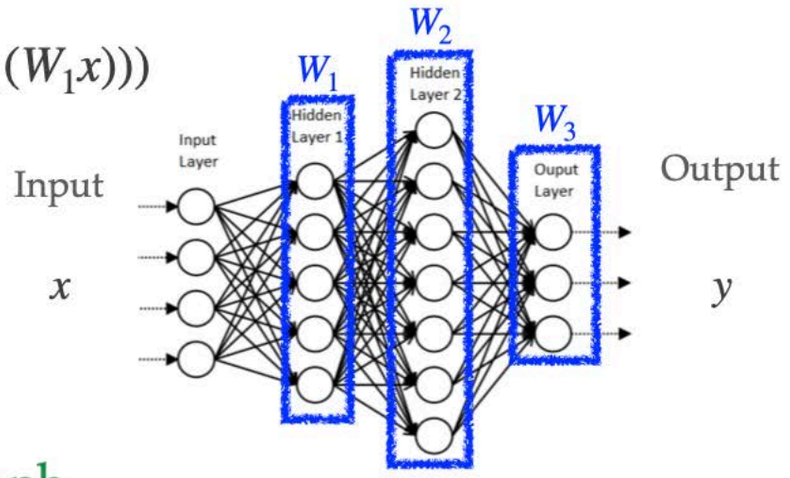
FC-n_filters

FC-2

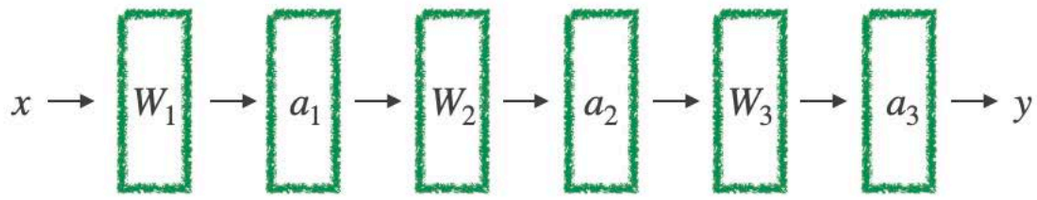
Outline

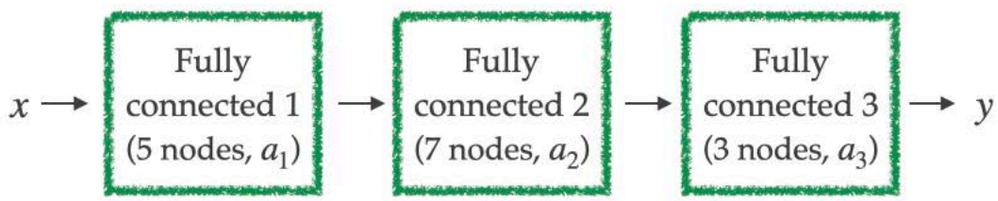
- Deep Learning Components
  - Fully connected layer
  - Computation graph
  - Activation functions
  - Loss function
  - Model optimization
    - Gradient descent
    - Backpropagation
    - Stochastic gradient descent (SGD)
  - Regularization
    - ℓ_2 and ℓ_1 regularization
    - Dropout
    - Batch normalization
  - Convolutional layer
  - Pooling layer
- Convolutional Neural Network (CNN)

$$y = a_3(W_3 a_2(W_2 a_1(W_1 x)))$$

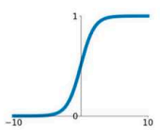


Computation graph

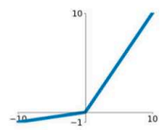




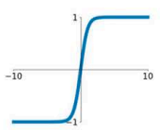
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

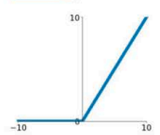


tanh
$\tanh(x)$

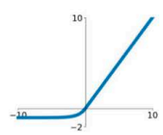


Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$

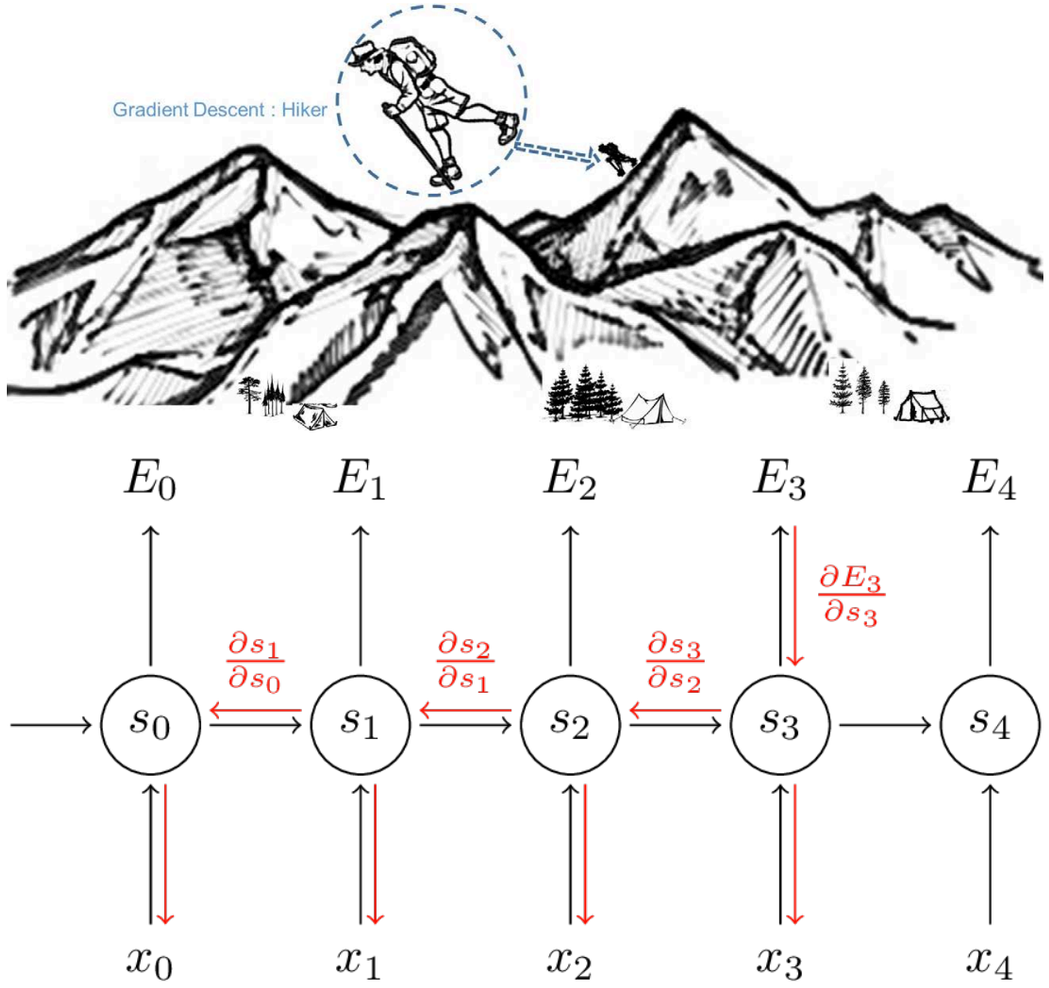


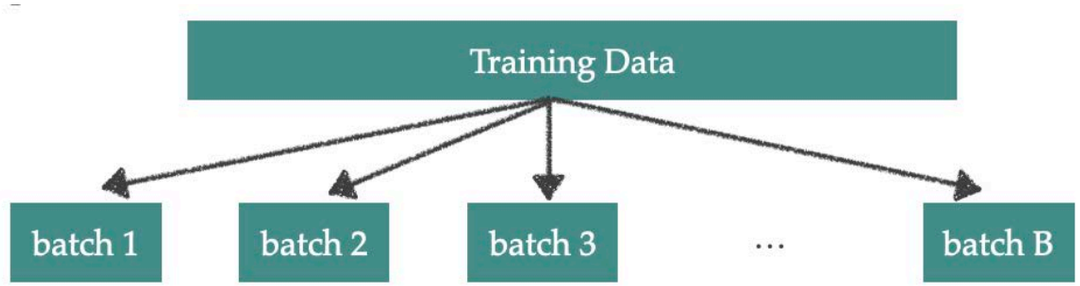
ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



Outline

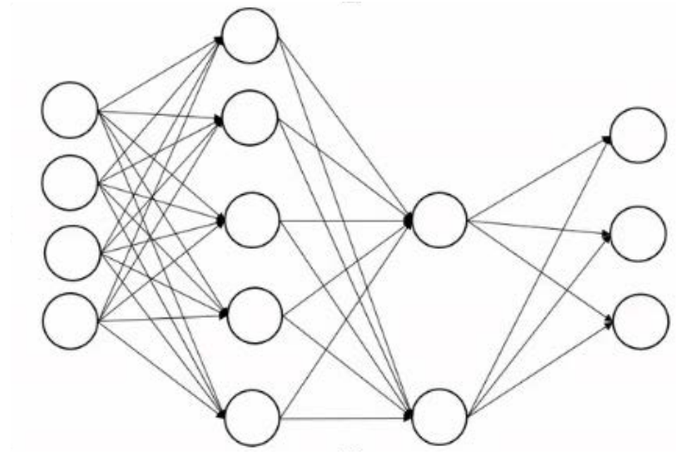
- Deep Learning Components
  - Fully connected layer
  - Computation graph
  - Activation functions
  - Loss function
  - Model optimization
    - Gradient descent
    - Backpropagation
    - Stochastic gradient descent (SGD)
  - Regularization
    - ℓ_2 and ℓ_1 regularization
    - Dropout
    - Batch normalization
  - Convolutional layer
  - Pooling layer
- Convolutional Neural Network (CNN)

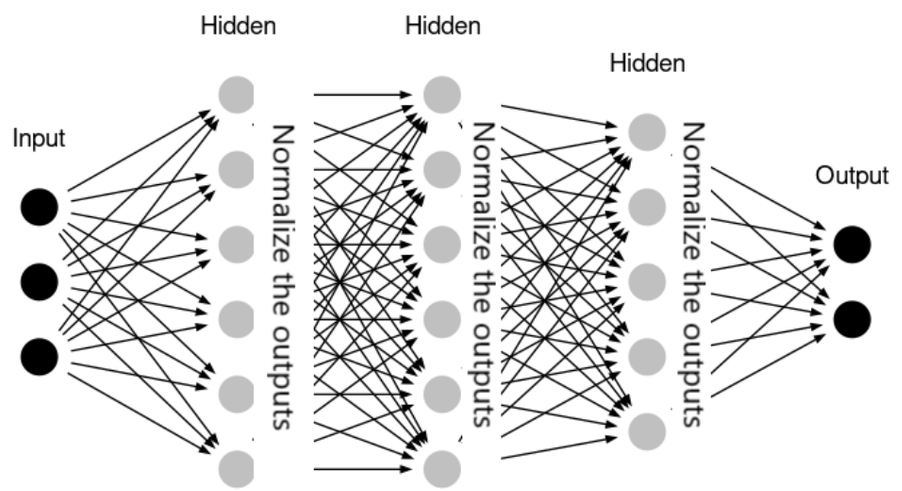


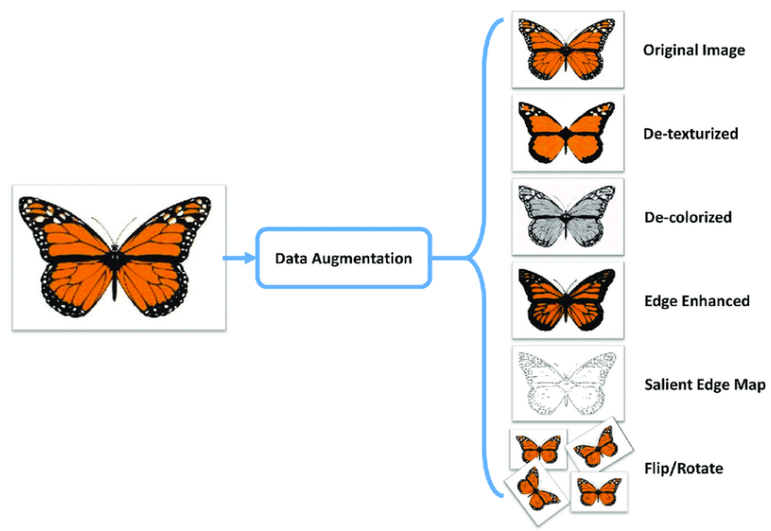


Outline

- Deep Learning Components
  - Fully connected layer
  - Computation graph
  - Activation functions
  - Loss function
  - Model optimization
    - Gradient descent
    - Backpropagation
    - Stochastic gradient descent (SGD)
  - Regularization
    - ℓ_2 and ℓ_1 regularization
    - Dropout
    - Batch normalization
  - Convolutional layer
  - Pooling layer
- Convolutional Neural Network (CNN)

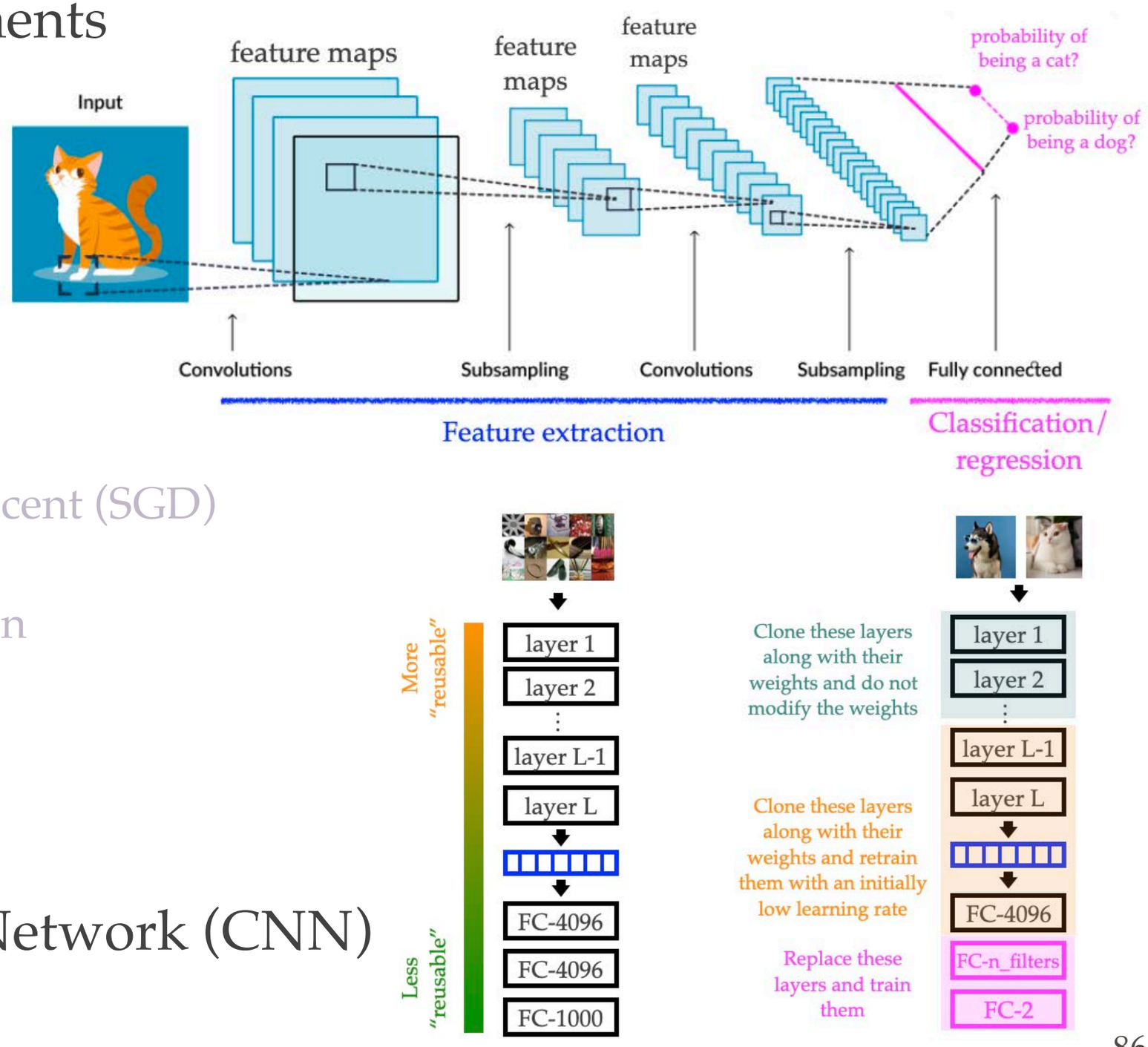




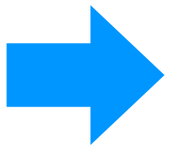


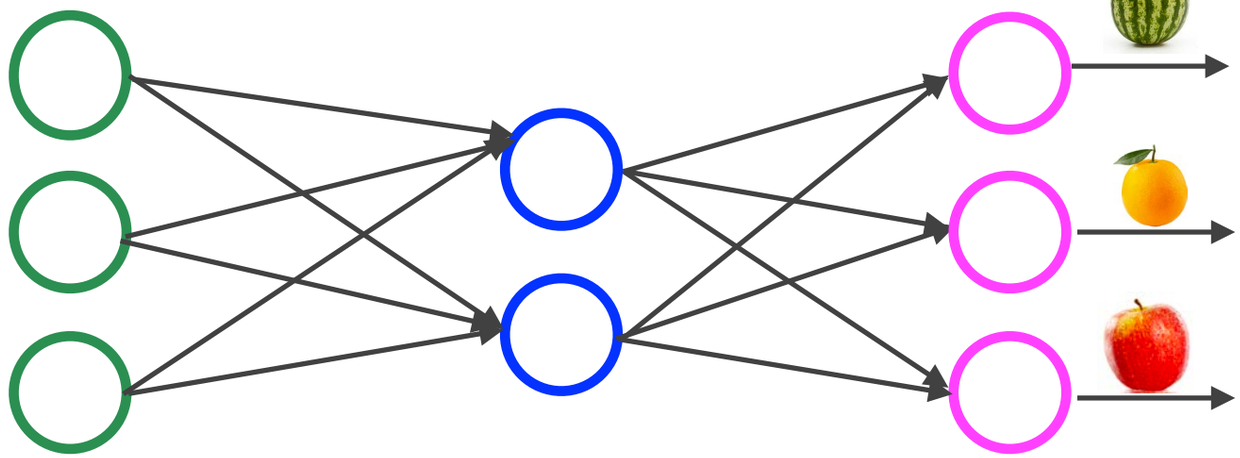
Outline

- Deep Learning Components
  - Fully connected layer
  - Computation graph
  - Activation functions
  - Loss function
  - Model optimization
    - Gradient descent
    - Backpropagation
    - Stochastic gradient descent (SGD)
  - Regularization
    - ℓ_2 and ℓ_1 regularization
    - Dropout
    - Batch normalization
  - Convolutional layer
  - Pooling layer
- Convolutional Neural Network (CNN)



x





y

probability of being class 1

probability of being class 2

probability of being class 3

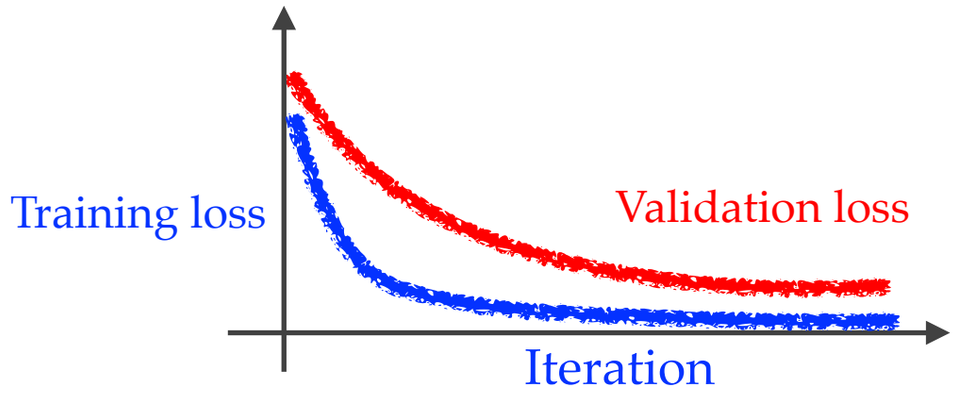
activation functions

ReLU

ReLU

Softmax

```
# Import necessary modules
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Create the model
model = keras.Sequential()
model.add(layers.Dense(3, activation="relu"))
model.add(layers.Dense(2, activation="relu"))
model.add(layers.Dense(3, activation="softmax"))

# Compile the model
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.CategoricalCrossentropy(),
)

# Train the model for 100 epochs with a batch size of 32
model.fit(x_train, y_train, batch_size=32, epochs=100, validation_data=(x_val,y_val))
```



Model

Loss function and optimizer