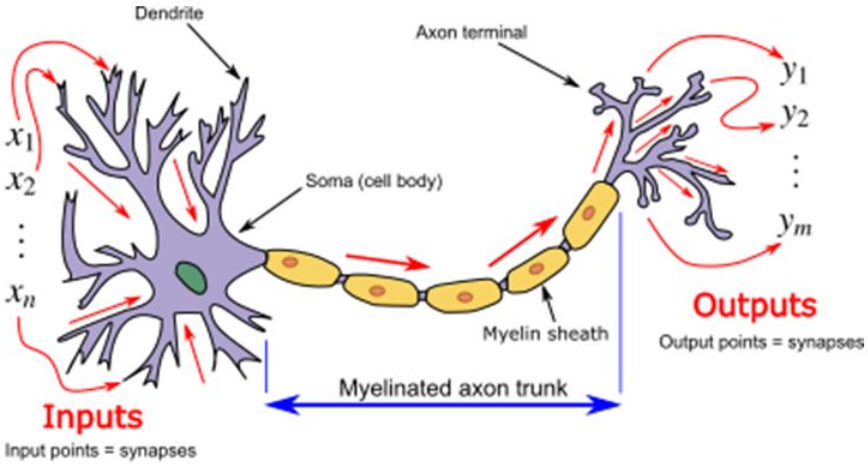
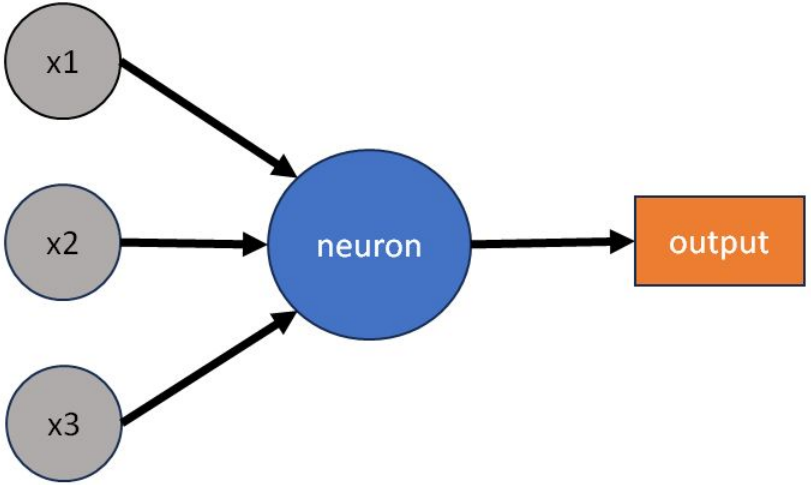
Neuron

- a single neuron is loosely based on the function of a brain neuron
- when presented with n inputs, it decides whether to send a signal (fire)
- this makes it similar to a binary classifier
- in the brain, x and output are electrical signals. In ML, they are real numbers



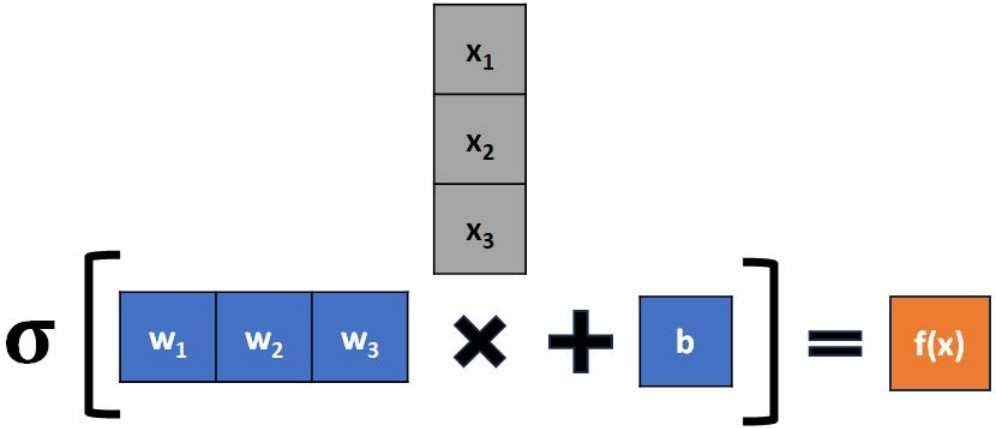


https://en.wikipedia.org/wiki/Neural_network_(machine_learning)#/media/File:Neuron3.png

Neuron: implementation

- a single neuron is just a linear function wrapped in a step-like function
  $f: \mathbb{R}^n \to \mathbb{R}$, $f(x) = \sigma(wx + b)$

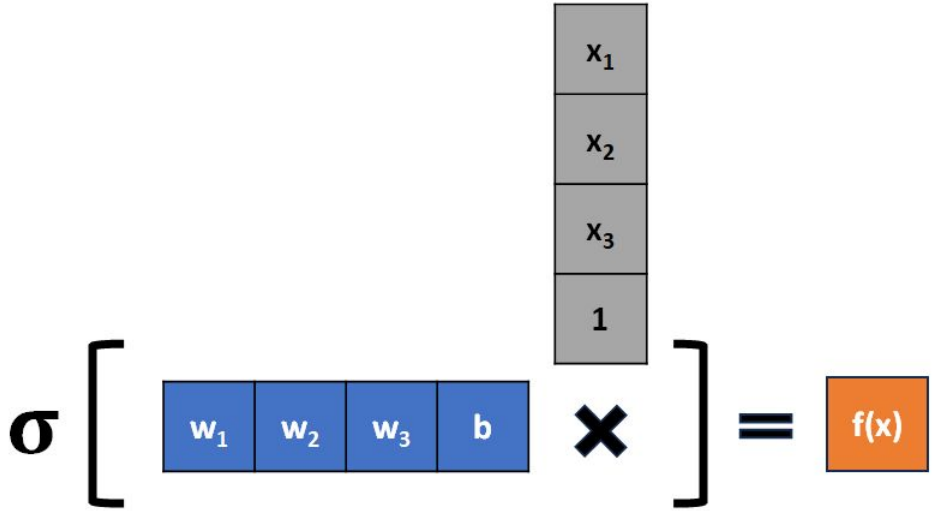
$x \in \mathbb{R}^n$ input
$w \in \mathbb{R}^n$ weights
$b \in \mathbb{R}$ bias
$\sigma: \mathbb{R} \to \mathbb{R}$ activation function



Neuron: weights and biases

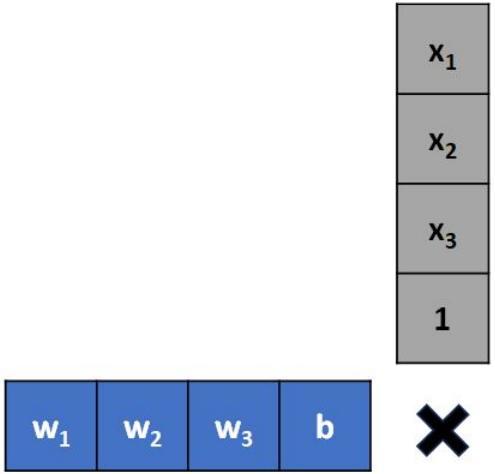
- a neuron has n+1 trainable parameters: one parameter for each input + a bias
- the bias can be viewed as a weight to an input that's always equal to one

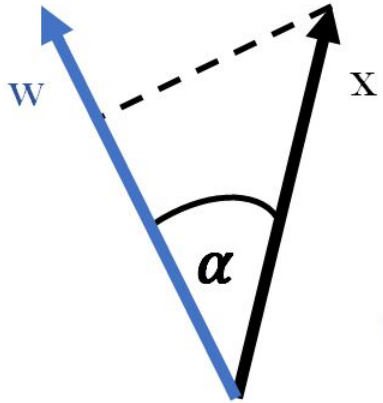
$x \in \mathbb{R}^n$	input
$w \in \mathbb{R}^n$	weights
$b \in \mathbb{R}$	bias
$\sigma: \mathbb{R} \to \mathbb{R}$	activation function



Neuron: weights and biases

- dot product wx is fast and easy to parallelize
- it has a nice intepretation: the neuron detects direction w
- the larger the component of the input x in that direction is, the larger the output





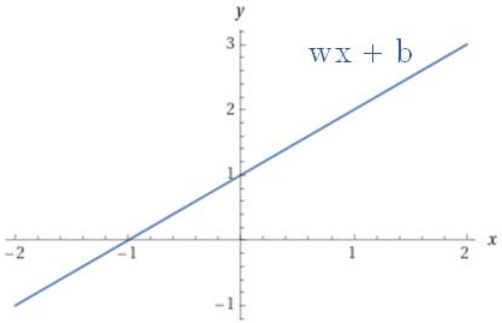
$$wx = |w|\,|x|\cos(\alpha)$$

Neuron: activation functions

- $xw + b$ is a linear function
- $\sigma(xw + b)$ turns it into a step-like function
- a decision boundary is created: $xw > b \rightarrow \text{output} > 0$
- a network of neurons will behave like a complex decision process

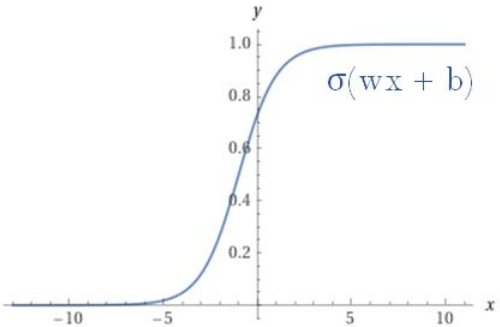
identity

$\sigma(z) = z$



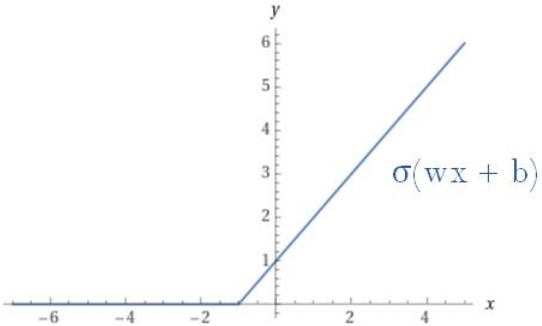
sigmoid

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



ReLU

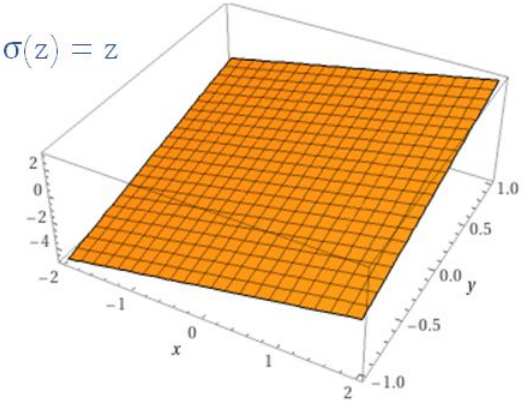
$\sigma(z) = \max(0, z)$



Neuron: activation functions

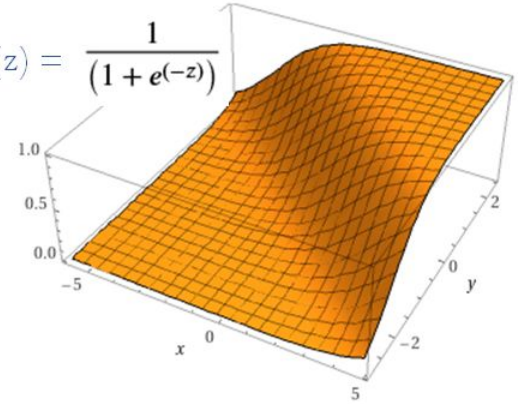
3D plot

$\sigma(z) = z$



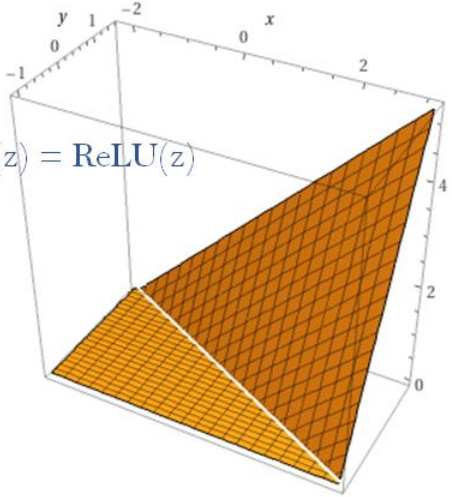
3D plot

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

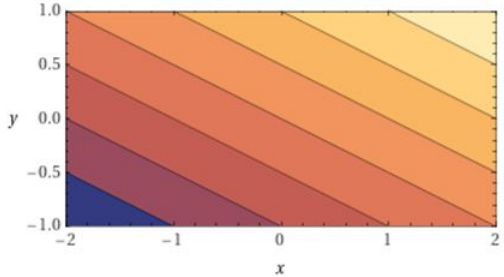


3D plot

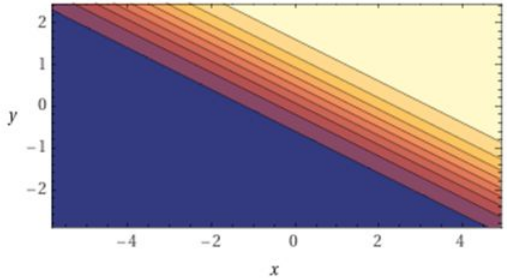
$\sigma(z) = \mathrm{ReLU}(z)$



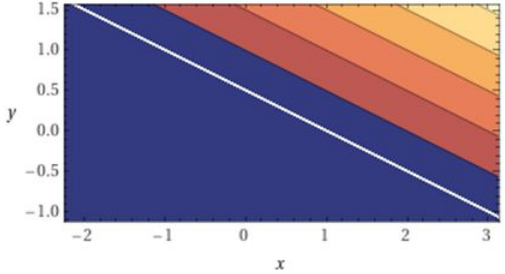
Contour plot



Contour plot

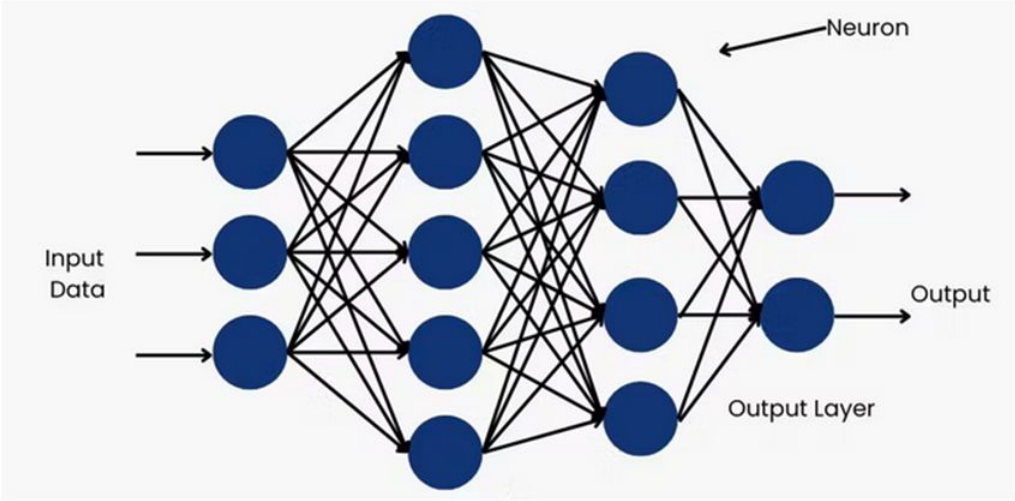


Contour plot



Multilayer perceptron

- neurons are arranged in layers
- output of neurons from the previous layer is fed to the next
- network on the picture has 64 parameters
- real deep networks have millions of them
- deeper neurons learn more complex features



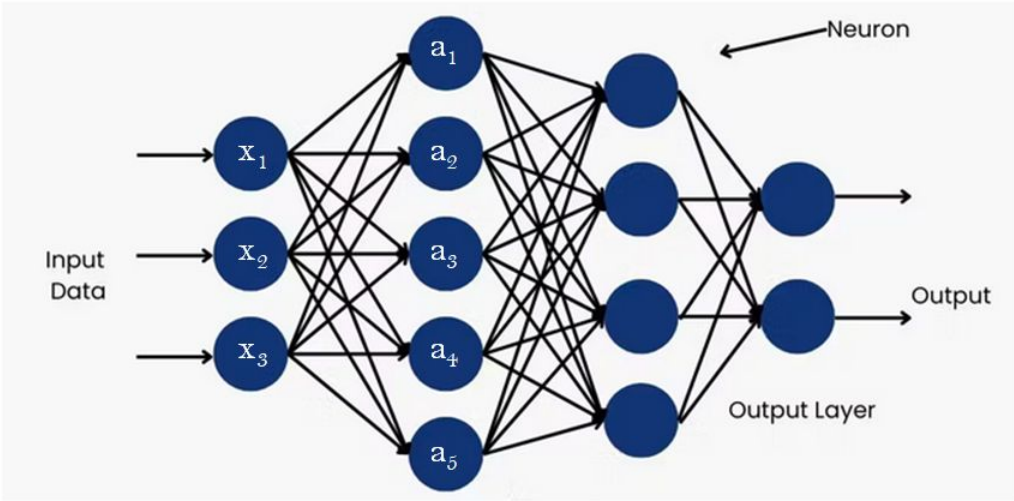
Multilayer perceptron

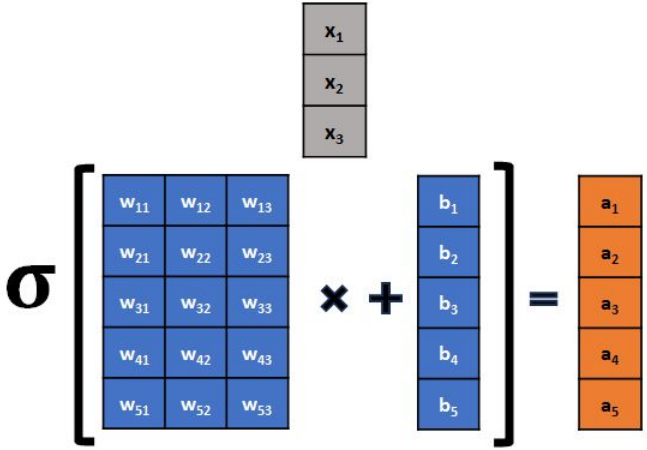
Equations for a single layer can be written in a matrix form

$$a_1 = \sigma(w_{1,1}x_1 + w_{2,1}x_2 + w_{3,1}x_3 + b_1)$$
$$a_2 = \sigma(w_{1,2}x_1 + w_{2,2}x_2 + w_{3,2}x_3 + b_2)$$
$$a_3 = \sigma(w_{1,3}x_1 + w_{2,3}x_2 + w_{3,3}x_3 + b_3)$$
$$a_4 = \sigma(w_{1,4}x_1 + w_{2,4}x_2 + w_{3,4}x_3 + b_4)$$
$$a_5 = \sigma(w_{1,5}x_1 + w_{2,5}x_2 + w_{3,5}x_3 + b_5)$$

$$\downarrow$$

$$a = \sigma(Wx + b)$$





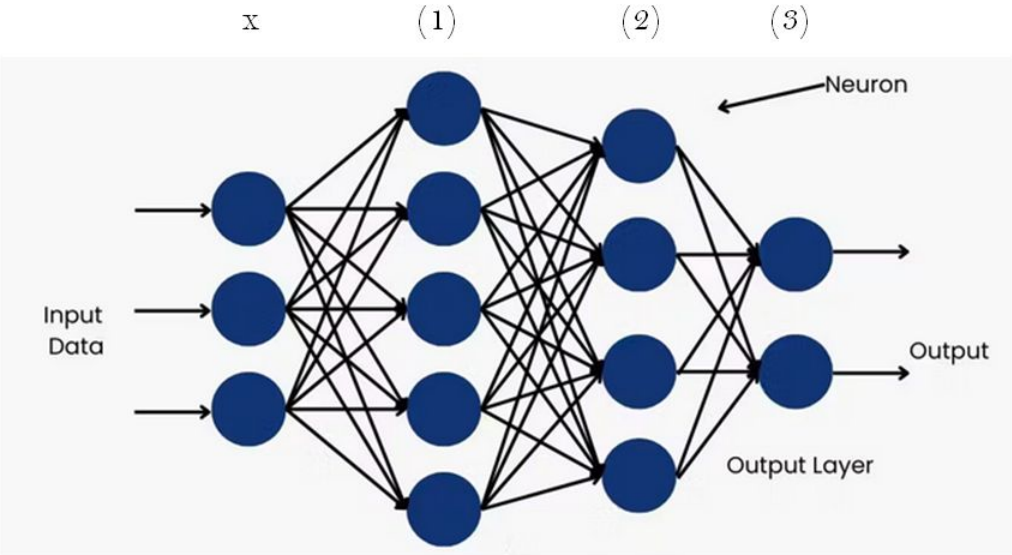
https://www.turing.com/kb/explanation-of-deep-neural-network-multilayer-perceptron-deep-q-network

Many inputs at once

$$\sigma\left[\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{bmatrix} \times \begin{bmatrix} x_{11} & x_{21} & \\ x_{12} & x_{22} & \\ x_{13} & x_{23} & \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}\right] = \begin{bmatrix} a_{11} & \\ a_{12} & \\ a_{13} & \\ a_{14} & \\ a_{15} & \end{bmatrix}$$

A network is a series of simple operations

$$x \rightarrow \sigma(W^{(1)} \cdot + b^{(1)}) \rightarrow \sigma(W^{(2)} \cdot + b^{(2)}) \rightarrow W^{(3)} \cdot + b^{(3)}$$




```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class SmolMultilayerPerceptron(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(3, 5)
        self.fc2 = nn.Linear(5, 4)
        self.fc3 = nn.Linear(4, 2)

    def forward(self, x):
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x

model = SmolMultilayerPerceptron()
x = torch.FloatTensor([0.1, 0.2, -0.3])
print(model(x).detach().numpy())
```

✓ 3.1s

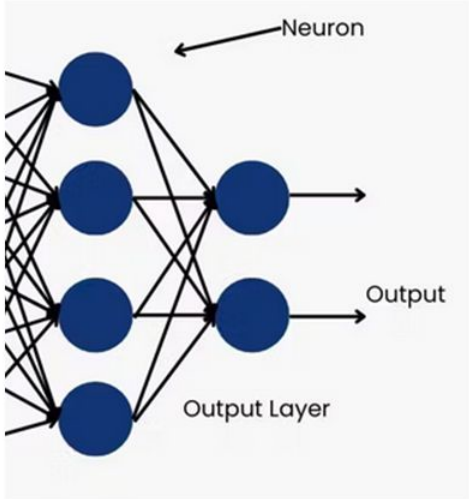
```
[-0.12244508 -0.19386089]
```

https://www.turing.com/kb/explanation-of-deep-neural-network-multilayer-perceptron-deep-q-network

Network output

$$\longrightarrow \sigma(W^{(2)} \cdot + b^{(2)}) \longrightarrow W^{(3)} \cdot + b^{(3)}$$

(2) (3)



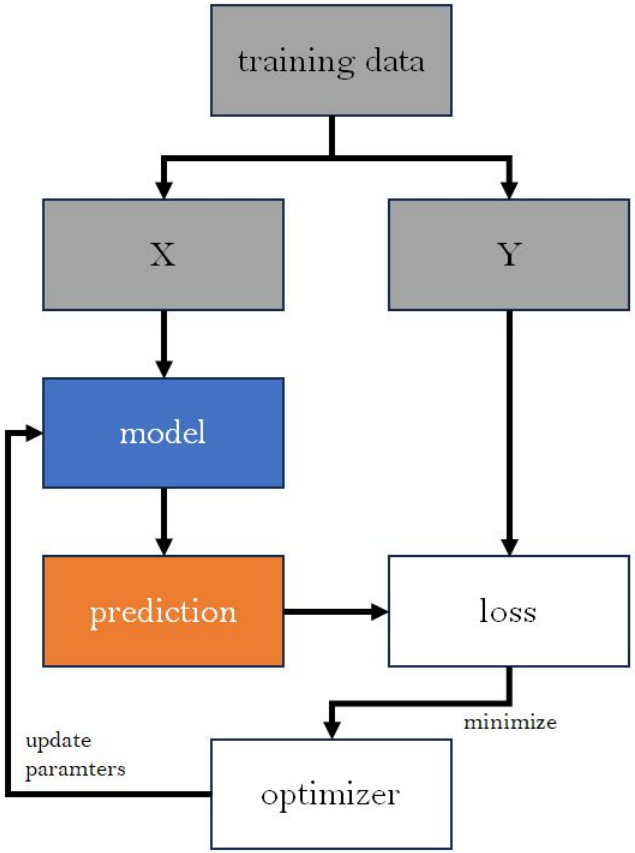
- the last layer usually doesn't have activations (it returns values from –inf to inf)
- for regression, we set the same number of outputs as the values we're trying to predict
- binary classification: one output represents the result (negative = the first class)
- non-binary classification: one output represents a single class
- if we want to obtain probabilities of classes, we apply softmax to the output layer

$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

https://www.turing.com/kb/explanation-of-deep-neural-network-multilayer-perceptron-deep-q-network

Training neural networks

- we want to use a neural network to predict something
- the network operates on numbers: all input must be turned into numbers, and the output will also be numbers
- a training set is used to teach the intended relationship between the input and the output
- the difference between the current output and the intended output is measured with a loss function
- optimization procedure is used to find paramters that minimize the loss



Training neural networks: loss

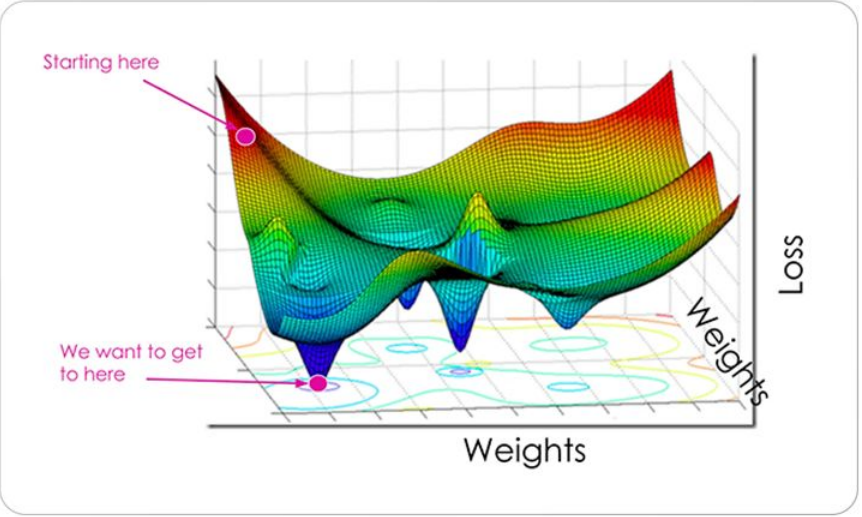
- loss measures the difference between the model's output and the intended output, creating a goal for the optimization procedure
- different loss functions are used for different problems

regression		
Mean Absolute Error (MAE)	Mean Squared Error (MSE)	Root Mean Squared Error (RMSE)
$\frac{1}{n}\sum\lvert Y-\hat{Y}\rvert$	$\frac{1}{n}\sum(Y-\hat{Y})^2$	$\sqrt{\frac{1}{n}\sum(Y-\hat{Y})^2}$

classification
Cross-entropy (a.k.a. negative log-likelihood or log loss)
$H(y,\hat{y}) = -\sum_{c\in Classes} y_c log\ \hat{y}_c$ Where $\hat{y}$ are predicted probabilities

Training is an optimization problem

- training looks for parameters w^* that minimize the loss function
- problem: w is high-dimensional and non-convex: we don't have an analytical solution for finding the global minimum
- even a simple grid search (k values of each parameter) leads to too many combinations
- solution: start from a random initialization and try to improve iteratively



Optimization: how to know where to go

- at step i, the parameters are in the point w_i. What step to take?
- we can calculate some of the loss'es derivatives in w_i analytically

$$\mathcal{L}(w_i + s) = \sum_{n=0}^{\infty} \frac{\mathcal{L}^{(n)}(w_i)}{n!} s^n$$

$$\mathcal{L}(w_i + s) \approx \mathcal{L}(w_i) + \nabla\mathcal{L}(w_i)s$$

$$\mathcal{L}(w_i + s) \approx \mathcal{L}(w_i) + \nabla\mathcal{L}(w_i)s + \nabla^2\mathcal{L}(w_i)\frac{s^2}{2}$$

- then, approximate the loss function around w_i with a Taylor expansion
- pick a step that minimizes (or just lowers) this expansion
- methods that use the second derivative (the Hessian) are called second-order methods. They were popular around 2010
- first-order methods (using only the gradient) are more popular now

Optimization: calculating the gradient

- as a network is a series of simple operations, we can calculate the derivatives of those operations…

$$x \rightarrow \sigma(W^{(1)} \cdot + b^{(1)}) \rightarrow \sigma(W^{(2)} \cdot + b^{(2)}) \rightarrow W^{(3)} \cdot + b^{(3)} \rightarrow \frac{1}{n} \sum (Y - \cdot)^2$$

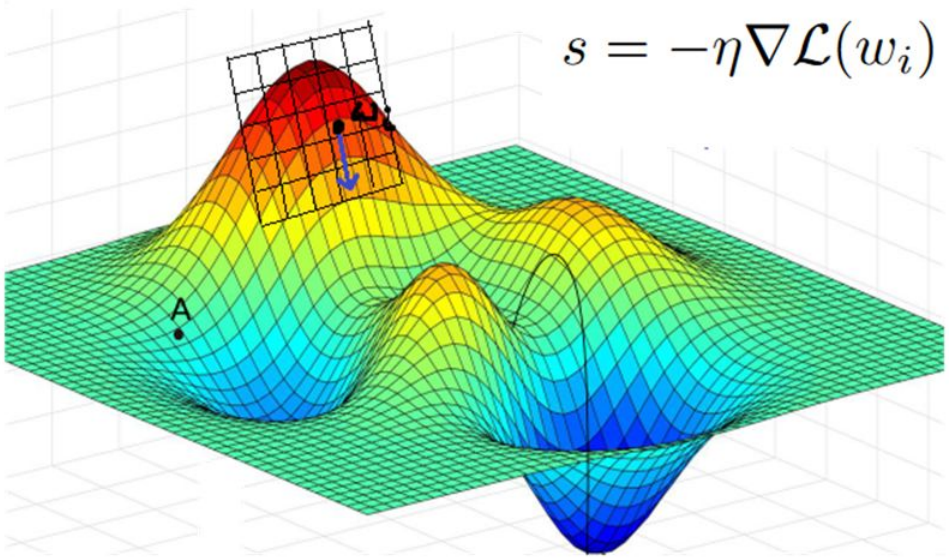
$$\frac{\partial}{\partial \cdot} \qquad W^{(1)} \sigma'(W^{(1)} \cdot + b^{(1)}) \rightarrow W^{(2)} \sigma'(W^{(2)} \cdot + b^{(2)}) \rightarrow W^{(3)} \rightarrow -\frac{1}{n} \sum 2 \cdot$$

- …and use the chain rule to get partial derivatives of any parameter. Doing this for the first derivative is implemented efficiently in an algorithm called Backpropagation
- we won't go into details here; it's only important that calculating the gradient of the loss in a given point w is as easy as calculating the loss itself
- derivatives of higher order can be calculated too, but take more time

Optimization: Gradient Descent

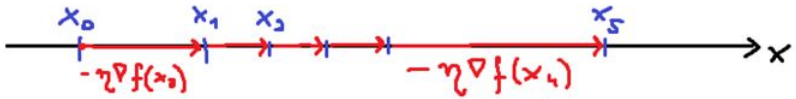
$$\mathcal{L}(w_i + s) \approx \mathcal{L}(w_i) + \nabla\mathcal{L}(w_i)s$$

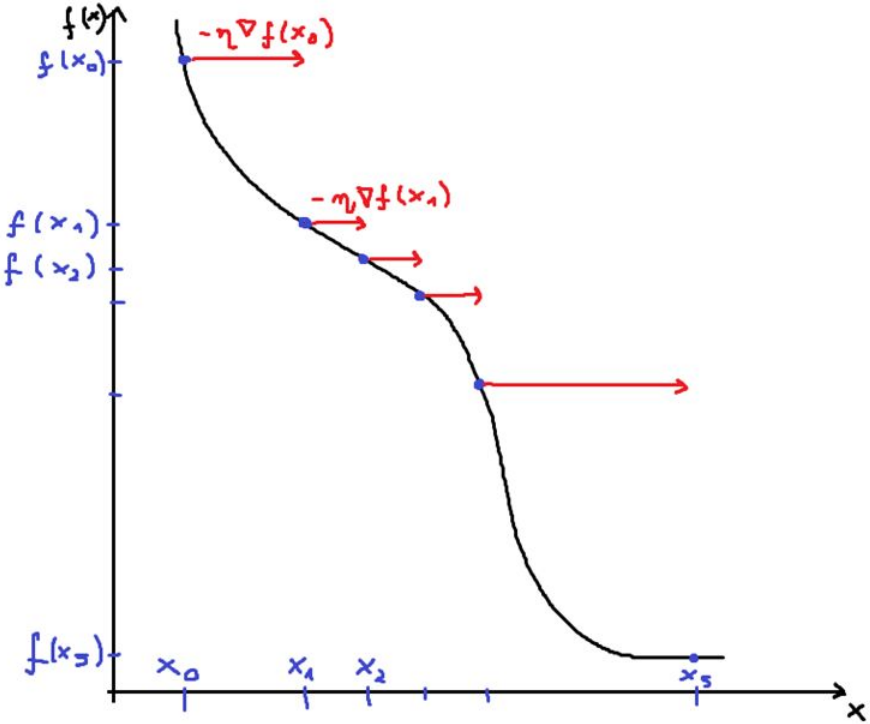
$$s = -\eta\nabla\mathcal{L}(w_i)$$

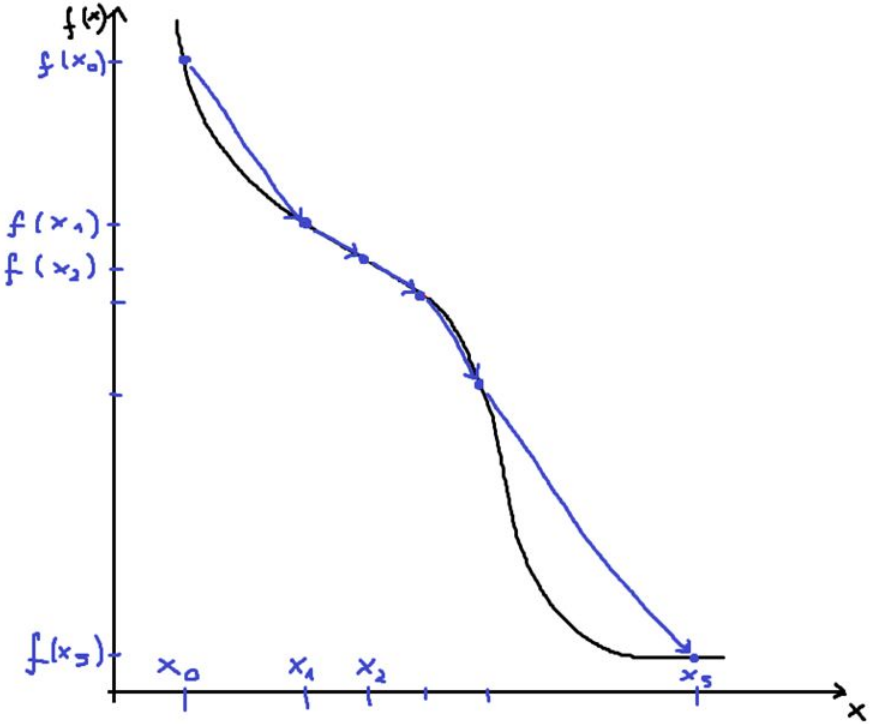


- the „simplest" first-order optimization method
- creates a first-order approximation and picks the direction in which it decreases the fastest (in relation to the step's length)
- because of some properties of quadratic functions, this best step's direction is a vector identical to the gradient
- learning rate is used to control step size

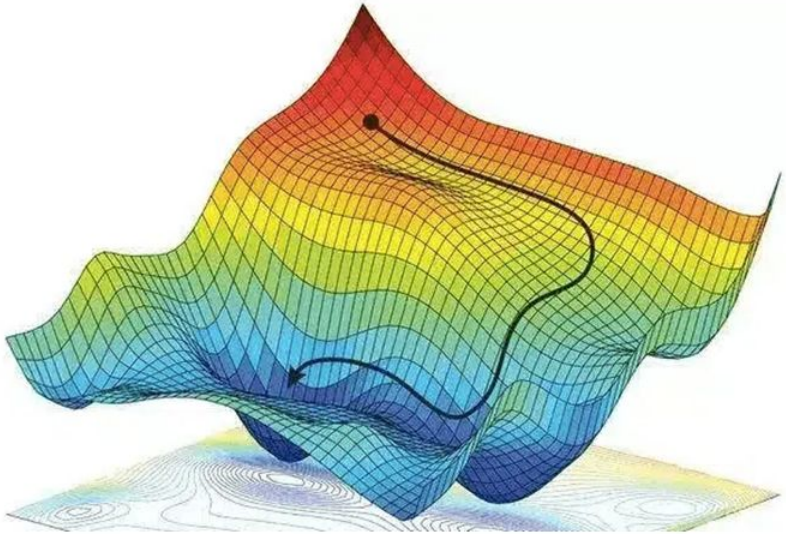
Visualizing GD





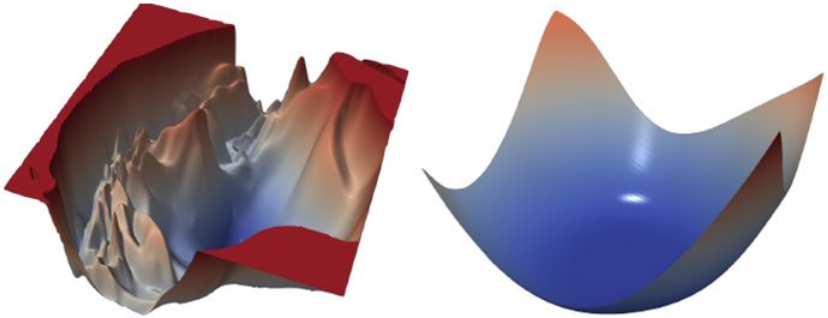


Visualizing GD



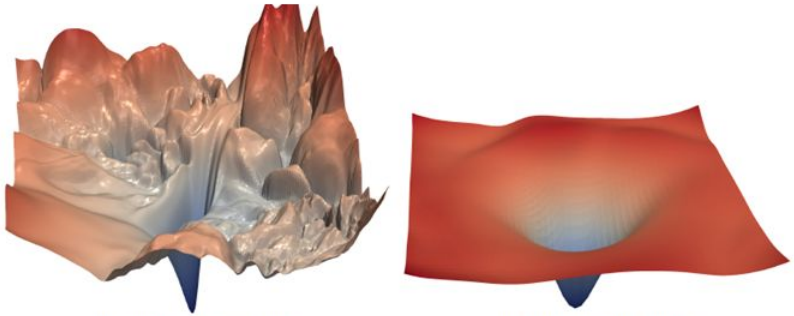
https://easyai.tech/en/ai-definition/gradient-descent/

Difficulty of optimization depends on the loss landscape, which depends on the architecture



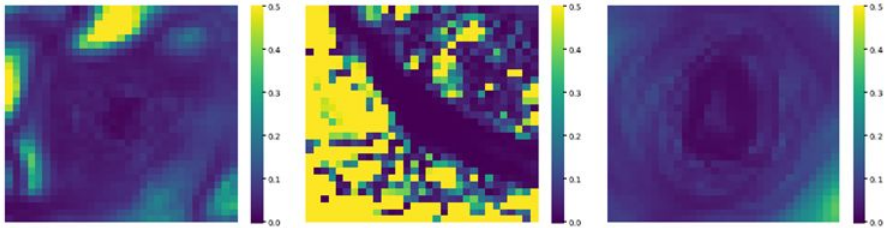
(a) ResNet-110, no skip connections (b) DenseNet, 121 layers

Figure 4: The loss surfaces of ResNet-110-noshort and DenseNet for CIFAR-10.



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.



(a) Resnet-56 (b) Resnet-56-noshort (c) DenseNet-121

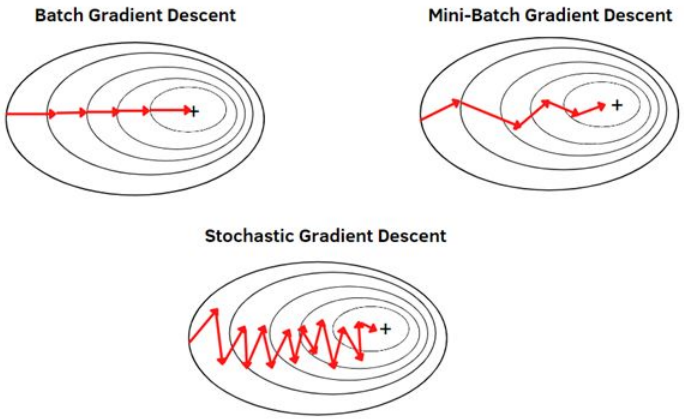
Figure 7: For each point in the filter-normalized surface plots, we calculate the maximum and minimum eigenvalue of the Hessian, and map the ratio of these two.

Visualizing the Loss Landscape of Neural Nets: Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein

Stochastic Gradient Descent

- in practice, the gradient is rarely calculated based on the whole training data
- instead, mini-batches are used to calculate estimates of the gradient
- individual steps have larger variance, but their mean is the same (it's an unbiased estimator)
- instead of taking a single very precise step using 100 samples we take 20 steps with 10 samples each
- this leads to faster convergence and might help with generalization

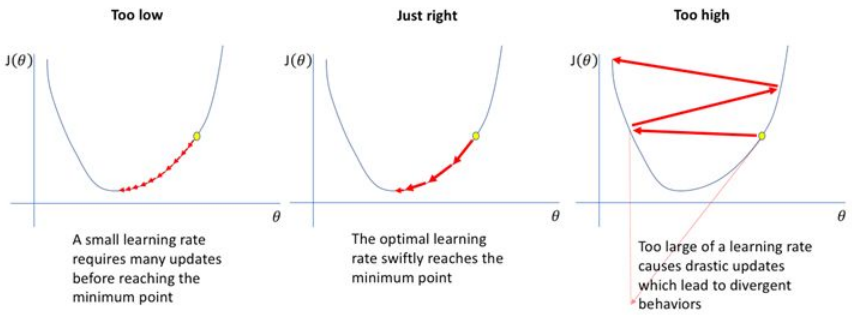
- SGD used to refer to the extreme case where a batch contains only a single sample. Later, the definition relaxed to include mini-batches



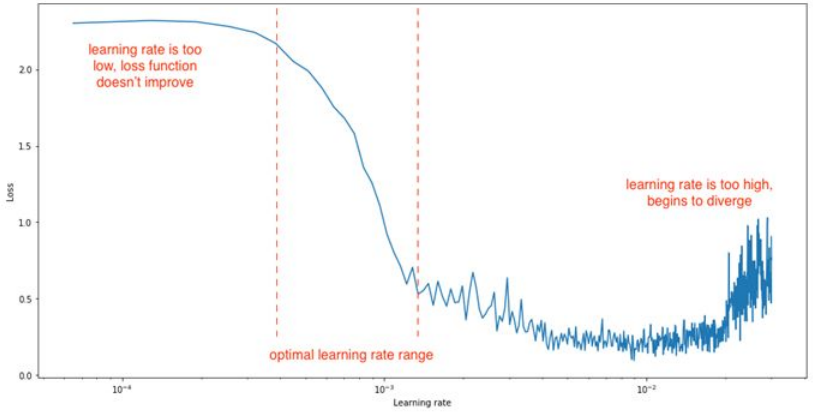
https://ml-explained.com/blog/gradient-descent-explained

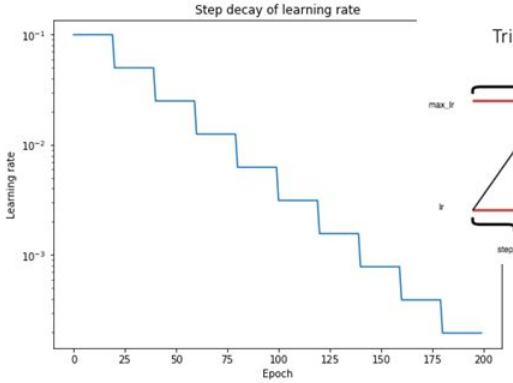
Learning rate

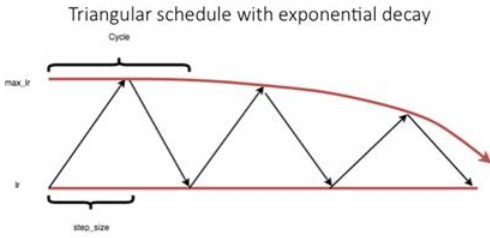
- the size of a single SGD step is controlled with the learning rate
- changing it has a profound effect on the training dynamics



- learning rate scheduling can be used to decrease it as training progresses







Tutorial: training a Multilayer Perceptron

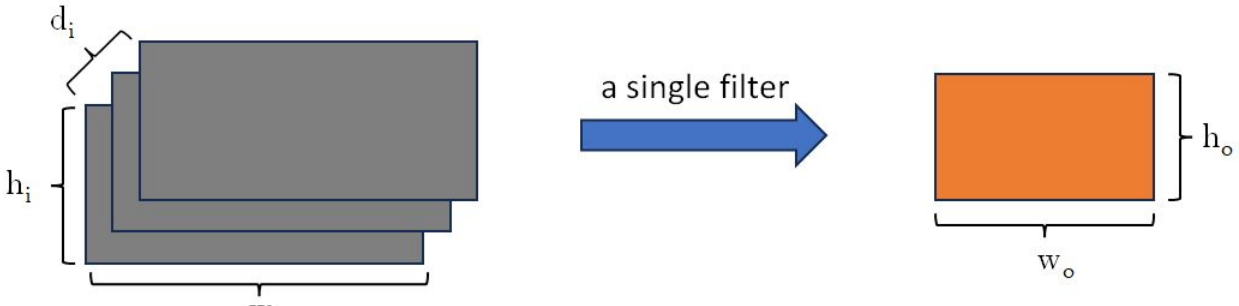
- https://github.com/JakubBilski/introduction-to-machine-learning
- https://api.wandb.ai/links/podcast-o-rybach-warsaw-university-of-technology/2qr5y1lj

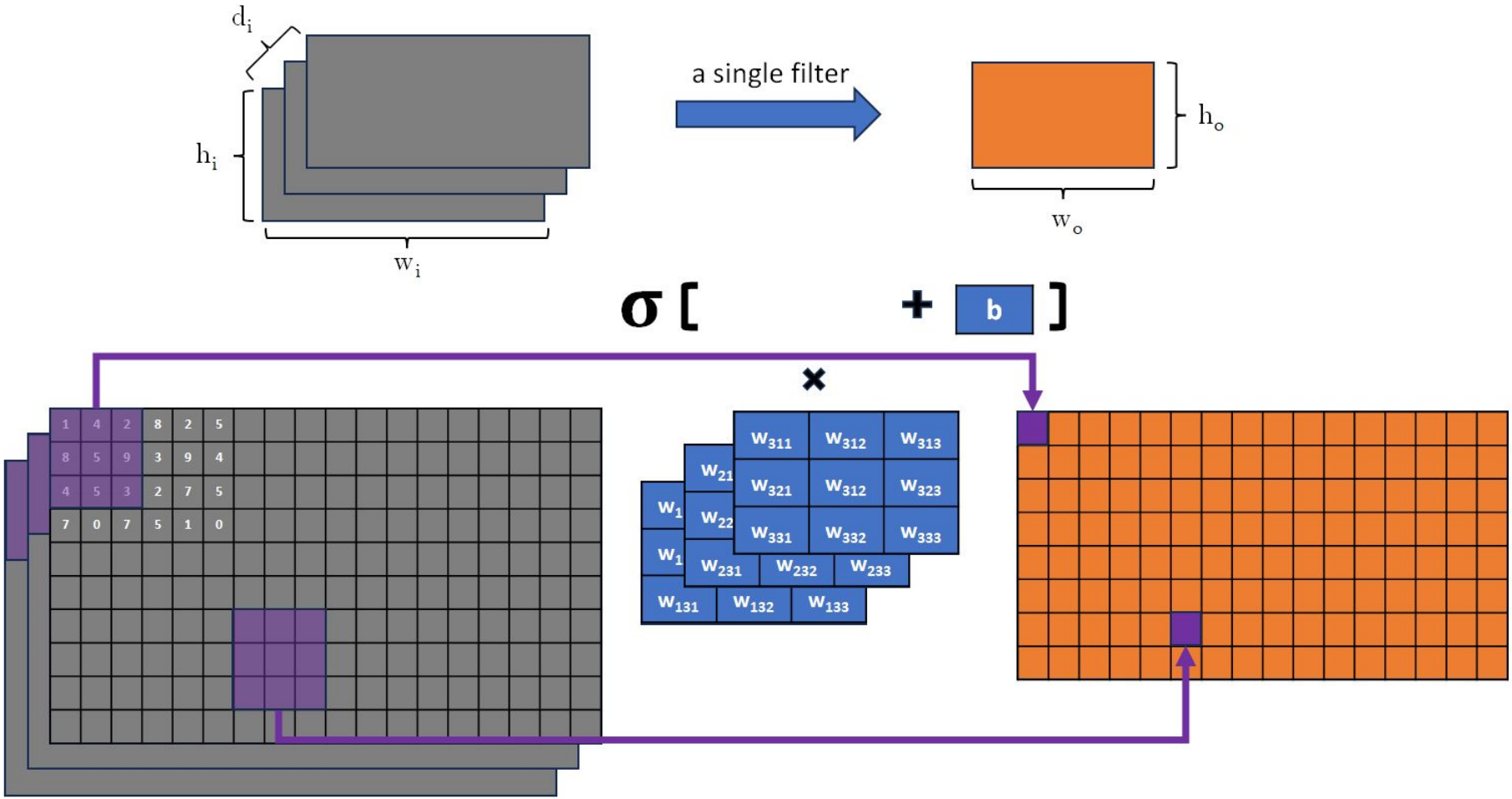
Problems with using MP for images

- the spacial information is discarded: images are flattened and pixels are processed in the same way regardless of the distance between them. As a result, most neurons try to predict something based on a random set of pixels scattered across the image
- there are too many parameters: individual sets of weights are assigned to every pixel. When the image resolution gets 2x better, the number of parameters in the first layer increases 4x
- intuition: we'd like to calculate something based on a small window of pixels, and apply the same weights across the whole image

Convolutional Neural Networks

- … are networks that use convolutional layers, typically close to the input
- the input is not flattened, it's now a 3d tensor (width, height, no. channels)
- convolutional layer will transform a 3d input into a 3d output. The width and height typically decreases, while the numer of channels grows
- neurons in a convolutional layer are connected to a small portion of inputs (close in the first and second coordinate), and they share the same weights with all neurons that generate the same channel
- before we think about neurons, let's imagine a convolutional layer as a set of d_o filters that take (w_i, h_i, d_i) and produce (w_o, h_o, d_o)




d_i
h_i
w_i
a single filter
h_o
w_o
σ [+ b]
×
1 4 2 8 2 5
8 5 9 3 9 4
4 5 3 2 7 5
7 0 7 5 1 0
W_{311} W_{312} W_{313}
W_{321} W_{312} W_{323}
W_{331} W_{332} W_{333}
W_{21}
W_1
W_{22}
W_1
W_{231} W_{232} W_{233}
W_{131} W_{132} W_{133}

Convolutional layer

- the same filter is applied across the whole image to create a single channel
- during backpropagataion, gradients from all inputs are aggregated to influence filter's weights
- many channels can be created
- convolutional layers have many other parameters than height, width and the number of channels/filters. Some examples include padding, step and stride. Most of them influence the width and height of the output

output channels

input channels

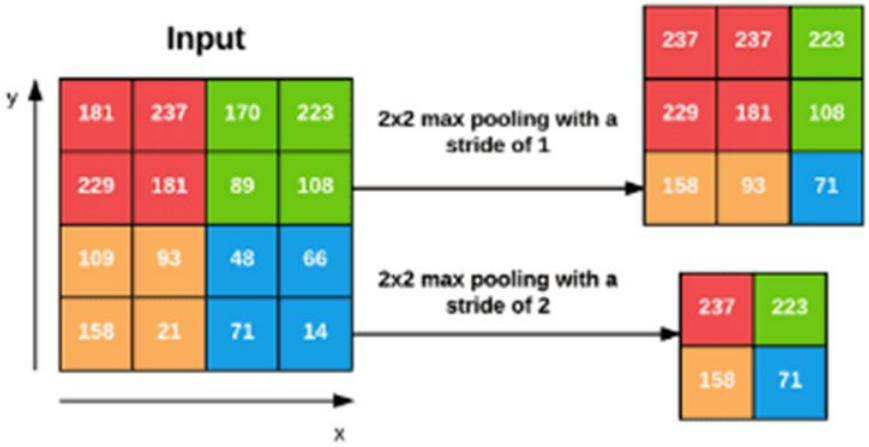
filter size

```
class OurCNN(nn.Module):
    def __init__(self):
        super(OurCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 18, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(18, 32, 3)
        self.fc1 = nn.Linear(32 * (INPUT_RESOLU
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, len(CLASS_NAME

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 32 * (INPUT_RESOLUTION//
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Pooling layer

- turns out that convolutional layers work better when they are followed by a „smoothing" layer that averages the outputs
- this makes the network more robust to noise and less likely to overfit
- types of pooling include max and average



https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/

Tutorial: training a CNN

- https://github.com/JakubBilski/introduction-to-machine-learning
- https://api.wandb.ai/links/podcast-o-rybach-warsaw-university-of-technology/2qr5y1lj