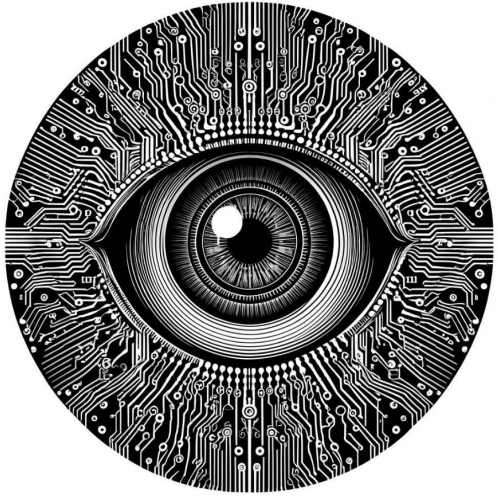
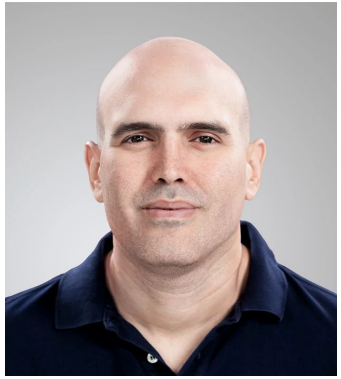
Workshop 2 - Introduction to Neural Network Training



Antonio Rueda-Toicen

About me

- External PhD Candidate at Hasso Plattner Institute, AI Engineer & DevRel for Voxel51
- Organizer of the Berlin Computer Vision Group
- Instructor at Nvidia's Deep Learning Institute and Berlin's Data Science Retreat
- Preparing a MOOC for OpenHPI (to be published in May)



LinkedIn

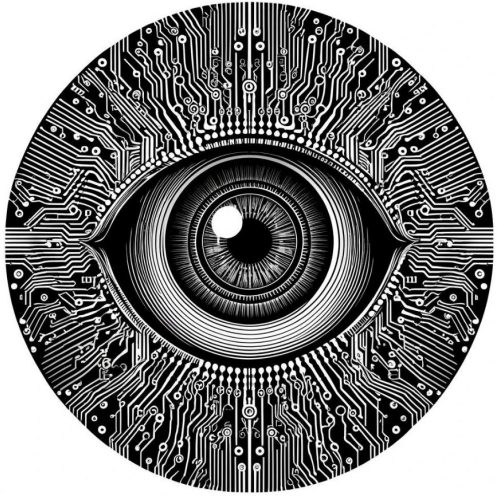
Agenda

- Introduction to neural networks
- Building a multilayer perceptron (MLP) for regression
- Matrix multiplication, non-linear activations, and network shape

Notebook

- Training an MLP for car price prediction (memorizing a price)
  - Colab notebook

Predicting Car Prices from their Images



Antonio Rueda-Toicen

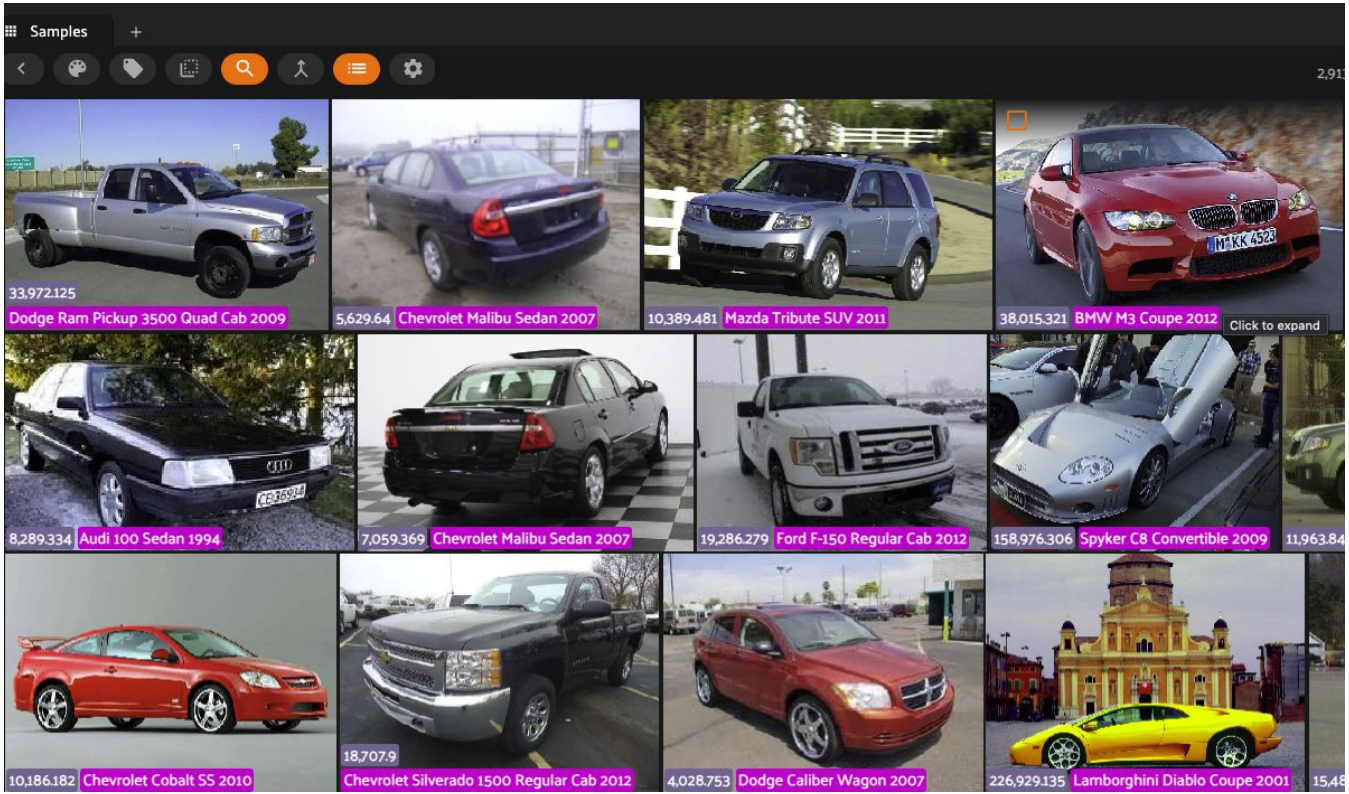
cars & bids Auctions Sell a Car Community What's Cars & Bids? Daily Email Search

2012 BMW M3 Coupe

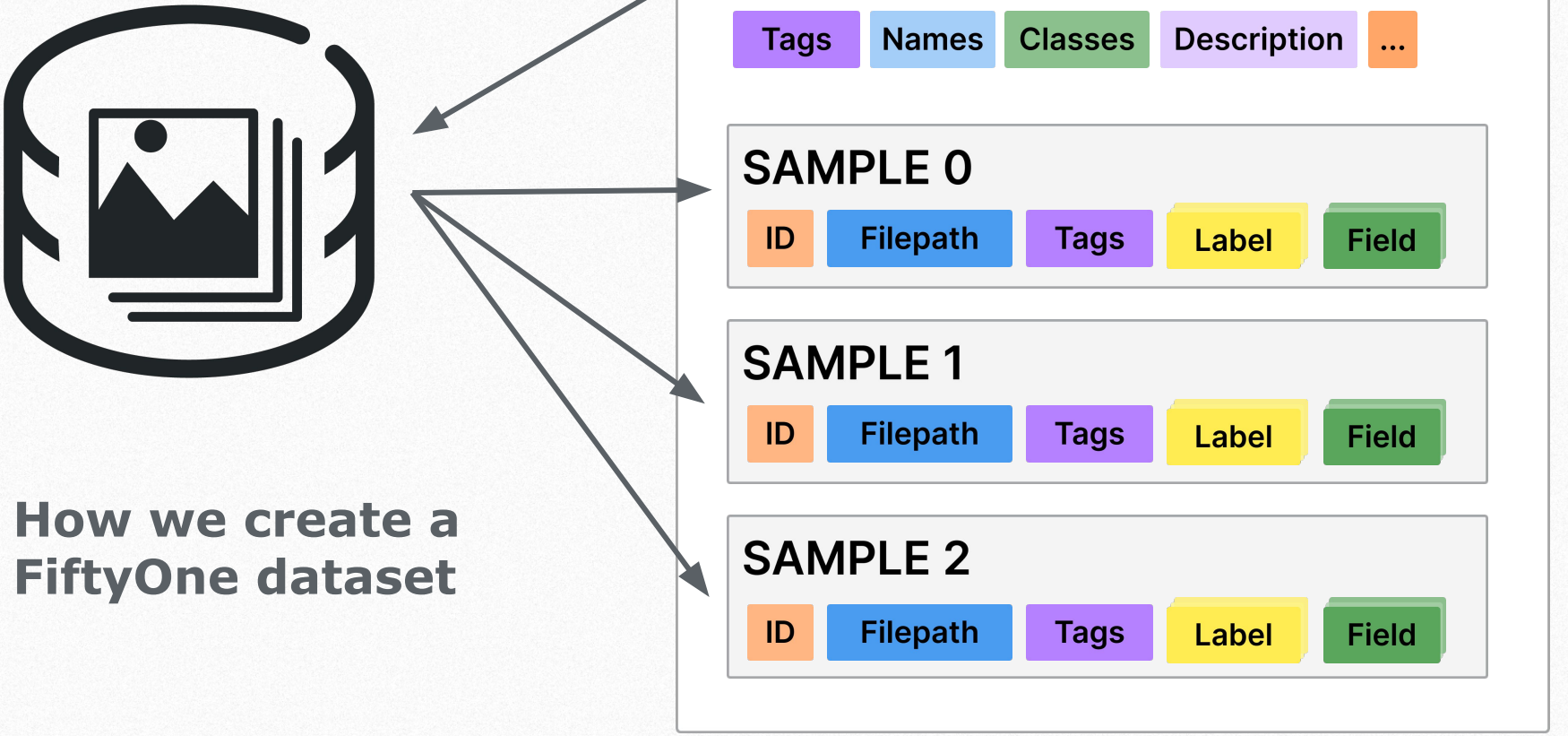
NO RESERVE ~46,700 Miles, 6-Speed Manual, Premium Package, Some Modifications

How much is this car worth?

Our dataset



[Exploring the dataset in FiftyOne and training a model to predict prices]



How we create a FiftyOne dataset

Actual: $13045.57
Predicted: $12750.80
Absolute Error: $294.77

Actual: $18327.76
Predicted: $18665.31
Absolute Error: $337.55

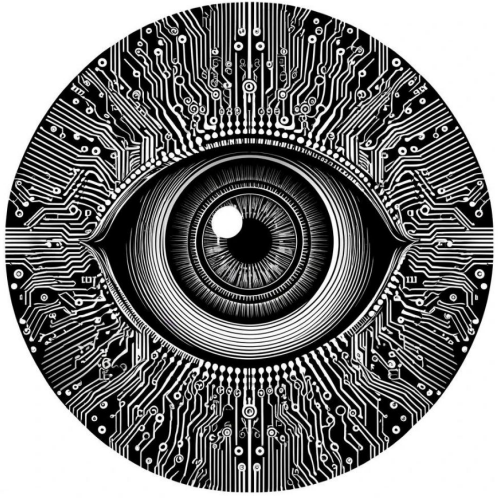
Actual: $13632.83
Predicted: $12904.04
Absolute Error: $728.79

Actual: $13892.51
Predicted: $14672.02
Absolute Error: $779.51

Actual: $17756.41
Predicted: $16876.02
Absolute Error: $880.39

Homework for advanced students: add predictions to our FiftyOne dataset

Introduction to Neural Networks

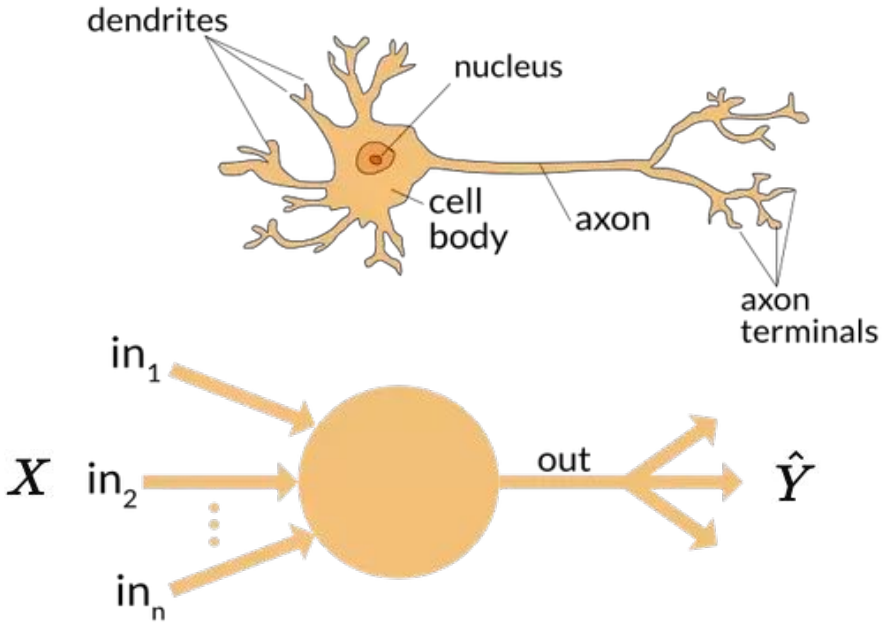


Antonio Rueda-Toicen

Learning goals

- Explain how neural networks learn during training
- Identify the components of input units, hidden layers, and output units
- Apply concepts of weights, gradients and loss
- Understand the purpose of validation and test data

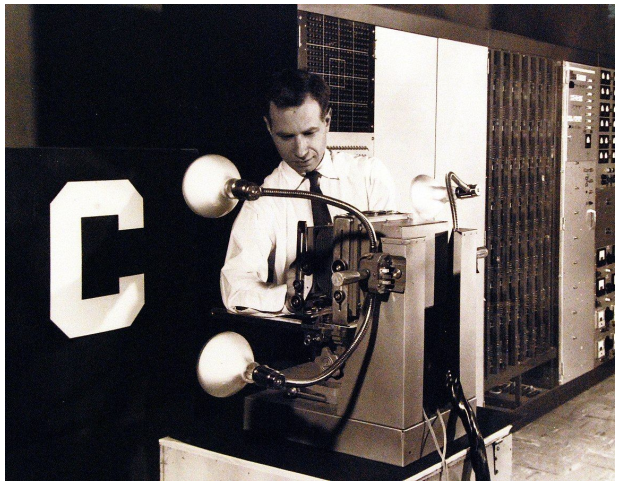
What is an artificial neural network?



- Artificial "neurons" are also called "processing units"
- We can think of them as functions that map X inputs to Ŷ outputs

Image from source

The perceptron



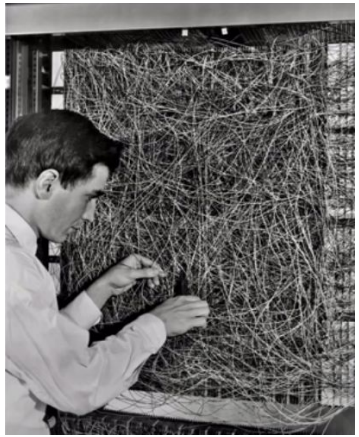
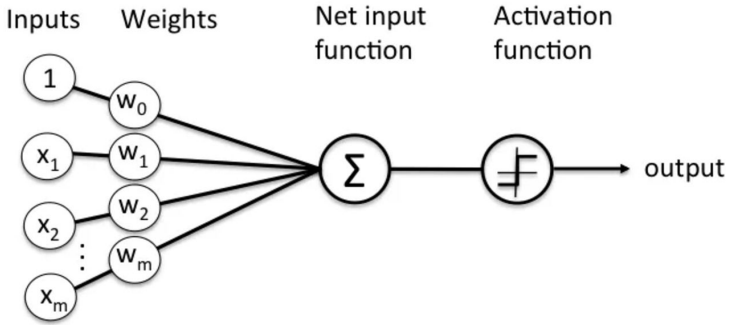


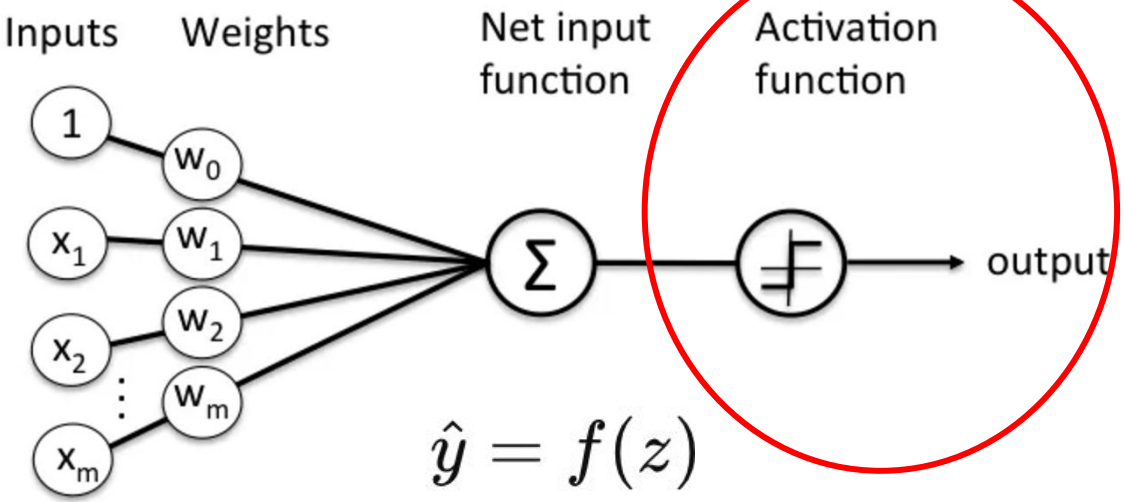
Image from source



$$\hat{y} = f\left(\sum_{j=0}^{m} w_j x_{ij}\right)$$

Images: Mark I Perceptron at Cornell University (1961) source

A modern fully connected layer

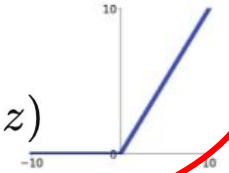


$$\hat{y} = f(z)$$

$$z = \sum_{i=0}^{m} w_i x_i$$

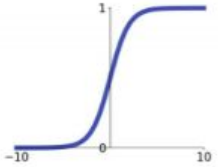
ReLU

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



Sigmoid

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



Making a perceptron learn the price of a vintage car

training

1964 Chevrolet Corvette Stingray

Image source

y = 83999 EUR (real price, target value)

$\hat{y}$ = 84000 EUR (a possible prediction)

Absolute error = $|\hat{y} - y|$ = 1 EUR

validation

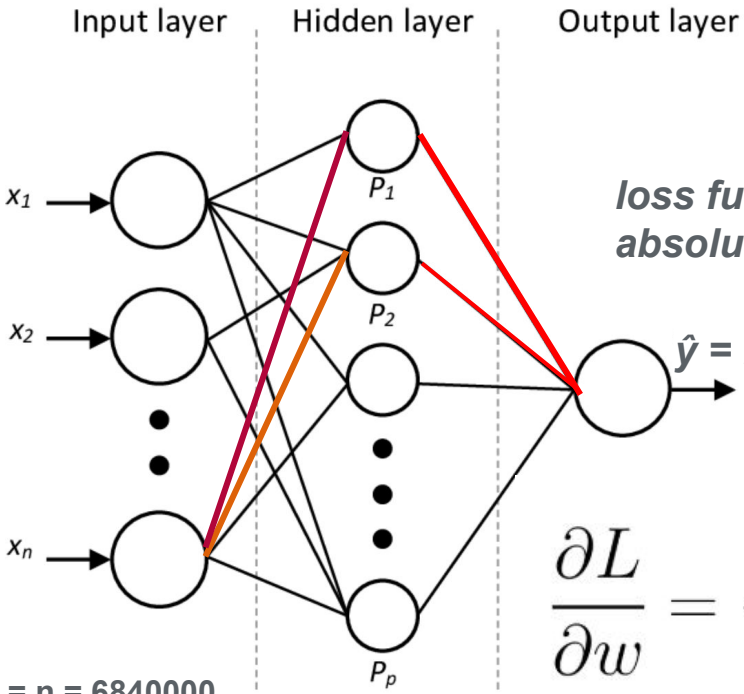
Image source

testing

Image source

Training a fully connected network (multilayer perceptron)

y = 83999 EUR



loss function =
absolute error = |ŷ - y| = 12188

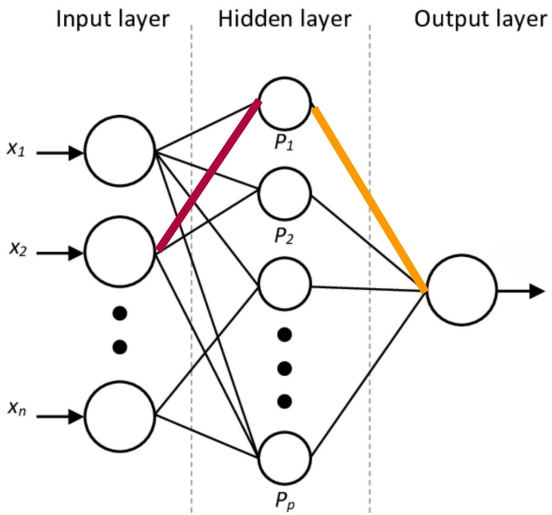
ŷ = f(z) = 71811

$$\frac{\partial L}{\partial w} = -\operatorname{sign}(y - f(x))\frac{\partial f(x)}{\partial w}$$

```
H = 1900
W = 1200
C = 3 (RGB)
```

C * H * W = n = 6840000

Understanding network weights and their updates



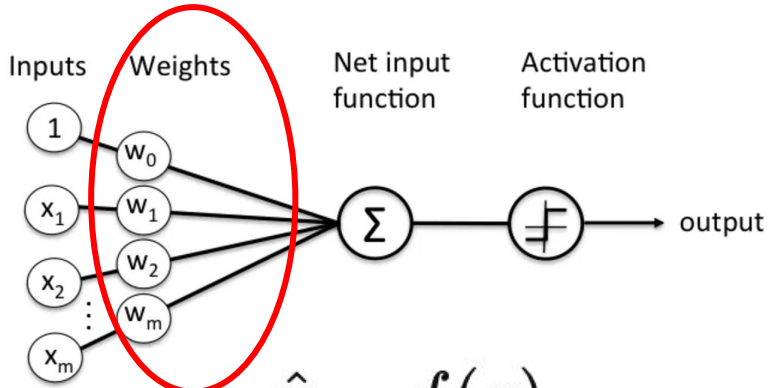
w_{ij} = weight from neuron i to neuron j

learning_rate = step size for updates, a constant like 0.01

L = loss function

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Modern perceptrons use loss functions as feedback



$$\hat{y} = f(z)$$

$$z = \sum_{i=0}^{m} w_i x_i$$

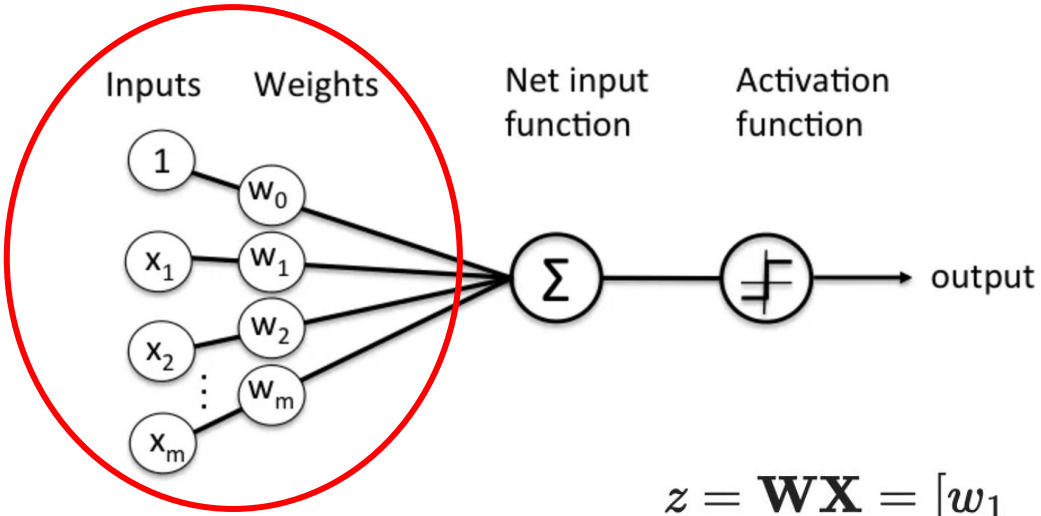
Regression

$$\text{Mean Absolute Error} = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i|$$

Classification

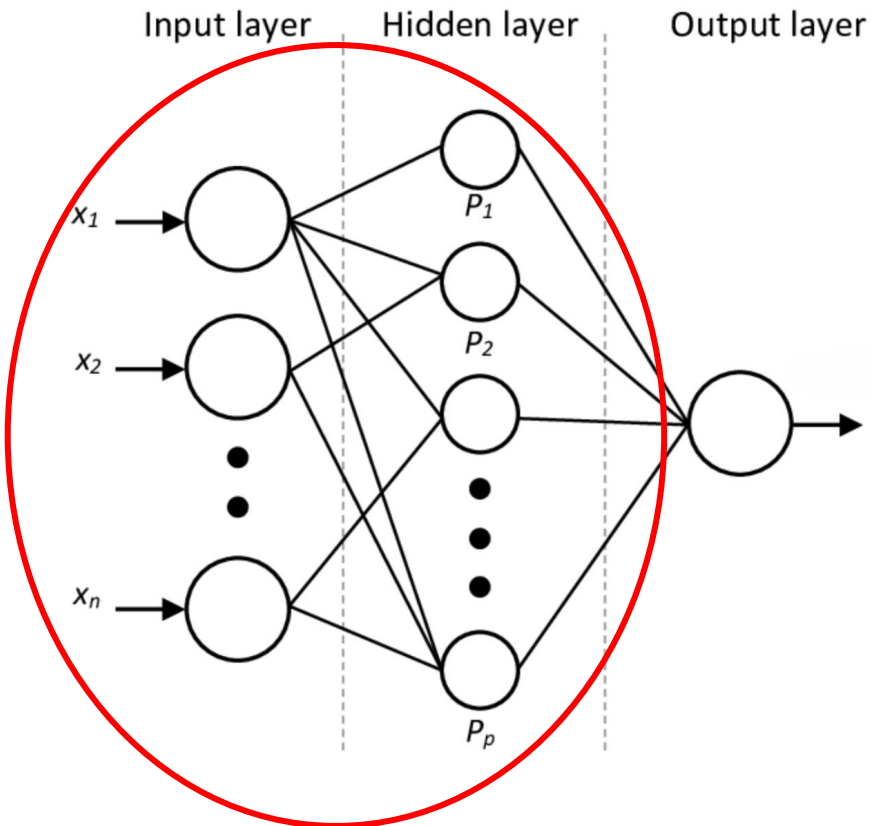
$$\text{Cross Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

Matrix multiplication: features times weights



$$z = \mathbf{WX} = \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \sum_{i=1}^{m} w_i x_i$$

Dimensions of the weight matrix define the output's



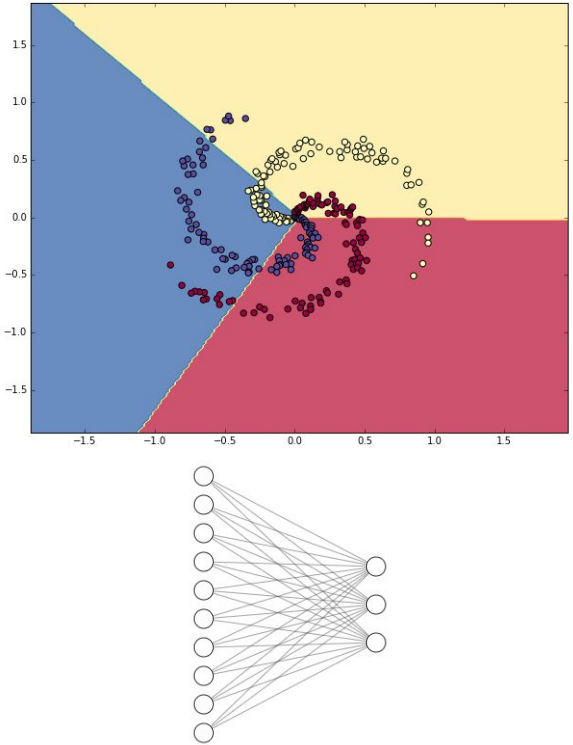
$$P = WX$$

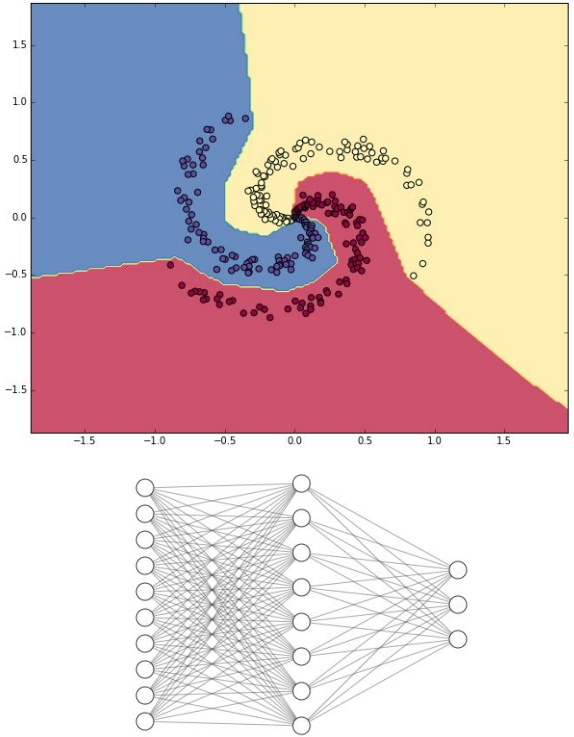
$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_p \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p1} & w_{p2} & \cdots & w_{pn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

For example:

$$p = 4 \quad n = 3$$

Why do we add hidden layers?





Validating and testing the model

sample to decide when to stop training: validation

y = 82999
Image source

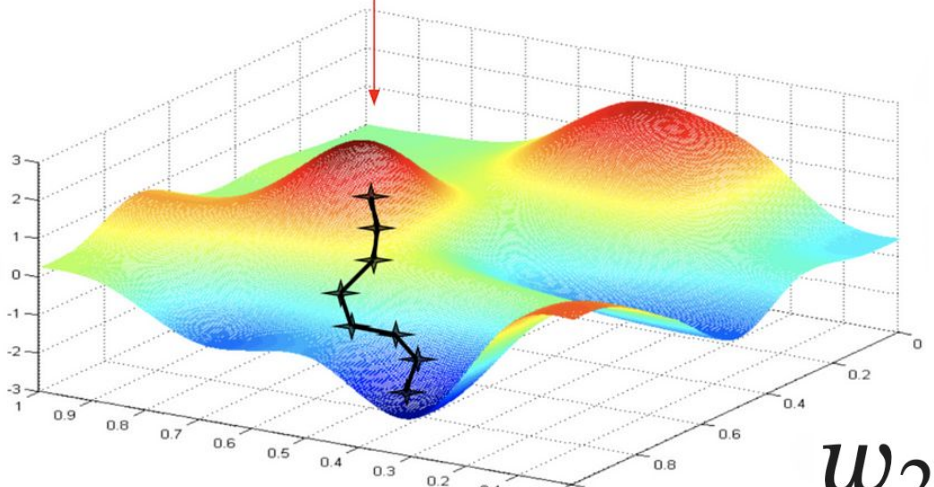
sample to use as a final benchmark: test

y = 82699
Image source

Gradient descent

we start at a random position in the loss landscape

$L(w_1, w_2)$



w_1

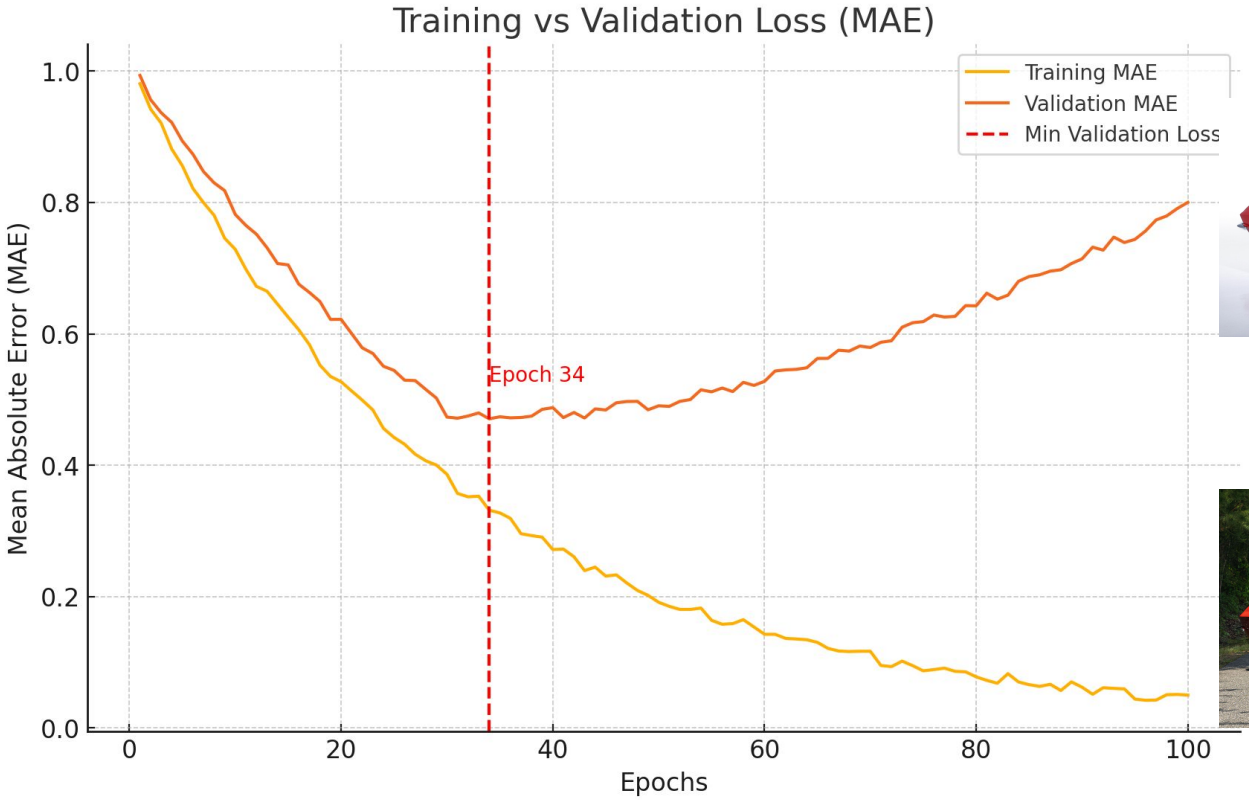
w_2

validation image
role: evaluate best weights

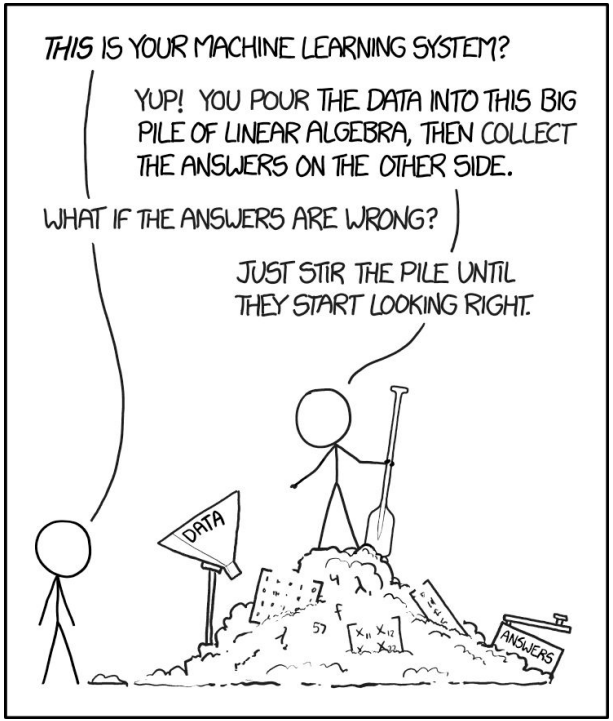
training image
role: update weights

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Training vs validation loss



The test set gives the benchmark of generalization



***y* = 82699**
ŷ = 84015
test set error = |y - ŷ| = 1316

Image from https://xkcd.com/1838/

Summary

Neural networks are simplified mathematical models of biological neurons

- They are functions with trainable parameters (aka ‘weights’) that we stack in layers with variable number of processing units (aka ‘neurons’)
- The size of our weight matrices define the dimensions of our hidden and output layers

We use loss functions and gradients to update the weights of a network

- Loss functions provide feedback to the model
- The weight update is an iterative process (gradient descent)

Deciding on training settings (aka “hyperparameters”) is an explorative process

- We use validation sets to decide how and when to stop training. We use tests to provide final benchmarks

Further reading

On the origin of neural networks

- https://nautil.us/the-man-who-tried-to-redeem-the-world-with-logic-2885/

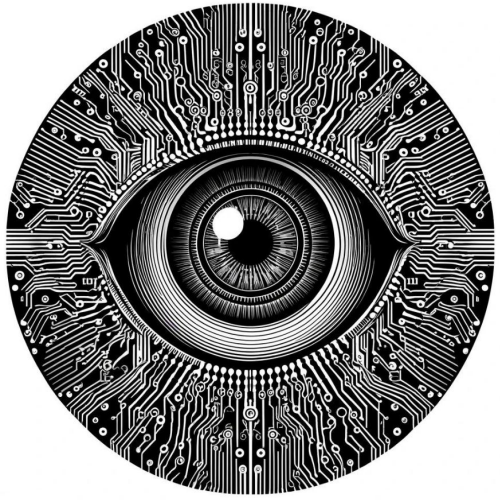
Annotated history of modern AI and deep learning by Jurgen Schmidhüber

- https://people.idsia.ch/~juergen/deep-learning-history.html

Frank Rosenblatt's Perceptron

- **https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon**

Building a Multilayer Perceptron for Regression in PyTorch

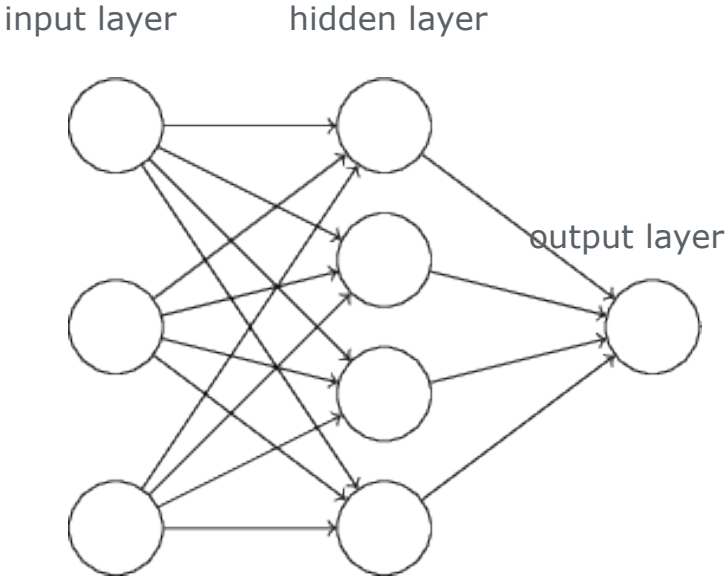


Antonio Rueda-Toicen

Learning goals

- Implement a fully connected network in PyTorch with `nn.Linear` and `nn.Sequential` syntax
- Explore the necessary agreements between processing units and input size
- Understand the motivations for image input resizing
- Code input units, hidden layers, and output units
- Examine the representation of weights, gradients, and loss in PyTorch

A minimal network




```
import torch.nn as nn

# Input layer to hidden layer (3 -> 4)
i_h_layer = nn.Linear(in_features=3, out_features=4, bias = False)

# Hidden layer to output layer (4 -> 1)
h_o_layer = nn.Linear(in_features=4, out_features=1, bias = False)


# Connecting the layers, notice that 'hidden layer' is implicit
model = nn.Sequential(
            i_h_layer,
            nn.ReLU(),
            h_o_layer,
        )

# The input has as many entries as we have input units
input_data = torch.tensor([1.0, 2.0, 3.0], dtype=torch.float32)

# Feedforward pass
prediction = model(input_data)

# The value that we want to match
target = torch.tensor([11.], dtype=torch.float32)

# Our measure of error
loss_function = nn.L1Loss()

# Evaluate how good our prediction is
loss_value = loss_function(target, prediction)
```

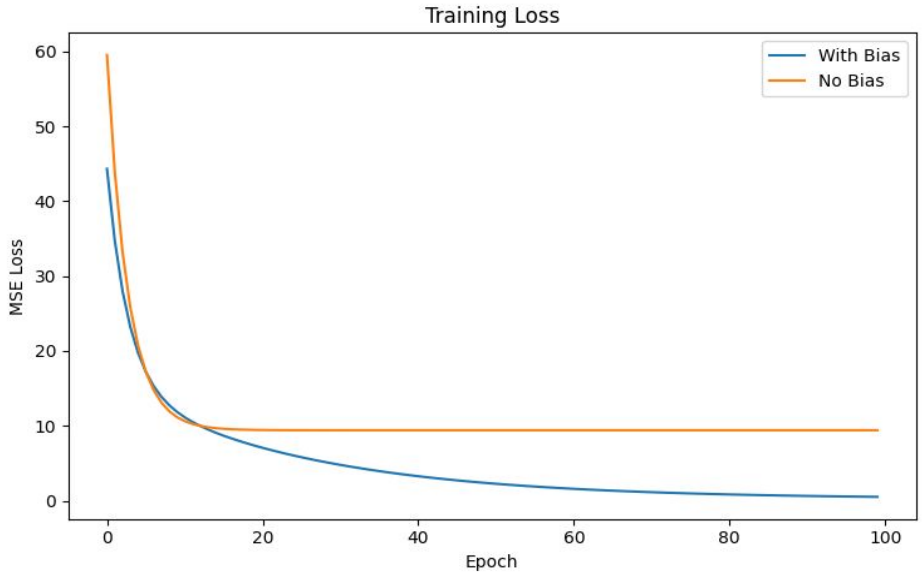
Inspecting nn.Linear

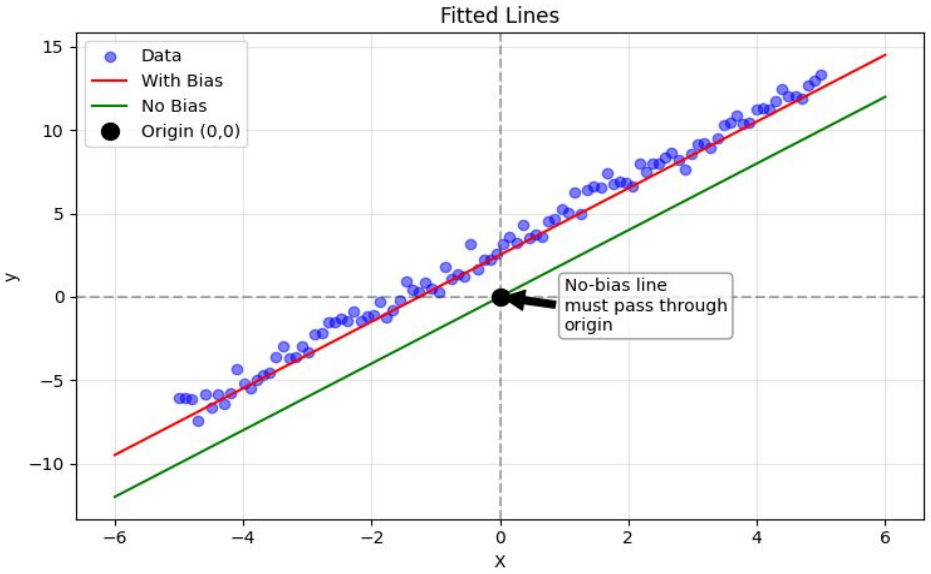
```python
import torch
import torch.nn as nn

# Define a linear layer
layer = nn.Linear(in_features=4, out_features=1, bias=True)


# Examine parameters, weights and bias
for name, param in layer.named_parameters():
    print(f"{name}: {param.shape}")
    print(param)

# Forward pass
x = torch.randn(5, 4)  # Batch of 5 samples, 4 features each
output = layer(x)

# Batch of 5 samples, single output value
print(output.shape)
```

$$z = \mathbf{W}\mathbf{X} = \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \sum_{i=1}^{m} w_i x_i$$

```
weight: torch.Size([1, 4])
Parameter containing:
tensor([[-0.0576,  0.3385, -0.3604, -0.1795]], requires_grad=True)
bias: torch.Size([1])
Parameter containing:
tensor([0.3461], requires_grad=True)
```

Bias vs no bias



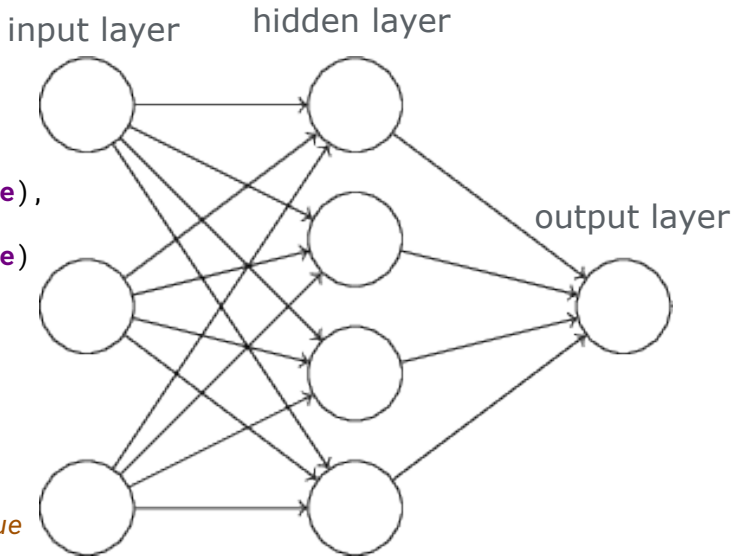


$$\text{Mean Squared Error (MSE) Loss} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

A Colab demo notebook

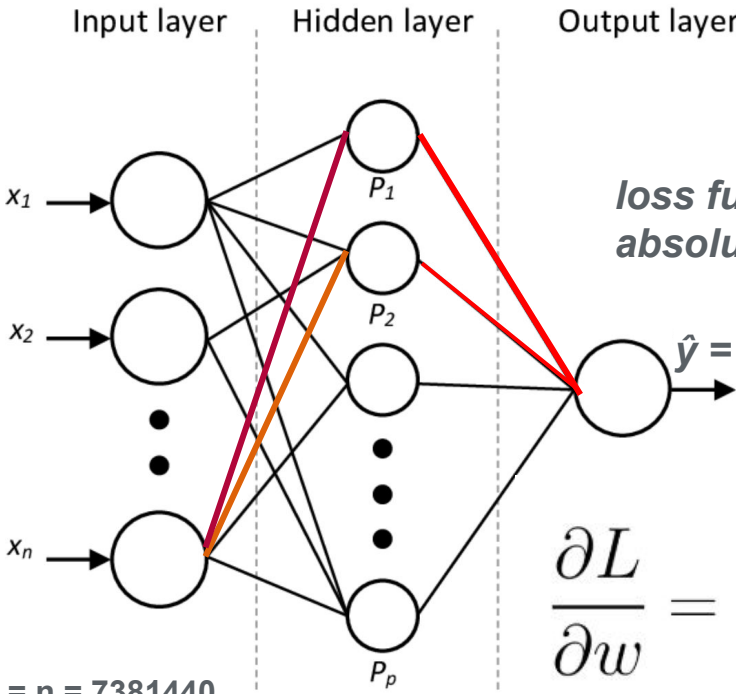
Inspecting nn.Sequential

```
# Connecting the layers,
# notice that 'hidden layer' is implicit

model = nn.Sequential(
          nn.Linear(in_features=3, out_features=4, bias = False),
          nn.ReLU(),
          nn.Linear(in_features=4, out_features=1, bias = False)
        )

# Forward pass
x = torch.randn(5, 3) # Batch of 5 samples, 3 features each
output = model(x)

# Batch of 5 outputs, each one having one regression target value
print(output.shape) # prints torch.Size([5, 1])
```



Fully connected networks (aka multilayer perceptrons)

y = 83999 EUR



loss function =
absolute error = $|\hat{y} - y|$ = 12188

$\hat{y} = f(x_1, \ldots, x_n)$ = 71811

$$\frac{\partial L}{\partial w} = -\operatorname{sign}(y - f(x))\frac{\partial f(x)}{\partial w}$$

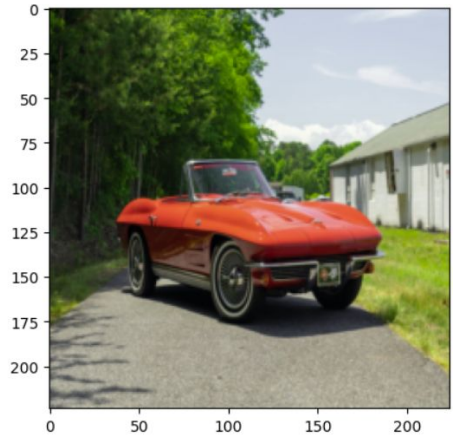
```
H = 1920
W = 1282
C = 3 (RGB)
```

C * H * W = n = 7381440

Resizing the input to reduce the number of parameters

H = 1920
W = 1282
C = 3 (RGB)

```
transforms.Compose([
 transforms.ToImage(),
 transforms.Resize((224, 224)),
transforms.ToDtype(torch.float32,scale=True)
])
```

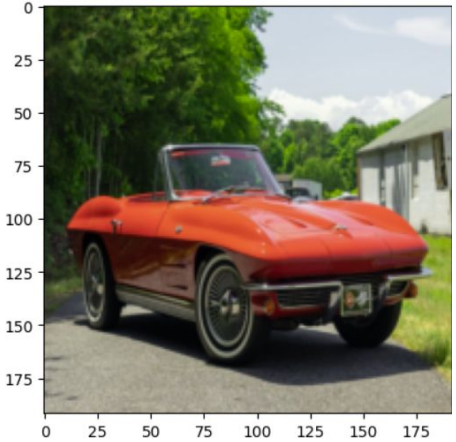


H = 224
W = 224
C = 3 (RGB)

CenterCrop to capture more of the object of interest

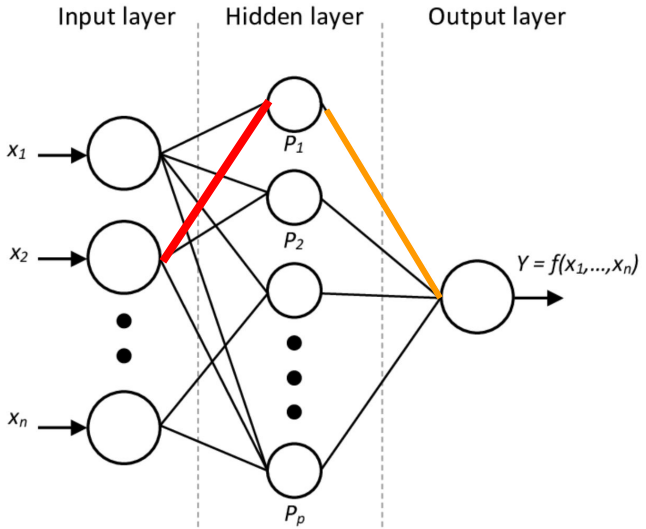
H = 1920
W = 1282
C = 3 (RGB)

```
transforms.Compose([
 transforms.ToImage(),
 transforms.Resize((256, 256)),
 transforms.CenterCrop(192),
transforms.ToDtype(torch.float32,scale=True
)
])
```



H = 192
W = 192
C = 3 (RGB)

Stochastic gradient descent - the weight update rule



w_{ij} = weight from neuron i to neuron j

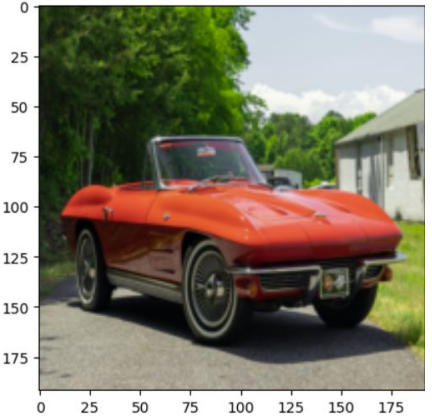
learning_rate = step size for updates, a constant like 0.01

L = loss function

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Implementing the network

```
# Number of units on the hidden layer, this number is arbitrary

p = 100

# We use nn.Sequential to concatenate PyTorch modules in order

multilayer_perceptron=nn.Sequential(
                    nn.Flatten(), # Flattens the image tensors within a batch
                    nn.Linear(in_features= C * H * W, # This matches the image shape
                              out_features= p,
                              bias=True),
                    nn.ReLU(),
                    nn.Linear(in_features=p,
                              out_features=1, # We are predicting a single value
                              bias=True),
)
```



H = 192
W = 192
C = 3 (RGB)

y = 83999 EUR

Implementing the weight update rule

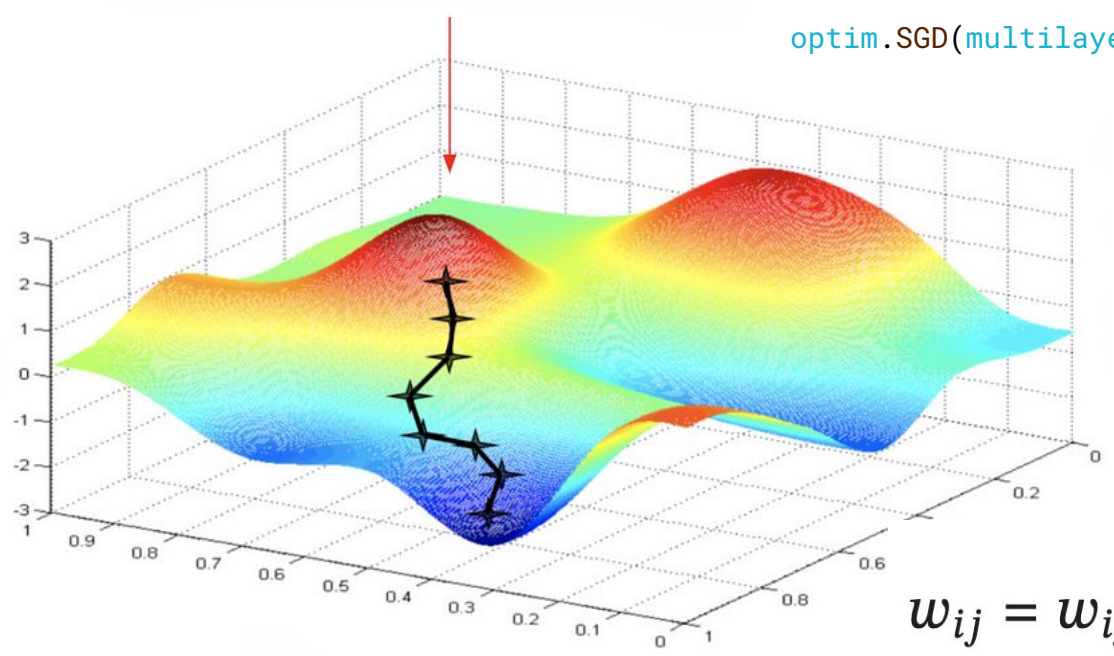
```
def update_rule(weight, grad, learning_rate):
    return nn.Parameter((weight - learning_rate * grad))

learning_rate = 0.1
new_weights_0 = update_rule(multilayer_perceptron[0].weight,
                            multilayer_perceptron[0].weight.grad,
                            learning_rate)

new_weights_2 = update_rule(multilayer_perceptron[2].weight,
                            multilayer_perceptron[2].weight.grad,
                            learning_rate)

# We reassign the weights to our model
multilayer_perceptron[0].weight = new_weights_0
multilayer_perceptron[2].weight = new_weights_2
```

Image from ideogram.ai

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Training with an optimizer

```
import torch.optim as optim

optim.SGD(multilayer_perceptron.parameters(), lr=1e-2)
```



$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Training with an optimizer

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define an Stochastic Gradient Descent optimizer
optimizer = optim.SGD(multilayer_perceptron.parameters(), lr=1e-2)

# Define a loss function
criterion = nn.L1Loss()

# Forward pass
pred = multilayer_perceptron(x)

# Compute loss
loss = criterion(pred, y)

# Backprop
optimizer.zero_grad()
loss.backward()

# Update weights
```

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

The importance of the learning rate

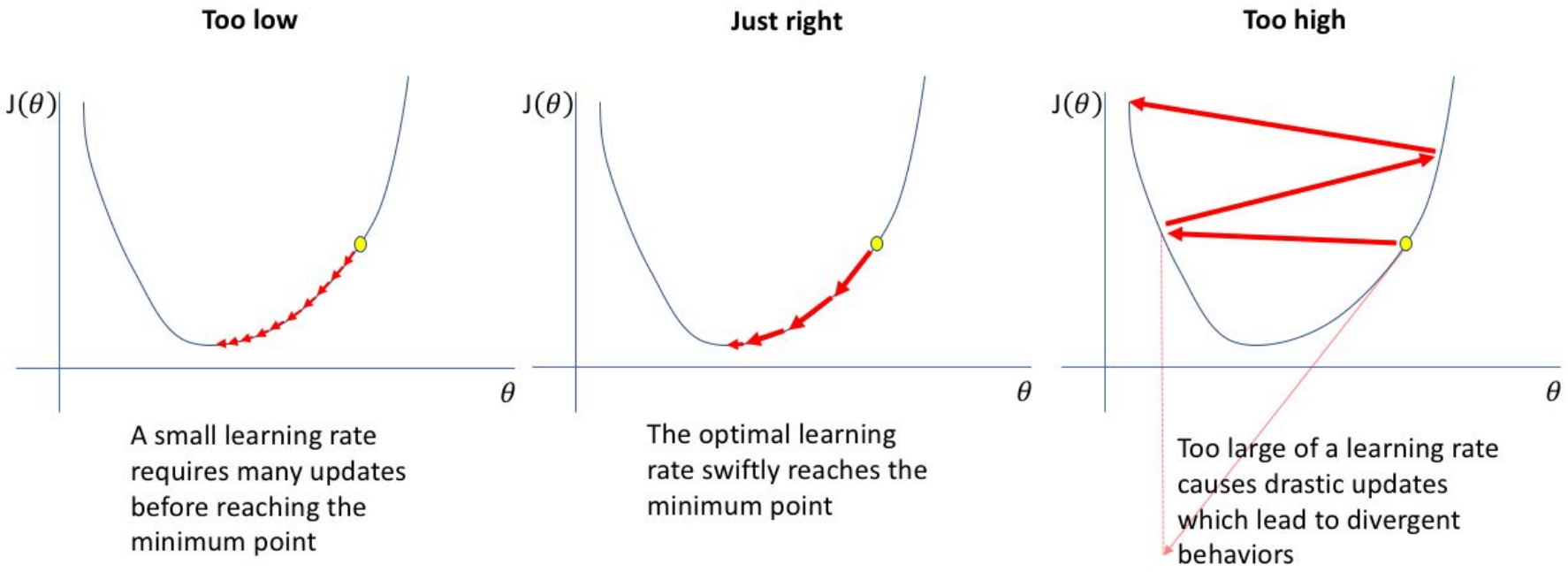


Image from Deep Learning for Coders

Summary

PyTorch building blocks

- Multilayer perceptrons in PyTorch combine `Linear` layers, activation functions, and loss functions. The framework handles gradient computation through autograd.
- We can use the `Sequential` syntax to wrap up components, this is an alternative to subclassing `nn.Module`

Input processing

- Image input requires preprocessing through resizing and normalization to create manageable feature tensors. The input size determines the network architecture.

Model training

- Training combines forward passes, loss computation, backpropagation, and weight updates. The learning rate controls optimization stability and convergence speed.

Further reading

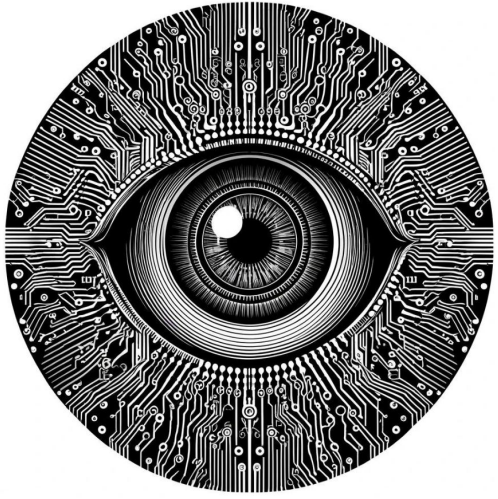
What is torch.nn really?

- https://pytorch.org/tutorials/beginner/nn_tutorial.html

Tensorflow playground

- https://playground.tensorflow.org/

Matrix Multiplication, Non-linear Activations, and Network Shape

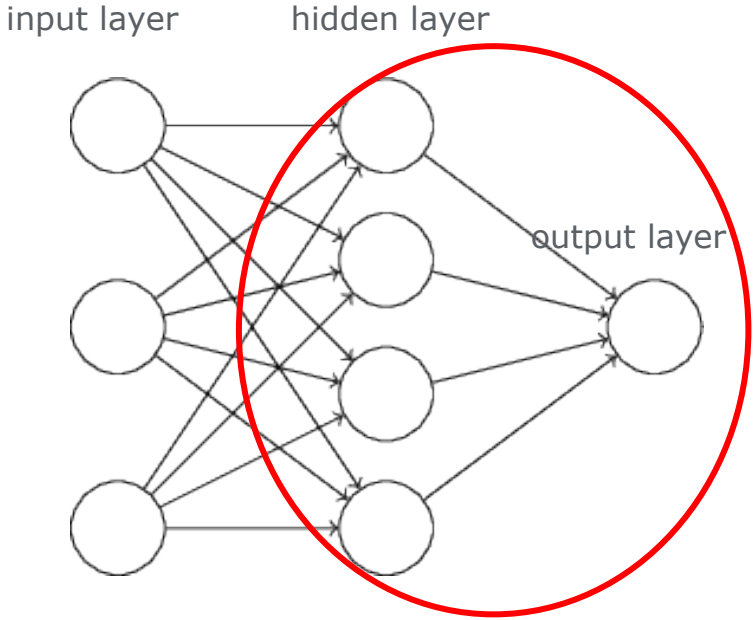


Antonio Rueda-Toicen

Learning goal

- Understand the correspondence between the matrix multiplication dimensions and the network’s architecture
- Recognize the importance of passing the output of matrix multiplication operations to non-linear activation functions

The feedforward pass - output layer



To compute the **output Z** with **1 unit**:

Explicitly:

$$Z = XW$$

$$Z = \begin{bmatrix} X_1 & X_2 & X_3 & X_4 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

Resulting in a **scalar output**

$$Z = X_1 w_1 + X_2 w_2 + X_3 w_3 + X_4 w_4$$

Matrix multiplication and network shape - output layer

```python
import torch

# Create input with batch_size = 1
batch_size = 1
input_features = 4  # X1, X2, X3, X4 as shown in the image
output_features = 1  # Scalar output Z as shown

# Create input tensor X = [X1, X2, X3, X4]
X = torch.randn(batch_size, input_features)  # Shape: (1, 4)
print(f"Input X: {X[0]}")  # Print without batch dimension for clarity

# Create weights w = [w1, w2, w3, w4]
weights = torch.randn(input_features, output_features)  # Shape: (4, 1)
print(f"Weights w: {weights.squeeze()}")  # Print without extra dimensions

# Approach 1: Direct matrix multiplication (Z = XW)
output_matmul = torch.matmul(X, weights)  # Shape: (1, 1)
print(f"Output using matrix multiplication: {output_matmul.item():.4f}")

# Approach 2: Using nn.Linear
layer = torch.nn.Linear(in_features=input_features,
                        out_features=output_features,
```

$$Z = XW$$

$$Z = \begin{bmatrix} X_1 & X_2 & X_3 & X_4 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

$$Z = X_1 w_1 + X_2 w_2 + X_3 w_3 + X_4 w_4$$

Common errors with tensor dimensions

```python
import torch

# Create input with batch_size = 1
batch_size = 1
input_features = 4  # X1, X2, X3, X4 as shown in the image
output_features = 1  # Scalar output Z as shown

# Wrong: Missing batch dimension
X = torch.randn(input_features)  # Shape: (4,) instead of (1,4)
# Error: Expected 2D input, got 1D

# Wrong: Reversed multiplication order
output_matmul = torch.matmul(weights, X)  # Wrong order
# Error: size mismatch, got [4,1] x [1,4]

# Wrong: Inconsistent dimensions
layer = torch.nn.Linear(in_features=3,  # Wrong input size
                        out_features=output_features, bias=False)
# Error: size mismatch, expected input of size [*,3], got [*,4]
```

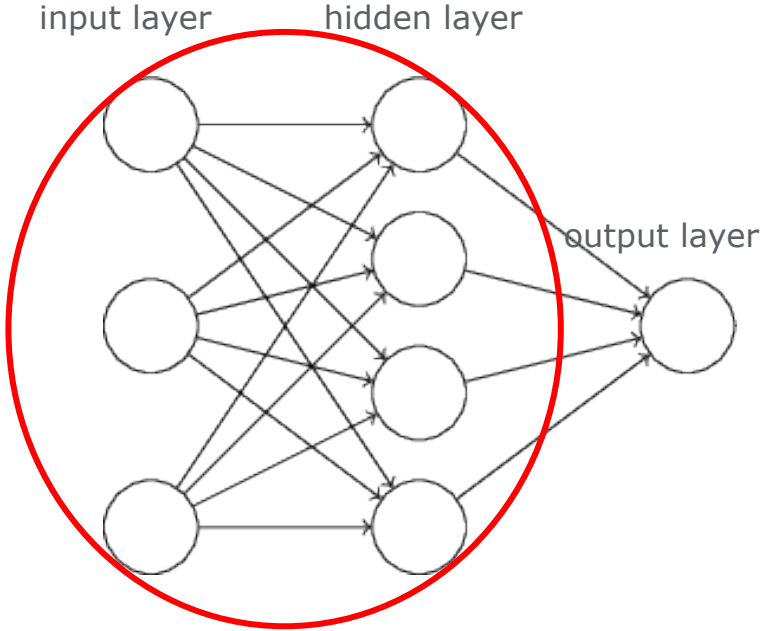
Tip:
always check the shape of tensors with a_tensor.shape or a_tensor.size()

$$Z = XW$$

$$Z = \begin{bmatrix} X_1 & X_2 & X_3 & X_4 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

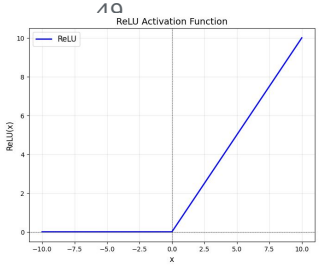
$$Z = X_1 w_1 + X_2 w_2 + X_3 w_3 + X_4 w_4$$

Adding non-linearity between layers



$$Z = ReLU(XW)$$

$$Z = \text{ReLU}\left(\begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix} \right)$$

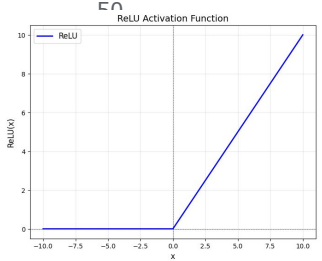


Adding non-linearity between layers

```python
import torch
from torch.nn.parameter import Parameter

X = torch.randn(batch_size, 3)

# Create weight matrix as Parameter
weights = Parameter(torch.randn(3, 4))  # Shape: (3, 4)

# Approach 1: Direct matrix multiplication with ReLU
output_matmul = torch.relu(torch.matmul(X, weights))

# Approach 2: Using nn.Linear with ReLU
layer = torch.nn.Sequential(
    torch.nn.Linear(in_features=input_features,
                    out_features=output_features,
                    bias=False),
    torch.nn.ReLU()
)
```

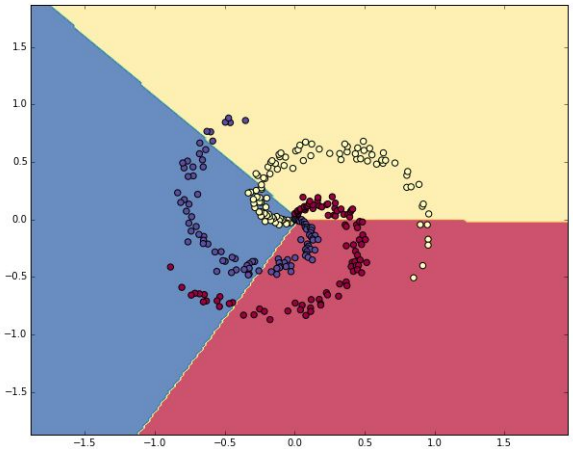
$$Z = ReLU(XW)$$

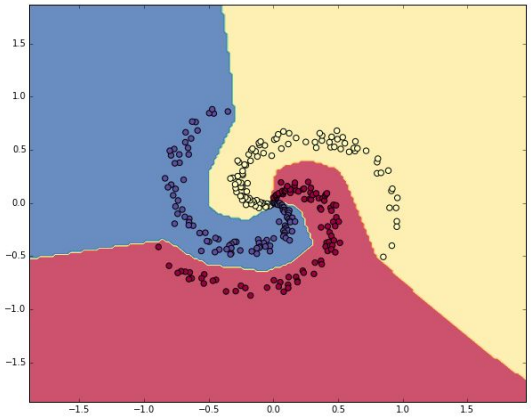
$$Z = \text{ReLU}\left(\begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix} \right)$$



Deep learning is the art of making squiggly functions

- Squiggly function = ‘sophisticated decision boundary’
- More weights and non linear activations ~ more squiggly boundaries
-
-





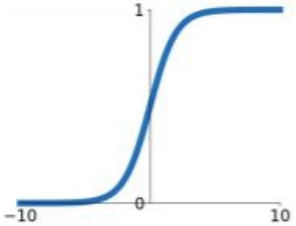
https://cs231n.github.io/neural-networks-case-study
Experiment notebook

Activation functions

ReLU is a good default choice for most problems

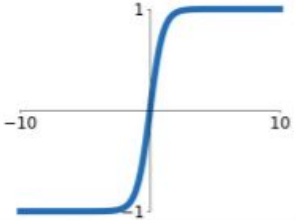
Sigmoid

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



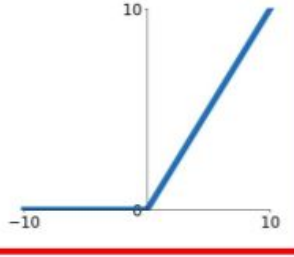
tanh

$$\tanh(x)$$



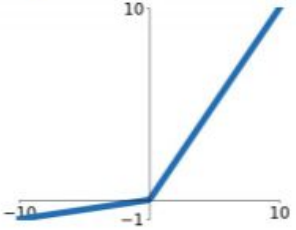
ReLU

$$\max(0, x)$$



Leaky ReLU

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

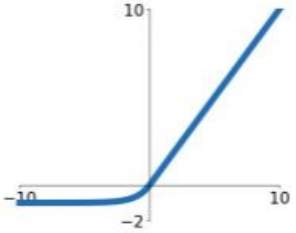


Maxout

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

ELU

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



Practicing matrix multiplication

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix} \times \begin{bmatrix} 0.11 & 0.12 & 0.13 & 0.14 \\ 0.21 & 0.22 & 0.23 & 0.24 \\ 0.31 & 0.32 & 0.33 & 0.34 \\ 0.41 & 0.42 & 0.43 & 0.44 \end{bmatrix} = \begin{bmatrix} 0.31 & 0.32 & 0.33 & 0.34 \end{bmatrix}$$

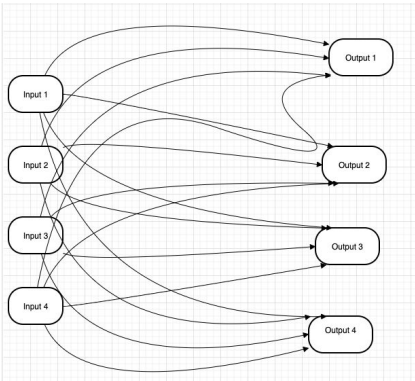
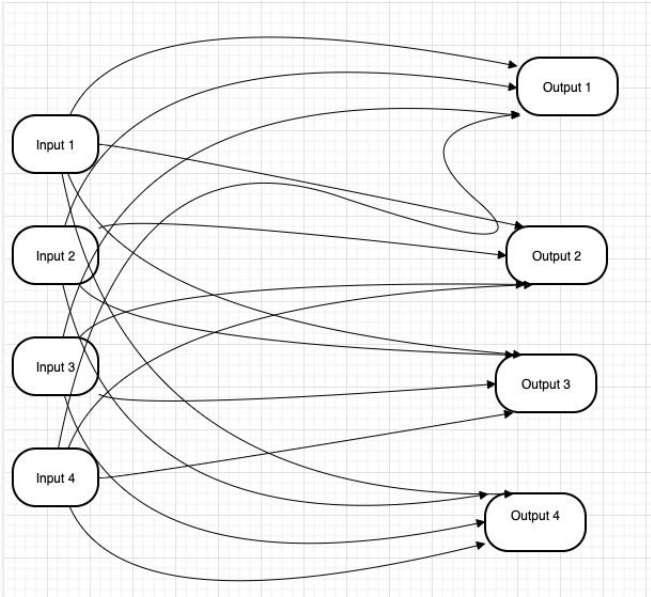


Image from http://matrixmultiplication.xyz/

Practicing matrix multiplication

```
import torch

# Create the input matrices with exact values from the example
X = torch.tensor([[0.1, 0.2, 0.3, 0.4]], dtype=torch.float32)  # Shape: (1, 4)

W = Parameter(torch.tensor([
    [0.11, 0.12, 0.13, 0.14],
    [0.21, 0.22, 0.23, 0.24],
    [0.31, 0.32, 0.33, 0.34],
    [0.41, 0.42, 0.43, 0.44]
], dtype=torch.float32))  # Shape: (4, 4)

# Perform matrix multiplication
result = torch.matmul(X, W)

# Print results with clear formatting
print("Matrix multiplication verification:")
print(f"Input shape: {X.shape}")
print(f"Weight shape: {W.shape}")
print(f"Output shape: {result.shape}")
```



Summary

Matrix multiplication fundamentals

- Matrix shapes determine network architecture
- Input dimension (n,m) maps to weight matrix (m,p) producing output of (n, p)

Implementation in PyTorch

- `torch.matmul` provides direct matrix multiplication
- `nn.Linear` layer add structure for neural networks
- Shape verification prevents common errors (e.g. `tensor_name.shape`)

Beyond linear operations

- Non-linear functions like ReLU enable the learning of complex patterns and must be added to the output of matrix multiplication operations

Further reading and references

torch.matmul

- https://pytorch.org/docs/main/generated/torch.matmul.html

A case study on neural network training

- https://cs231n.github.io/neural-networks-case-study/

GitHub Repository, YouTube Playlist, Discord channel

- Repository
- Playlist
- Discord (**#practical-computer-vision-workshops**)