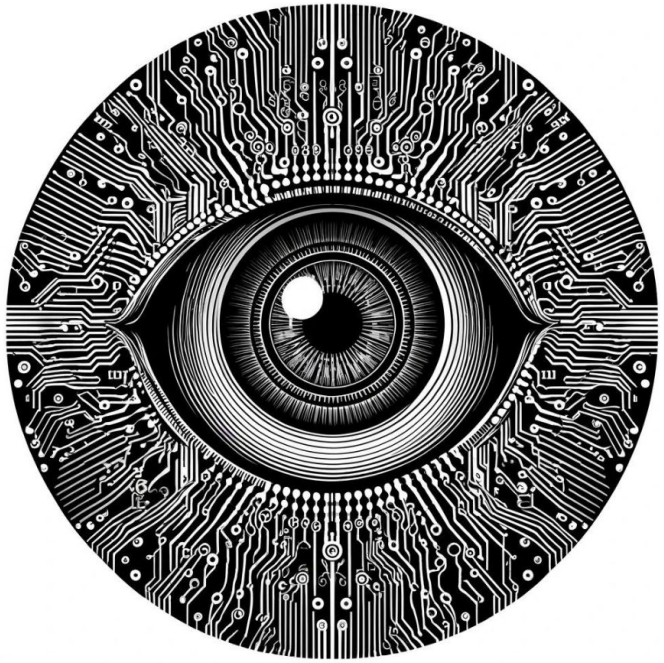
Matrix Multiplication, Non-linear Activations, and Network Shape



Antonio Rueda-Toicen

KI Service Zentrum by Hasso-Plattner-Institut

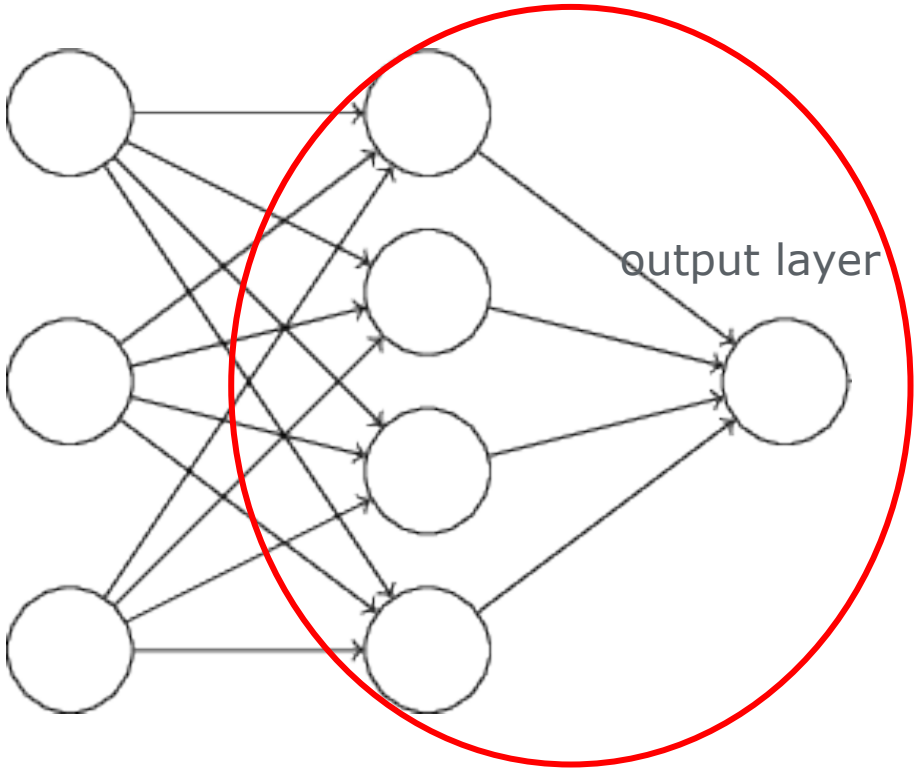
SPONSORED BY THE

Federal Ministry of Education and Research

Learning goals

- 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,
                        bias=False)
```

$$Z = XW$$

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

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

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]
```

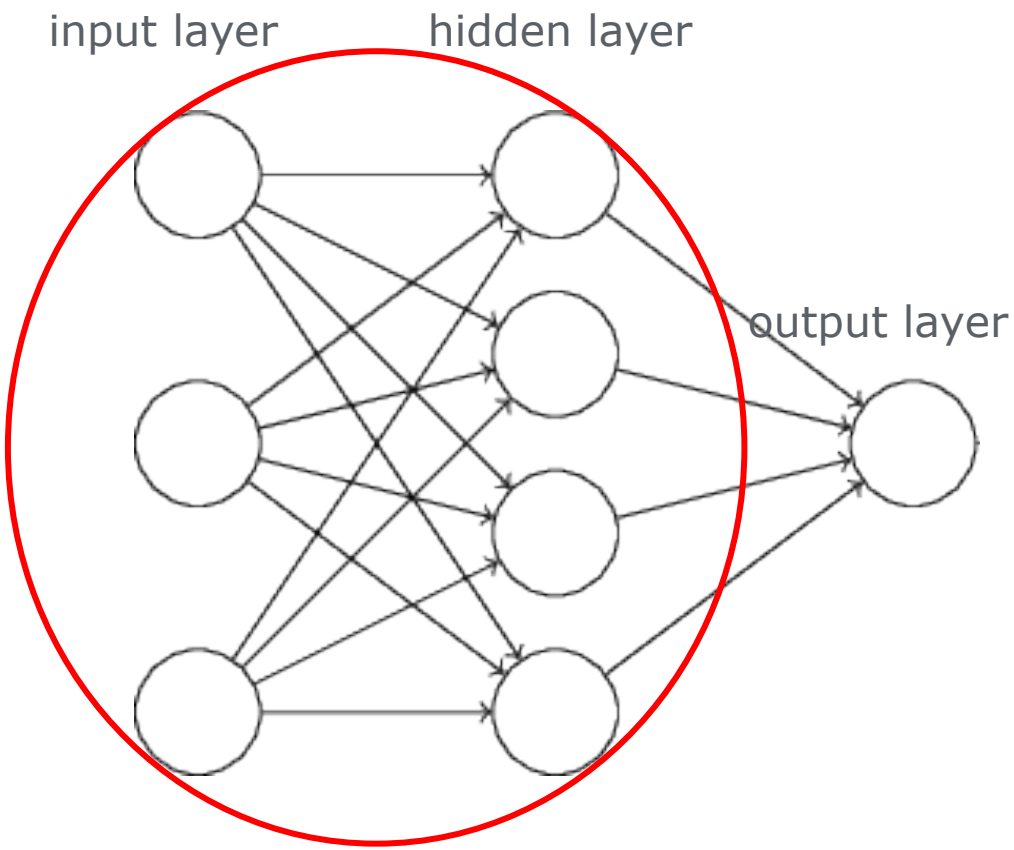
Tip:
always check the shape of tensors with a_tensor.shape or a_tensor.size()

$$Z = XW$$

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

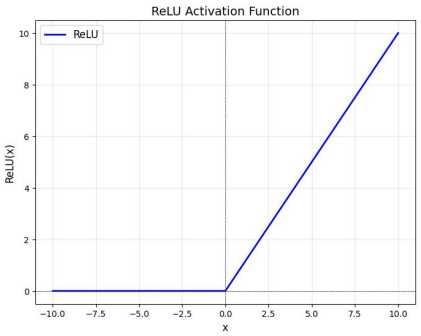
$$Z = X_1 w_1 + X_2 w_2 + X_3 w_3 + X_4 w_4$$

Adding non-linearity between layers



$$Z = ReLU(XW)$$

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

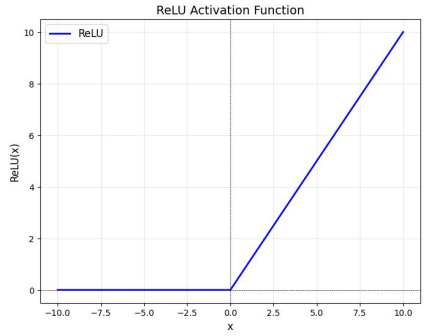


Adding non-linearity between layers

```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()
)
```

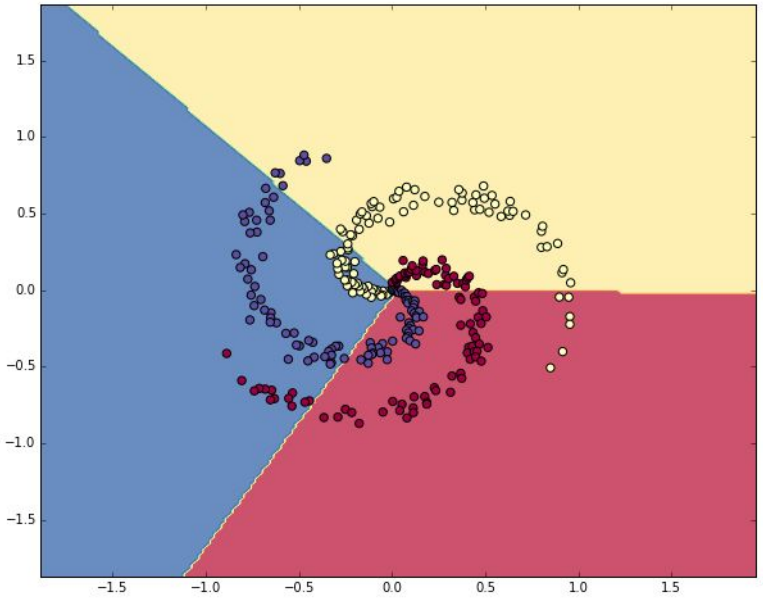
$$Z = ReLU(XW)$$

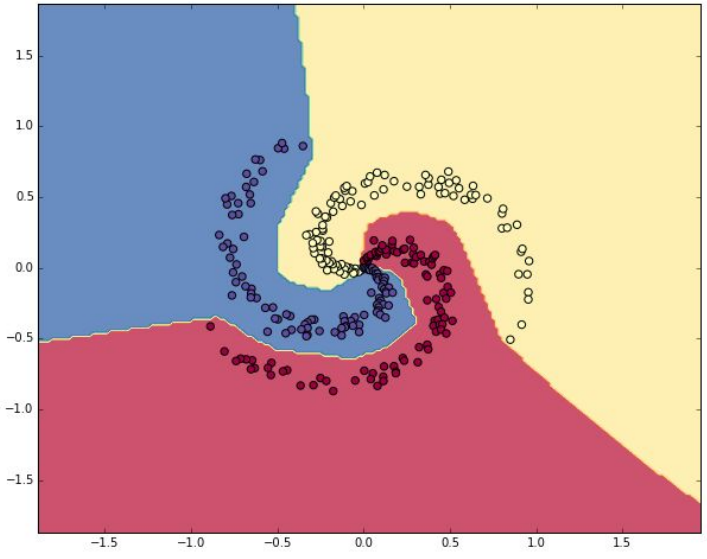
$$Z = \text{ReLU}\left(\begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix} \right)$$



Deep learning is the art of making squiggly functions

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





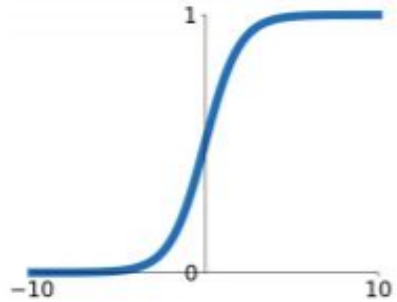
https://cs231n.github.io/neural-networks-case-study
Experiment notebook

Activation functions

ReLU is a good default choice for most problems

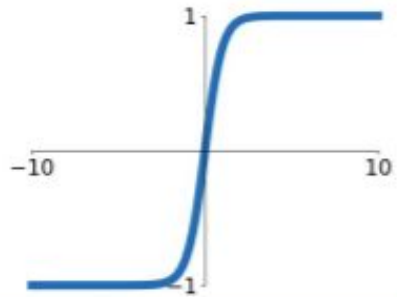
Sigmoid

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



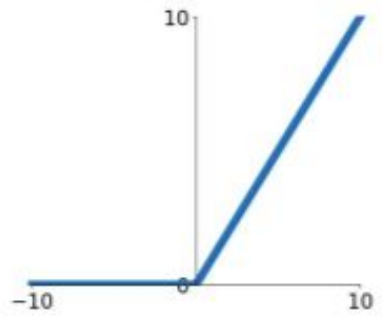
tanh

$$\tanh(x)$$



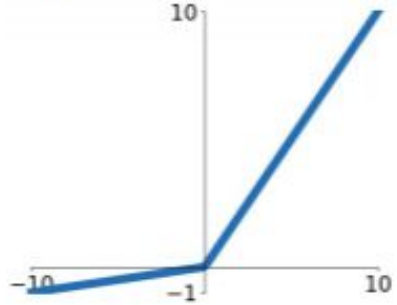
ReLU

$$\max(0, x)$$



Leaky ReLU

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

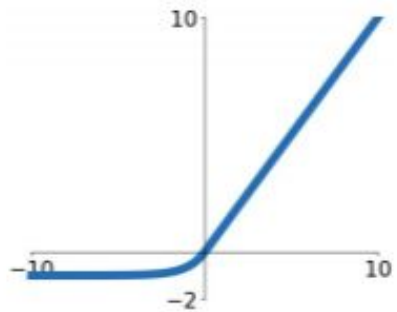


Maxout

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

ELU

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



Practicing matrix multiplication

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

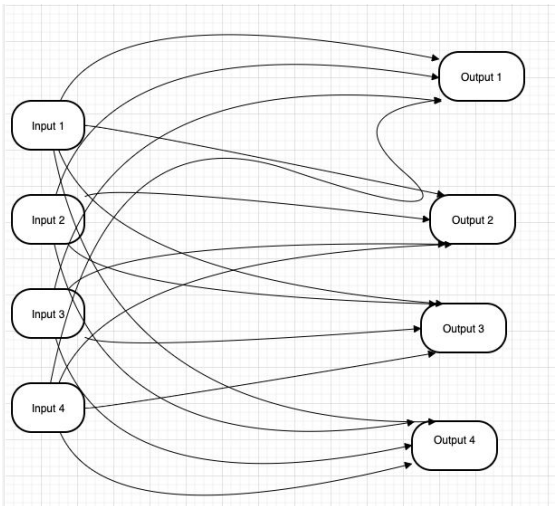
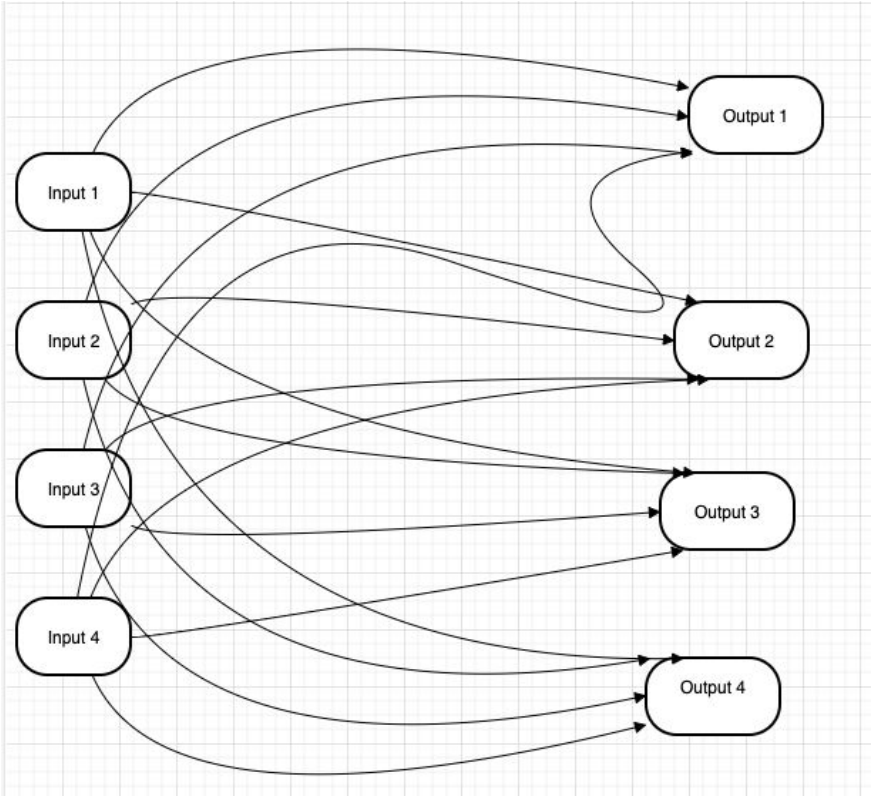


Image from http://matrixmultiplication.xyz/

Practicing matrix multiplication

```python
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

SPONSORED BY THE
Federal Ministry of Education and Research

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/

SPONSORED BY THE
Federal Ministry
of Education
and Research