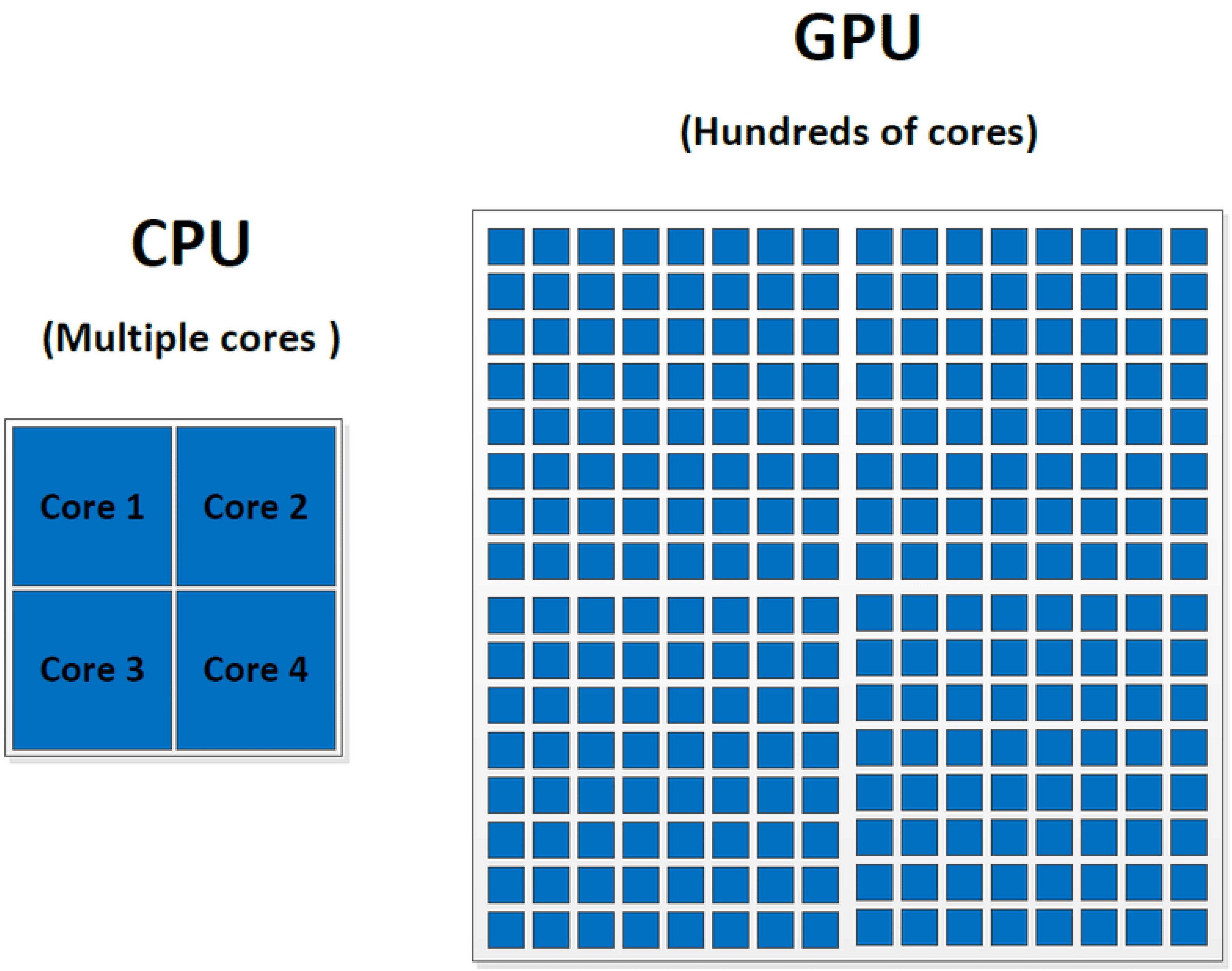
Deep Learning Introduction

Prepared By:

H. Fuat Alsan

Deep Learning Introduction

- High performace computing (HPC)
- High dimentional (tensor) data
  - (3, 128, 128) -> RGB 128x128 resolution image
- Lots of differential and linear algebra operations
- Hardware accelerators such as: GPU, TPU, custom tensor processors


GPU
(Hundreds of cores)
CPU
(Multiple cores)
Core 1
Core 2
Core 3
Core 4

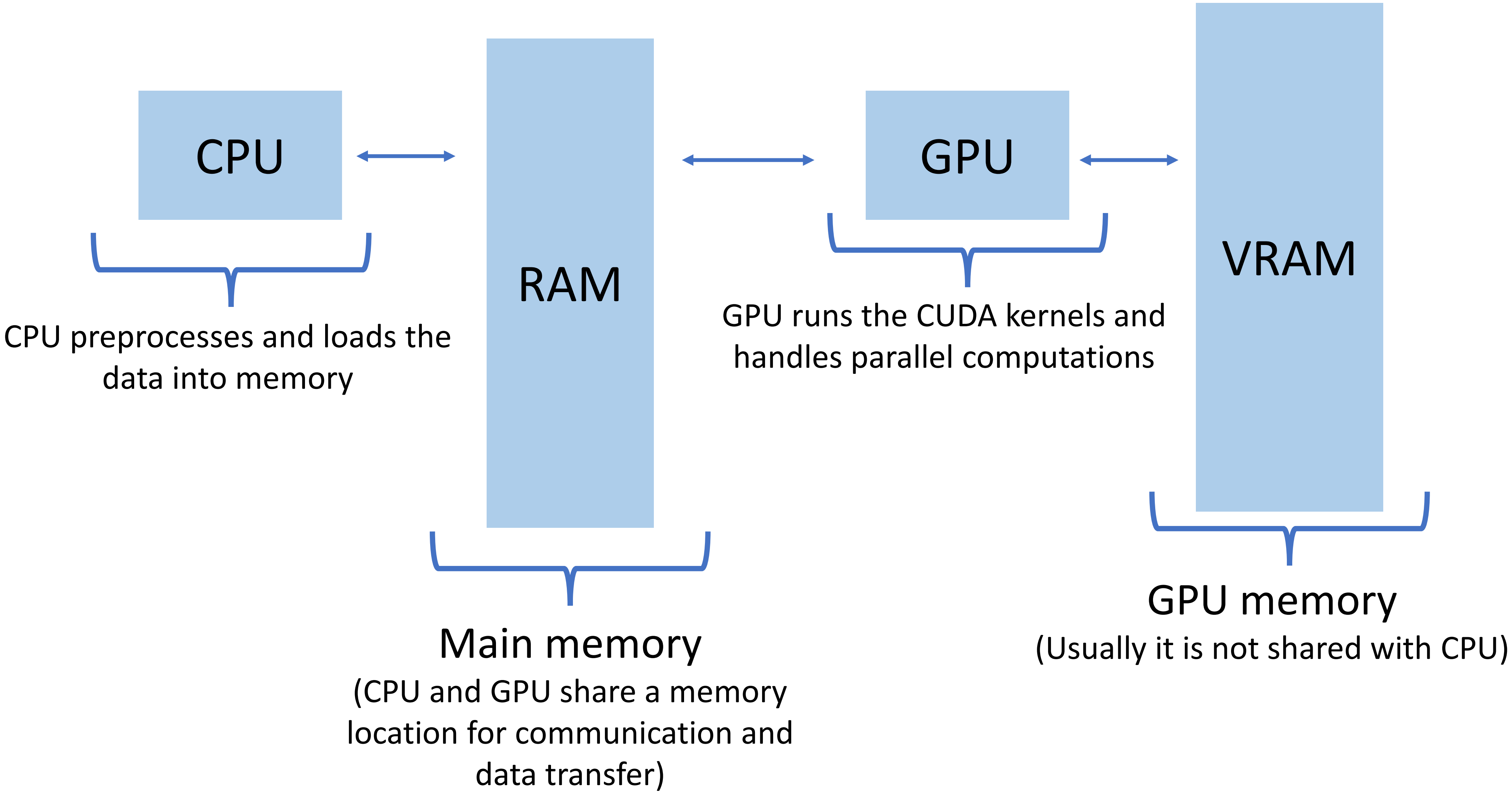
GPU Basics

- Shader
  - Geometric math operations
- GPGPU
  - General-purpose computing on graphics processing units
- CUDA Cores
  - Number of processing cores in GPUs (NVIDIA)
- CUDA Kernel
  - Small programs that run in CUDA cores. Used for parallel computing
- Video RAM (VRAM)
  - Amount of dedicated GPU memory

Other GPGPU Alternatives

- OpenCL (Open Computing Language)
- AMD ROCm (Radeon Open Compute)
- Apple MPS (Metal Performance Shaders)
  - Usually used in M1 and M2 macs
- Intel IPEX (Intel Extension for PyTorch) (Both CPU, GPU)
  - Best used with Xeon data center class CPUs

Deep Learning Pipeline



Deep Learning Pipeline (cont.)

1. Read the data, preprocess it and load into the RAM
2. Move the model and data to the VRAM
3. Do the computations described in model layers using GPU and VRAM
4. After operations are finished, move the results back to the RAM (main memory)
5. (Note: both data and model should be on VRAM for GPU computing or in RAM for CPU computing)
   - data.to('cuda')
   - model.to('cuda')
   - data.to('cpu')
   - model.to ('cpu')

PyTorch

- Very Pythonic
- Efficient Tensor operations
- Autograd
- Ready-to-use deep learning layers and functions
- Handles complex CUDA operations for us
- Uses cuDNN (CUDA Deep Neural Network)

PyTorch Hierarchy

- **nn.Module** is the base class for everything in PyTorch
- nn.Module2 **(contains)** nn.Module1 **(contains)** nn.Parameter **(contains)** tensors
- Commonly used layers:
  - nn.Linear
  - nn.Conv2d
  - nn.ConvTranspose2d
  - nn.ReLU
  - nn.Sigmoid

Datasets

- In deep learning we use very large datasets
- Ex: ImageNet
  - Millions of images, approx. 150 GB of data
- Unfortunately the whole dataset can’t fit into RAM or VRAM
- We split the dataset into smaller chunks called «**batch»**
- We only load batch into RAM and VRAM

Dataset Bacthing

- Assume we have 10000 datapoints
- Let’s say we want 32 as batch size:
  - Each batch would have 32 data points
  - We would have 10000 // 32 = 312 batches
  - But we can’t equally divide 10000 by 32, so there is a last batch
  - Last batch would have 10000 % 32 = 16 datapoints
  - So actually we have **312 batch of 32 datapoints** and **a single batch of 16 datapoints**
  - Total 313 batches
  - Dataloader of PyTorch has **drop_last** option to ignore the last (unequal) batch

Stochastic Gradient Descent

- Normally we would compute the gradients **over all the dataset**
- However, in batch processing we don't access all the data
- So instead we compute the gradients over the batch data
- This is called **stochastic** gradient descent
- NOTE: Size of batch can effect learning.
  - Smaller batch size: longer training time, risk of overfitting
  - Larger batch size: more memory consuption, risk of underfitting
  - Still debated in deep learning research
  - Ex: Batch size is usually 1 for Pix2Pix (one to one mapping)

Example Data: Grayscale Image

- $\begin{matrix} 12 & 255 & 56 \\ 8 & 255 & 15 \\ 255 & 255 & 255 \end{matrix}$

- 3 by 3 grayscale image (each value is a pixel)
- Image is read as uint8 (unsigned 8-bit integer)
  - Values are between 0-255
- Converted to tensor and float32
  - Float32 is the default for deep learning models

Flatten Operation

- Converts N dimentional tensor to single dimensional tensor
- Original:
- $$\begin{matrix} 12 & 255 & 56 \\ 8 & 0 & 15 \\ 255 & 255 & 255 \end{matrix}$$

- Flatted:
- $12 \quad 255 \quad 56 \quad 8 \quad 0 \quad 15 \quad 255 \quad 255 \quad 255$

FlatNet

- Each pixel value have a seperate weight

$$[12, 255, 56, 8, 0, 15, 255, 255, 255] \quad * \quad \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \\ W_8 \\ W_9 \end{bmatrix} \quad + b$$

$*$: dot product
b: bias

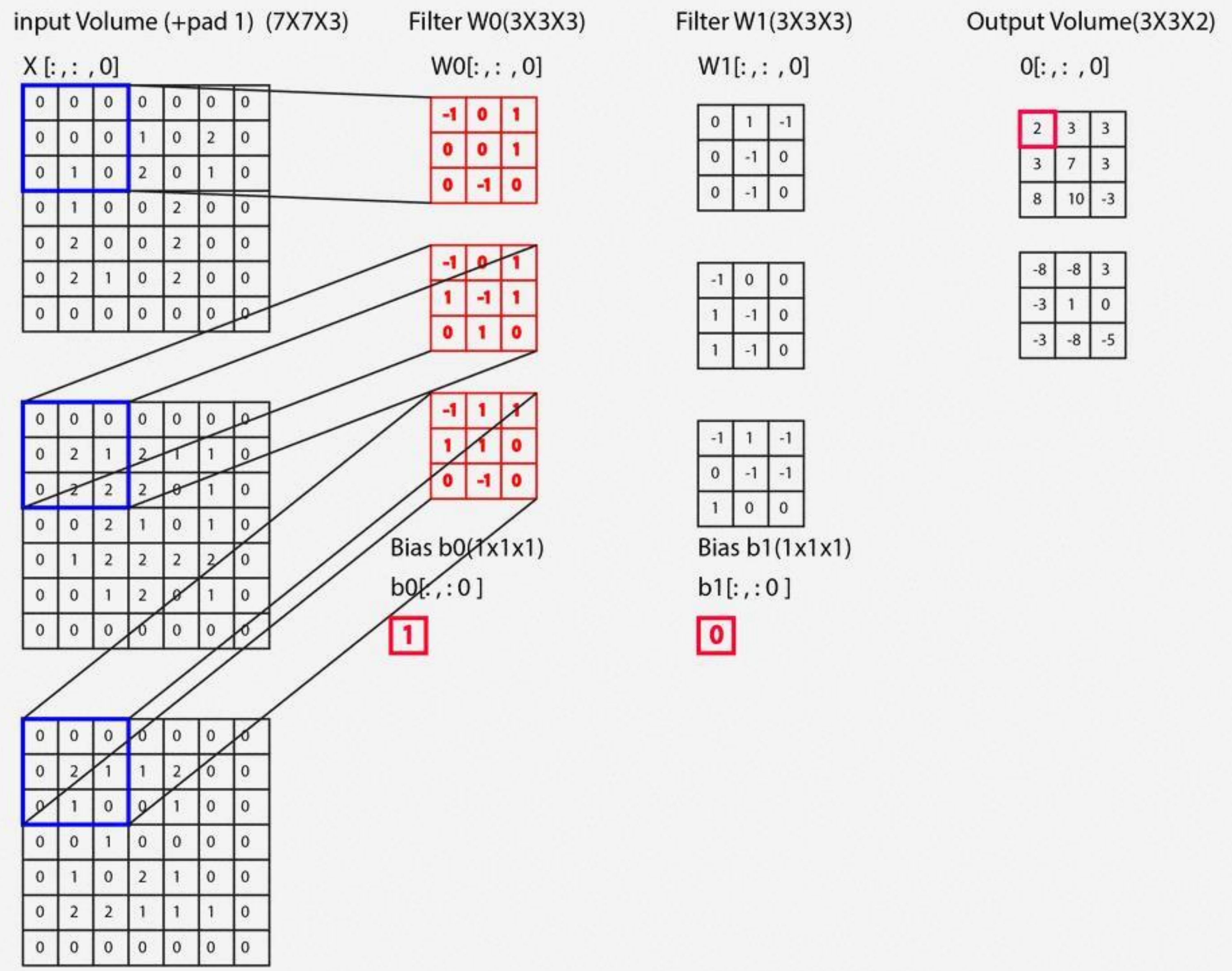
Result:

$[W_1 12 + W_2\ 255 + W_3\ 56 + W_4\ 8 + W_5\ 0 + W_6\ 15 + W_7\ 255 + W_8\ 255 + W_9\ 255 + b]$

(Note: here the results is a scalar but in practice we use matrix multiplication and get a vector as a result)

Convolution Operation

- Uses filters instead of flat pixel values
- Filters have learnable weights and biases
- Filters move on image data and extract features
  - Edge detection
  - Color change
  - Etc.
- Unlike flat linear operations, filter weights shared



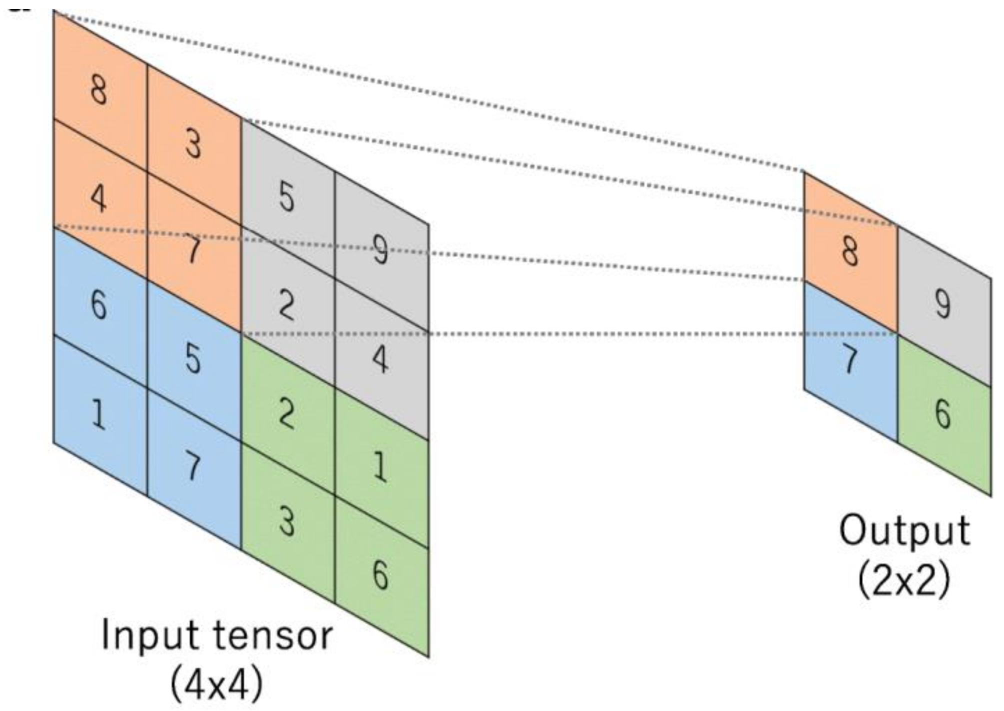
input Volume (+pad 1) (7X7X3)
X [:, :, 0]
Filter W0(3X3X3)
W0[:, :, 0]
Filter W1(3X3X3)
W1[:, :, 0]
Output Volume(3X3X2)
0[:, :, 0]
Bias b0(1x1x1)
b0[:, :0]
1
Bias b1(1x1x1)
b1[:, :0]
0

Convolution Visualized

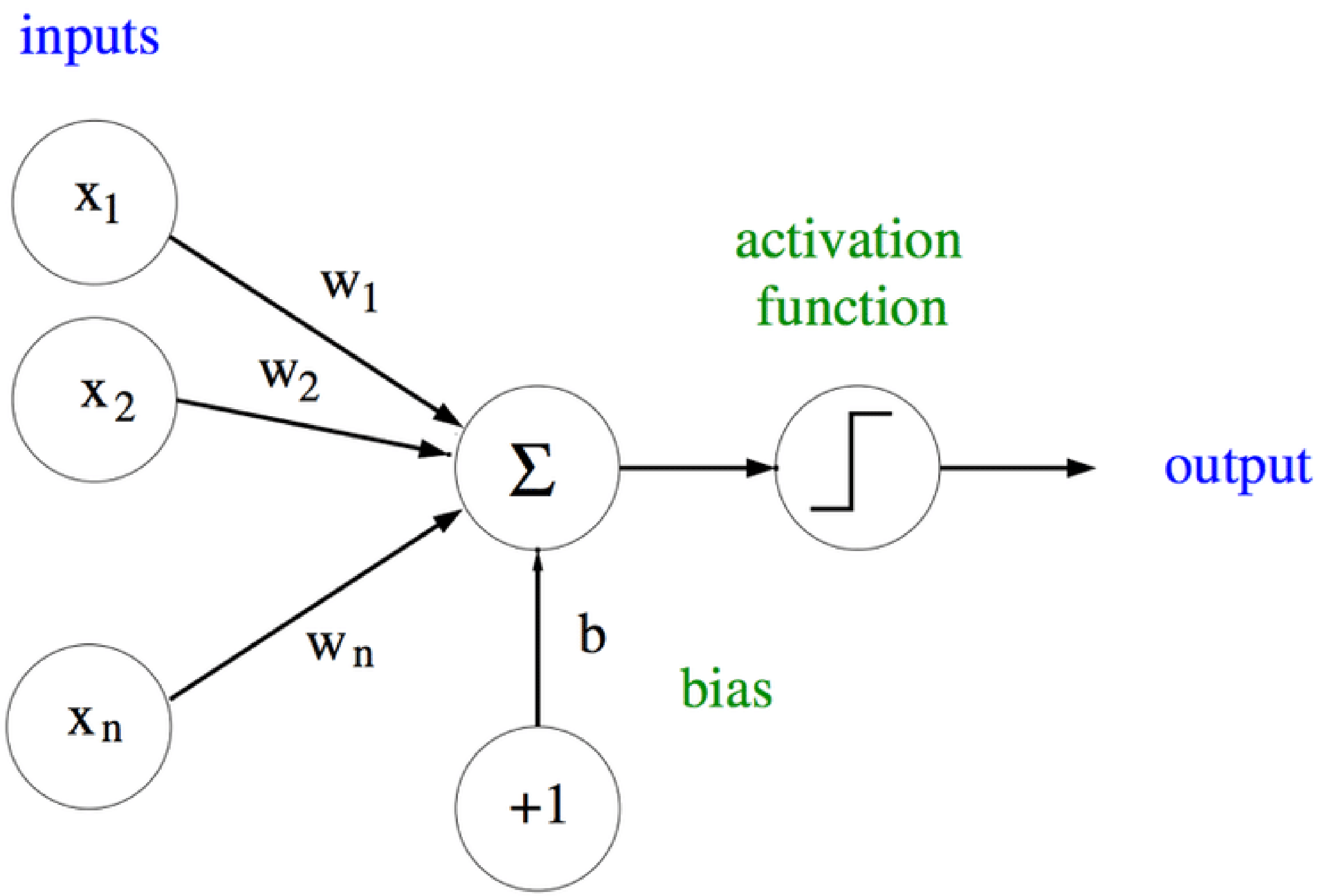
- https://ezyang.github.io/convolution-visualizer/
- https://github.com/vdumoulin/conv_arithmetic
- Input filter count (for RGB, 3)
- Output filter count
- Kernel size
- Padding
- Stride
- Dilate

Max Pooling

- Downsampling of image features
- **Rare case: nn.Module with no learnable parameter!**
- Ex with 2x2 kernel size:



Non-Linearity with Activation Functions

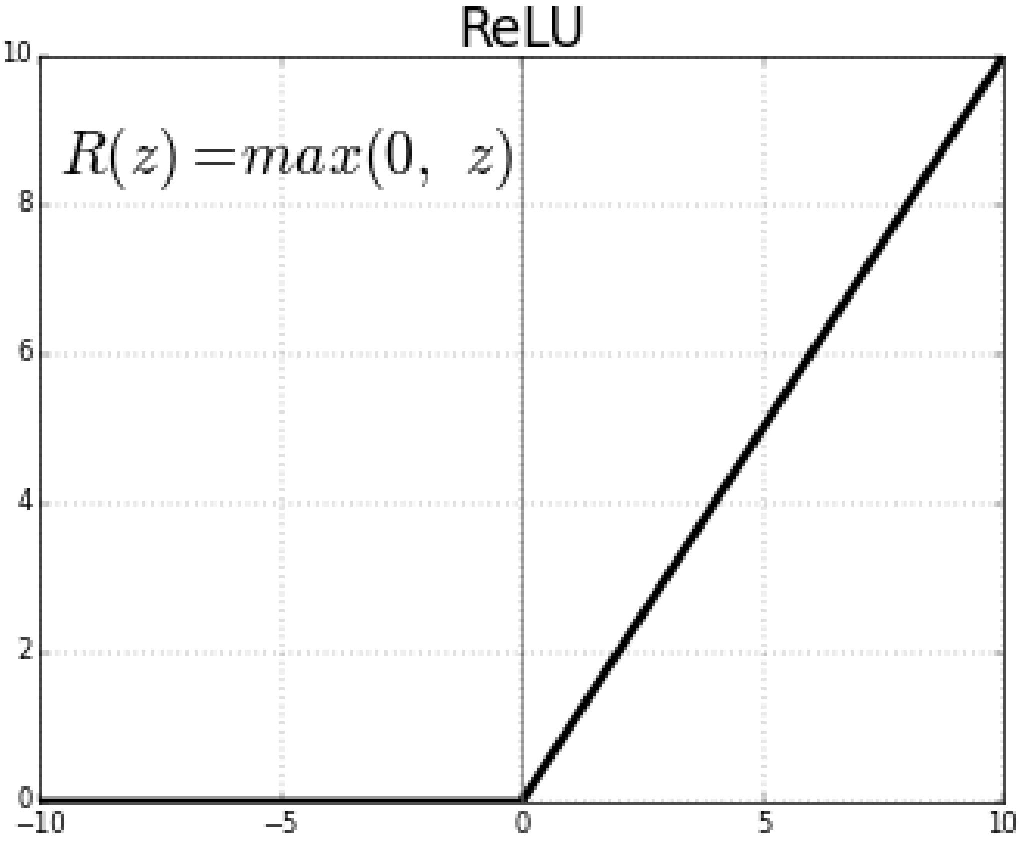


Why Non-Linearity?

- Linear functions (like nn.Linear) can only create straight lines, but many real-world problems have curved or non-linear patterns
- It allows neural networks to model complex, non-linear relationships in data
- See: XOR problem
- Also image a neural network with only weights:
- $W_i^3\left(W_i^2\left(W_i^1(x_i)\right)\right) \approx W_i(x_i)$
- **It doesn't matter how many layers the model have!!**

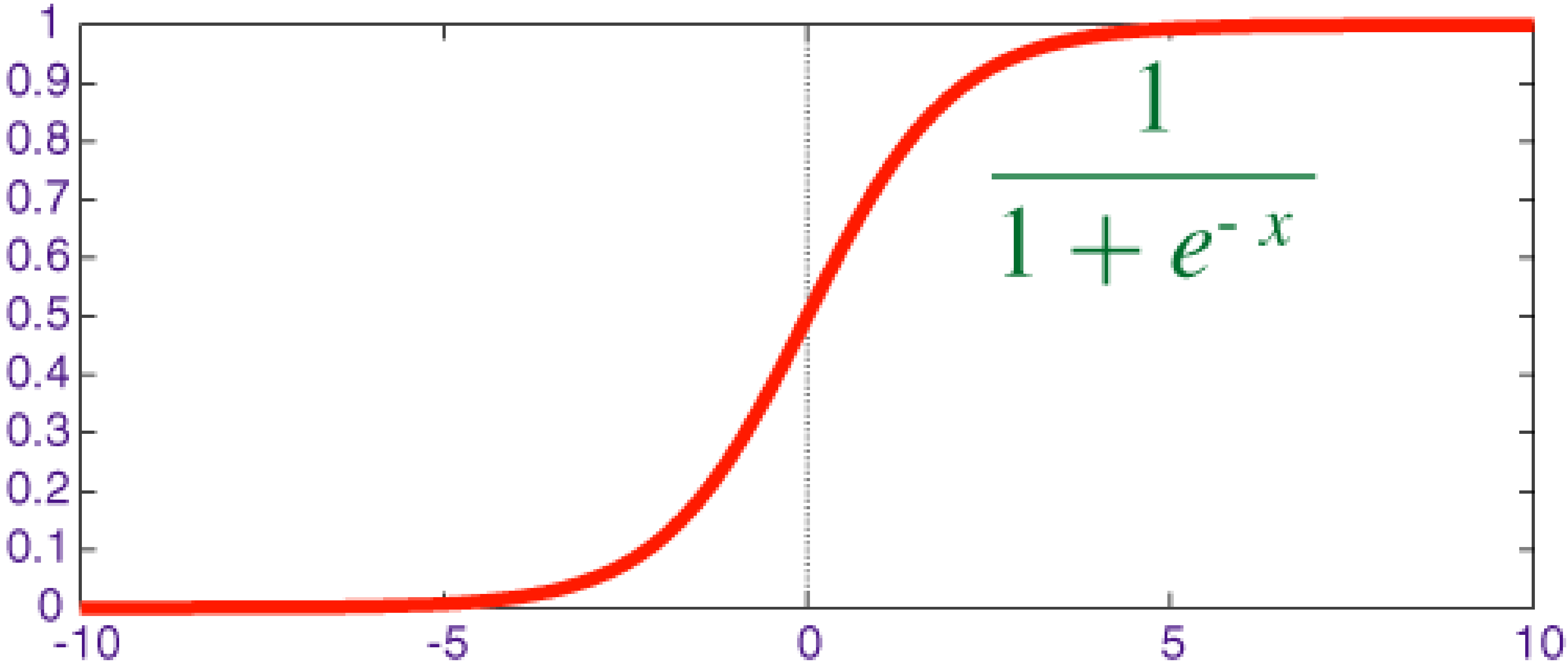
Rectified Linear Unit (ReLU)

- Simple and popular choice for non-linear activation



Sigmoid

- Another popular activation function
- Usually used for squashing values in the range of [0.0, 1.0]



Classification Model

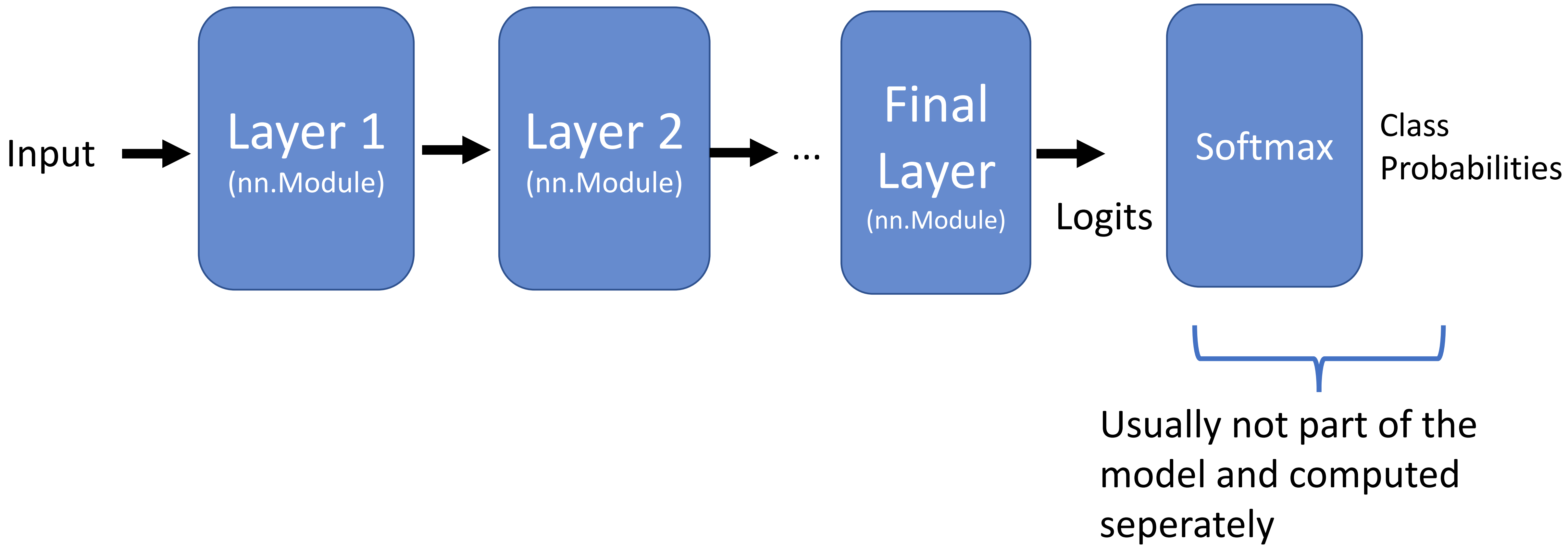
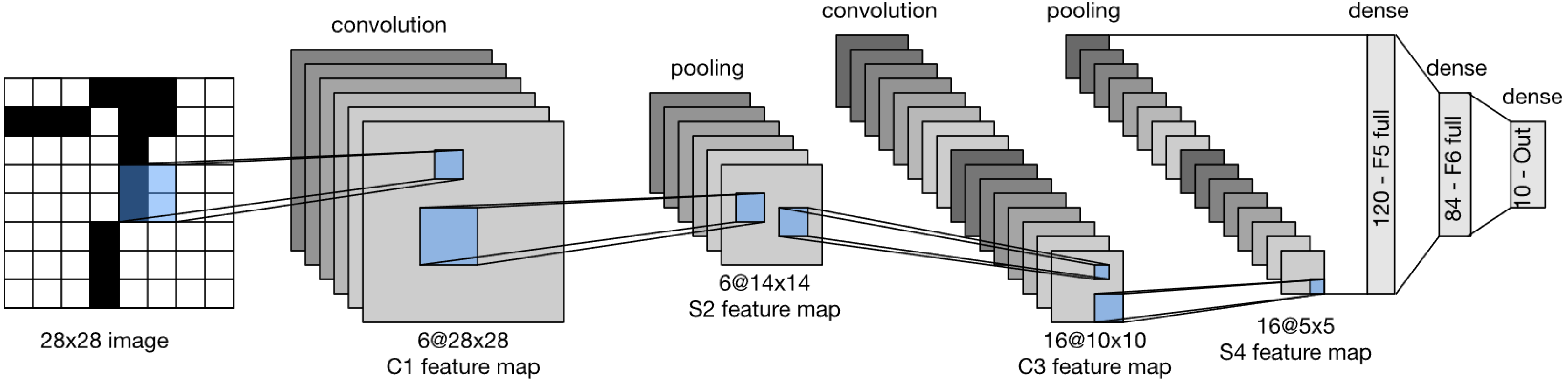
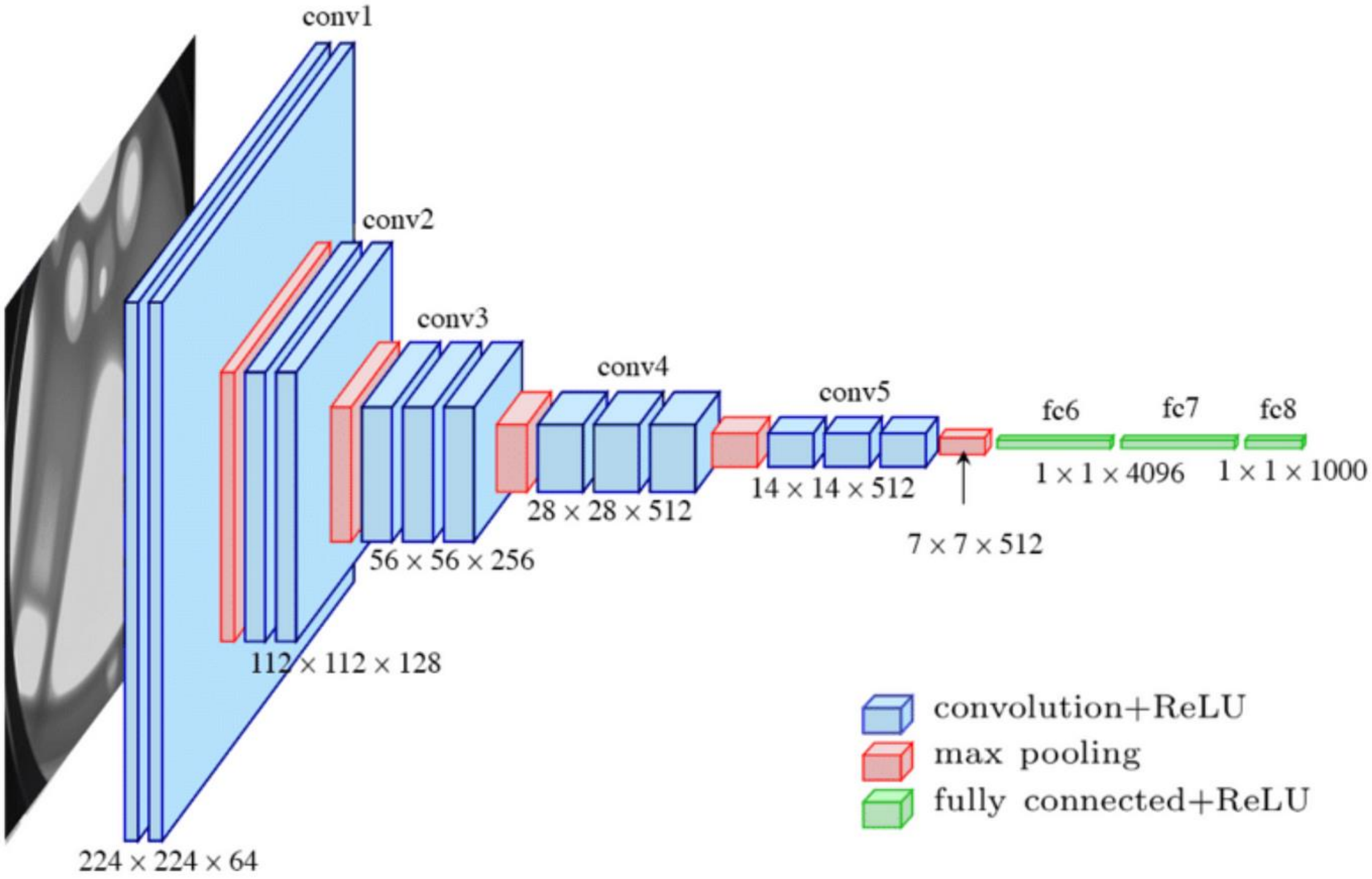


Image Classifier for MNIST Dataset



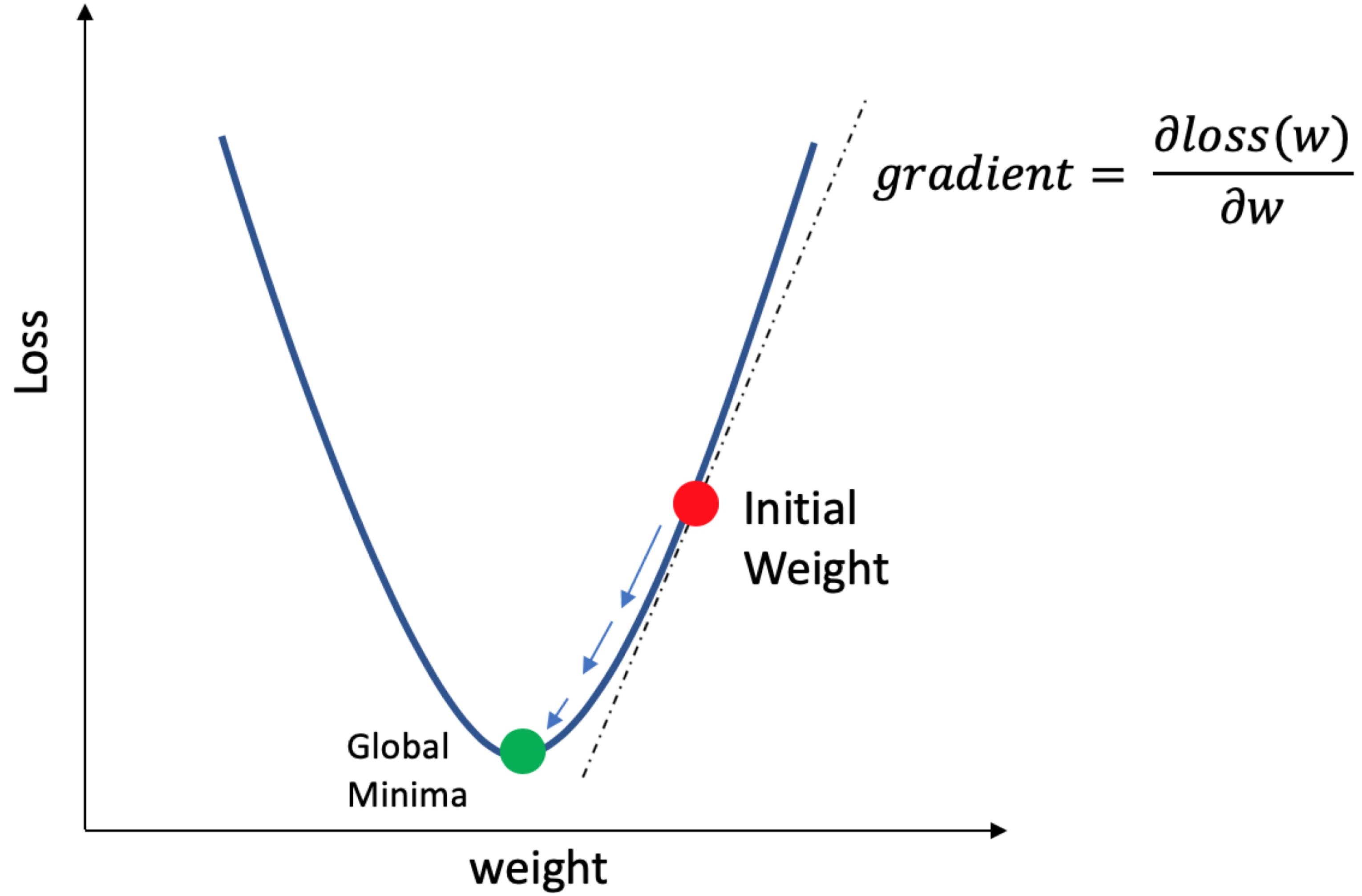
A Modern Image Classifier (VGG16)



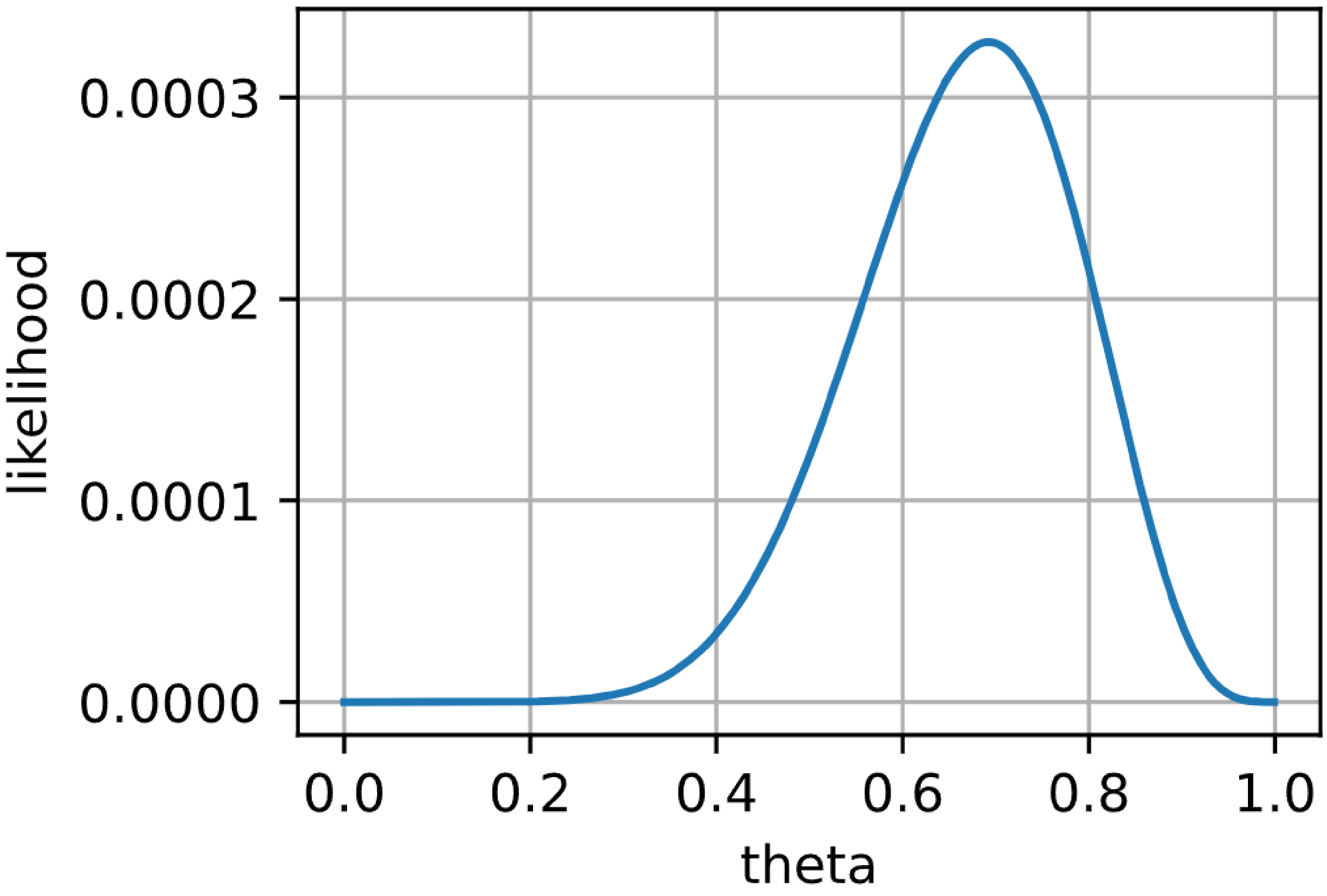
Training Classifiers

- In regression we had continuos values, we used MSE
- However, this is not the case in classification
- In classification, we have fixed classes
  - Each class is represented by an ingeter ID
  - For ex: Cat/Dog classifier
  - Cat -> 0
  - Dog -> 1
- In classification, we use a loss function called **cross entropy**
  - Actually it is **negative log-likelihood** loss

Loss vs Likelihood



Loss vs Likelihood



Multiclass Classification

- Binay Case:
- y_i: class id (0 or 1, each indicating a class)
- $\hat{y}_i$: probability of class (predicted by the model)

$$\text{Loss} = -\frac{1}{\substack{\text{output}\\\text{size}}} \sum_{i=1}^{\substack{\text{output}\\\text{size}}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

Multiclass Classification (Cont.)

$$logloss = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij}\log(p_{ij})$$

- N is the number of rows
- M is the number of classes