Introduction to Deep Learning

Keith T. Butler

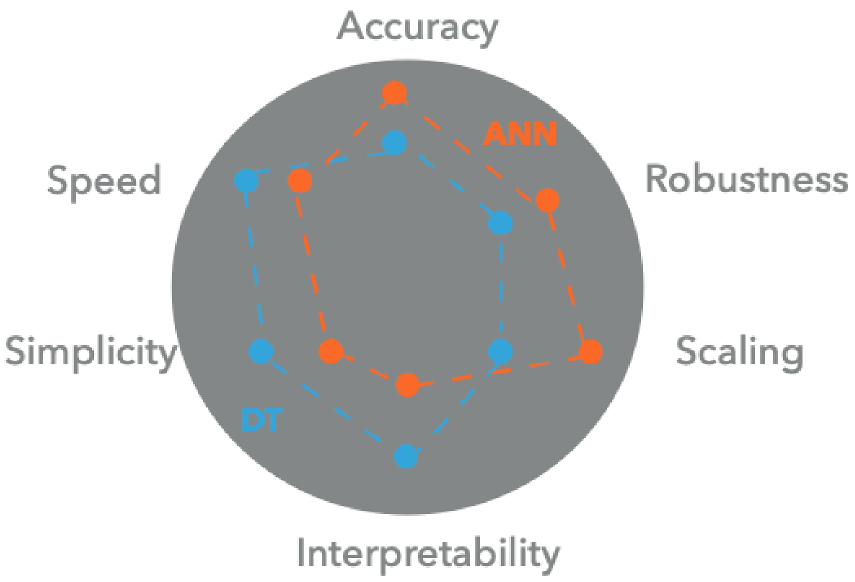
mdi-group.github.io @keeeto2000 k.t.butler@ucl.ac.uk

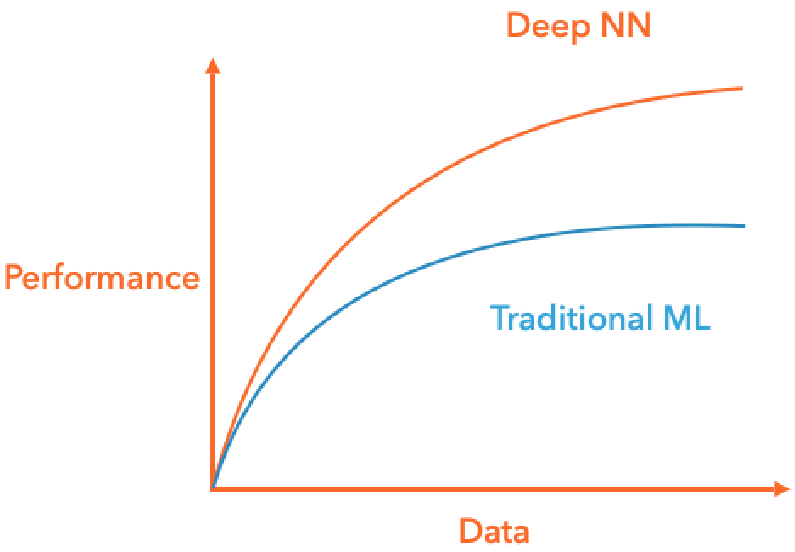
What we will cover

- The difference between deep and classical learning
- The concept of representation learning
- The structure of a simple multi-layer perceptron
- How to write an MLP in PyTorch
- How a NN learns – optimisation and backpropagation
- The power of inductive bias
- The structure of a simple convolutional neural network

Classical/deep methods

- Classical: linear regression, trees etc..
- Deep: neural network type models





Deep learning as representation learning

“Hand-crafted” features

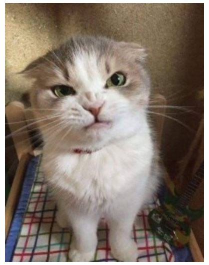
Classical ML $$f(x) = y$$

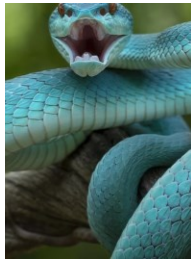
Deep learning $$f(f(\Theta)) = y$$

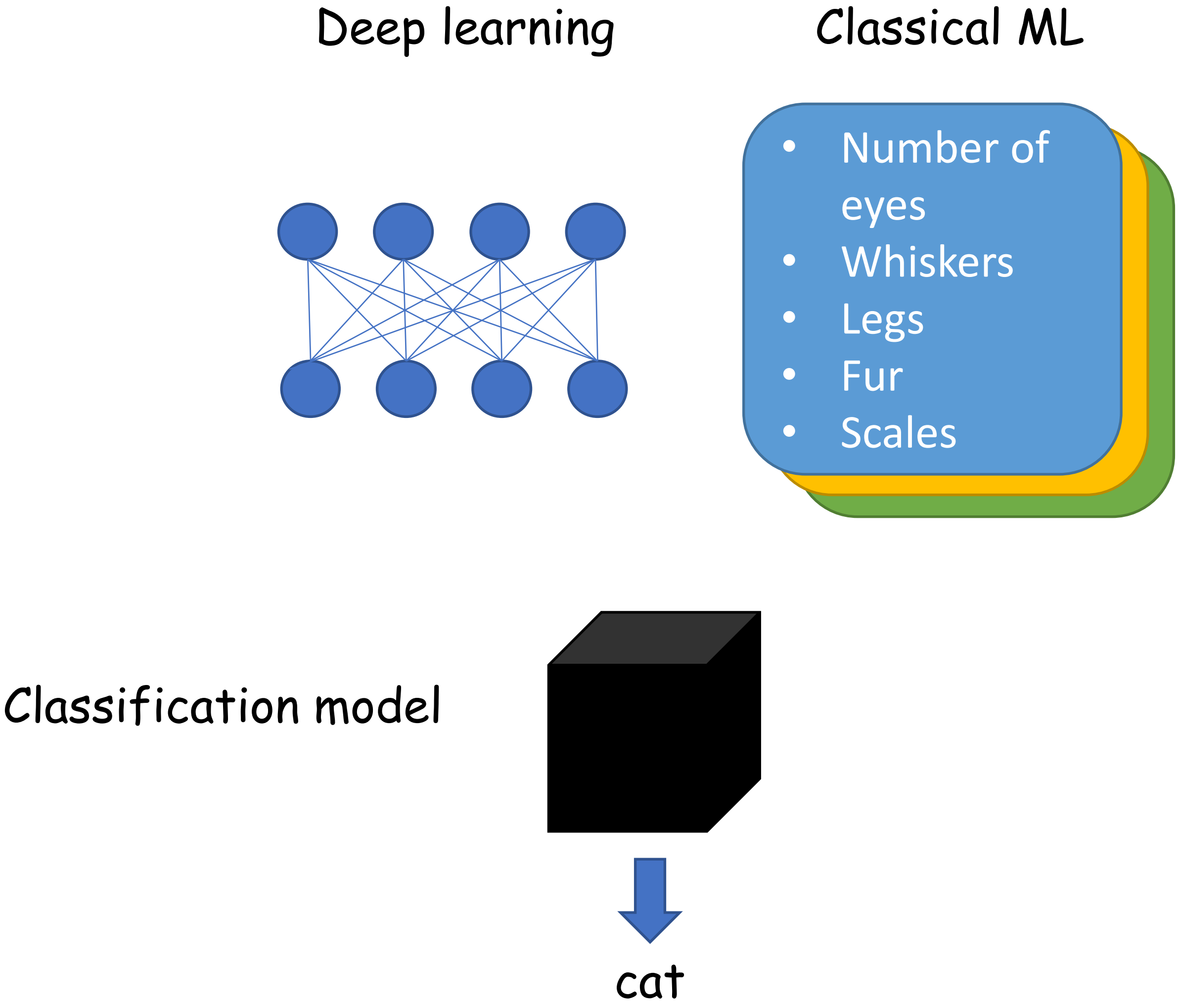
Unstructured data

Deep learning as representation learning

- Traditional ML relies heavily on feature engineering before learning
- Deep learning learns the features as well as the relational model of interest
- Deep learning requires less manual input; but more data





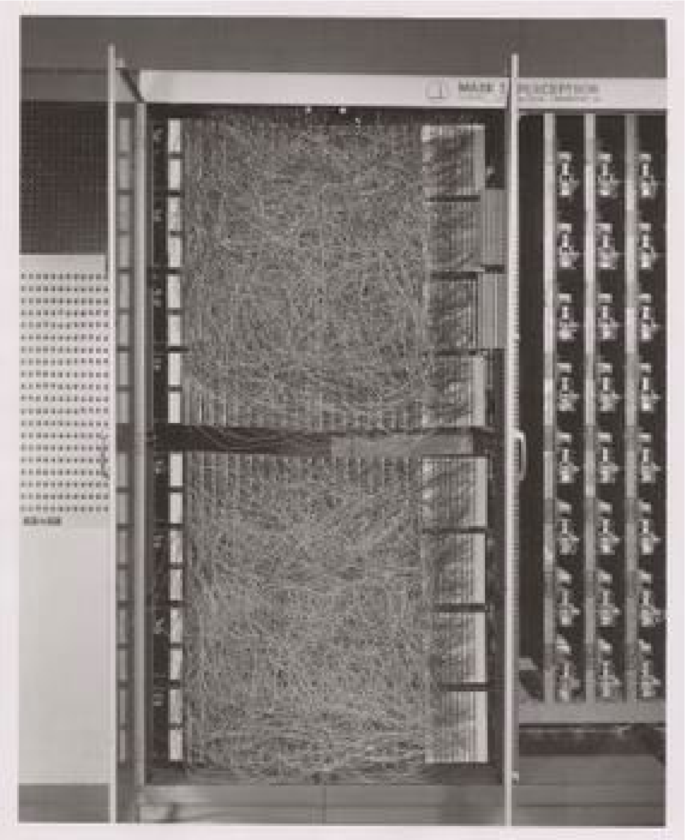


Neural networks

- Early NN
- Originally a device
- Intended for binary classification

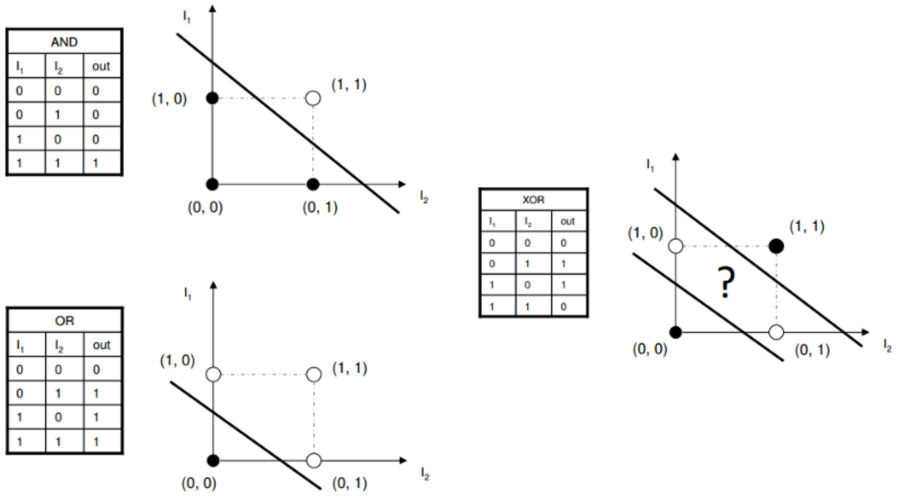
$$y = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^T \mathbf{x} + b)$$

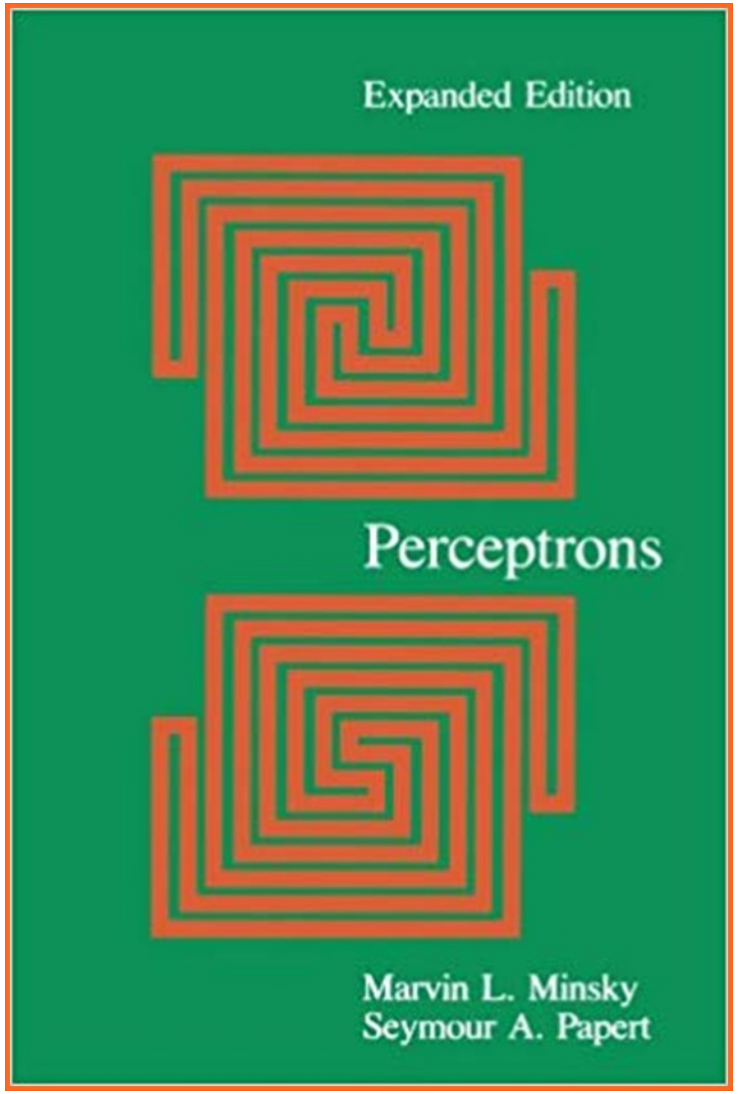
- Produces a single output from a matrix of inputs, weights and biases



Neural networks

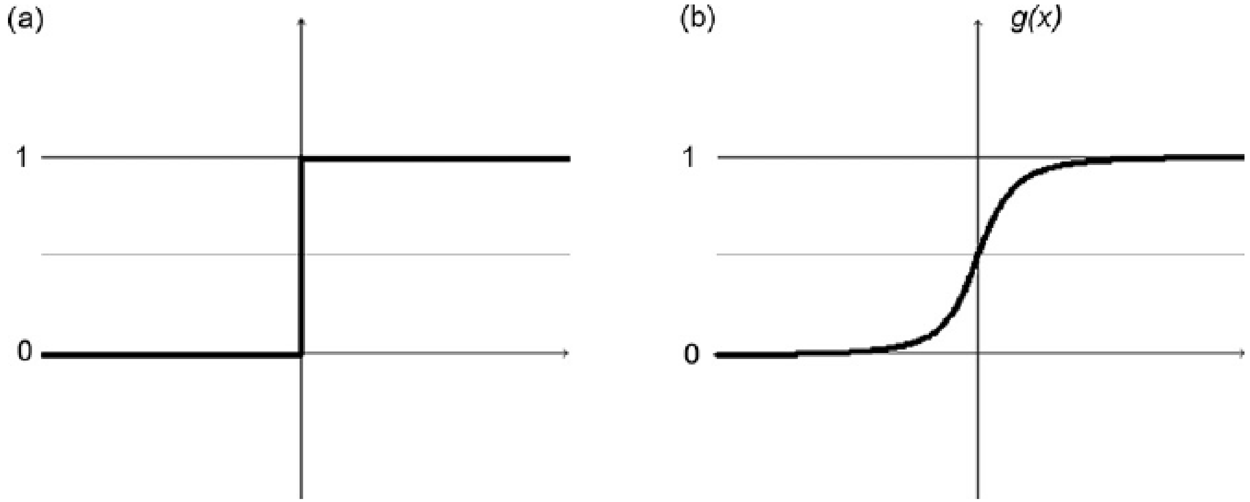
- Single layer
- Minsky and Papert showed they could not solve non-linear classification





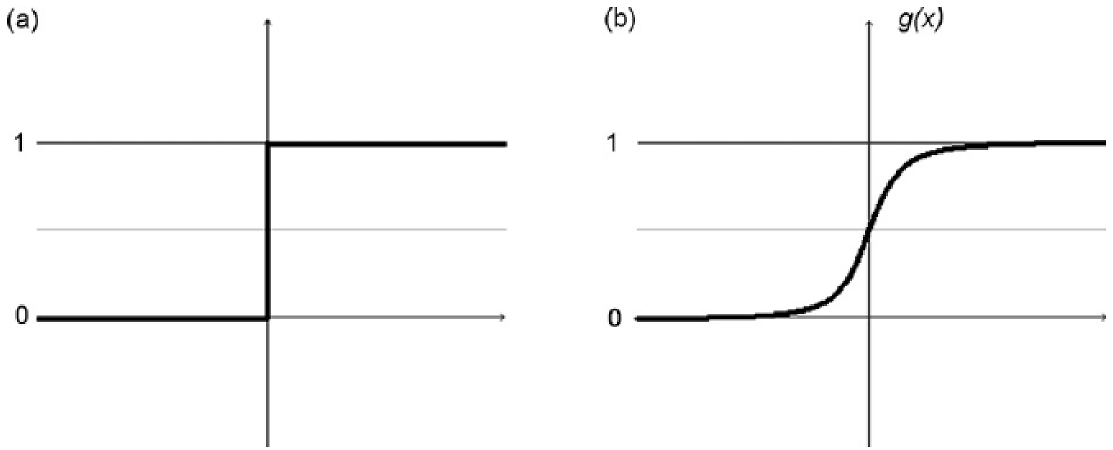
Change of function

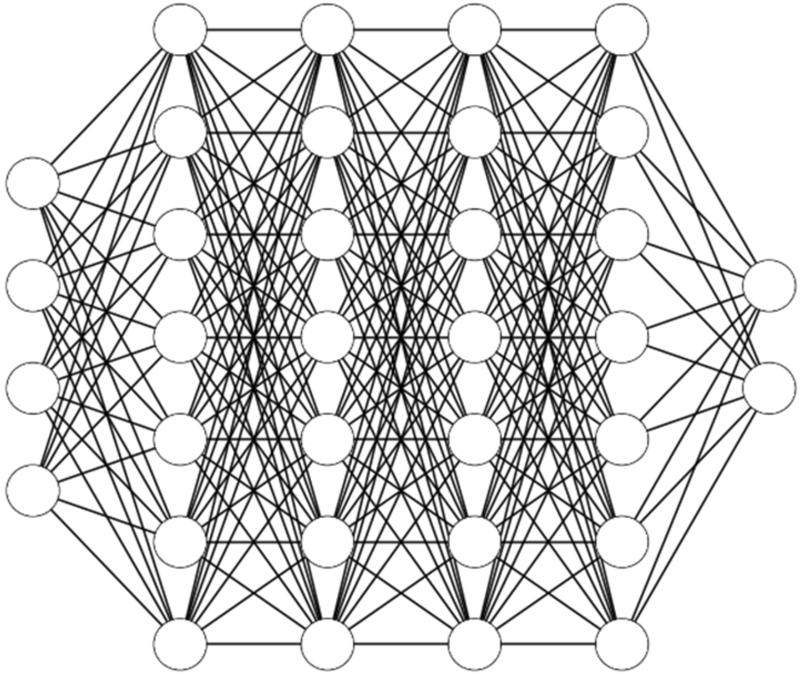
$$y = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^T \mathbf{x} + b)$$



Neural networks

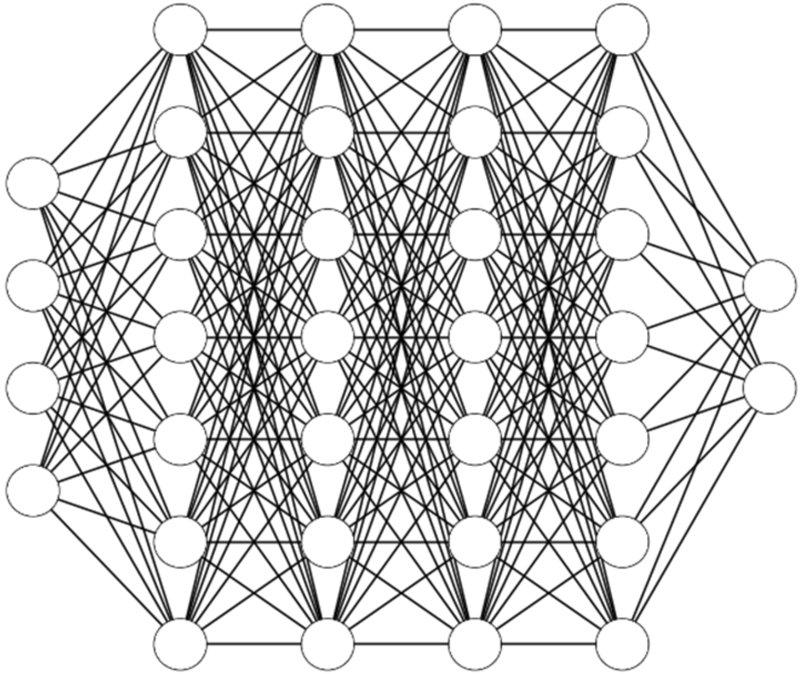
- Back propagation
- Now gradients could be used to minimise error
- Modifications back propagate through the network using the chain rule





Deep neural networks: Multi layer perceptron

Input layer

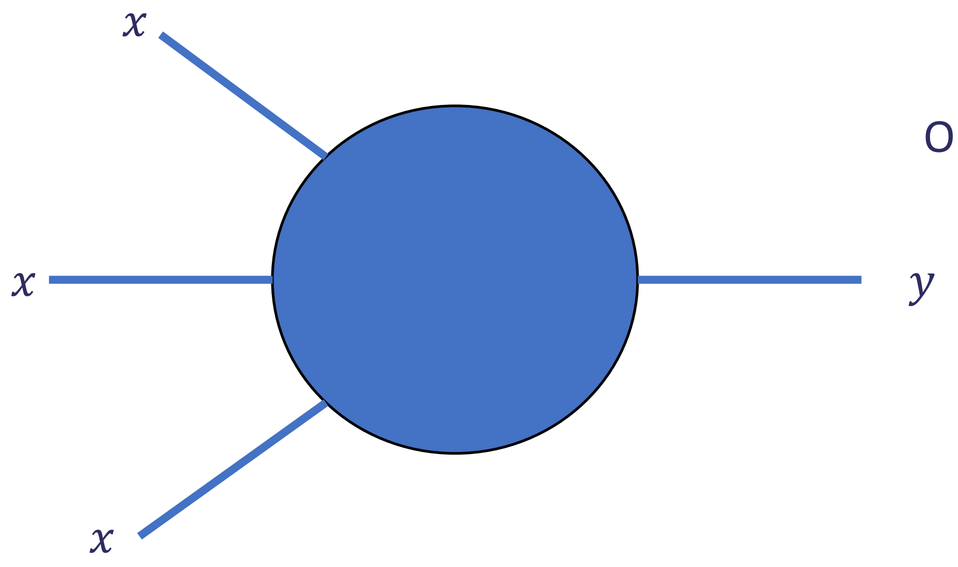


Output layer

Hidden layers

Dense layers

- Also called fully connected layers



Activation function

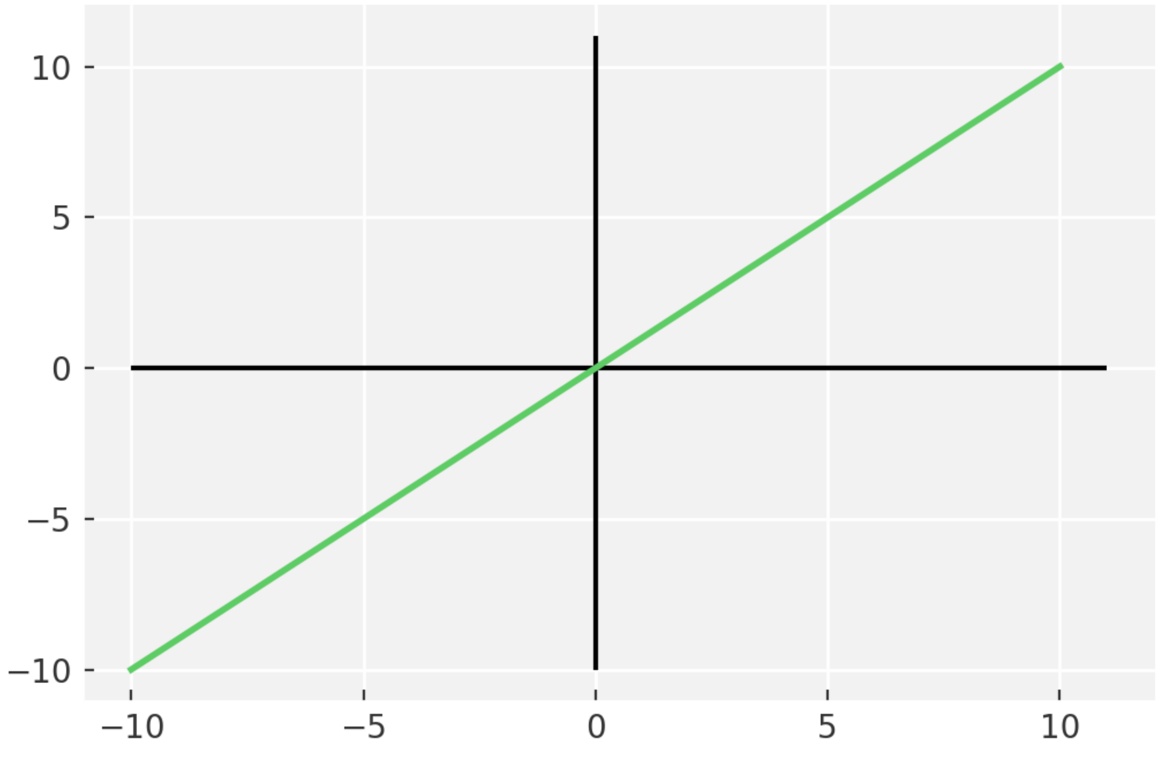
Output signal $y = g(z)$

$$z = \boldsymbol{w}^T . \boldsymbol{x} + b$$ Bias

Weights Input signals

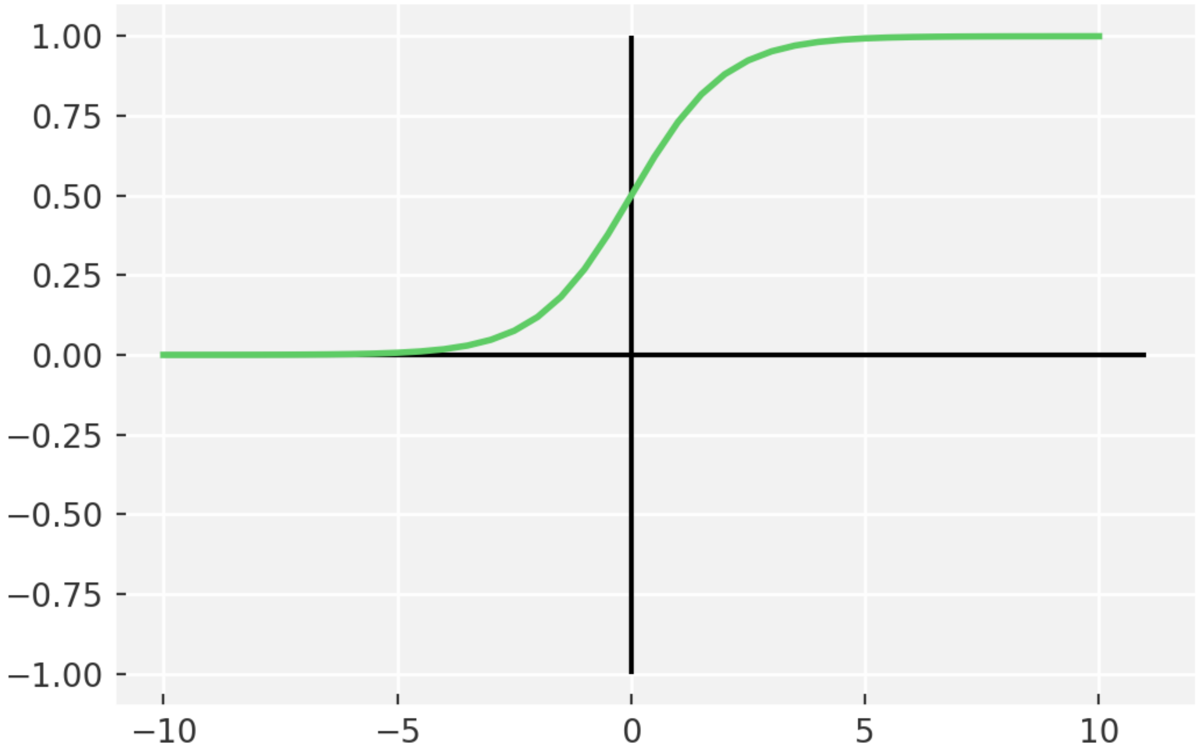
Activation function: Linear

- The simplest form of activation function



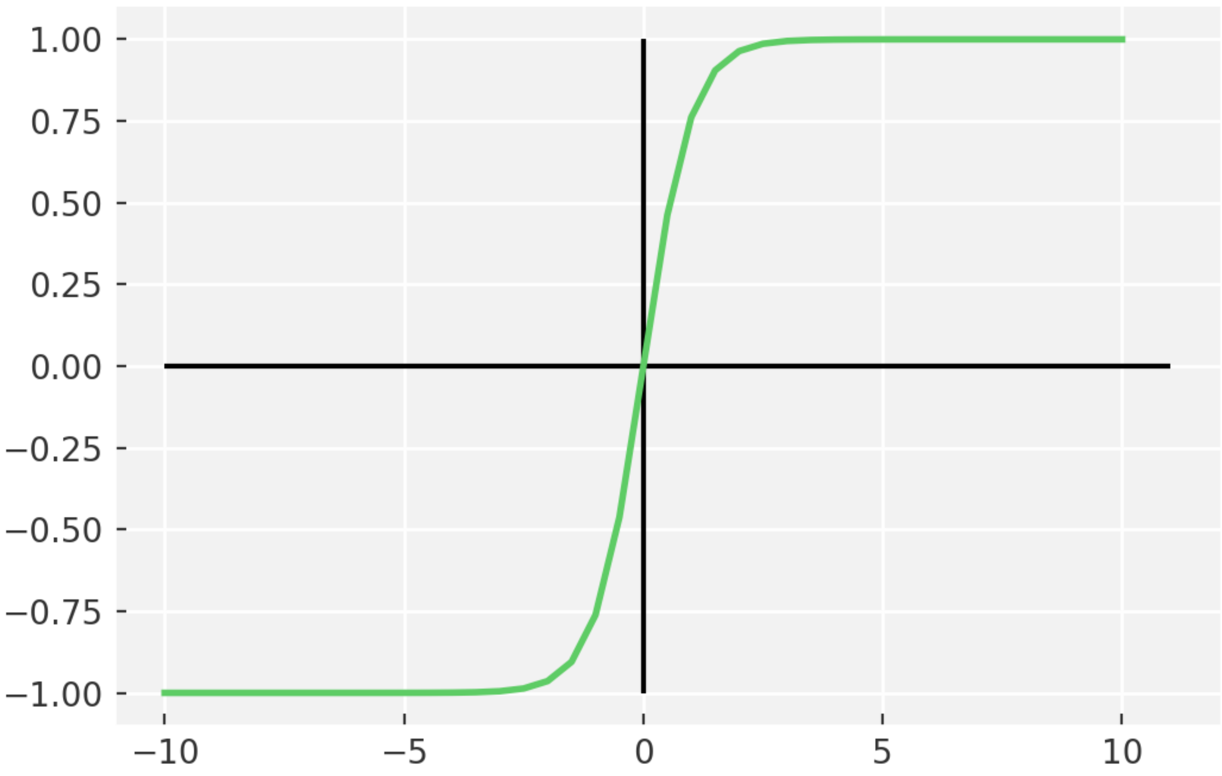
Activation function: Sigmoid

- Vanishing gradient problem
- Secondly , its output isn’t zero centered. It makes the gradient updates go too far in different directions. **0 < output < 1, and it makes optimization harder.**
- Sigmoids saturate and kill gradients.
- Sigmoids have slow convergence.



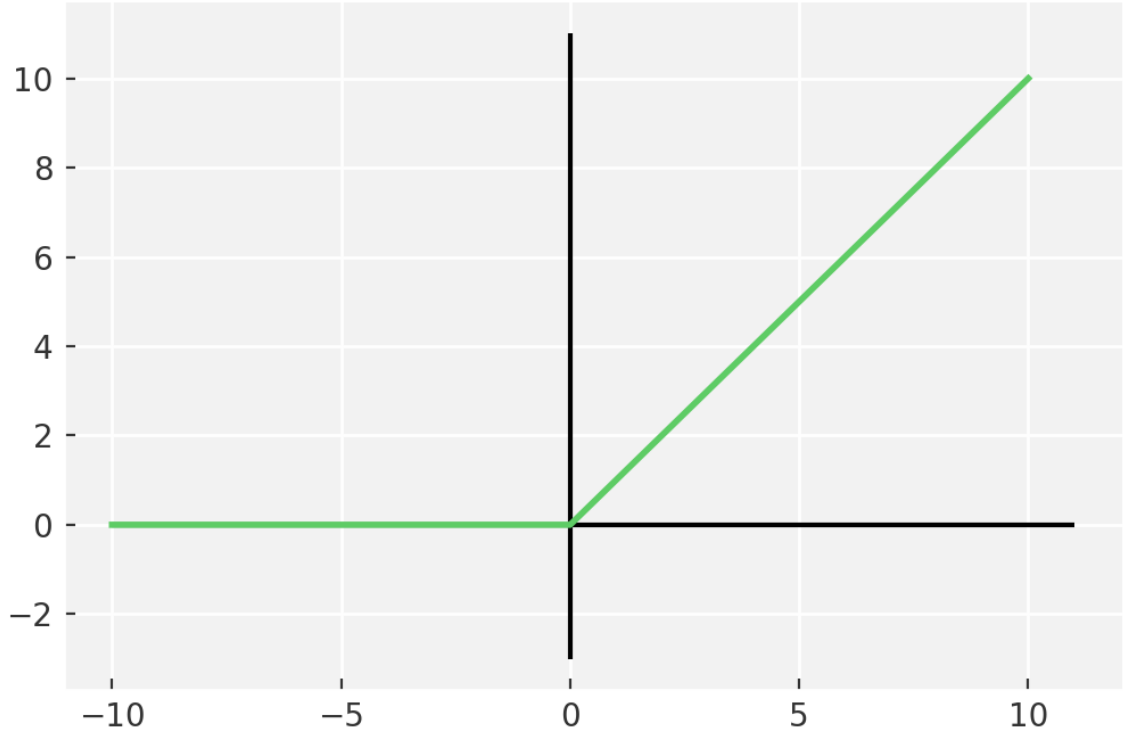
Activation function: Tanh

- Output is zero centered
- Usually preferred to sigmoid as it converges better
- Still it suffers from vanishing gradient problem



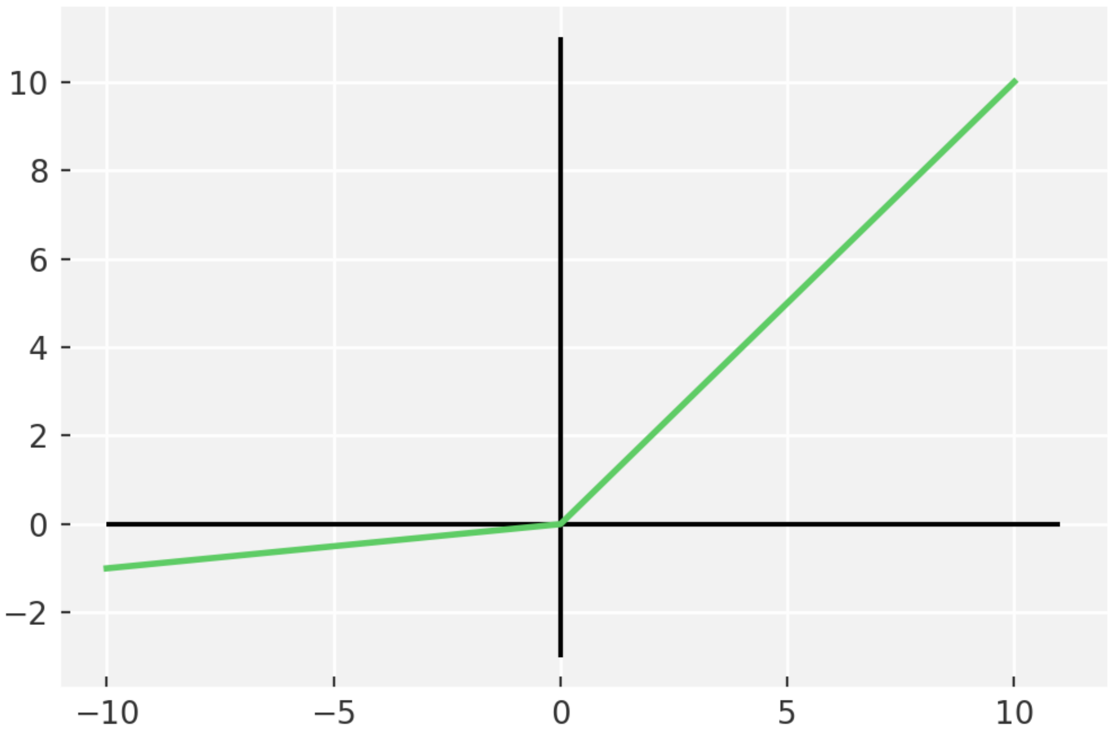
Activation function: ReLU

- 6 times improvement in convergence from Tanh function
- Should only be used within Hidden layers of a neural network model



Activation function: LeakyReLU

- Some ReLu gradients can be fragile during training and can die.
- Cause a weight update which will makes it never activate on any data point again.
- ReLu could result in Dead Neurons.



Writing a DNN in PyTorch

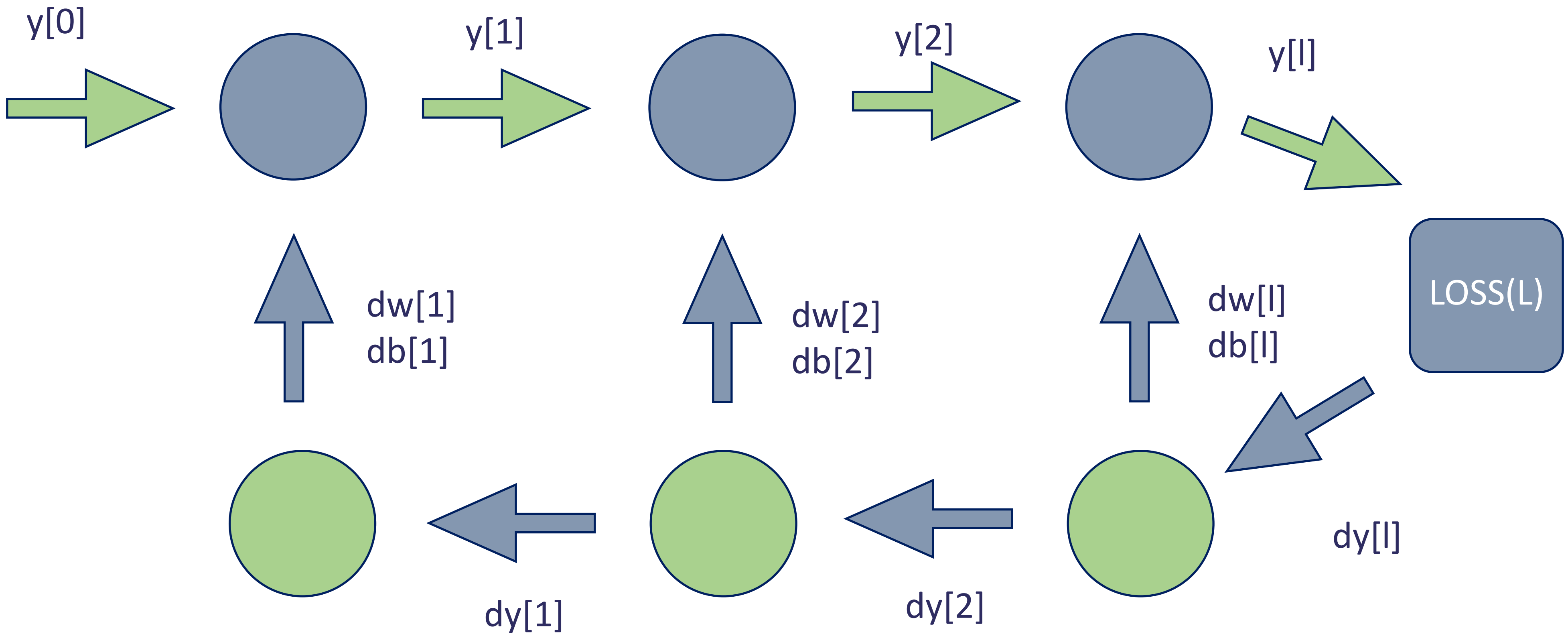
```
class MLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.input_fc = nn.Linear(input_dim, 250)
        self.hidden_fc = nn.Linear(250, 100)
        self.output_fc = nn.Linear(100, output_dim)

    def forward(self, x):

        batch_size = x.shape[0]
        x = x.view(batch_size, -1)
        h_1 = F.relu(self.input_fc(x))
        h_2 = F.relu(self.hidden_fc(h_1))
        y_pred = self.output_fc(h_2)

        return y_pred, h_2
```

Go to notebook

Back Propagation

- Backprop

Notation:
dL/dy[l] => dy[l]



Optimisation Stochastic gradient descent

- Gradient descent – calculate the gradient of the loss of the entire set with respect to parameters
- SGD – calculated per sample rather than on the entire batch
  - Much quicker to calculate, but can lead to high variance
- Mini-batch SGD – calculate loss gradient on batches of set size
  - Best of both worlds

Optimisation: Adaptive methods

- Some parameters update much more often than others
- Therefore different learning rates can be appropriate for different parameters
- *Adagrad* modifies the learning rate η at each time step for every parameter based on the past gradients computed for that parameter

Current gradient

Old parameter

New parameter

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \varepsilon}} g_t$$

Sum of previous gradients

Optimisation: Adam

- Similar to Adagrad
- Add in information about the mean of the momentum of previous steps too
- Works very well in most situations

Mean of last n gradients

Old parameter

New parameter

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v} + \varepsilon} m$$

Variance of last n gradients

Building block: Adam optimizer

```
import torch.optim as optim

optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()
```

Building block – a training loop

```
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in tqdm(iterator, desc="Training", leave=False):

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Go to notebook

Convolutional Neural Nets: The power of inductive bias

The Need for Biases in Learning Generalizations

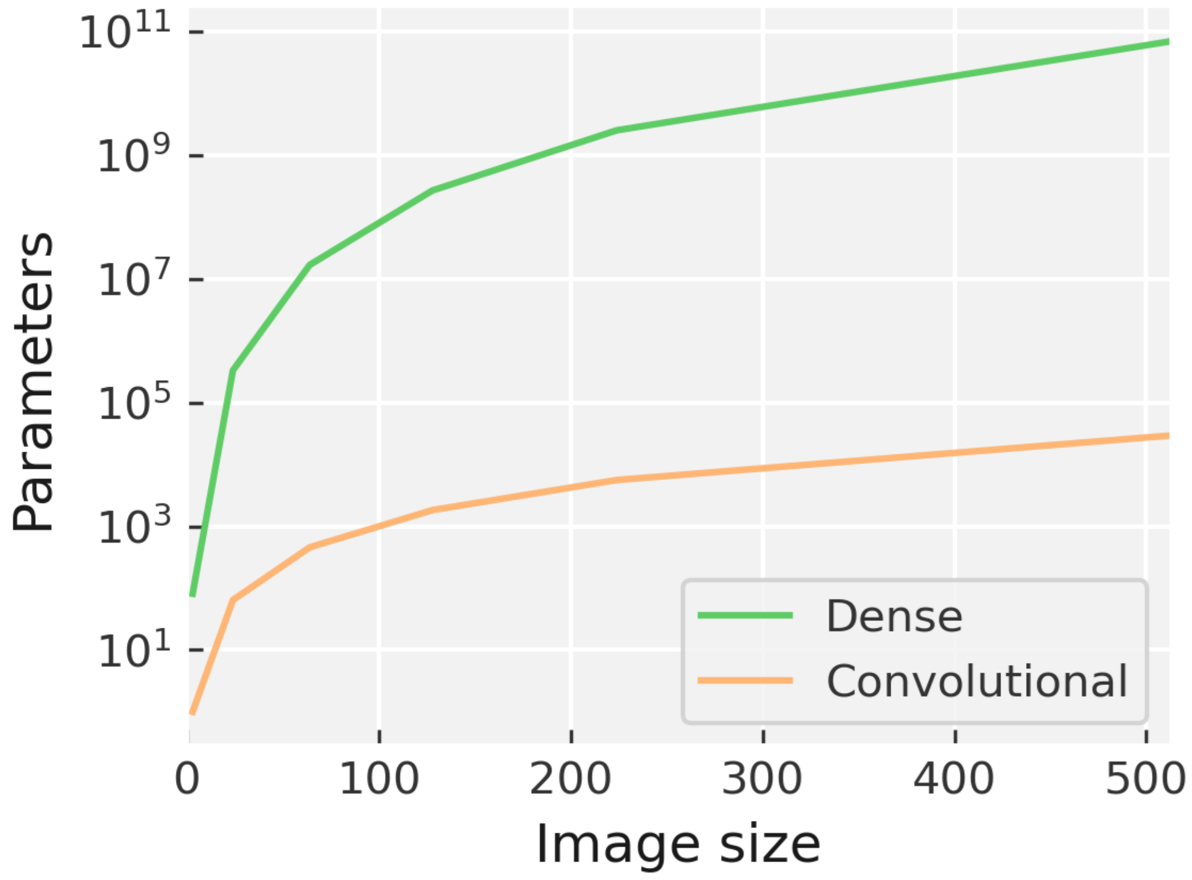
Tom M. Mitchell

The **inductive bias** (also known as **learning bias**) of a learning algorithm is the set of assumptions that the learner uses to predict outputs of given inputs that it has not encountered.

The need for biases in learning generalizations, CBM-TR 5-110, New Brunswick, New Jersey, USA: Rutgers University

Some pitfalls with MLPs for images

- No spatial awareness
- Parametric explosions



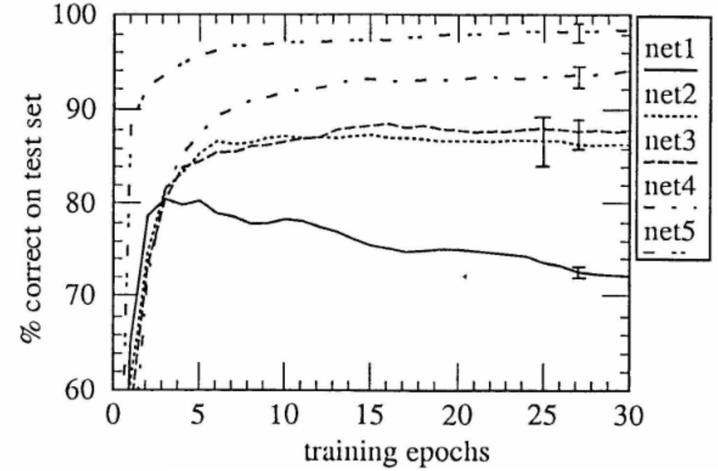
Early CNNs

- LeCun – restricting the number of parameters in a NN leads to better generalisation
- Also makes it possible to fit in memory
- Originally trained for digit recognition for the postal service

Generalization and Network Design Strategies

Y. le Cun
Department of Computer Science
University of Toronto

Technical Report CRG-TR-89-4
June 1989



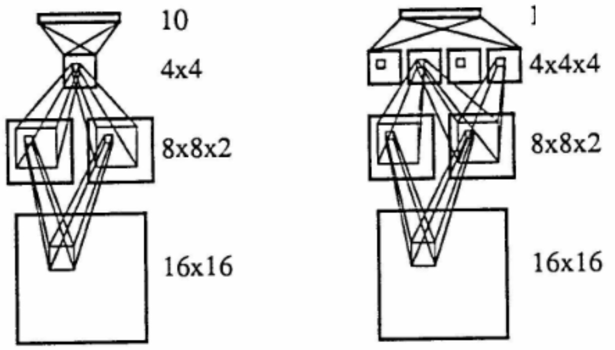


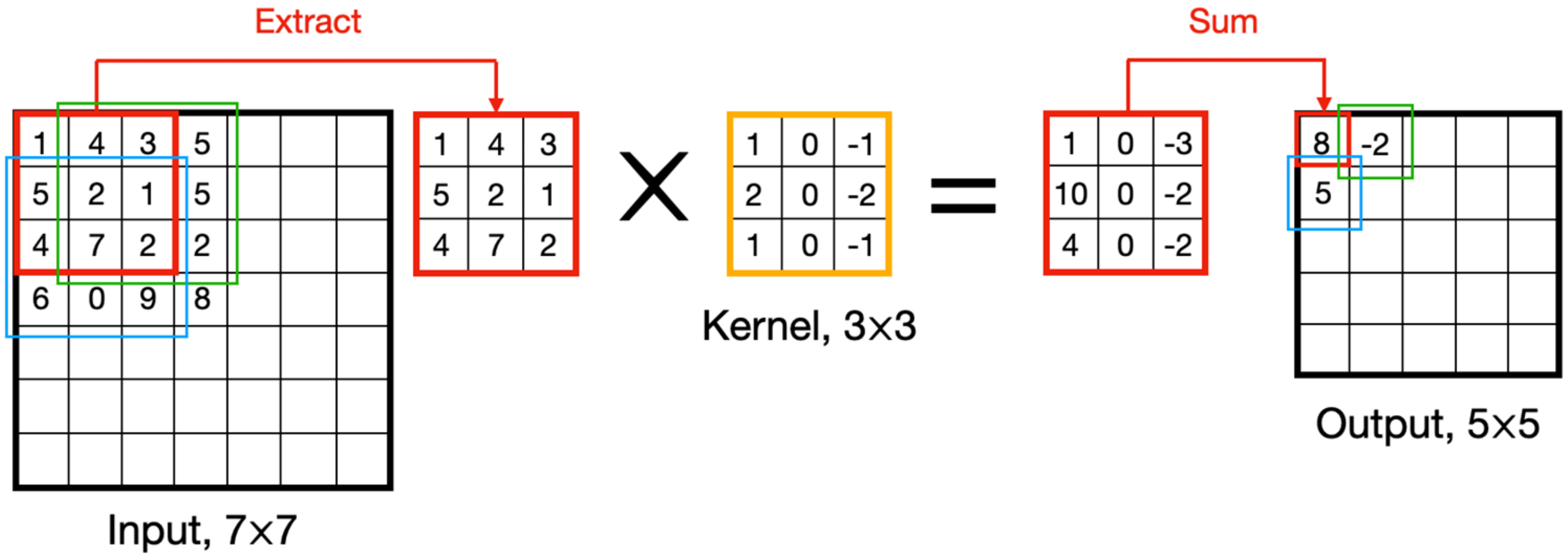
Figure 5 two network architectures with shared weights: Net-4 and Net-5

Structure of a convolutional layer

- Kernel
- Pooling
- Activation

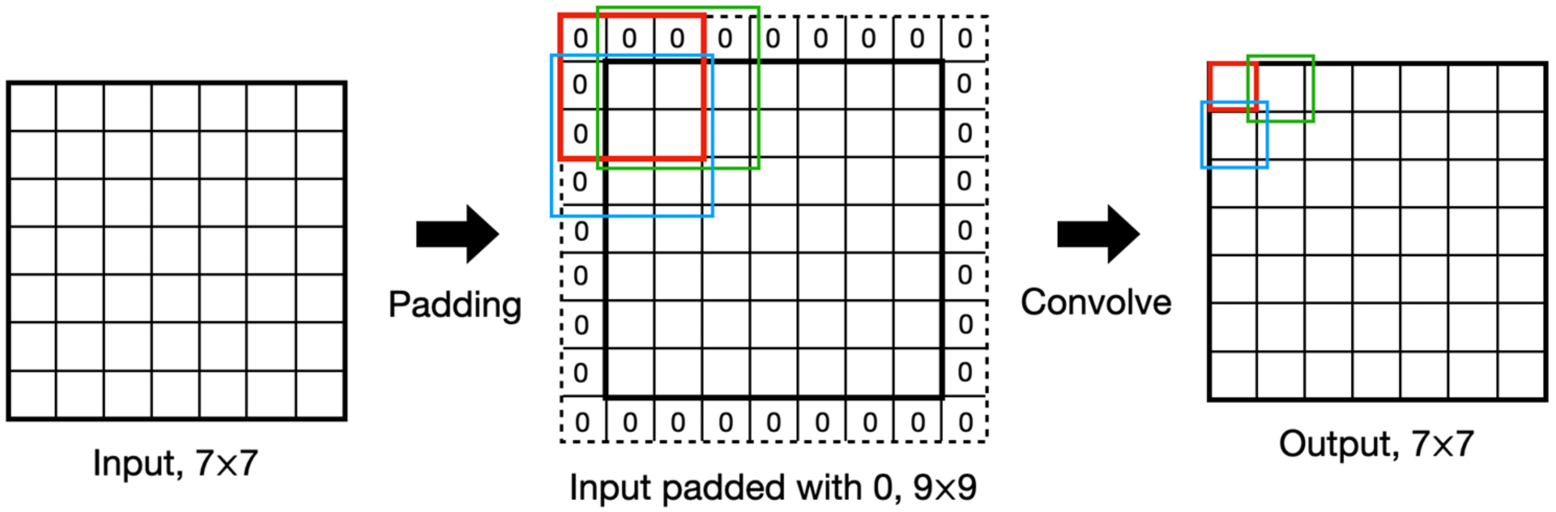
Convolution in action: Kernel

- Input + kernel -> activation map



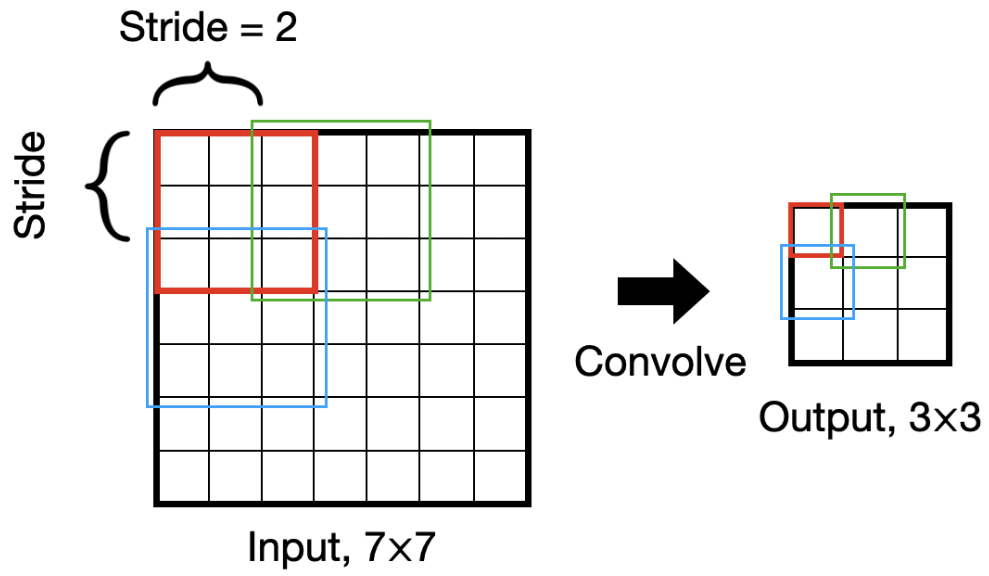
Convolution in action: Padding

- Padding around the outside of images
  - SAME: pad with zeros to make `output.shape == input.shape`
  - VALID: no padding `output.shape < input.shape`



Convolution in action: Striding

- Controls how the filter slides across the image

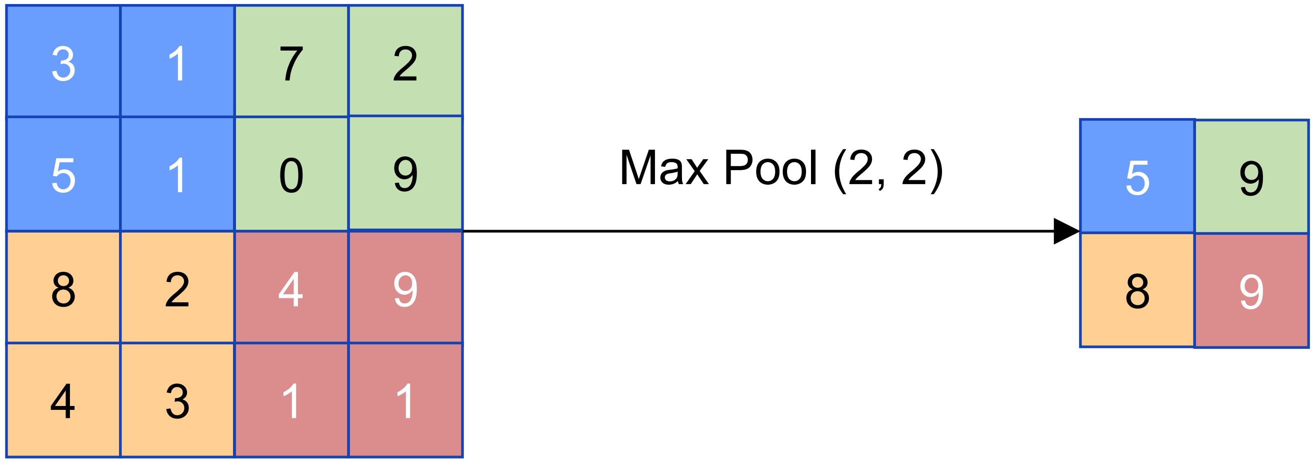


$$\text{output width} = \frac{W - F_w + 2P}{S_w} + 1$$

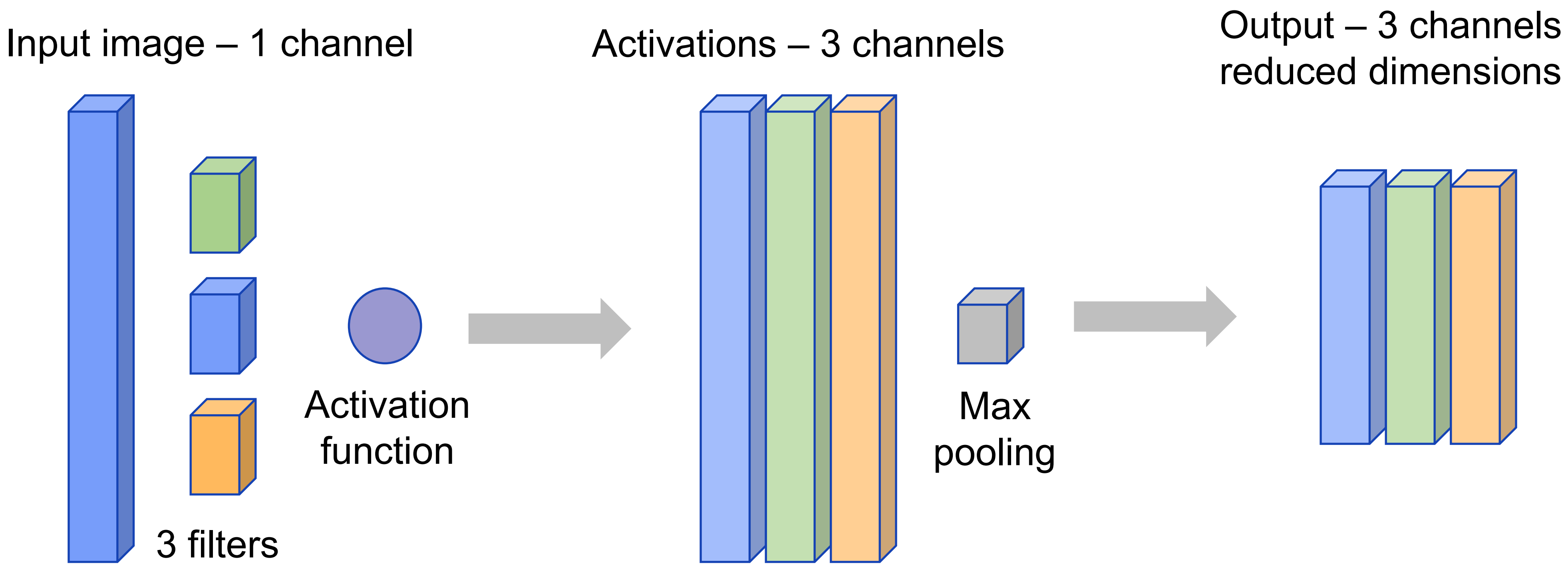
$$\text{output height} = \frac{H - F_h + 2P}{S_h} + 1$$

Convolution in action: Pooling

- Use to compress between layers

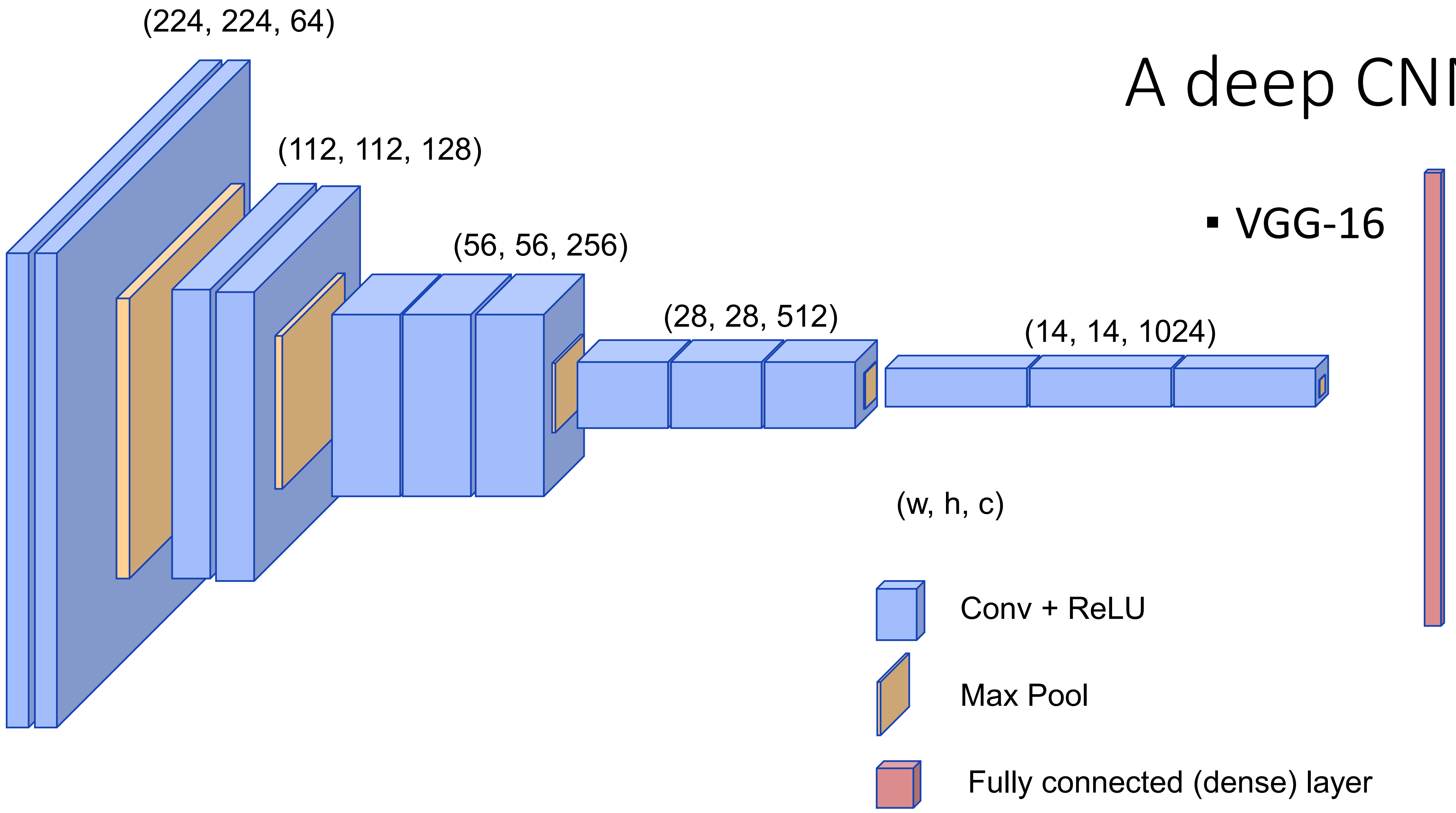


Convolution in action: Putting it together



A deep CNN

- VGG-16



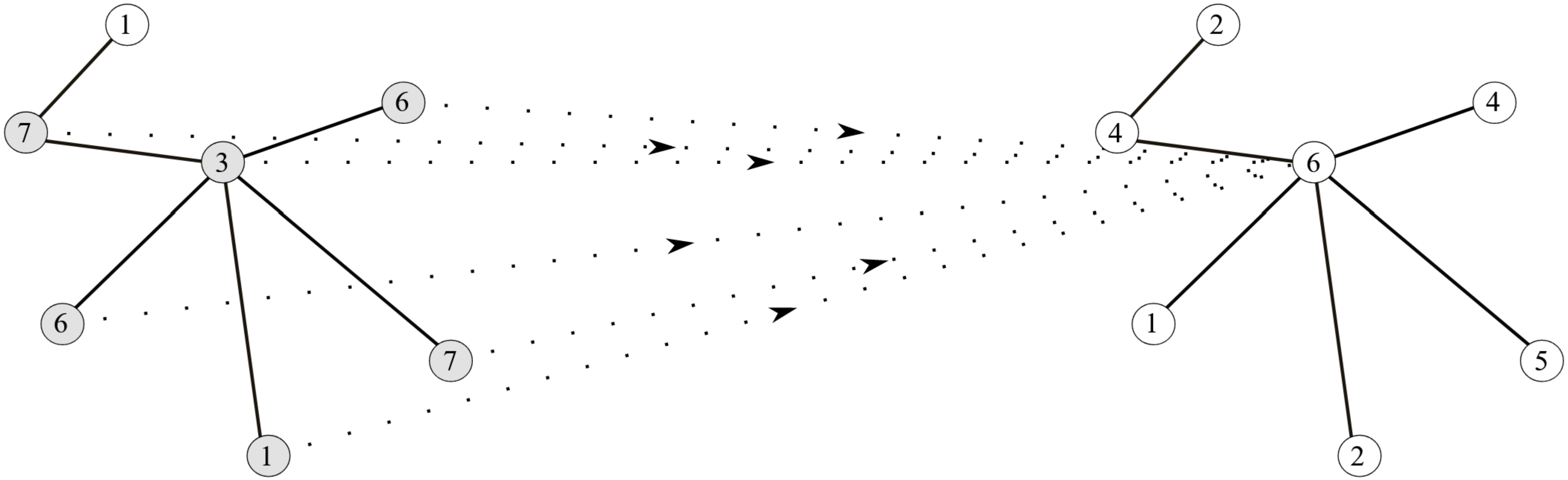
Building blocks: Convolution block

```
import torch.nn as nn

self.conv1 = nn.Conv2d(in_channels=1,
                       out_channels=6,
                       kernel_size=5)
```

Go to notebook

```
import torch.nn.functional as F

x = self.conv1(x)
x = F.max_pool2d(x, kernel_size=2)
x = F.relu(x)
```

Graphs: A more flexible convolution



https://distill.pub/2021/understanding-gnns/

Key concepts

- Deep learning is a qualitatively different process to classical ML
- Deep learning generally requires more data than classical ML
- Deep learning relies on representation learning
- How to write and train a neural network in PyTorch
- Inductive bias allows us to construct more general models
- Inductive bias can allow us to use deep learning on smaller datasets successfully