INTRODUCTION TO DEEP LEARNING

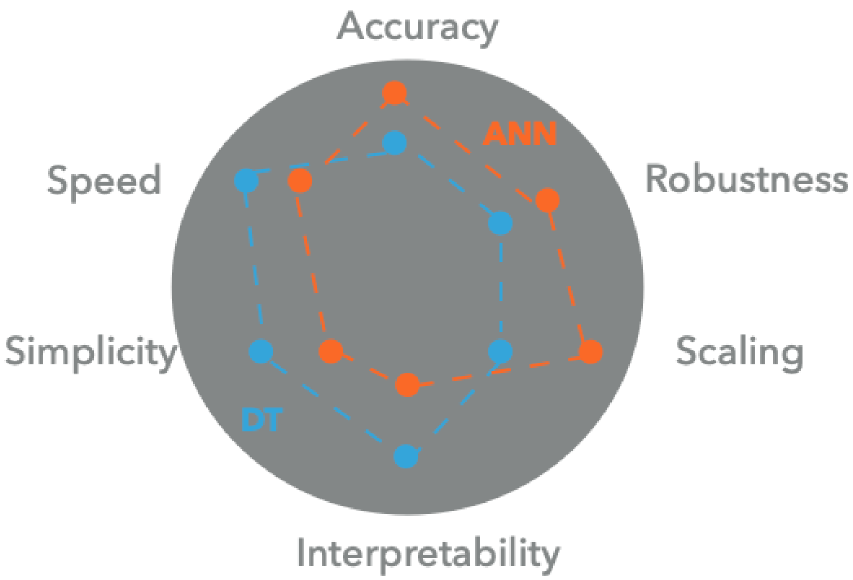
Keith Butler

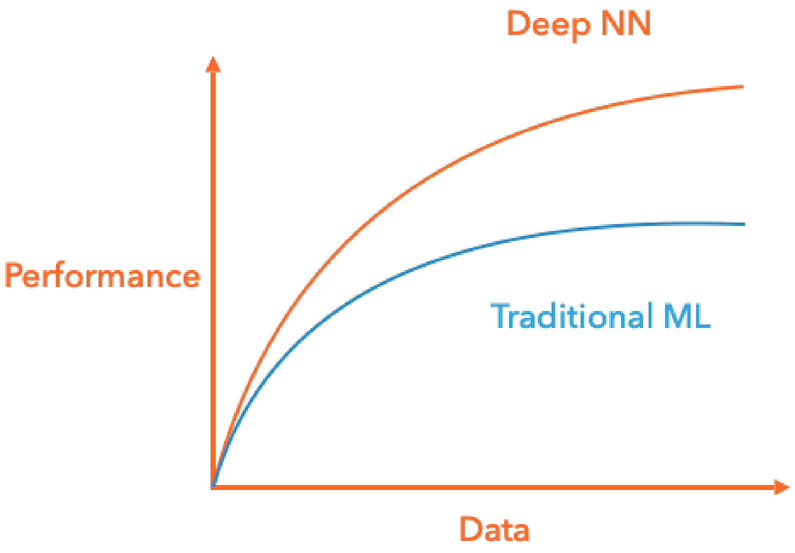
WHAT WE WILL COVER

- 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

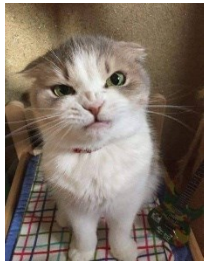
“Hand-crafted” features

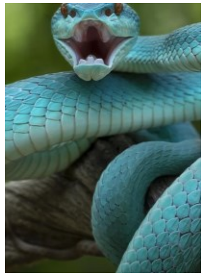
Classical ML $$f(x) = y$$

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

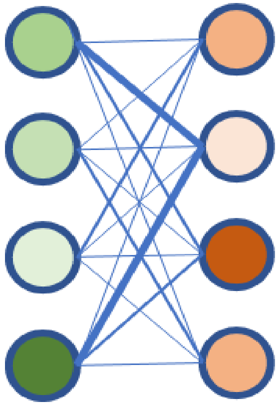
Unstructured data

DEEP LEARNING AS REPRESENTATION LEARNING





Deep learning



Classical ML

Number of eyes	2
Whiskers	N
Legs	N
…	
Scales	Y

Classification model

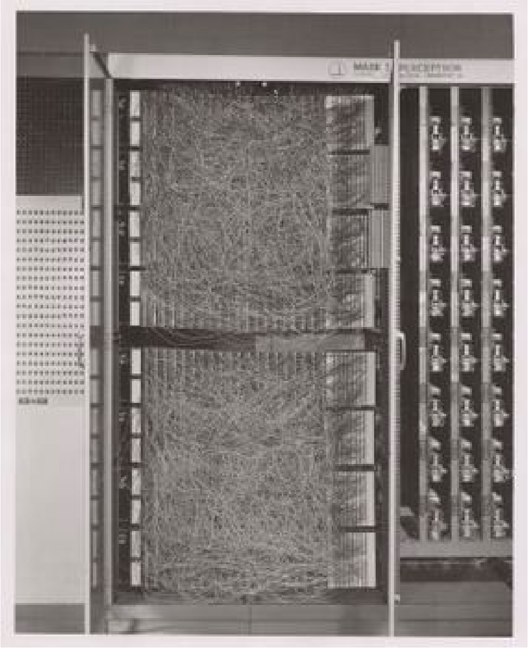
$$f(x) = y$$

Cat/Snake

NEURAL NETWORKS

Originally an analogue device intended for binary classification

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



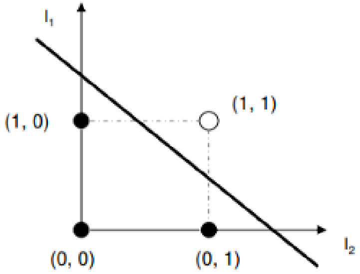
Produces a single output from a matrix of inputs, weights and biases

NEURAL NETWORKS

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

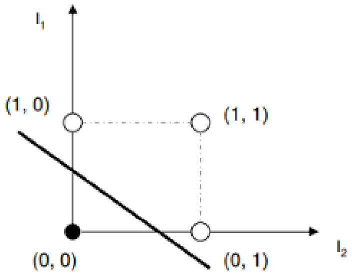
AND

I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



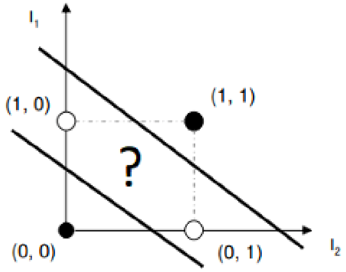
OR

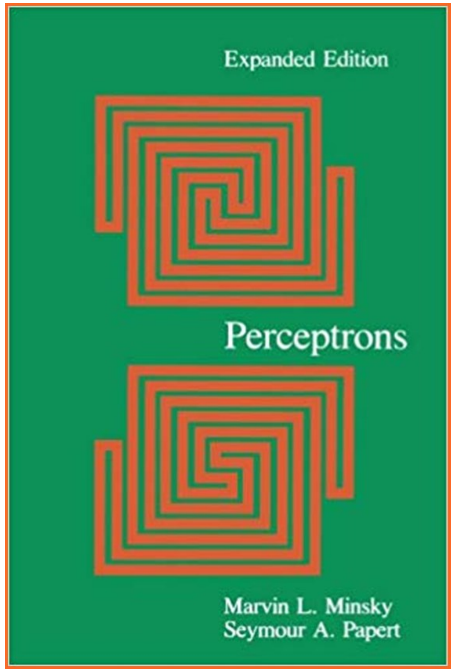
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1



XOR

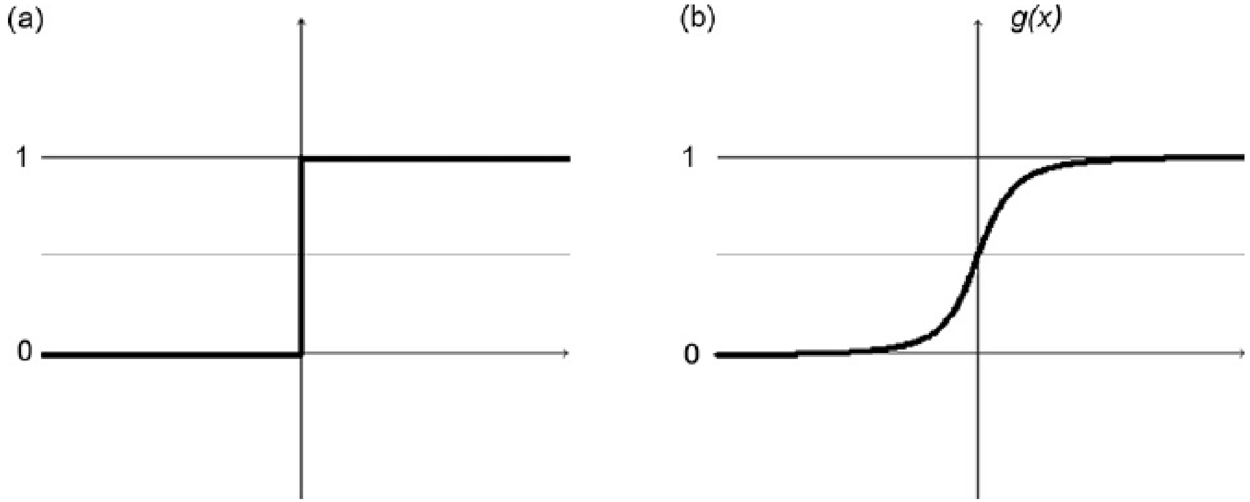
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0





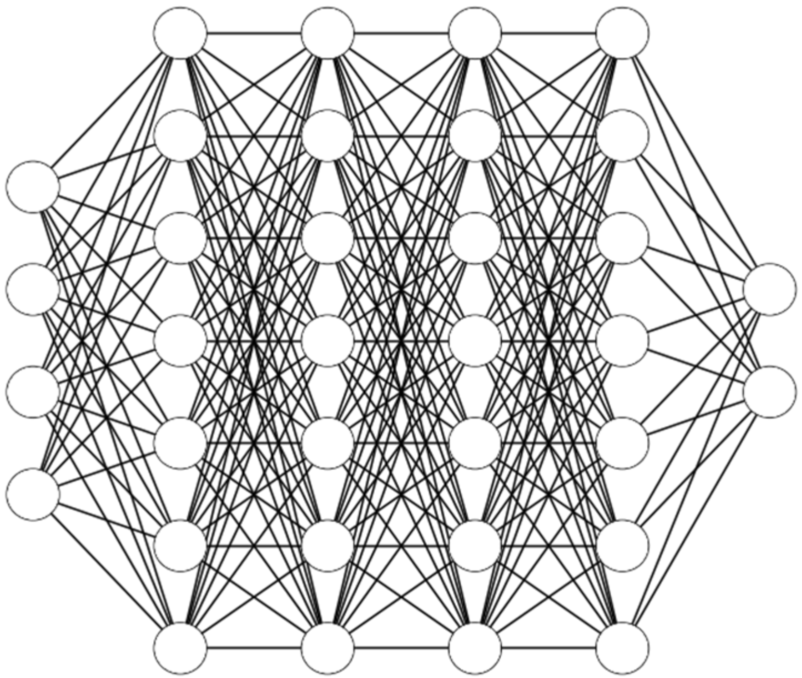
CHANGE OF FUNCTION

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



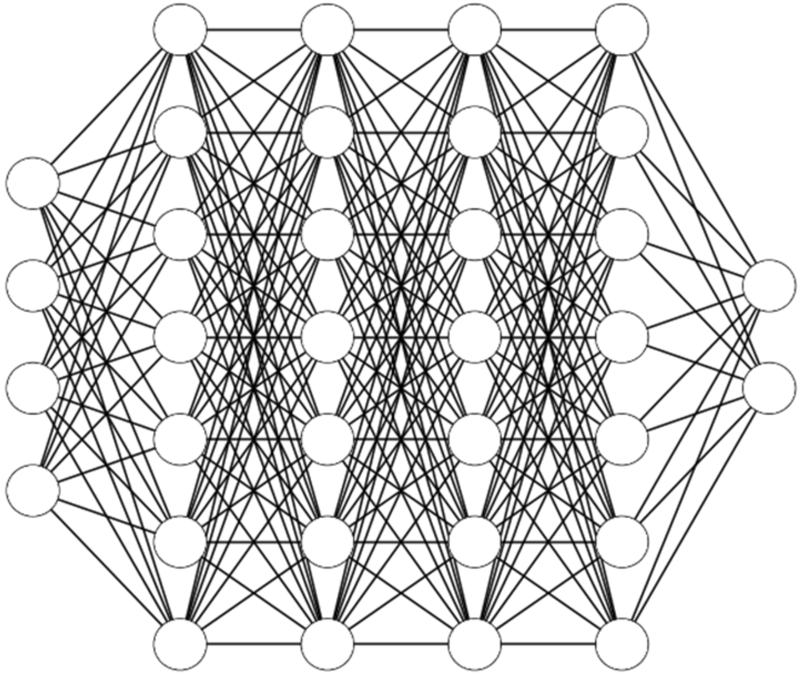
SIGMOID NON-LINEARITY

A differentiable non-linearity allows for multiple layers



DEEP NEURAL NETWORKS: MULTI LAYER PERCEPTRON

Input layer

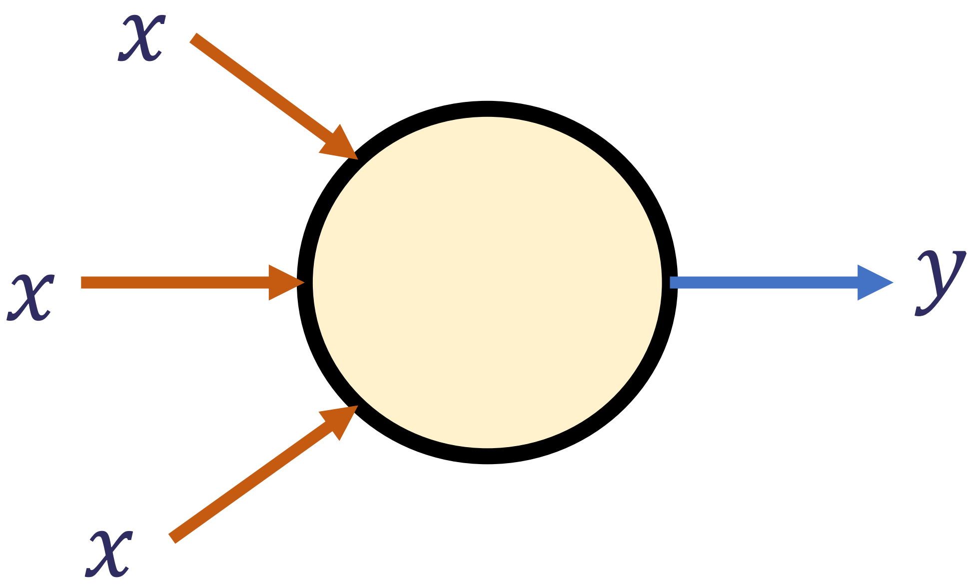


Output layer

Hidden layers

DENSE LAYERS

Also called fully connected layers as each node is connected to each node in the previous layer



Activation function

Output signal $$y = g(z)$$

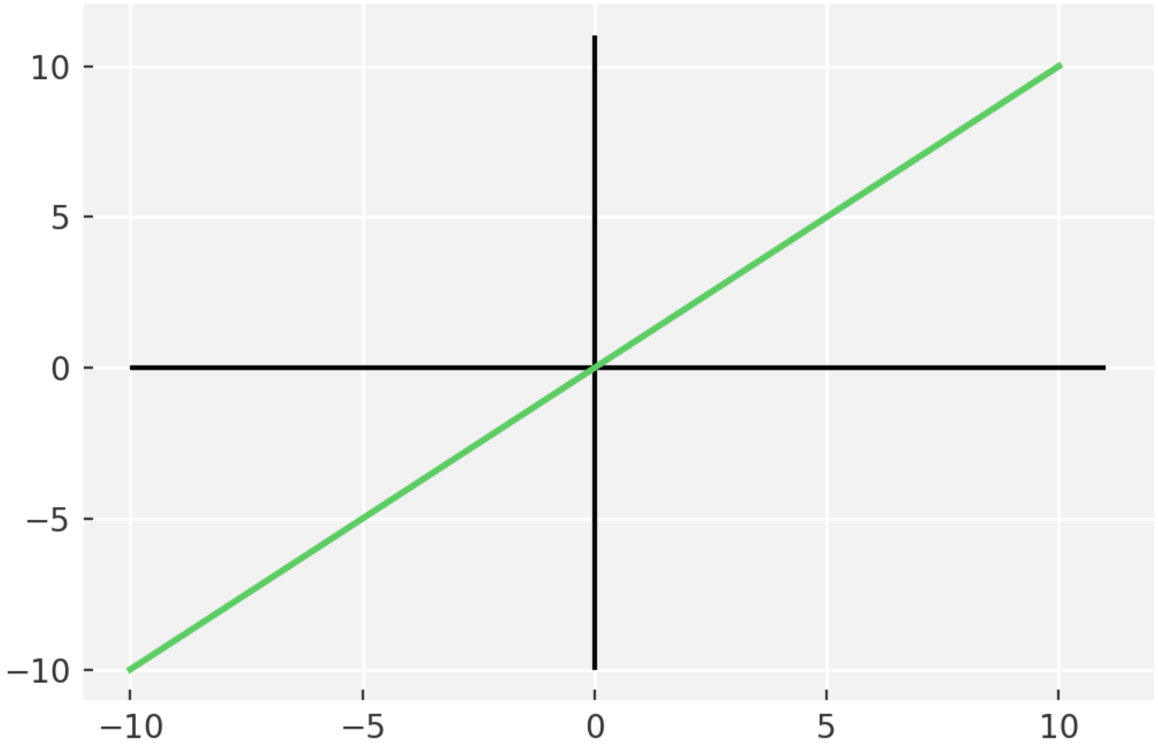
$$z = \mathbf{w}^T . \mathbf{x} + b$$ Bias

Weights

Input signals

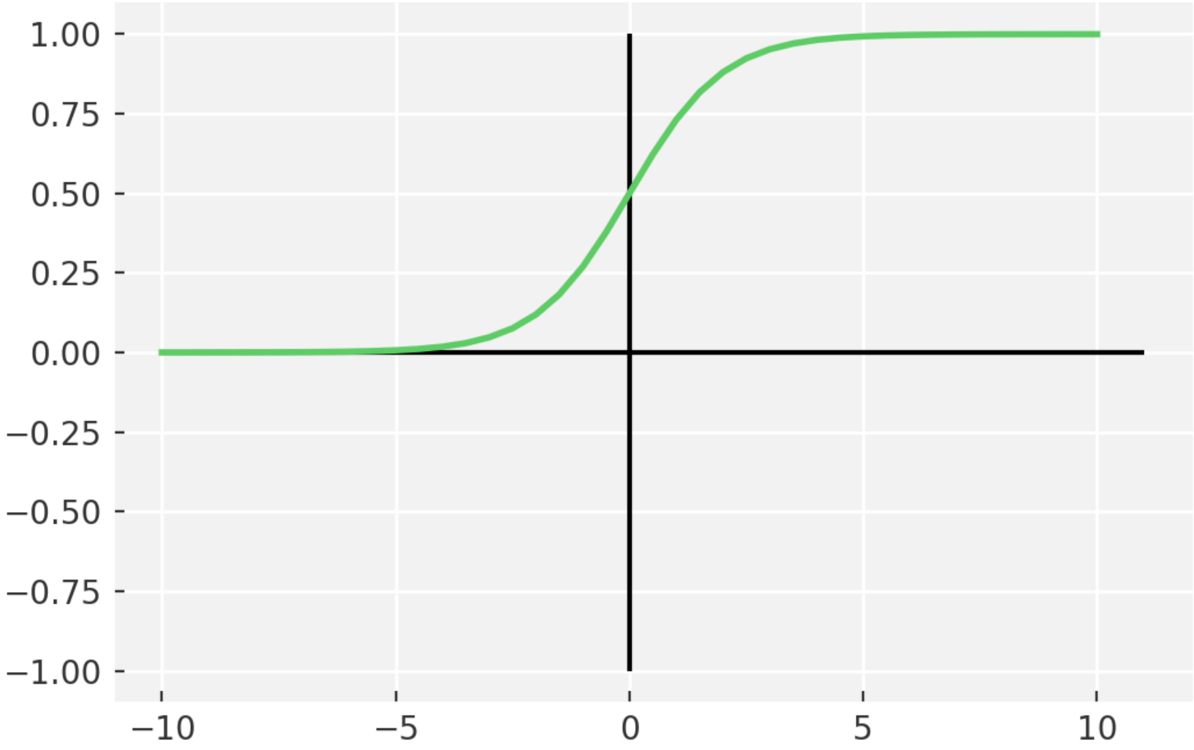
ACTIVATION FUNCTION: LINEAR

The simplest activation is a linear transformation of the weights matrix



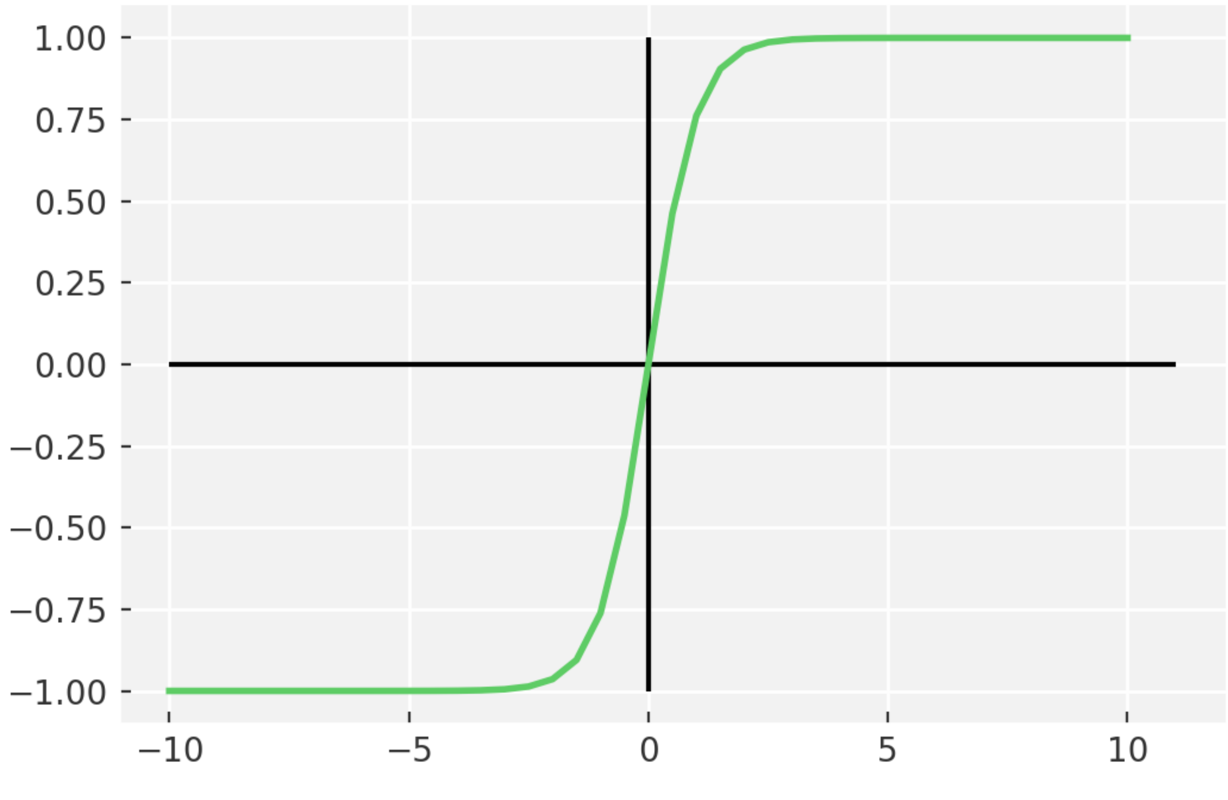
ACTIVATION FUNCTION: SIGMOID

As we saw earlier sigmoid was the first non-linearity (after the step function)



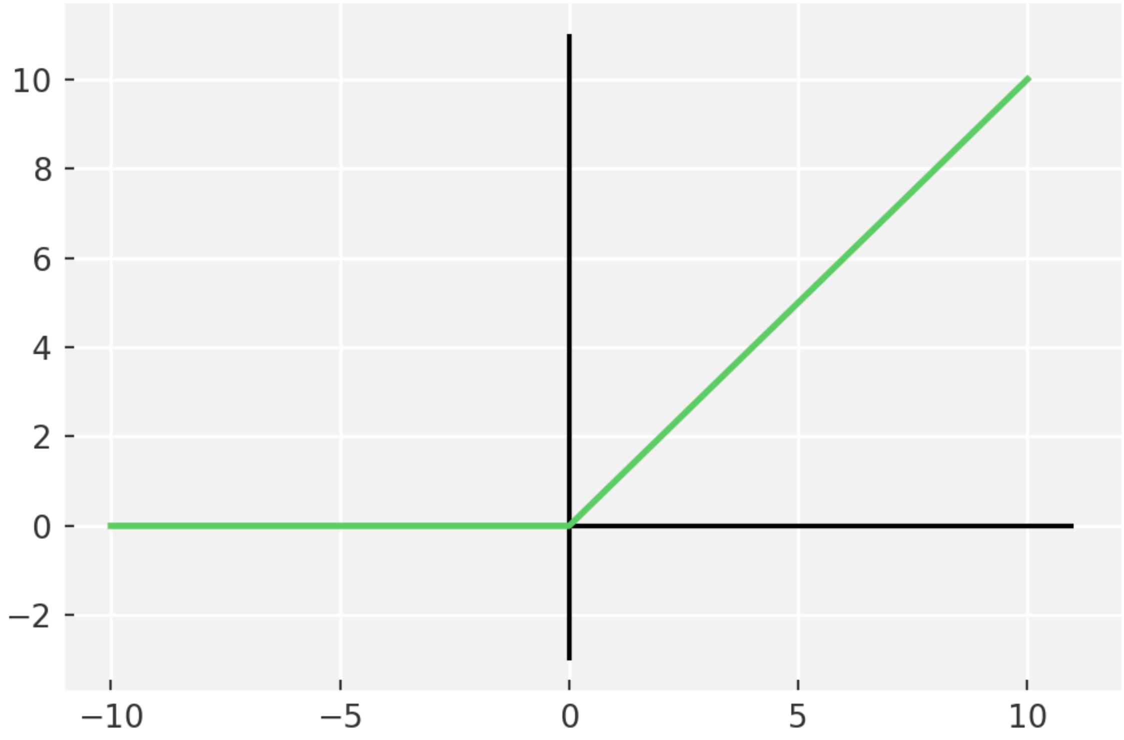
ACTIVATION FUNCTION: TANH

Like sigmoid, but zero-centered, **converges better** than sigmoid



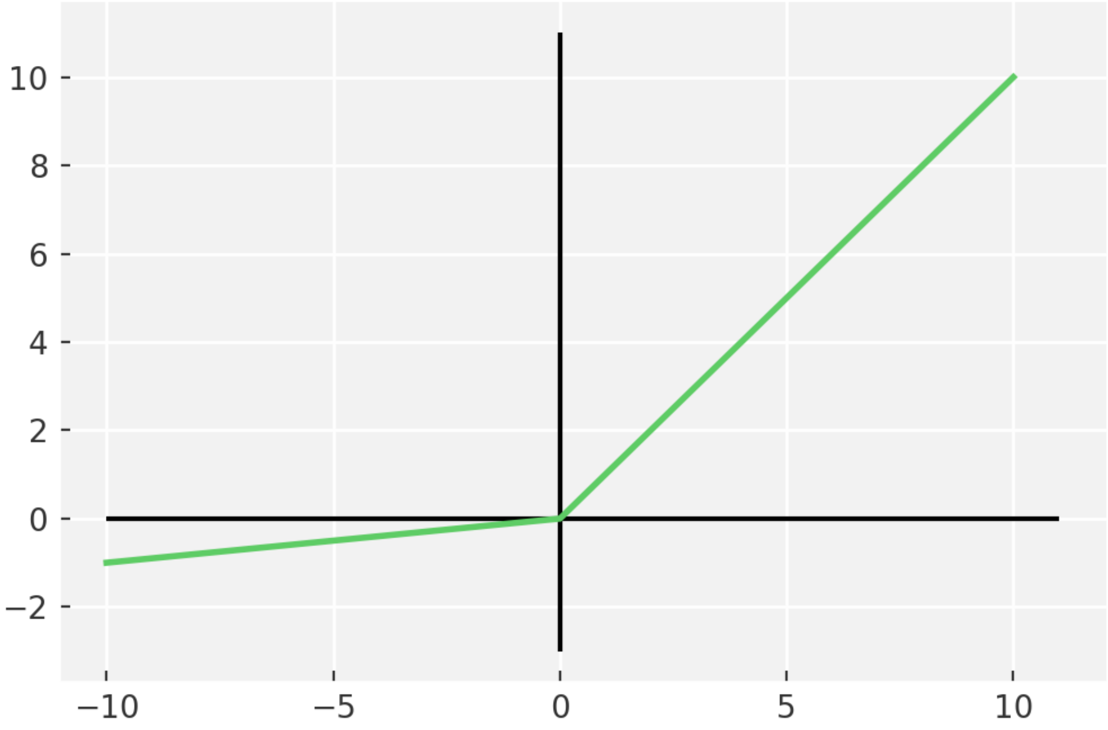
ACTIVATION FUNCTION: RELU

The rectified linear unit (ReLU) has 6 x improvement in convergence from Tanh function



ACTIVATION FUNCTION: LEAKYRELU

ReLU can still lead to vanisihing gradients, leaky ReLU attempts to circumvent this

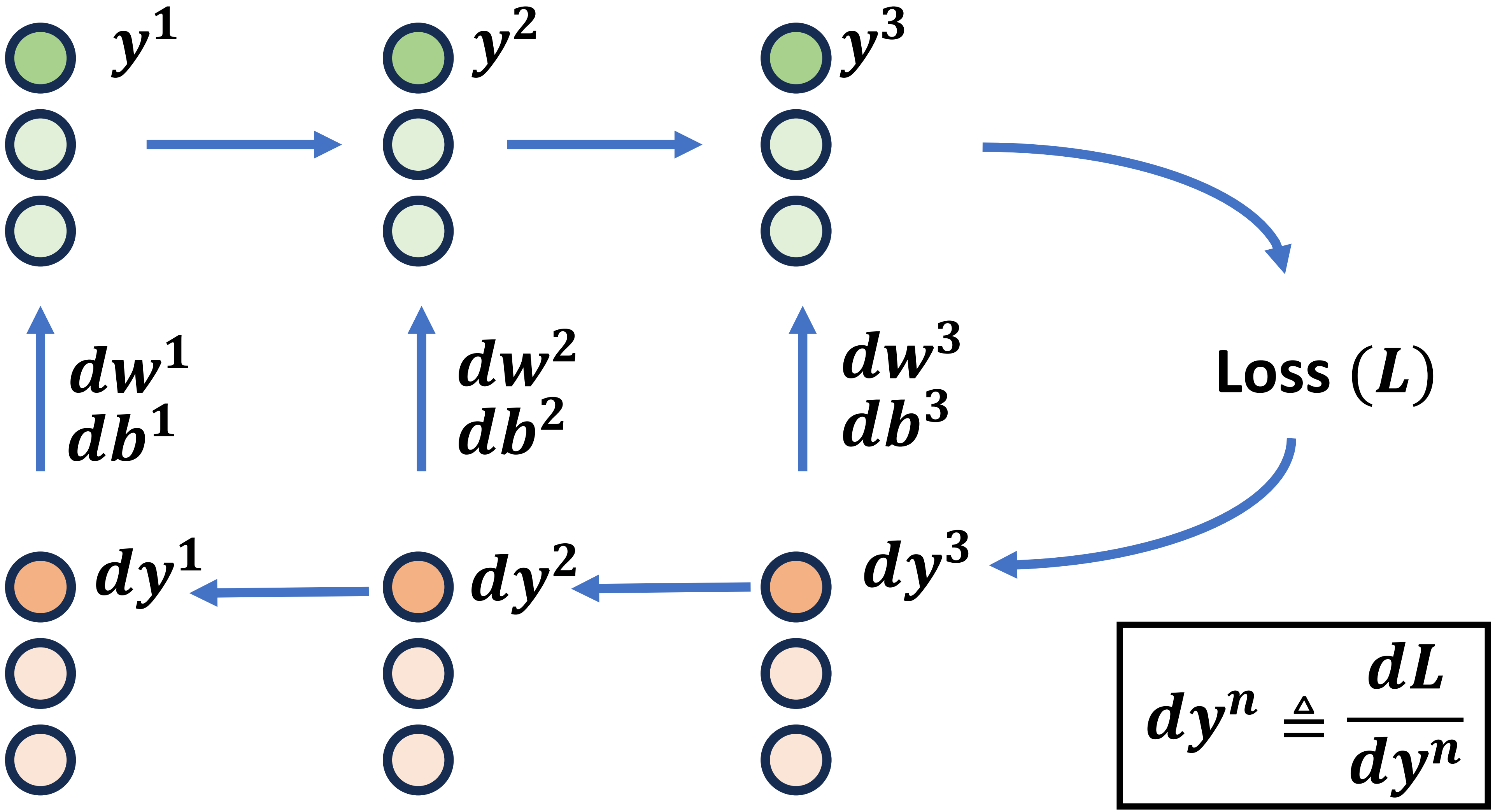


WRITING A DNN IN PYTORCH

```
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



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

CONCEPT CHECKLIST

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

THANK YOU

mdi-group.github.com

Elements