Artificial Neural Networks

John Stachurski

2025

Topics

- History
- Computational graphs
- Feedforward neural networks
- Gradient descent
- Why is DL so successful?

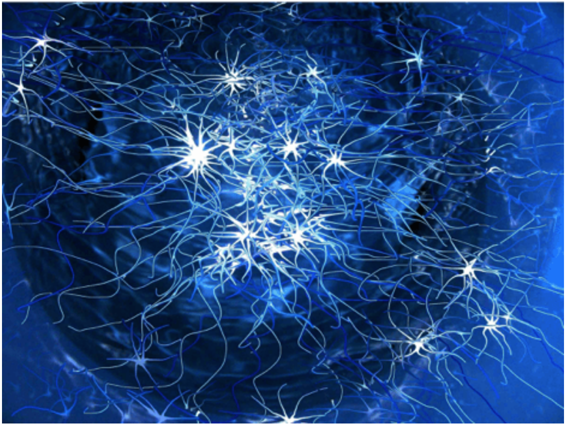
History

- 1940s: McCulloch & Pitts create mathematical model of NN
- 1950s: Rosenblatt develops the perceptron (trainable NN)
- 1960s-70s: Limited progress with single layer perceptrons
- 1980s: Backpropagation algorithm enables training of MLPs
- 1990s: SVMs temporarily overshadow ANNs in popularity
- 2000s: Deep learning finds successes in large problems

Last 10 years: Explosion of progress in deep learning

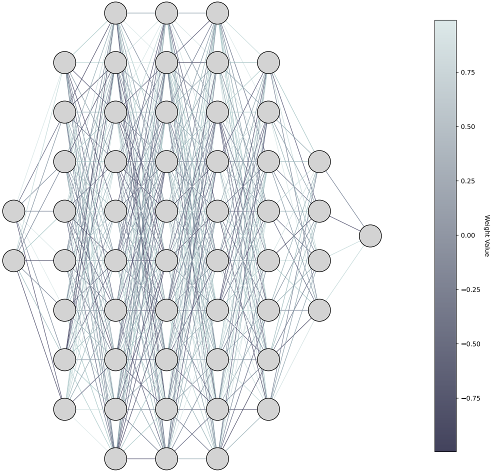
- CNNs, RNNs, LSTMs, transformers, LLMs, etc.

A model of the human brain

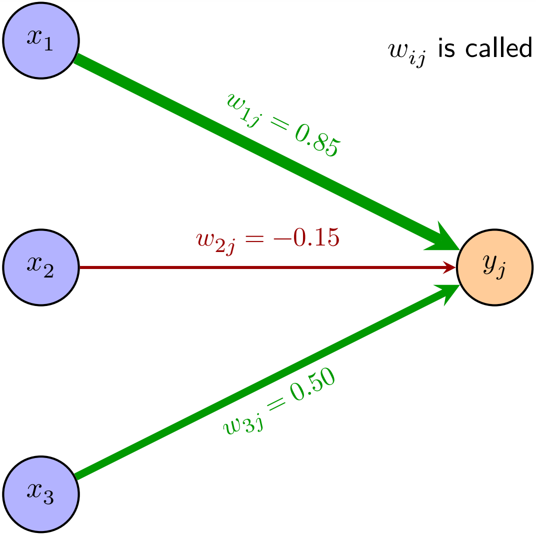


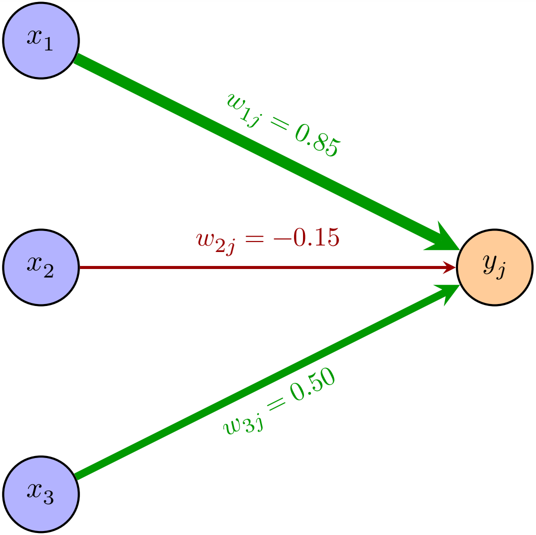
– source: Dartmouth undergraduate journal of science

A mathematical representation: directed acyclic graph

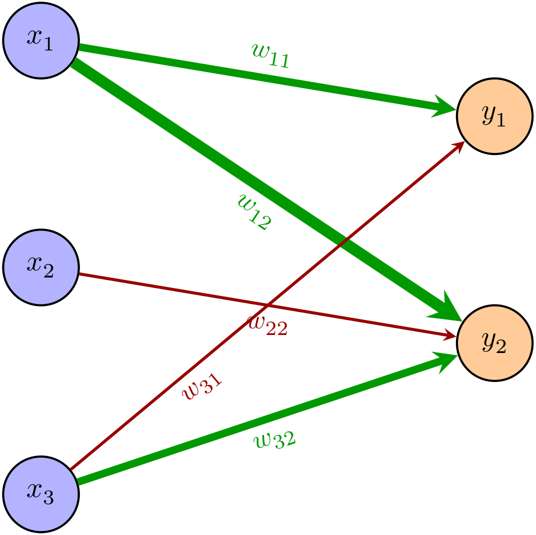


w_{ij} is called a "weight"





$$y_j = \sum_i w_{ij} x_i$$



$$y_1 = \sum_i w_{i1} x_i$$

$$y_2 = \sum_i w_{i2} x_i$$

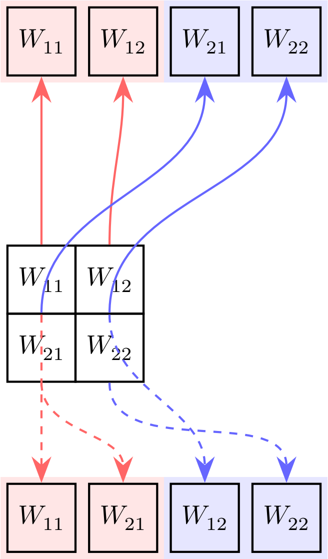
$$\implies y = xW$$

Note that we are using **row vectors**: $y = xW$

This is natural given our notation

- w_{ij} points from i to j
- hence $y_j = \sum_i w_{ij} x_i$
- hence $y = xW$

But it also has another advantage ...?



row-major storage

column-major storage

Computing xW

```python
for i in range(n):
    for j in range(m):
        # Access row elements of W contiguously
        y[j] += W[i, j] * x[i]
```

Computing Wx

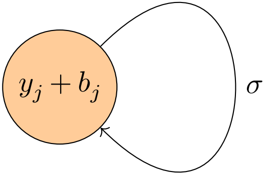
```python
for j in range(n):
    for i in range(m):
        # Non-contiguous access of W
        y[i] += W[i, j] * x[j]
```

Next steps

After computing $y_j = \sum_i w_{ij} x_i$ we

1. add a bias term b_j and
2. apply a nonlinear "activation function" $\sigma \colon \mathbb{R} \to \mathbb{R}$

applying activation function



First add bias:

$$y_j = \sum_i w_{ij} x_i \qquad \to \qquad y_j = \sum_i w_{ij} x_i + b_j$$

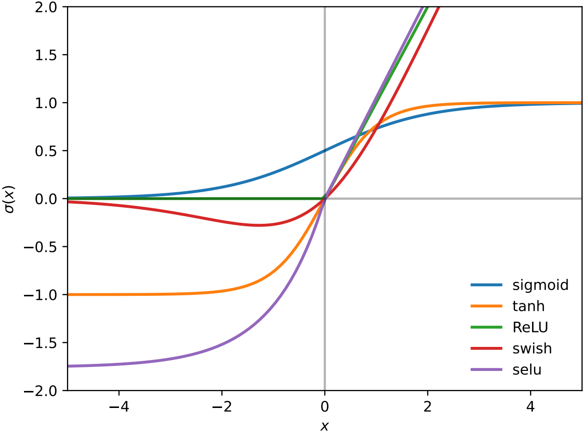
Then apply activation:

$$y_j = \sum_i w_{ij} x_i + b_j \qquad \to \qquad y_j = \sigma\left(\sum_i w_{ij} x_i + b_j\right)$$

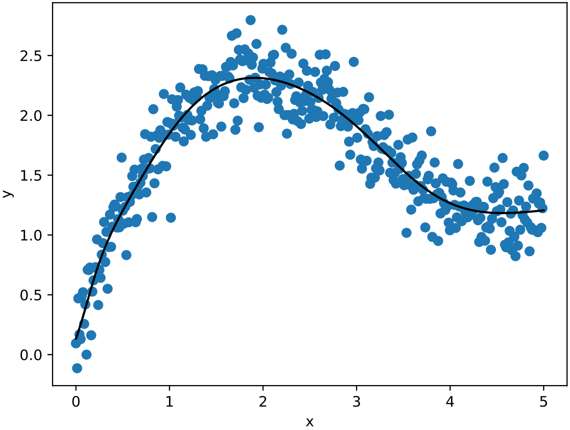
Applying σ pointwise, we can write this in vector form as

$$y = \sigma(xW + b)$$

Common activation functions



Training



Aim: Learn to predict output y from input x

- $x \in \mathbb{R}^k$
- $y \in \mathbb{R}$ (regression problem)

Examples.

- $x =$ cross section of returns, $y =$ return on oil futures tomorrow
- $x =$ weather sensor data, $y =$ max temp tomorrow

Problem:

- observe $(x_i, y_i)_{i=1}^n$ and seek f such that $y_{n+1} \approx f(x_{n+1})$

Nonlinear regression: choose model $\{f_\theta\}_{\theta\in\Theta}$ and minimize the empirical loss

$$\ell(\theta) := \frac{1}{n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2 \quad \text{s.t.} \quad \theta \in \Theta$$

In the case of ANNs, we consider all f_θ having the form

$$f_\theta = G_m \circ G_{m-1} \circ \cdots \circ G_2 \circ G_1$$

where

- $G_\ell x = \sigma_\ell(xW_\ell + b_\ell)$
- σ_ℓ is an activation function

Nonlinear regression: choose model $\{f_\theta\}_{\theta\in\Theta}$ and minimize the empirical loss

$$\ell(\theta) := \frac{1}{n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2 \quad \text{s.t.} \quad \theta \in \Theta$$

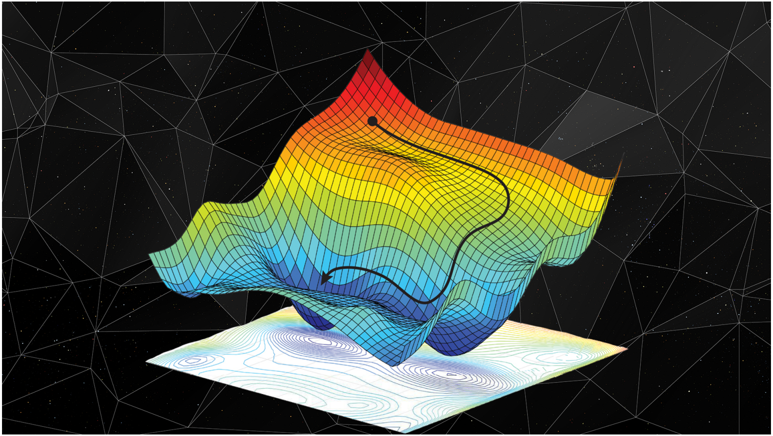
In the case of ANNs, we consider all f_θ having the form

$$f_\theta = G_m \circ G_{m-1} \circ \cdots \circ G_2 \circ G_1$$

where

- $G_\ell x = \sigma_\ell(xW_\ell + b_\ell)$
- σ_ℓ is an activation function

Minimizing the loss functions



Source: https://danielkhv.com/

Gradient descent

Algorithm:

$$\theta_{\text{next}} = \theta - \lambda \nabla_\theta \ell(\theta, x, y)$$

- take a step in the opposite direction to the grad vector
- λ is the **learning rate** – often changes at each step
- iterate until hit a stopping condition
- in practice replace $\ell(\theta)$ with batched loss

$$\frac{1}{|B|} \sum_{i \in B} (y_i - f_\theta(x_i))^2$$

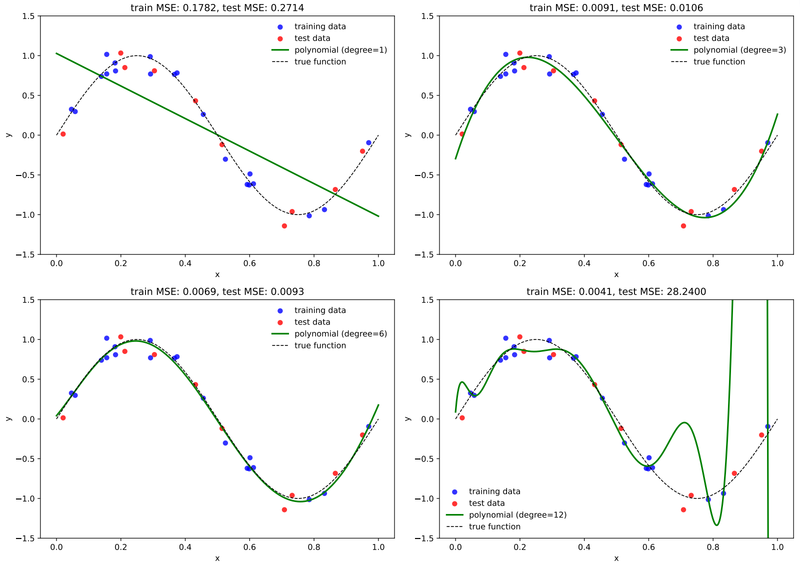
Using batches $\rightarrow$ **stochastic gradient descent**

Extensions

- Loss functions with regularization
- Cross-entropy loss (classification)
- Convolutional neural networks (image processing)
- Recurrent neural networks (sequential data)
- Transformers (LLMs)
- etc.

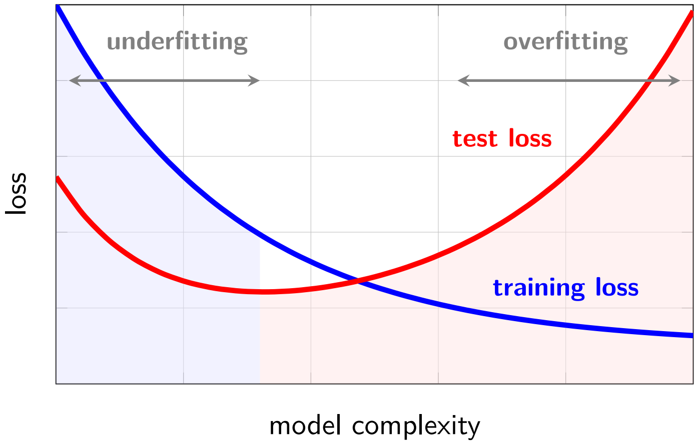
A mystery

What about overfitting?



train MSE: 0.1782, test MSE: 0.2714
training data
test data
polynomial (degree=1)
true function
train MSE: 0.0091, test MSE: 0.0106
polynomial (degree=3)
train MSE: 0.0069, test MSE: 0.0093
polynomial (degree=6)
train MSE: 0.0041, test MSE: 28.2400
polynomial (degree=12)
x
y

Overfitting and underfitting



If production-level DL models are so large, why don't they overfit?

Answer 1 Data sets are large and complex – need complex model

Answer 2 Engineers avoid using full complexity of the model

- Early stopping halts training when test loss starts to rise
- Drop out – shut down some neurons during training

Answer 3 Adding randomization to training prevents overfitting

- Related to benefits of random dropout (randomization)
- Related techniques in DL such as DropConnect and Stochastic Depth
- Stochastic gradient descent injects randomness

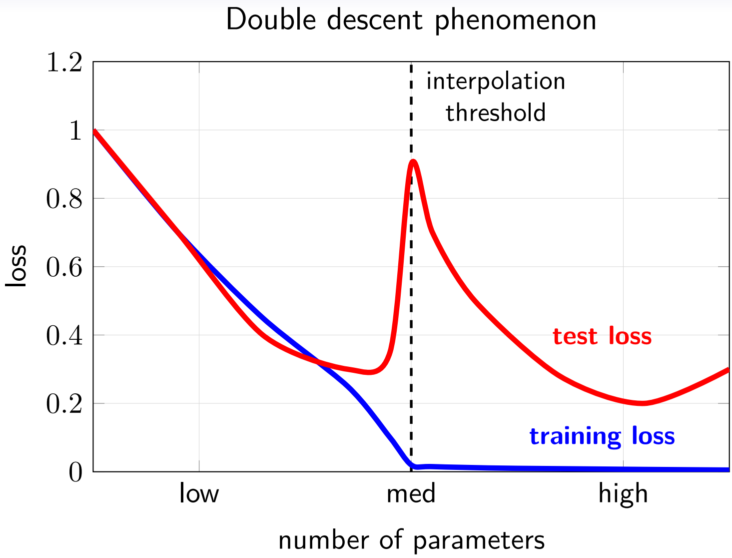
Randomness "adds some smoothing" to a given data set

"The model must learn to be robust to these perturbations, which encourages it to find smoother, more generalizable decision boundaries rather than fitting to noise in the training data."

Answer 4 Modern architectures have inductive biases that guide learning toward useful patterns

- translation invariance in CNNs (same pattern can be recognized anywhere in an image)
- localization in CNNs – pixels are influenced more by neighbors than pixels far away
- parameter sharing in RNNs – similarity of transformations across time
- Layer normalization and residual connections in transformers - create a bias toward stable training dynamics and information preservation across layers.

Finally, there is some evidence of “double descent” – test error starts to fall again when the number of parameters is very high


Double descent phenomenon
interpolation threshold
test loss
training loss
loss
1.2
1
0.8
0.6
0.4
0.2
0
low
med
high
number of parameters

Summary

Why can deep learning successfully generalize from limited observations?

Computer scientists' story

- Good because based on a model of the human brain!
- A universal function approximator!
- Can break the curse of dimensionality!

Alternative story (me)

- A highly flexible function fitting technique
- Easy to understand with limited maths background
- Extends naturally to high dimensions
- Can exploit low-dimensional structure
- Function evaluations are highly parallelizable
- Smooth recursive structure suited to calculating gradients
- Has received massive investment from the CS community
    - algos, software, hardware

- Many incremental improvements to improve regularization and training
- Many incremental improvements to inject domain-specific knowledge (CNNs, transformers, etc.)