Artificial Neural Networks & Deep Learning

CSCI4181/6802

Presented by Alex Manuele

About Me

DALHOUSIE UNIVERSITY

BenchSci

Objectives

You will learn:

- What is an artificial neural network?
- Ok, so what is a “Deep” neural network?
- The convolution operation
- Recurrence in neural networks
- Transformer: Not just robots in disguise
- Applications in bioinformatics

I assume you already know:

- What is machine learning?
- What is classification/regression?
- What is overfitting, and how does it happen (roughly)?
- What is a vector? (e.g. SVM)
- What is a feature?

WHAT IS AN ANN?

Broadly, an ANN is a machine learning model composed of “artificial neurons”

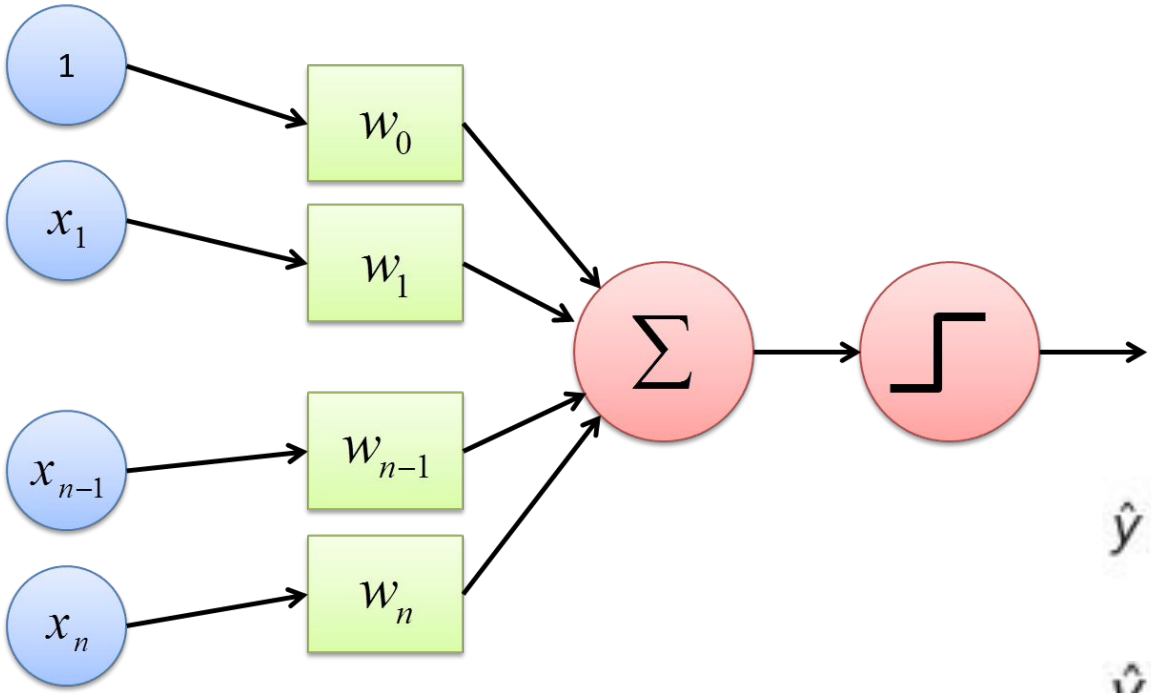
Broader still, all ANN designs (architectures) follow some essential principles:

“(i) takes some set of inputs;

(ii) generates a corresponding output

(iii) has a set of associated parameters that can be updated to optimize some objective function of interest.” -d2l.ai

Enter: The perceptron



X, an feature vector, goes in

w, a weight vector, is **learned**

ŷ, the predicted label, goes **out**

$$\hat{y} = f(\sum_{i=1}^{n} x_i w_i + w_0)$$

$$\hat{y} = f(\mathbf{w} \cdot \mathbf{x} + w_0)$$

Wait…. That looks familiar….

The Perceptron is a **linear classifier!**

i.e. Uses a linear combination of the input data features

$$\hat{y} = f(\mathbf{w} \cdot \mathbf{x} + w_0)$$

Recall: A line is y=mx+b

How perceptrons “learn”

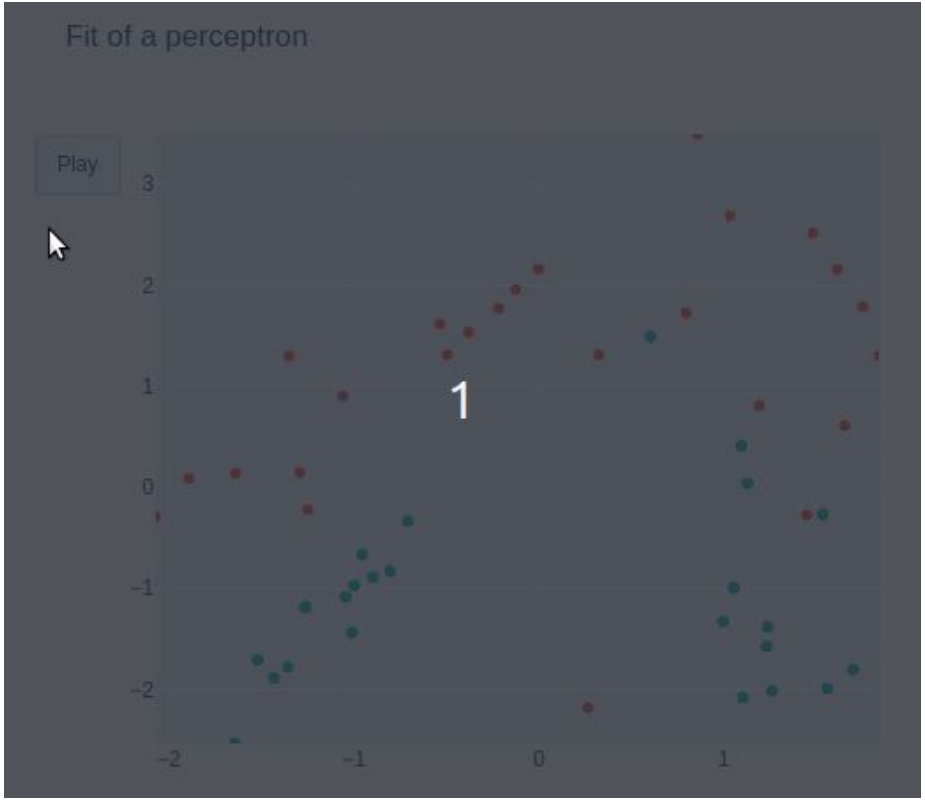
Train in steps: “Epochs”

At each step, calculate the predicted label of the training data(either 0 or 1)

Calculate the “loss” (i.e. a value that is a function of the error between predicted labels and true labels)

Calculate the derivative of the loss w.r.t the weights

Adjust the weights: Add the value of the derivative vector multiplied by some small value

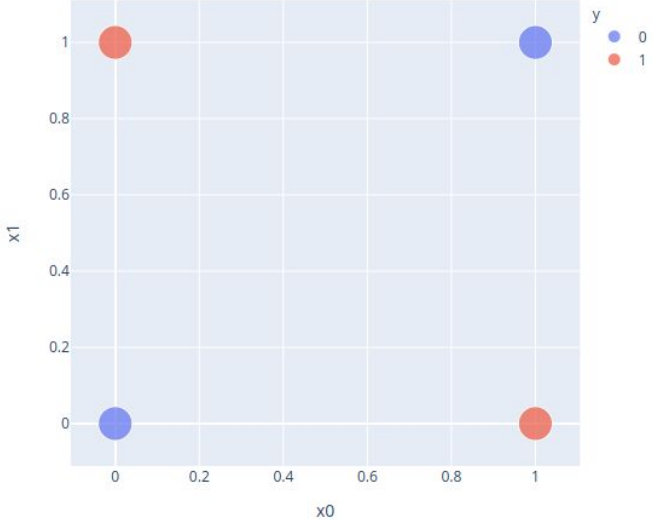


Limitations (Big ones!)

The perceptron can only solve **Linearly separable problems!**

Won’t necessarily converge on the best solution

XOR



x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Enter… the Multi Layer Perceptron (MLP)

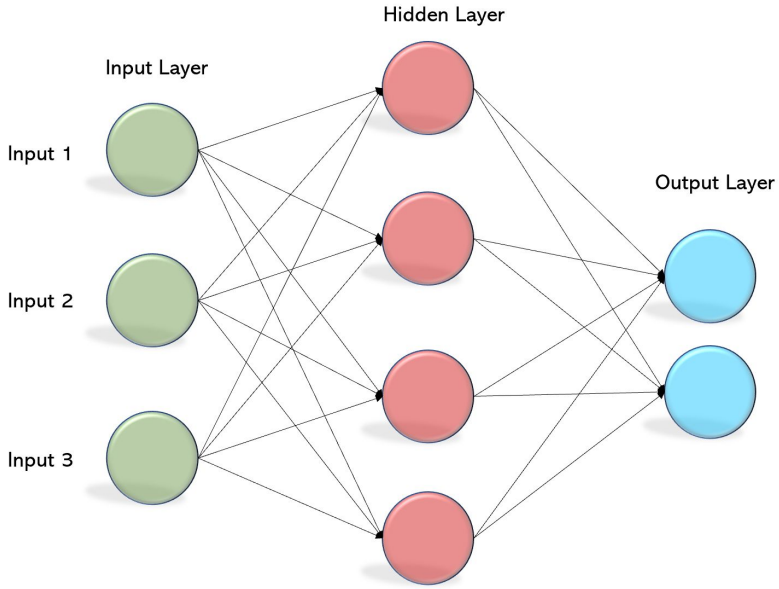
Introduce a “hidden layer” of perceptrons (nodes, neurons)

Outputs of one layer propagate forward

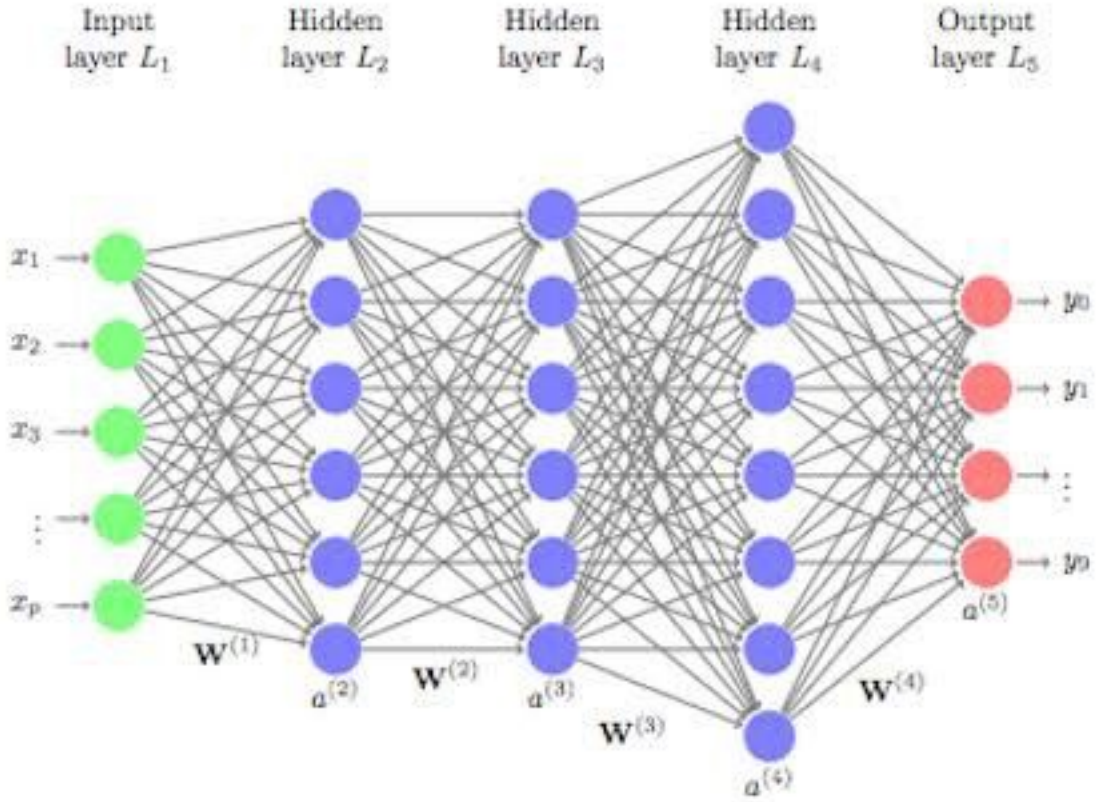
“Fully connected”: Each node in a layer receives the outputs of all nodes in the previous layer as an input vector

Additional layers allow the network to learn product-wise combinations of input features (consequence: Can fit non-linear distributions)

This is a neural network!



MLP Continued

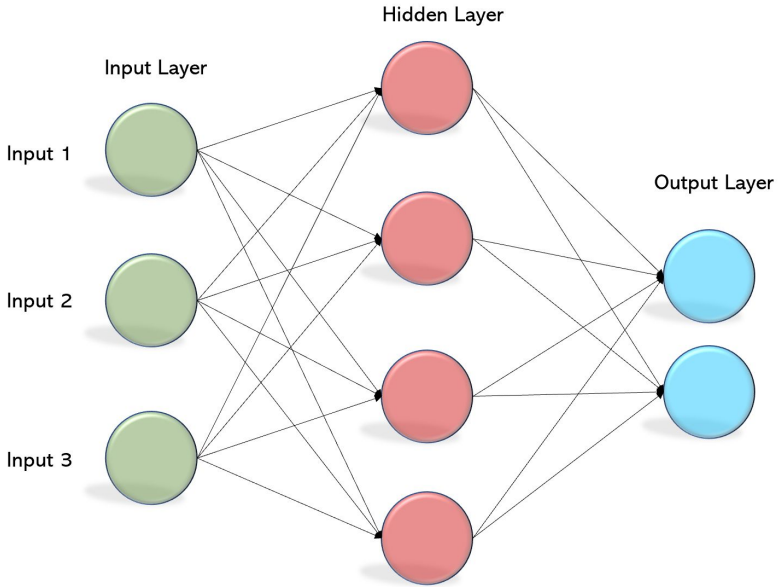


How a multi-layered network trains (high level!)

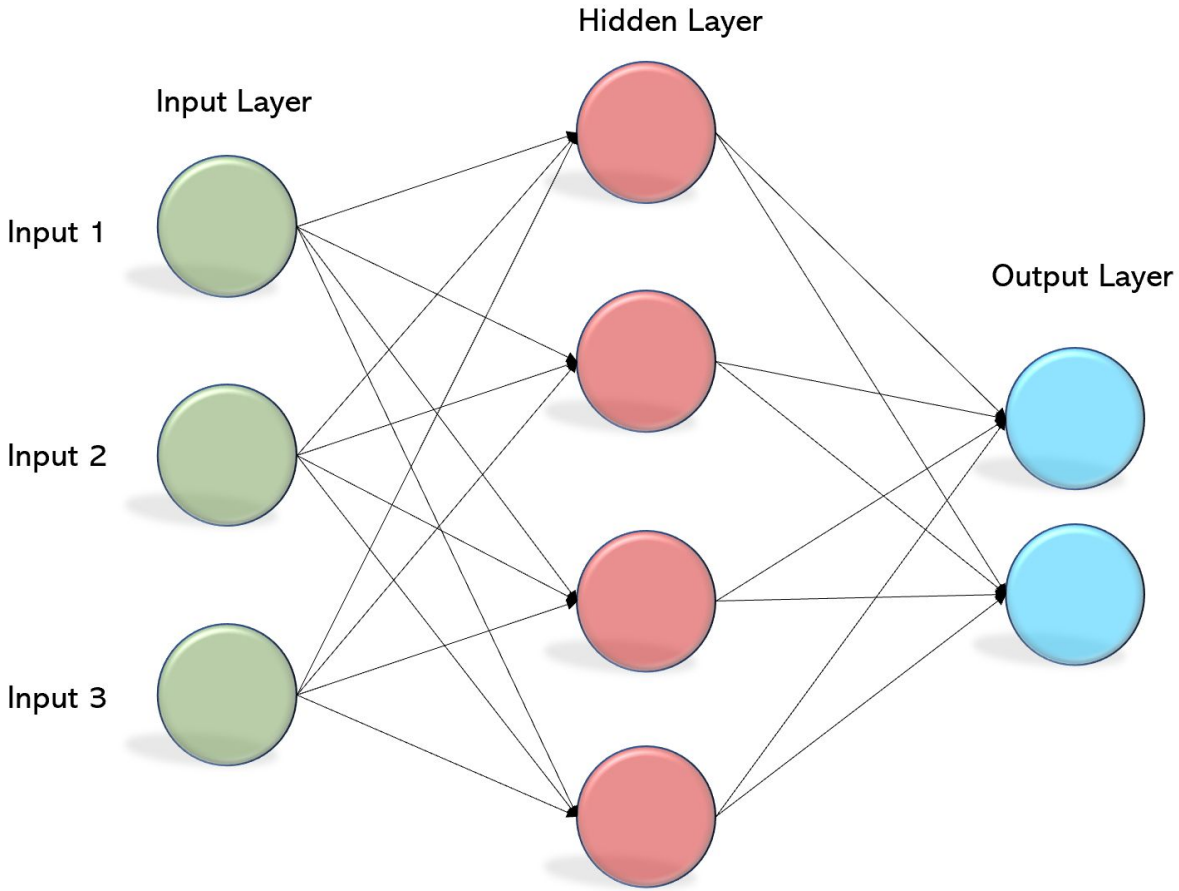
Forward Pass and Backward Pass

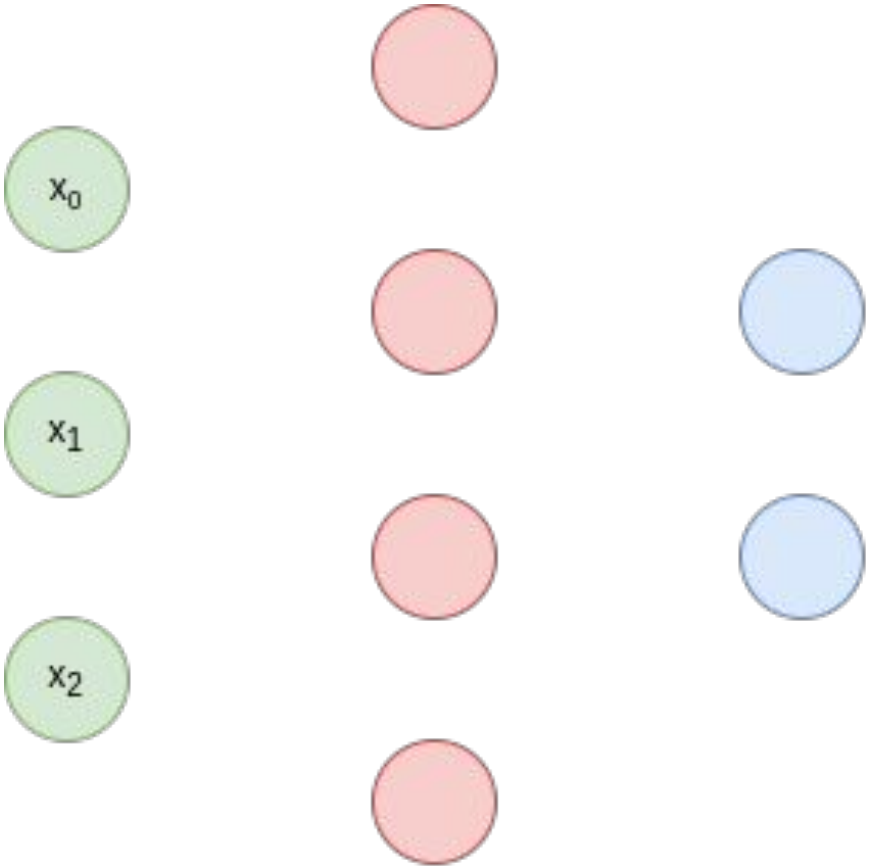
Forward is easy enough to understand: Each node in one layer passes its output as input to **every node in the subsequent layer**.

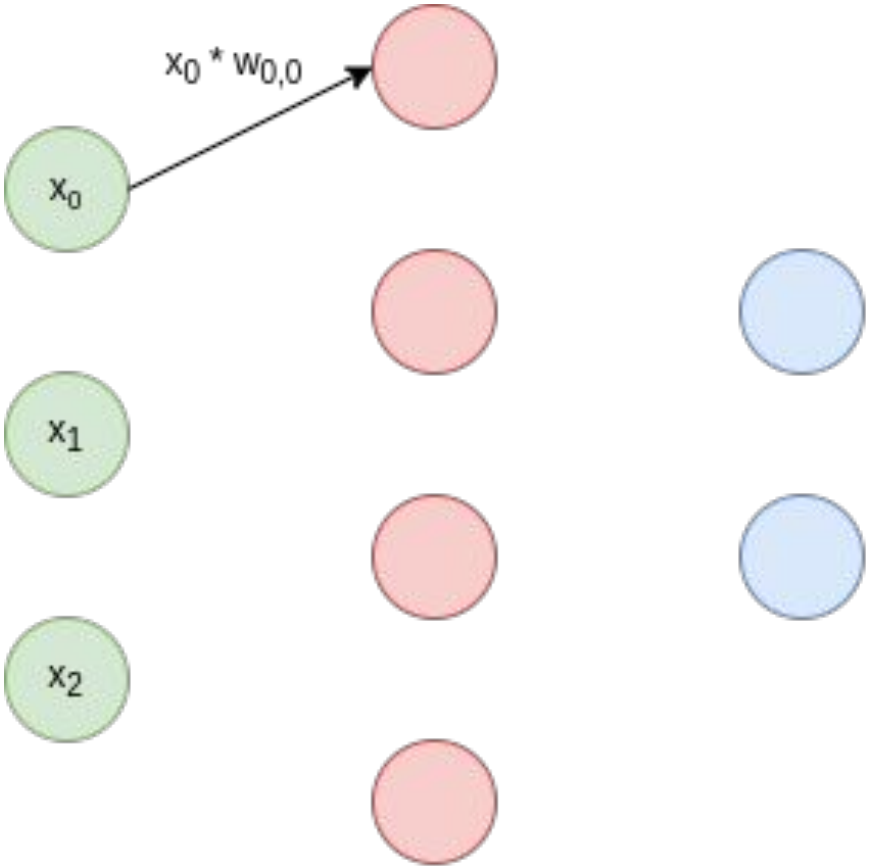
Backward pass: Calculate the **loss** of the network's predictions and **back-propagate** the error



Going Forward

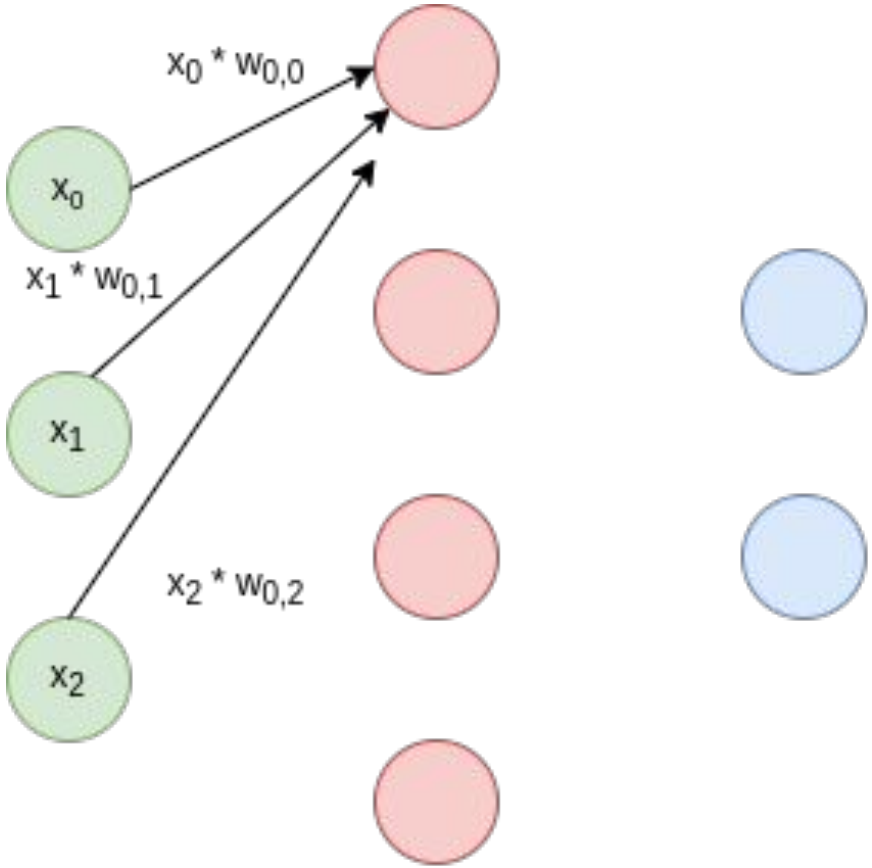

Hidden Layer
Input Layer
Input 1
Input 2
Input 3
Output Layer


x_0
x_1
x_2


$x_0 * w_{0,0}$
x_0
x_1
x_2

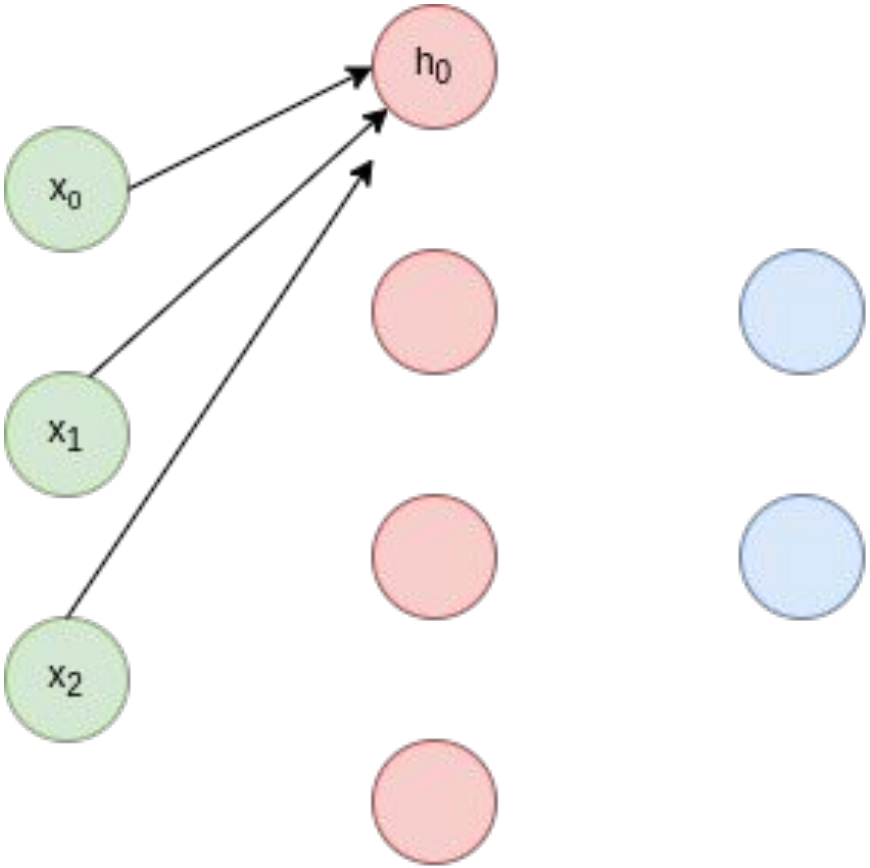
$$h_j = f(\sum_i^n x_i w_{i,j})$$

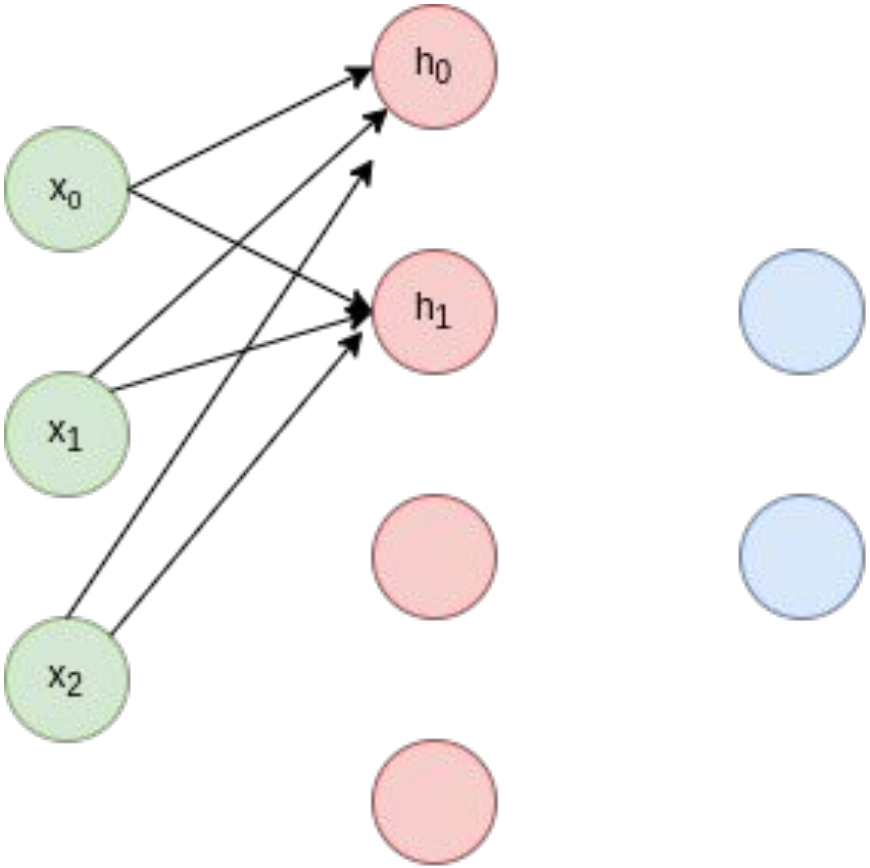
where *f* is an activation function

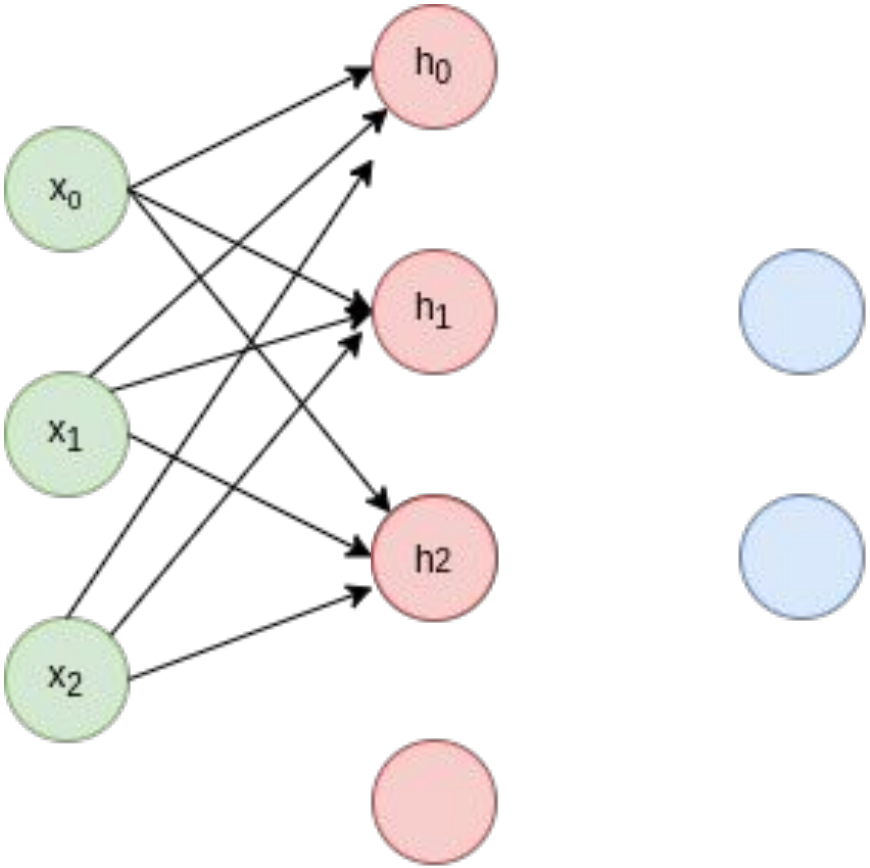


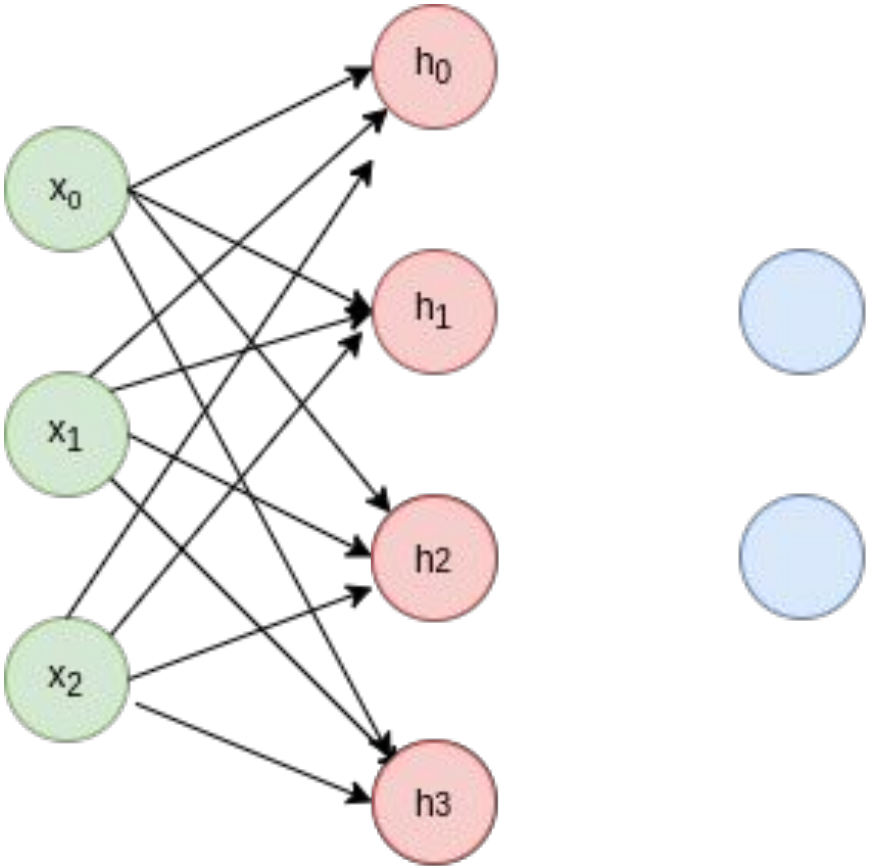
$$h_j = f(\sum_{i}^{n} x_i w_{i,j})$$

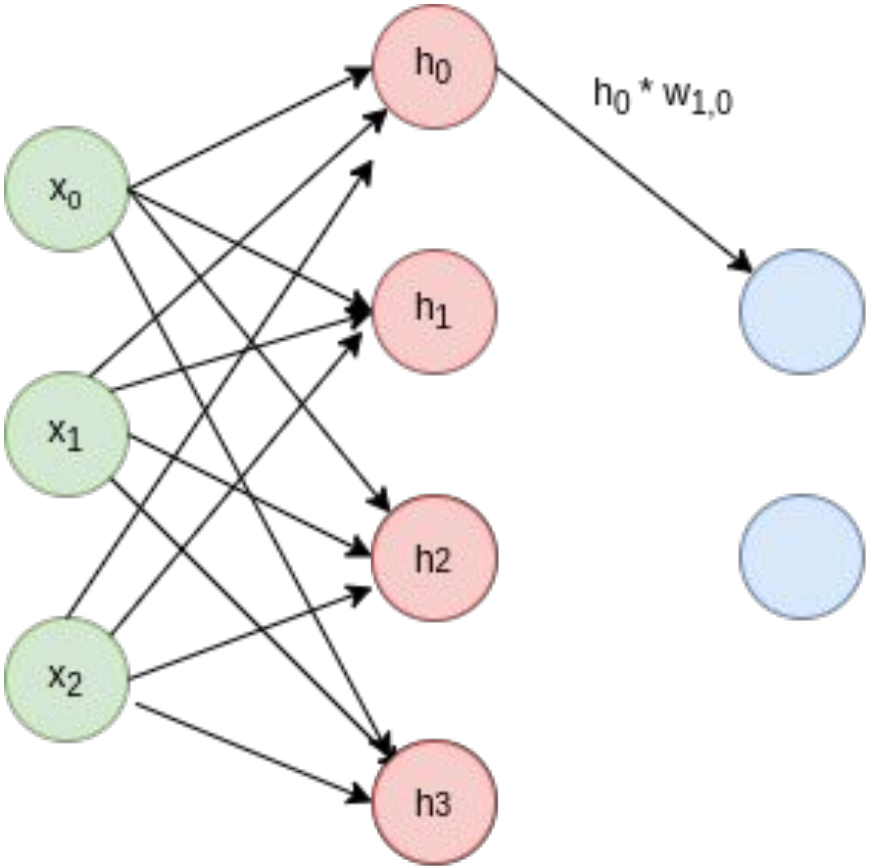
where *f* is an activation function

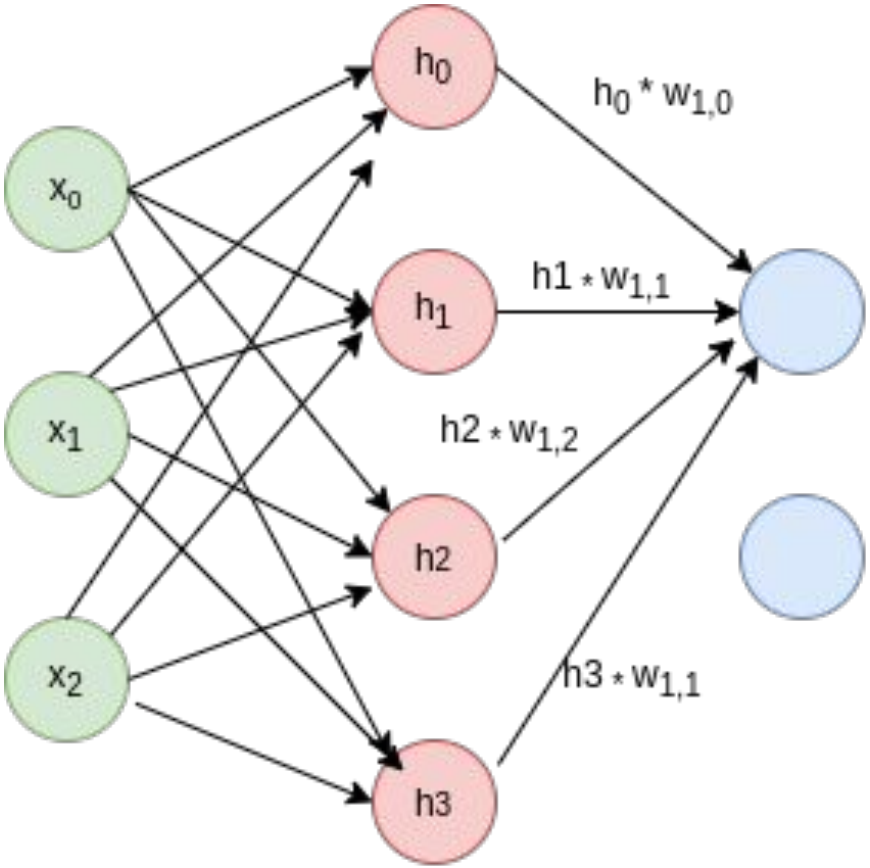


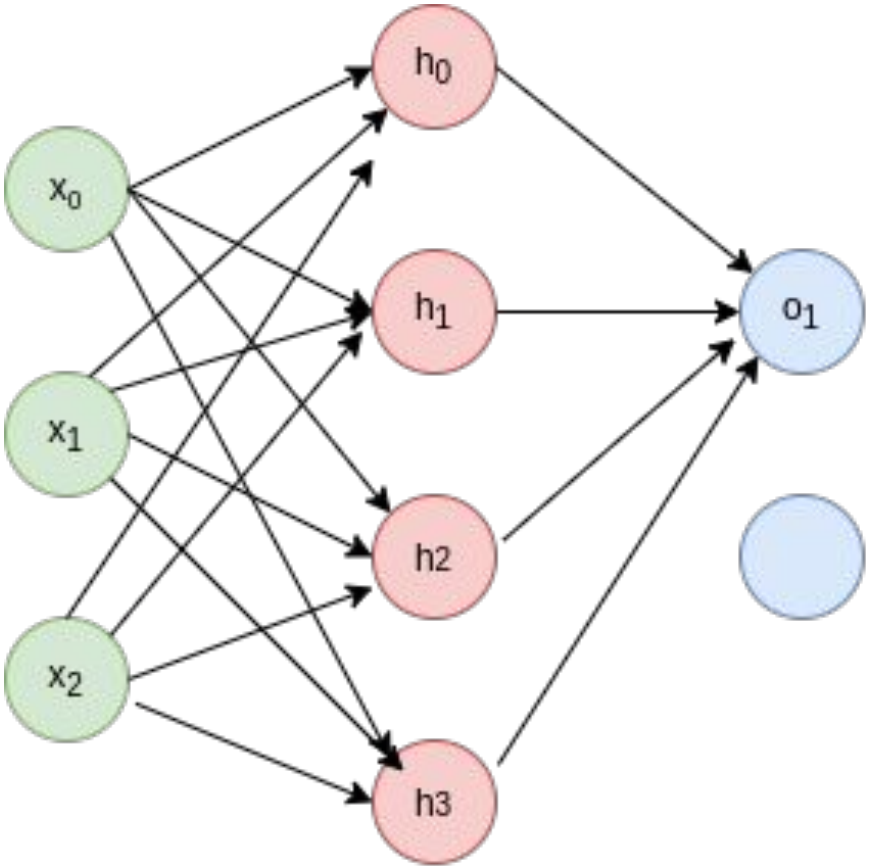

h_0
x_0
h_1
x_1
x_2

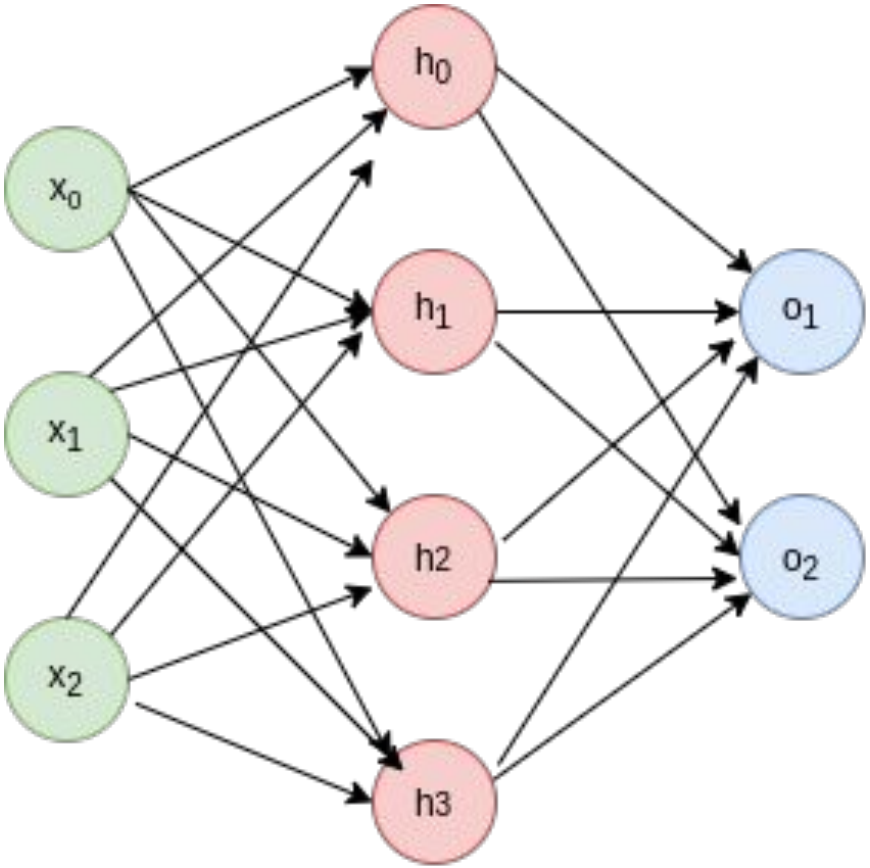

h0
x0
h1
x1
h2
x2


h_0
x_0
h_1
x_1
h2
x_2
h3


h0
h0 * w1,0
x0
h1
x1
h2
x2
h3


h0
h0 * w1,0
x0
h1
h1 * w1,1
x1
h2 * w1,2
h2
x2
h3 * w1,1
h3


h_0
x_0
h_1
o_1
x_1
h2
x_2
h3



$$C = L(y, \mathbf{o})$$

Going Backward

But first, a bit of calculus!

Introduce the concept of “Gradient descent”

The gradient is the transpose of the derivative of the output w.r.t the input. (For those well-versed in linear algebra, it is the Jacobian matrix of the weight vectors)

Intuitively, think of the gradient as the “direction” of the derivative of the loss function of the neural network

Recall from calculus: For some real valued function *f(x)*, the derivative *f’(x)* = 0 when *f(x)* is at a minima.

Gradient descent: Minimize the loss function by moving “down” the gradient

$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \ldots, \frac{\partial C}{\partial x_m}]$$

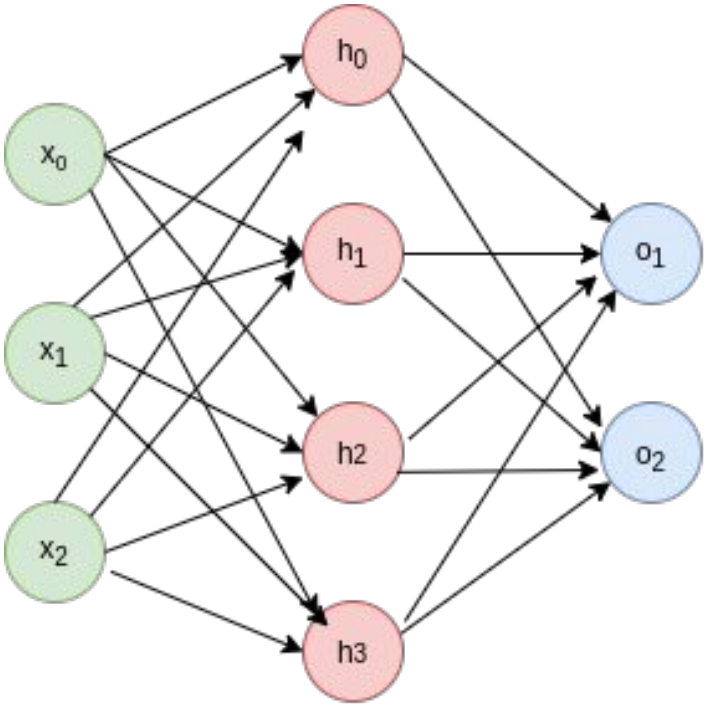
Gradient of a function C in vector **x**

In simple terms:

-Calculate the error
-Propagate the error using partial derivatives (gradient)
-Adjust weights using the value of the gradient

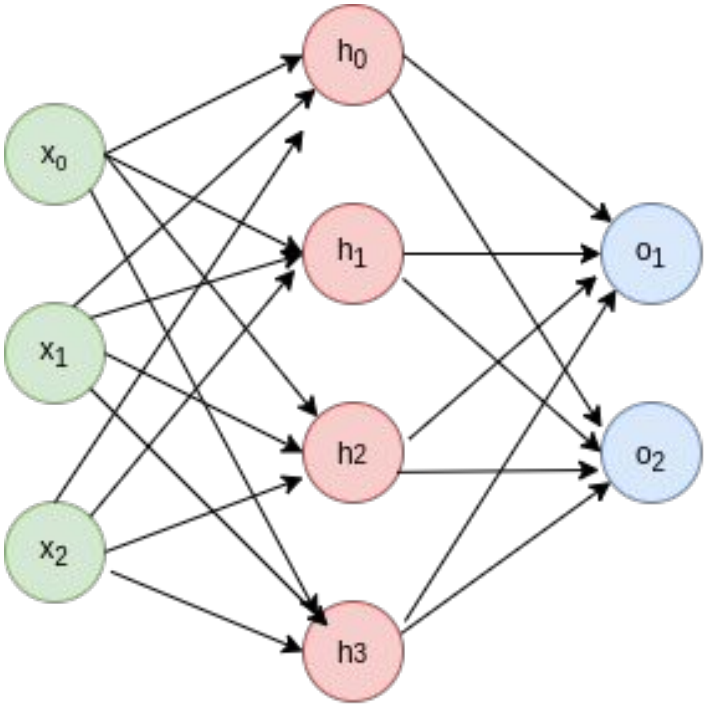
$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \ldots, \frac{\partial C}{\partial x_m}]$$

This is a gradient; the partial derivatives of a function C w.r.t a vector **x**



$$C = L(y, \mathbf{o})$$

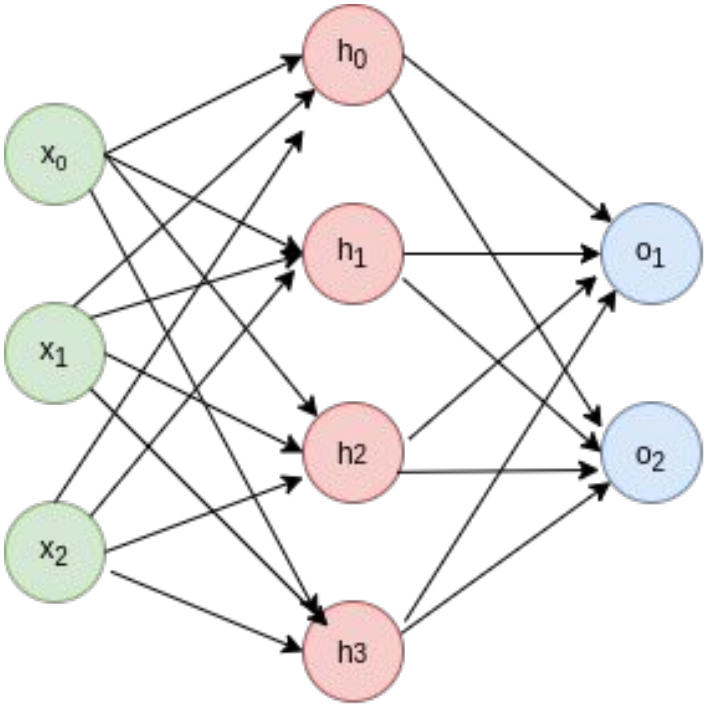
$$\delta^L = \frac{\delta C}{\delta \mathbf{o}}$$



$$C = L(y, \mathbf{o})$$

$$\delta^{\mathbf{l}} = ((\mathbf{w}^{l+1})^T \delta^{l+1}) \odot \frac{\delta C}{\delta z_{l-1}}$$

$$\delta^L = \frac{\delta C}{\delta \mathbf{o}}$$



$$C = L(y, \mathbf{o})$$

$$\delta^{\mathbf{l}} = ((\mathbf{w}^{l+1})^T \delta^{l+1}) \odot \frac{\delta C}{\delta z_{l-1}}$$

$$\delta^L = \frac{\delta C}{\delta \mathbf{o}}$$

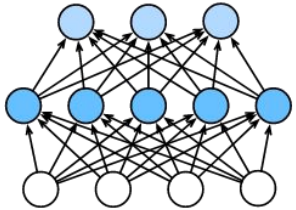
$$\mathbf{w} = \mathbf{w} - \alpha \delta^l$$

Going deeper: The Block

When constructing large neural networks, it is common to repeat a pattern multiple times

These patterns are called “Blocks”

Using blocks allows the design of complex and otherwise hard to describe architectures



Large models and what they can do

With layers and blocks, we can create hugely complex models

More complex models can solve more complex problems (e.g. image recognition)

Deeper models can learn underlying patterns in the data

More layers/blocks means more parameters, which require (LOTS) more data to train

More complex models are more susceptible to overfitting

More complex models are harder to interpret

More hyper parameters means more difficulty finding optimal model design

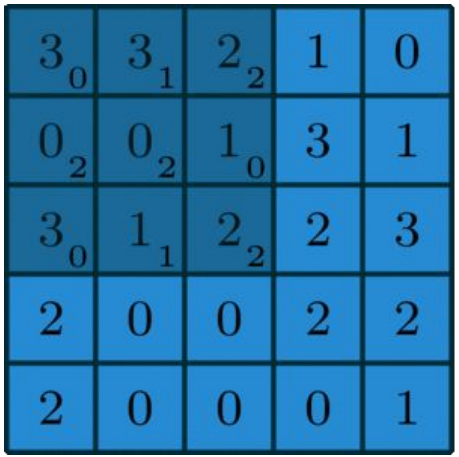
Going further: The Convolution Operation

The convolution is an operation that allows processing of multi-dimensional data

In convolution, a kernel is passed over the input data in a sliding window fashion. Each area that is covered by the kernel is transformed to a single scalar value.

These operations reduce the size of the matrix and encode its data

(Note: It is possible to prevent reduction in data size by “padding” after the convolution, but we’ll leave that as an aside)

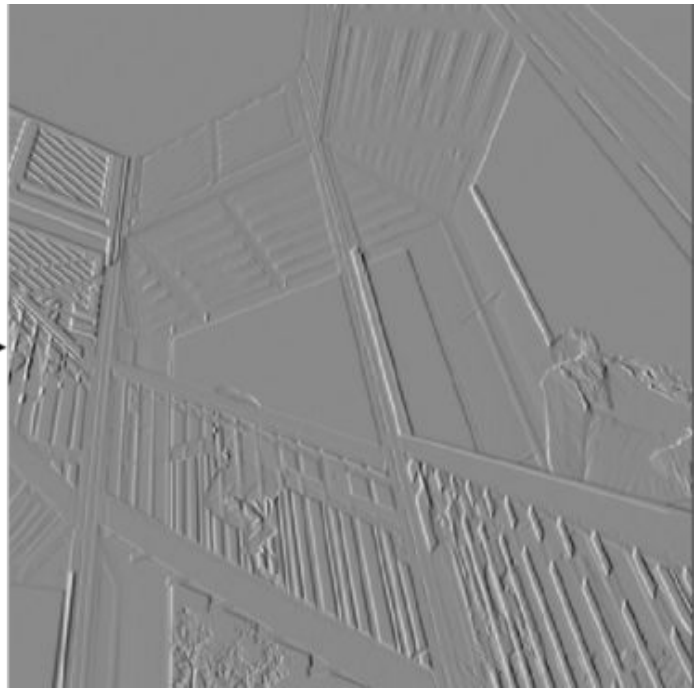


A 3x3 convolution kernel transforming a 5x5 matrix.

Convolutions can transform data in useful ways

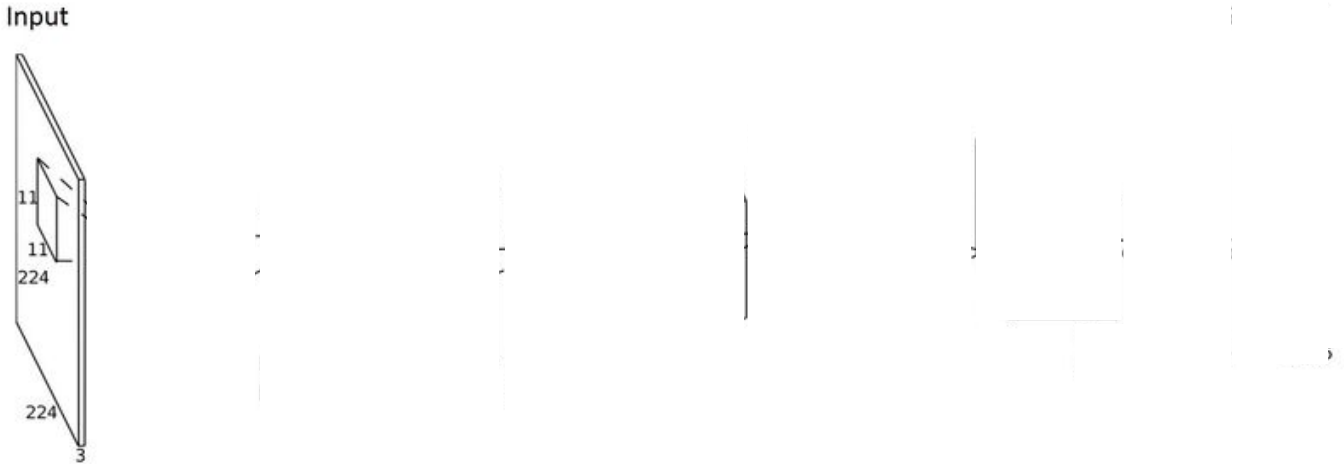
$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

Horizontal Sobel kernel



e.g. Applying this specific kernel to an image results in “edge detection”

Putting it together: The Convolutional Neural Network (CNN)



CNNs are probably what you think of when you hear “Deep Learning”

Apply a convolution operation to the outputs of a layer: “Convolutional layer”

Convolution kernel thought to capture high-level features of data

Kernel values are learned via gradient descent

“Compression” effect of convolution can result in larger networks due to reduced parameters

Applications of CNNs in Bioinformatics

Protein folding prediction (Alphafold)

Medical Image Analysis

Gene Expression Analysis (e.g. cancer survival prediction)

Various classification problems

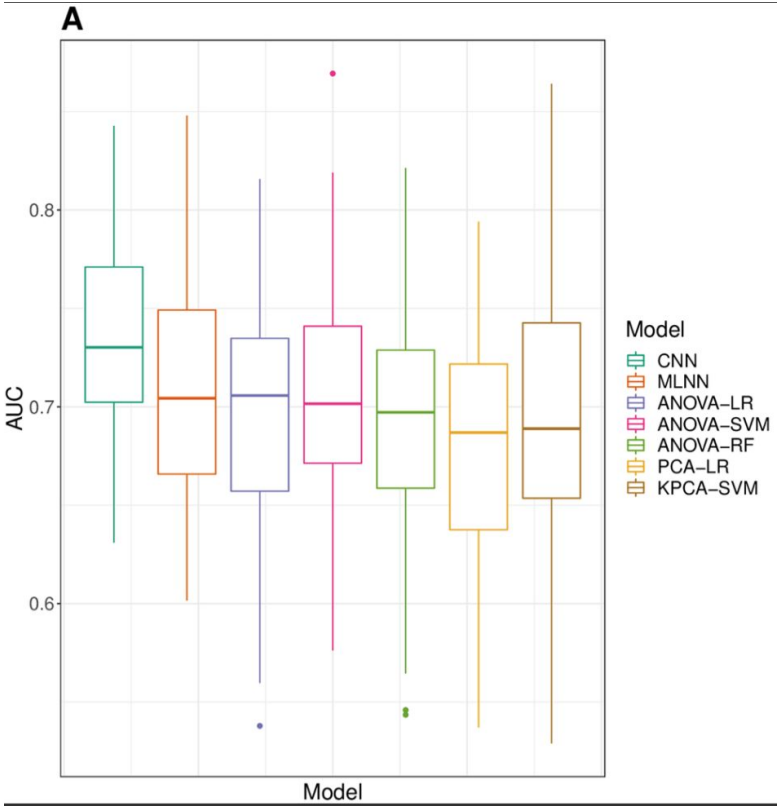
Representation learning

A word of caution

“Convolutional Neural Network”, “Deep Neural Network”, and “Deep Learning” have become somewhat trendy, leading to some questionable papers

Always read results carefully, and ask:

> Did the model significantly improve the state of the art on an important task?
>
> Is the model architecture itself new or interesting in some way?
>
> Can we learn something about our data that existing methods couldn’t teach us?

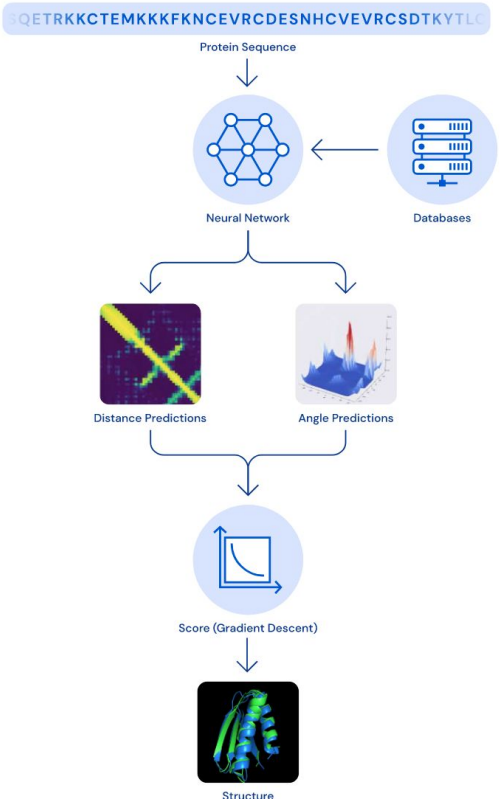


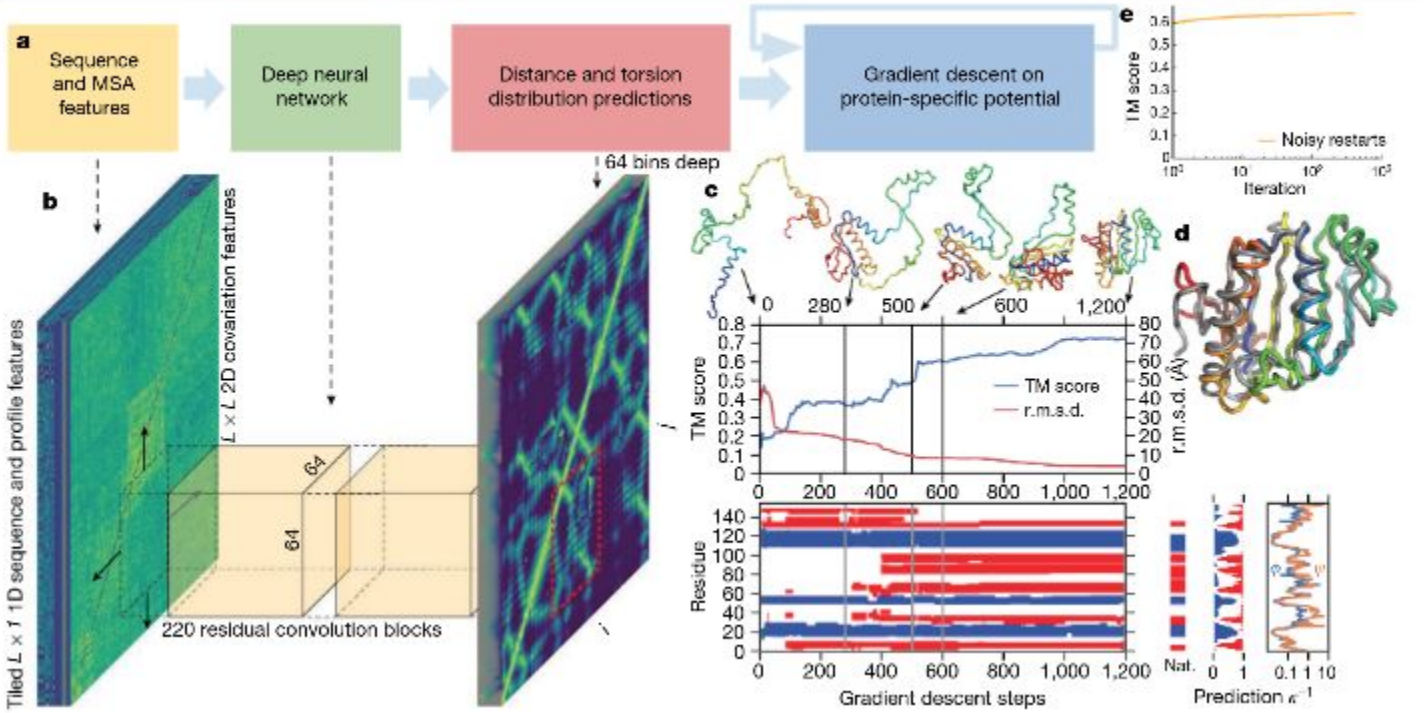
(Back to the good stuff) Protein Folding (AlphaFold)

Computationally intractable by first principles

Tertiary structure modelling is expensive and difficult; fuelling interest in prediction from primary structure

(Very) deep neural network finds most likely configuration; highly successful at modelling




a
Sequence and MSA features
Deep neural network
Distance and torsion distribution predictions
Gradient descent on protein-specific potential
b
64 bins deep
L × L 2D covariation features
Tiled L × 1 1D sequence and profile features
64
64
220 residual convolution blocks
c
0
280
500
600
1,200
TM score
r.m.s.d.
r.m.s.d. (Å)
Residue
Gradient descent steps
d
e
TM score
Noisy restarts
Iteration
Nat.
Prediction

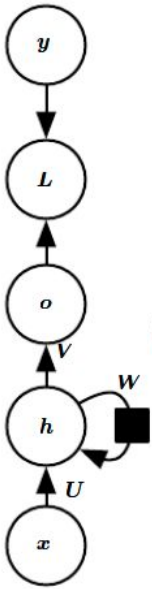
Analysing sequences: The Recurrent Neural Network (RNN)

ML with sequences is challenging: variable length inputs, long term dependencies

Recurrence-based neural networks model state-over-time

Multiple applications: Classification/regression; sequence modelling

Sequence length is limited: “Vanishing Gradient Problem”

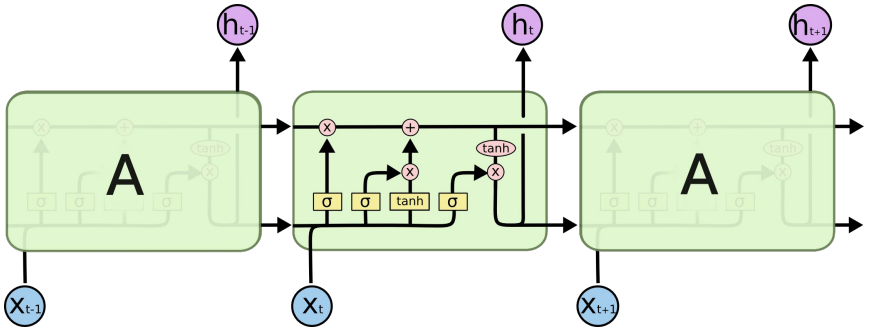


A computational graph showing how recurrent networks can be “unfolded”.

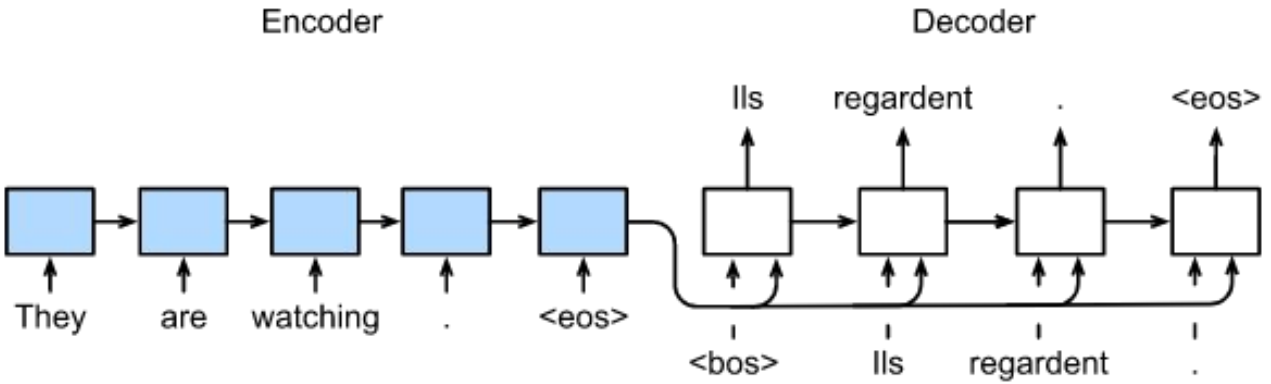
“Memory” in RNNs: GRU, LSTM

Gated Recurrent Unit architecture “gates” the hidden state, allowing the model to only selectively update hidden state based on sequence step (e.g. don’t update for white space

Long Short Term Memory: recurrence units similar to logic gates. Motivating principle is the same as the GRU; Update only when necessary



Sequence-to-Sequence



Sequence-to-sequence describes a family of models: Both input and output are sequences (e.g. machine translation)

An example of the “Encoder-decoder” architecture

Useful for conversion of sequences but also for “autoencoding”, representation learning

Applications of RNNs in Bioinformatics

Sequence classification; e.g. Subcellular localization

Sequence modelling: Viral escape mutation prediction (Seq2Seq)

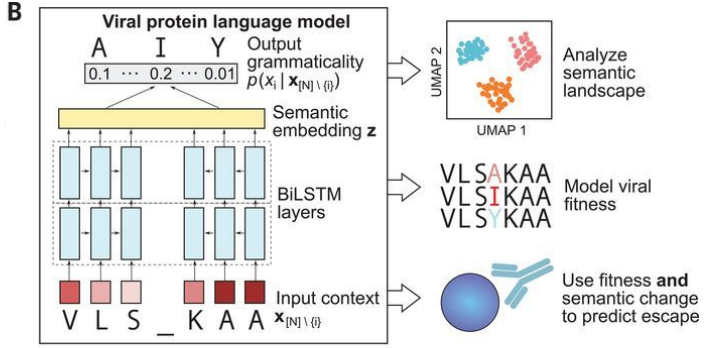
Viral Escape Modeling

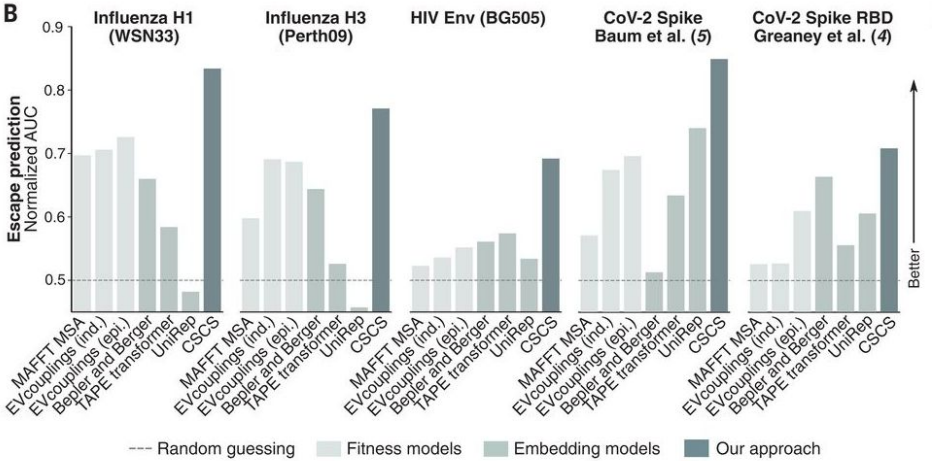
“Viral Escape” is when mutations of virus genomes result in mutations of antigen, allowing the virus to escape the immune system

LSTM “Language modelling” of viral protein sequences: Model learns the “semantics”, “grammar” of virus proteins

After training, predict viral escape based on semantic shift caused by mutations

(published in Science!!!!)





Long-term dependencies without recurrence: Attention

A relatively recent approach to ANNs, inspired by biological information processing (i.e. the brain filters input data using important cues)

Query, Key, and Value vectors

Output is a weighted sum of the Value vector; weights calculated via a function of the Query and Key vectors

Several different functions exist, but Scaled Dot-Product Attention is most commonly used

Unlike RNNs, allows for parallel computing

(drawback: Memory demand scales quadratically with sequence length)

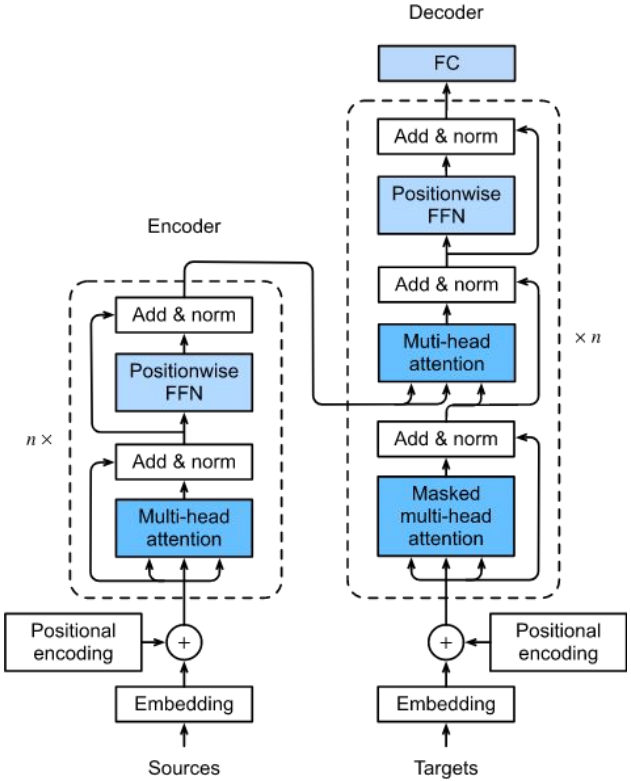
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Transformer

Transformer is a specific ANN architecture that has seen massive popularity and success recently

Positional encoding, self attention, and the position-wise feed-forward layer

Extremely powerful encoder-decoder model: Success in language modelling, “generative modelling”, and recently computer vision



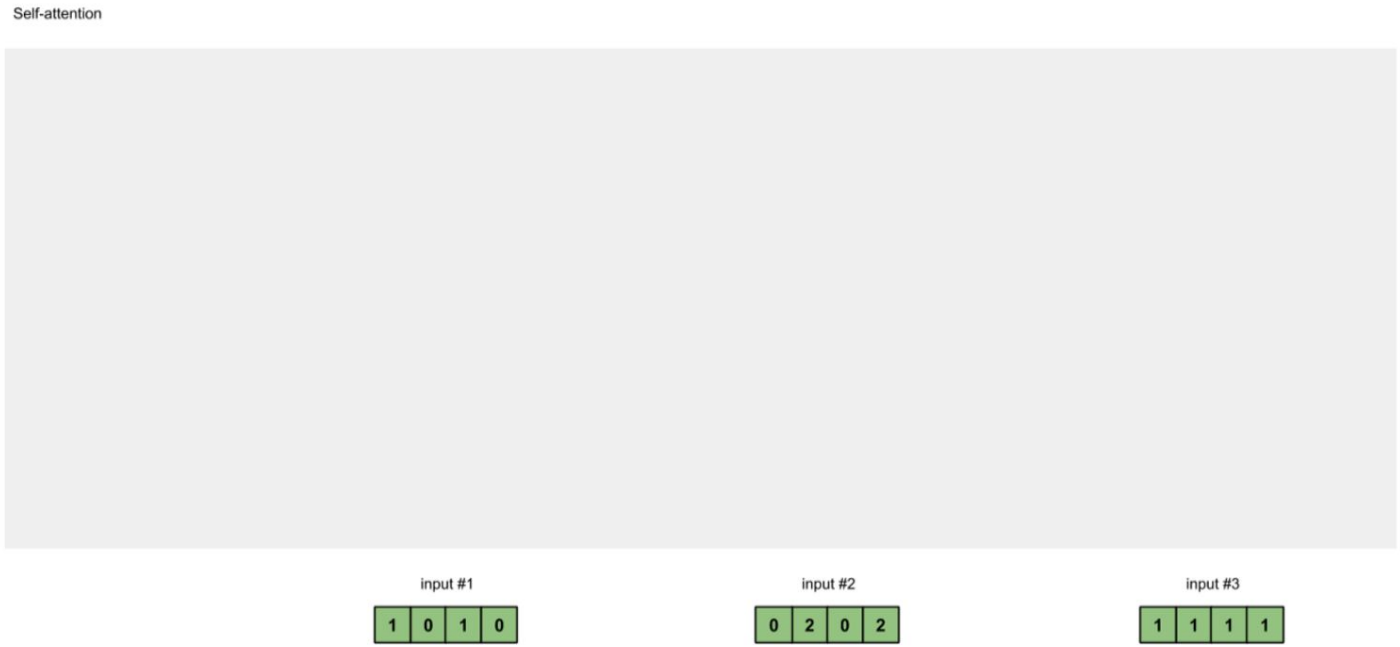
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



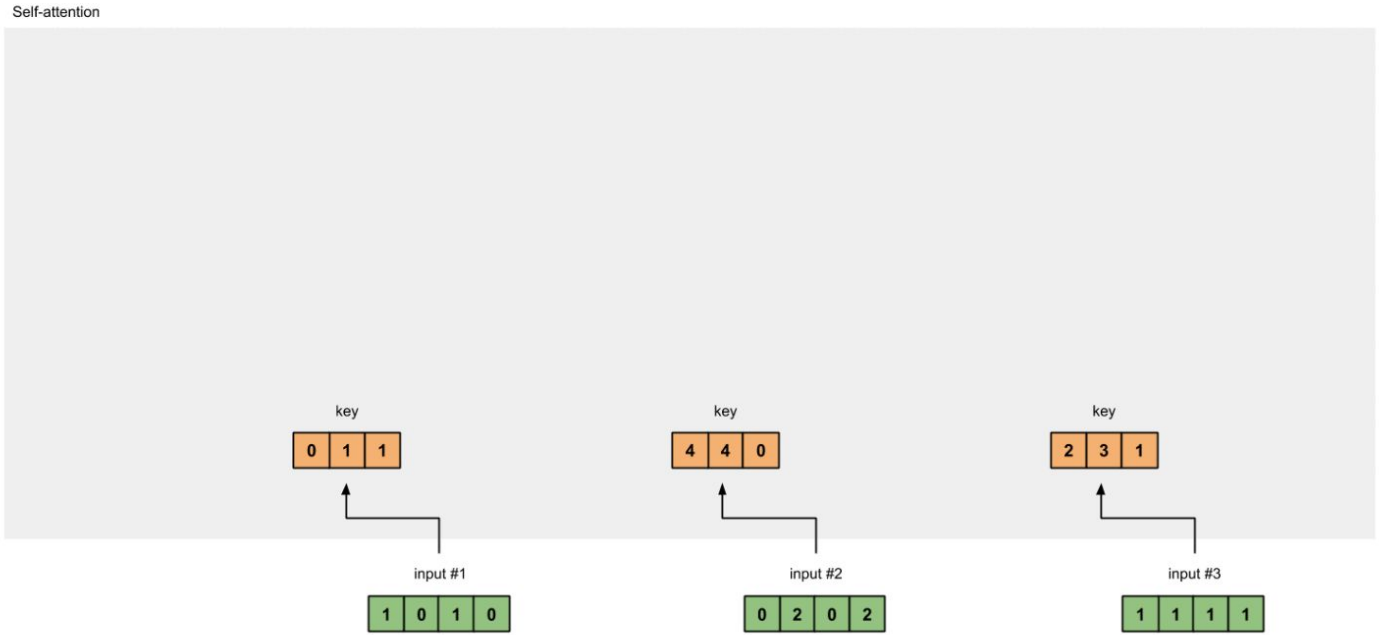
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



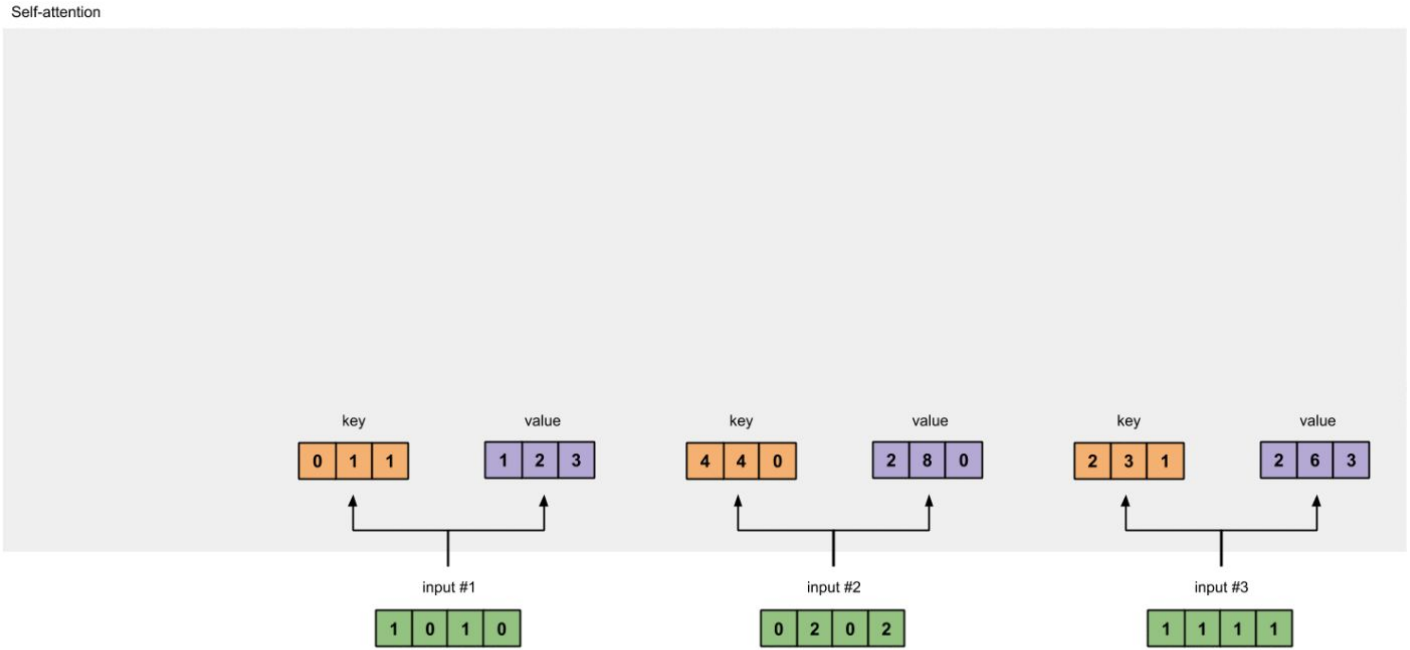
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



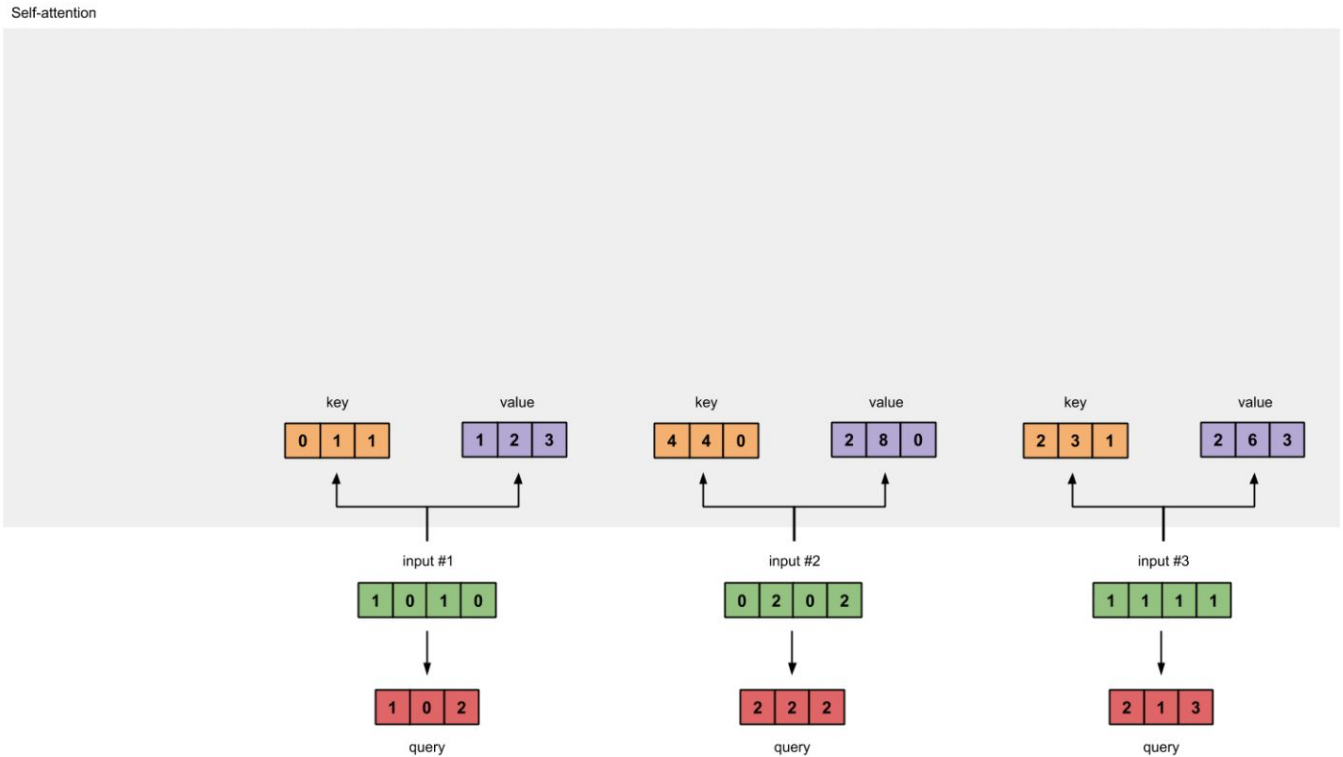
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



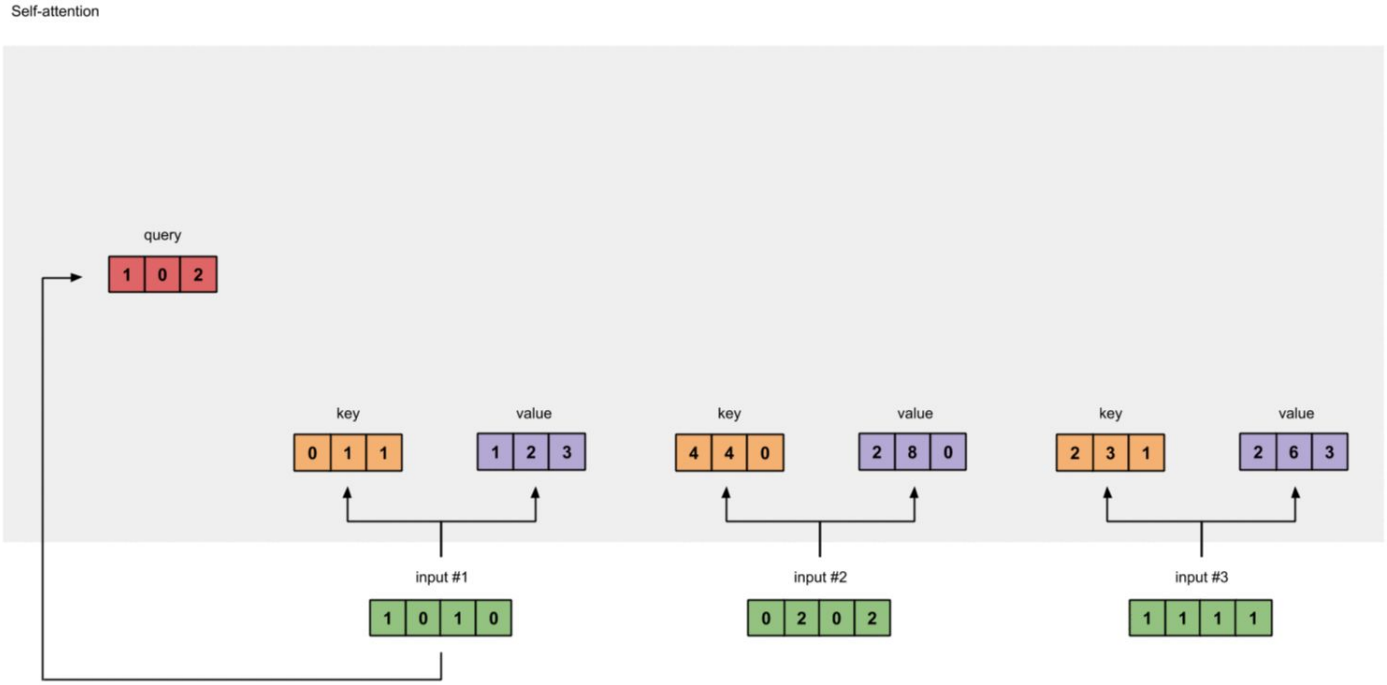
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



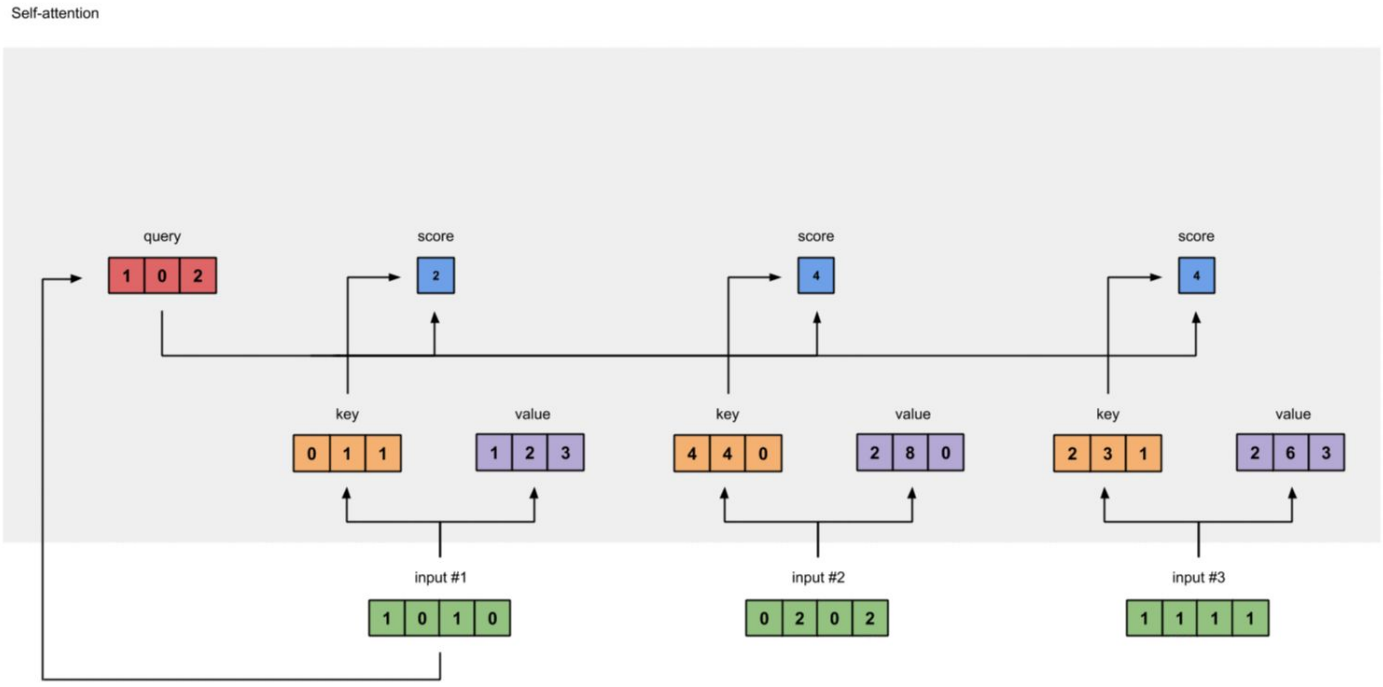
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



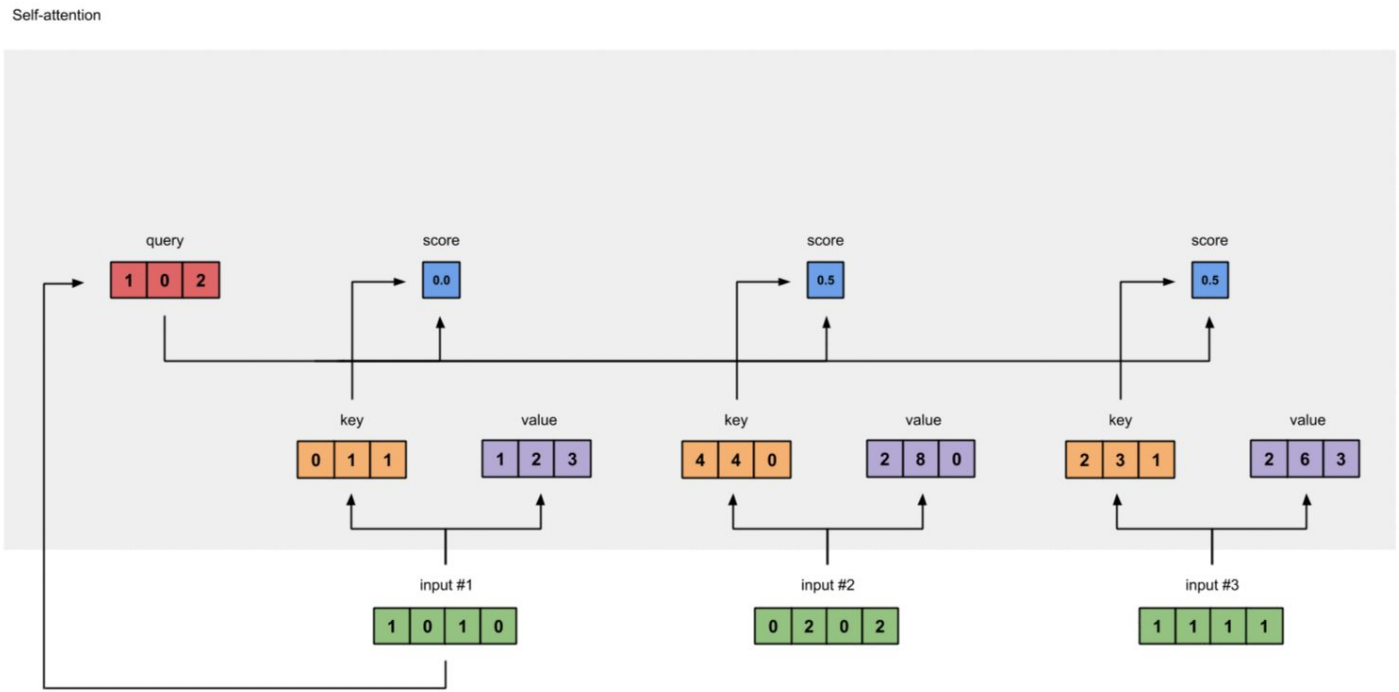
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



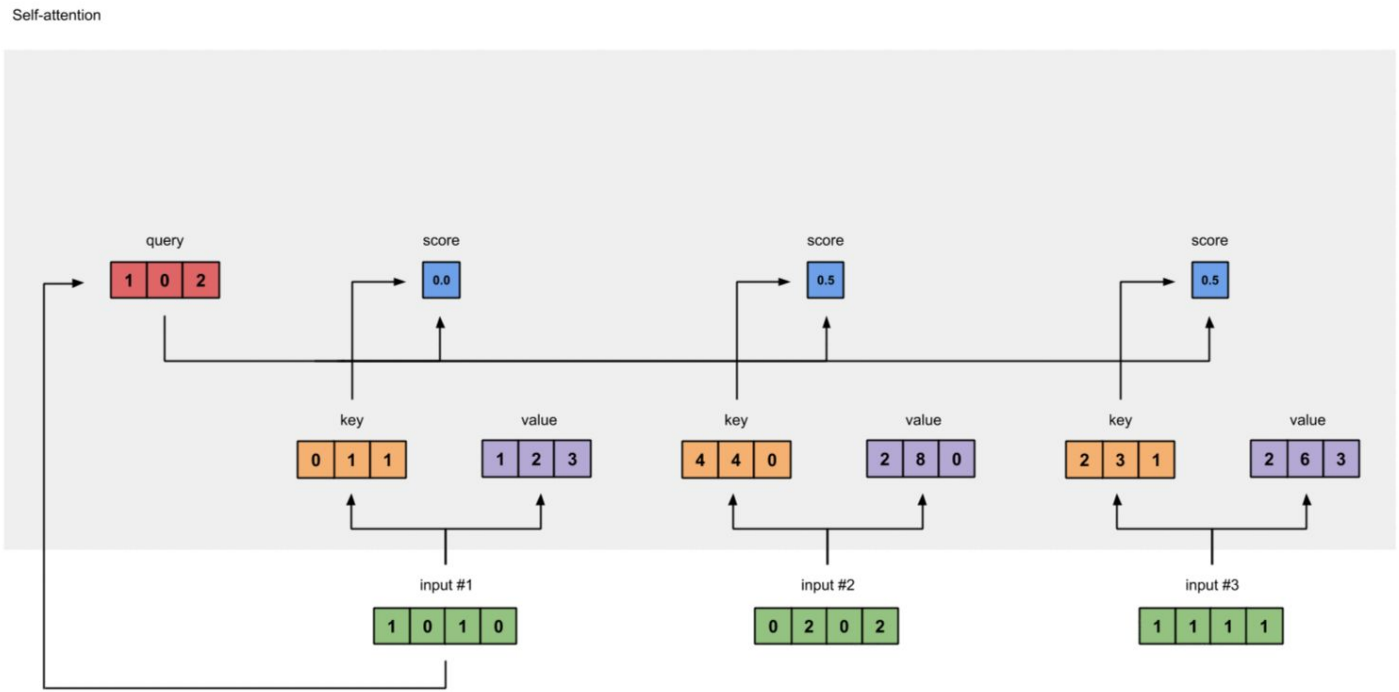
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



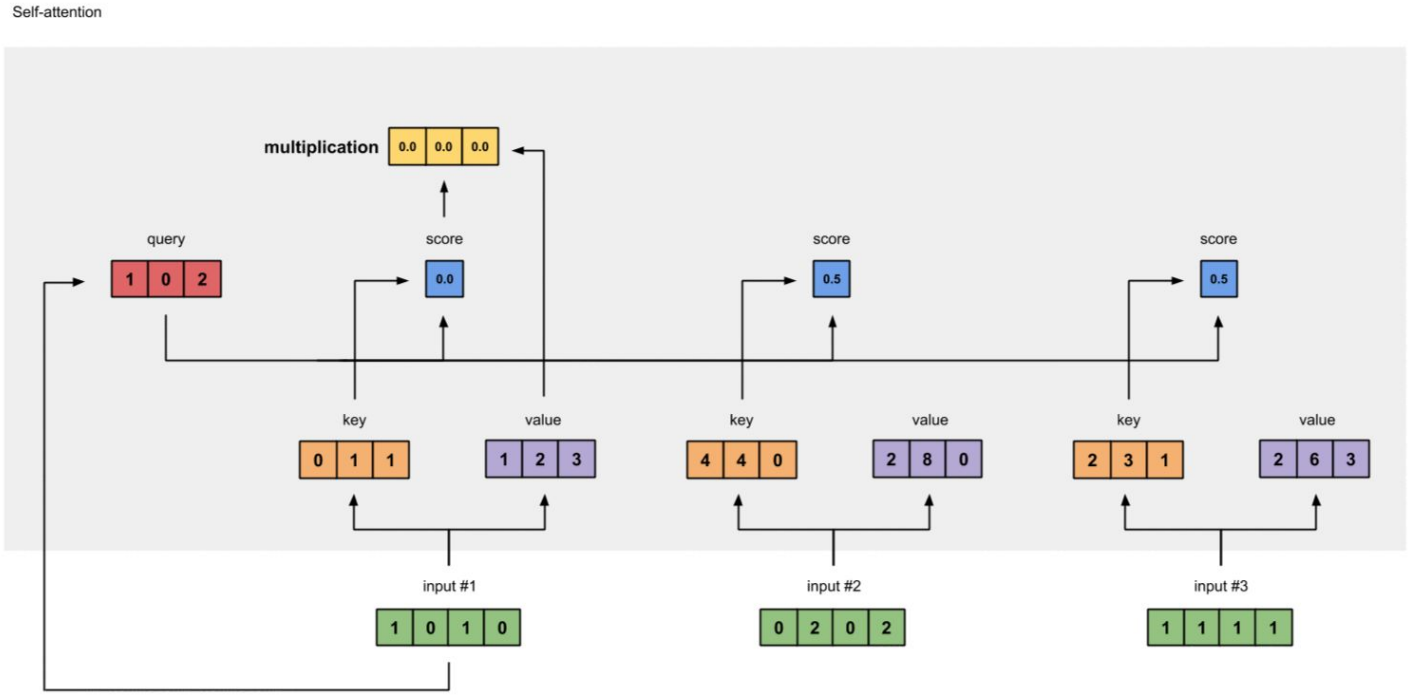
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



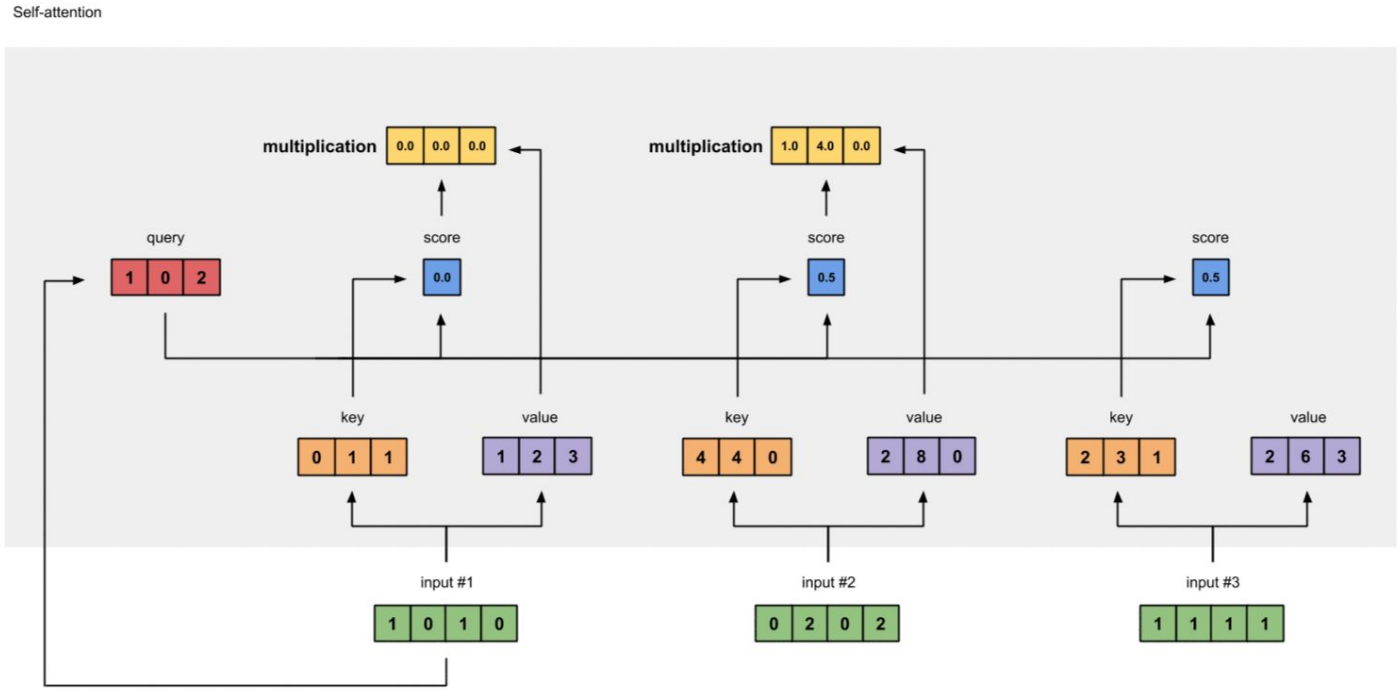
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



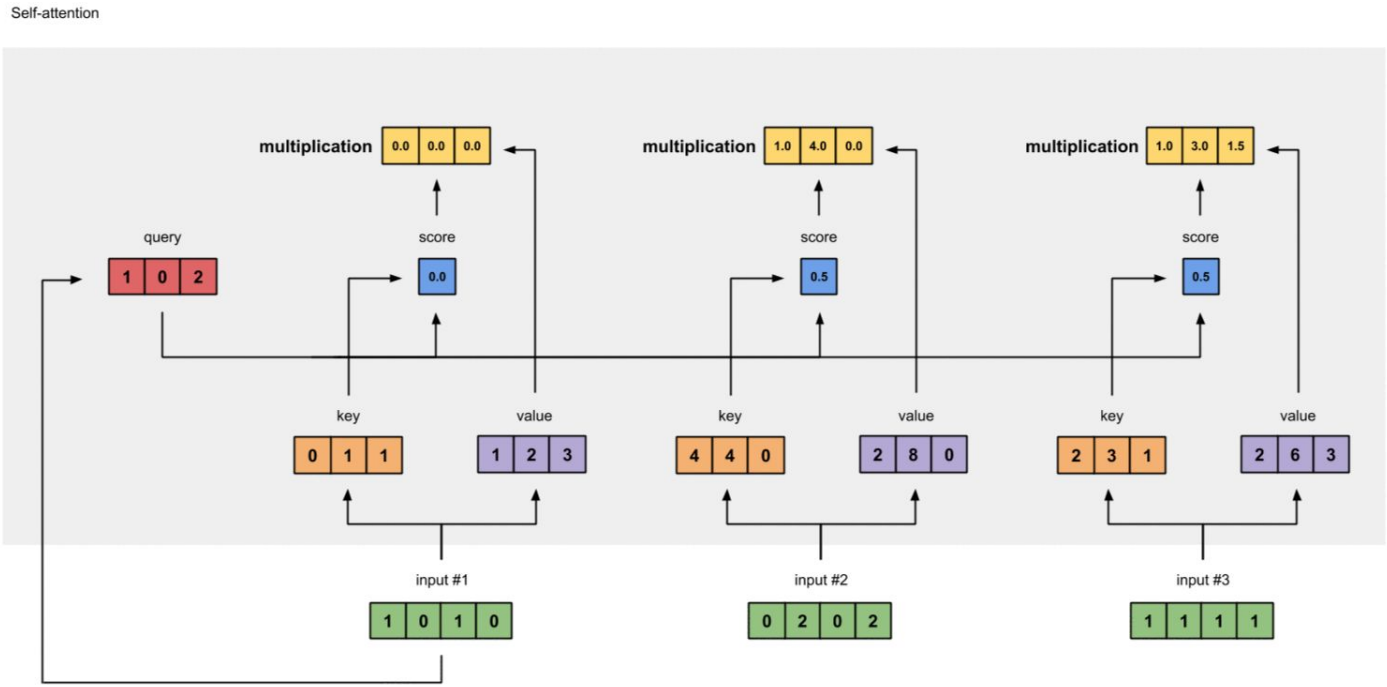
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



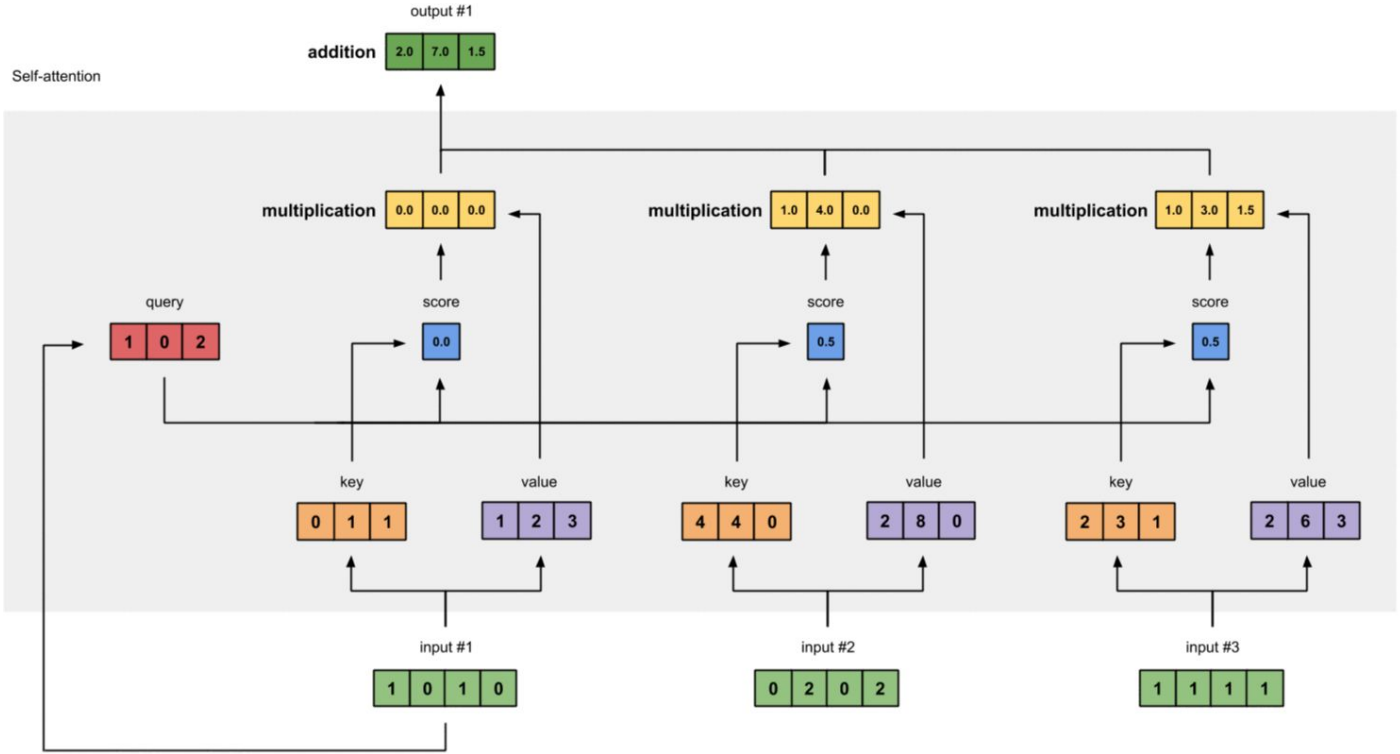
Self-attention demystified (hopefully)

Keep in mind:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, and V are weight vectors!

They are adjusted during training via backprop!



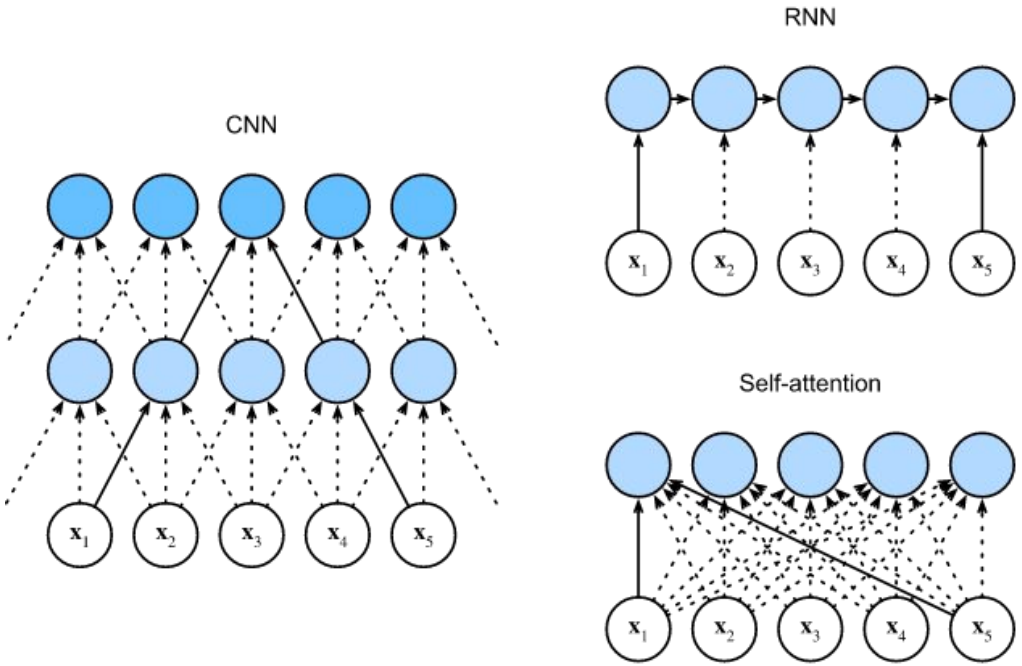


Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

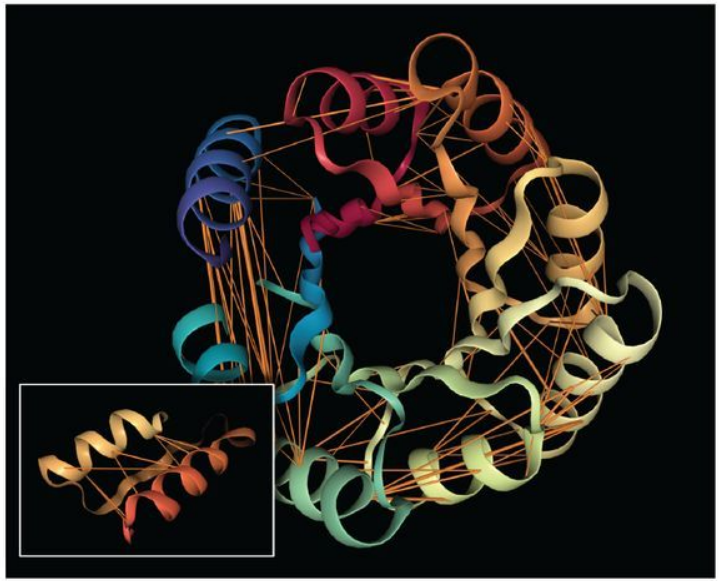
Attention/Transformer in Bioinformatics

Intra-sequence dependencies are very interesting in bioinformatics

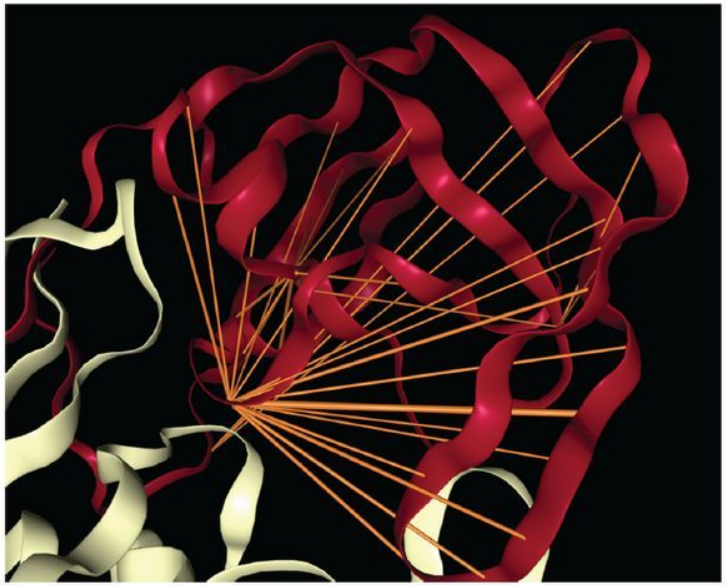
Primarily advances in Representation Learning:

- Protein “Language Modeling”: Use attention mechanism to visualize protein contact sites
- DNA “Language Modeling”: Identify important non-coding DNA sequences, such as promoter binding regions
- Sequence embedding: Create meaningful numeric representations of DNA sequences

Protein modeling: Protein folding contact sites

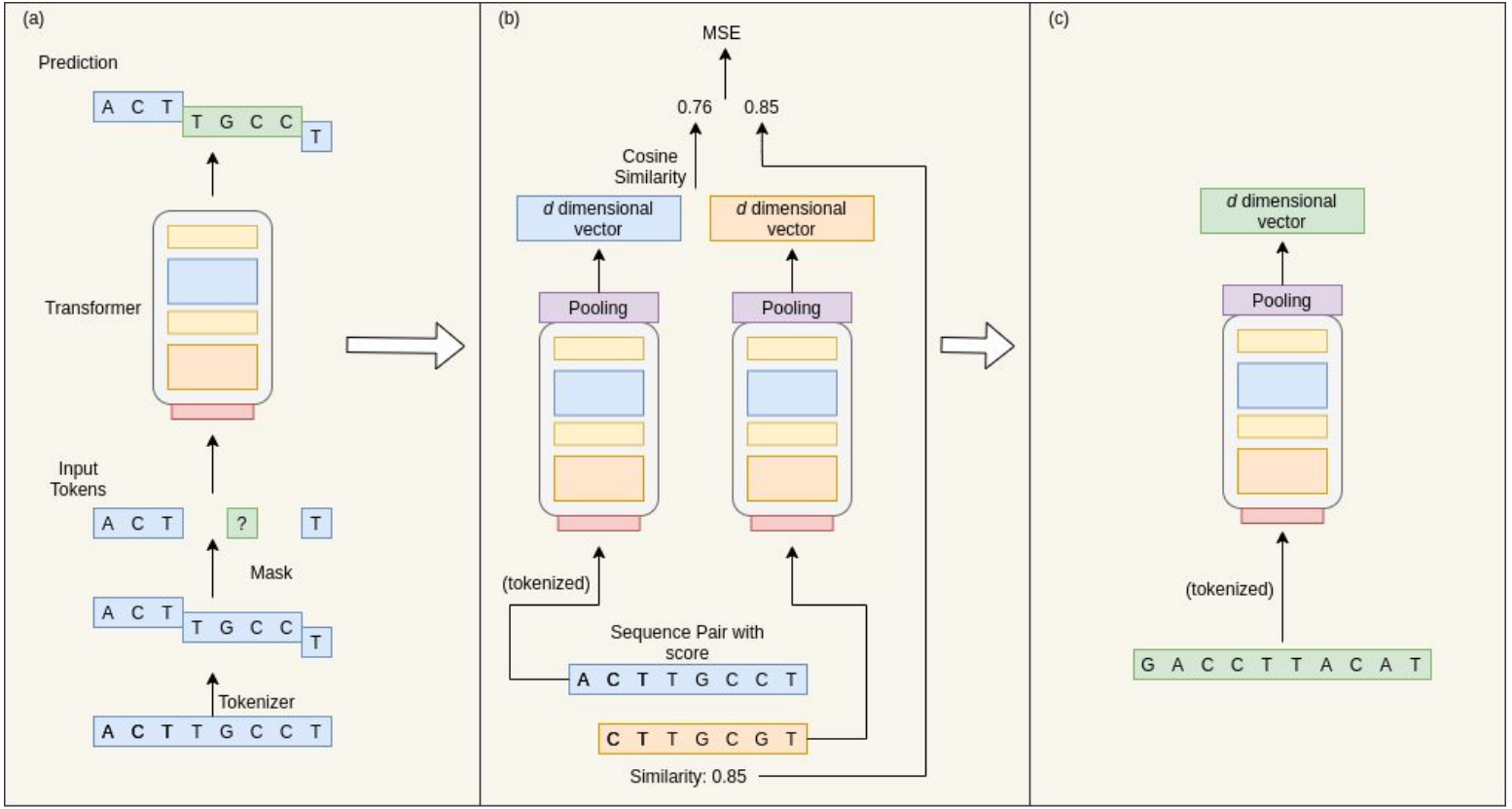


(a) Attention in head 12-4, which targets amino acid pairs that are close in physical space (see inset subsequence 117D-157I) but lie apart in the sequence. Example is a *de novo* designed TIM-barrel (5BVL) with characteristic symmetry.

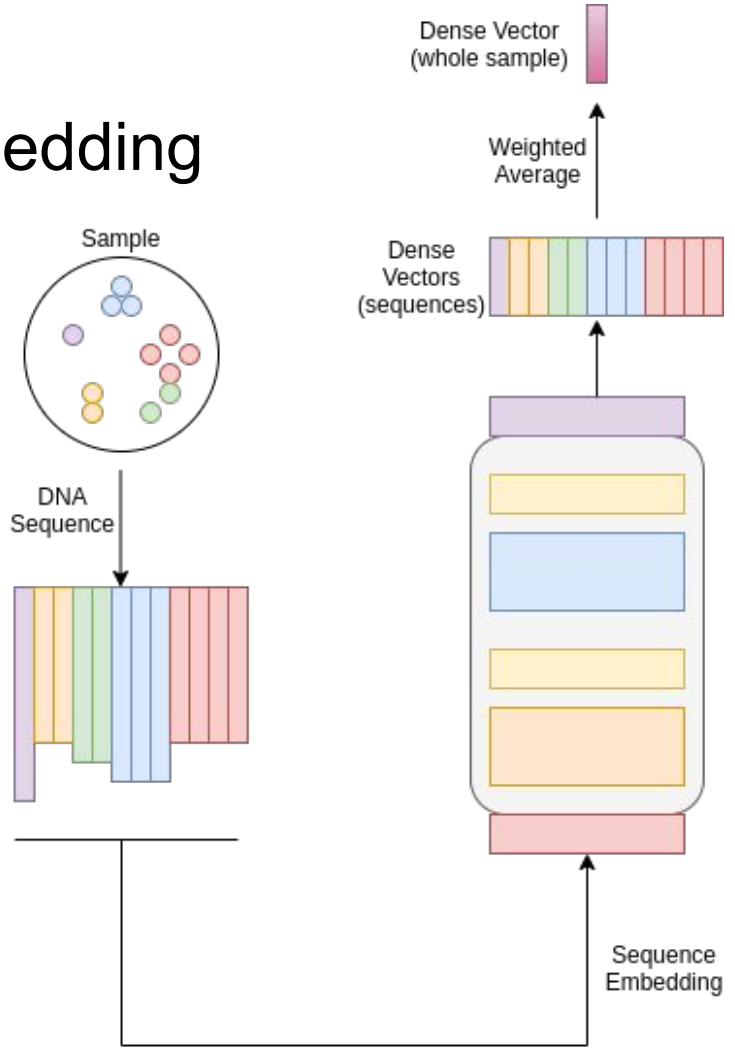


(b) Attention in head 7-1, which targets binding sites, a key functional component of proteins. Example is HIV-1 protease (7HVP). The primary location receiving attention is 27G, a binding site for protease inhibitor small-molecule drugs.

(My research!) Sequence embedding



(My research!) Sequence embedding



Extras

Additional Resources!

Here are some places you can go to learn more:

https://www.deeplearningbook.org/ : An excellent and fairly comprehensive resource describing design principles of many deep learning approaches

http://d2l.ai/ : Another learning resource for deep learning; more practically-minded than Deep Learning Book. Includes implementation details in Python

https://benjamin-lee.github.io/deep-rules/ 10 quick rules for deep learning for bioinformaticians (contributed to by our own Fin Maguire)

Backpropagation, detailed

How the weights of a multi layered network are computed

Introduces the concept of “Gradient descent”

The gradient is the transpose of the derivative of the output w.r.t the input. (For those well-versed in linear algebra, it is the Jacobian matrix of the weight vectors)

Intuitively, think of the gradient as the “direction” of the derivative of the loss function of the neural network

Recall from calculus: For some real valued function *f(x)*, the derivative *f’(x)* = 0 when *f(x)* is at a minima.

Gradient descent: Minimize the loss function by moving “down” the gradient

$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \ldots, \frac{\partial C}{\partial x_m}]$$

Gradient of a function C in vector **x**

1. **Input** x**:** Set the corresponding activation a^1 for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error** δ^L**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Subcellular localization

Simple enough: LSTM with a classification objective

State-of-the art classification accuracy

Representation learning example

“Inductive” ML: Learn about the data by investigating what the model used

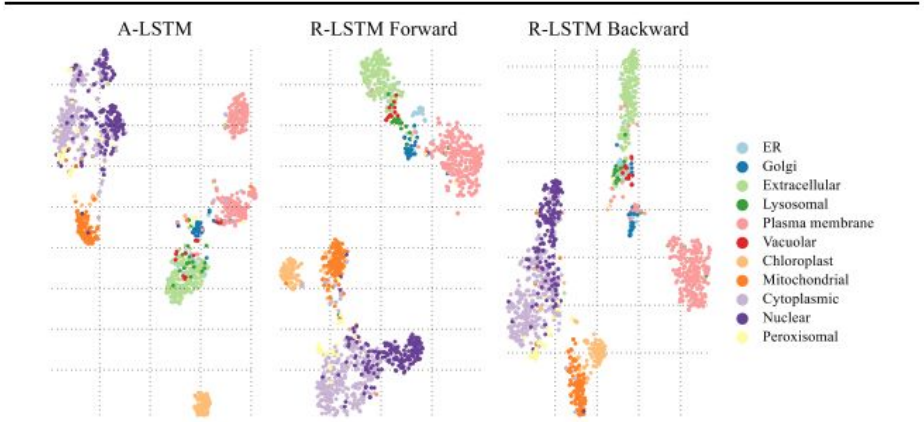
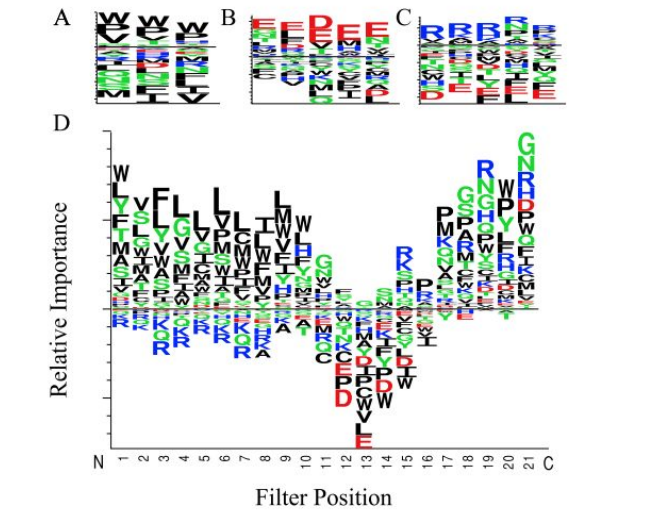


Figure 6. t-SNE plot of hidden representation for Forward and Backward R-LSTM and A-LSTM.



DNA modeling: Attention of nc-DNA motifs