DTU

Deep learning:

Feed-forward (continued) & NNAlign

Eric Bautista Farrerons and Pablo Vivero Fernandez

Immunoinformatics and Machine Learning (IML) group

Recap: Feed-forward Networks (FFNs)

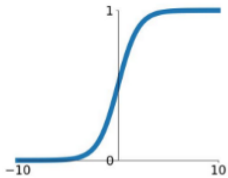
Keypoints:

- Weights connecting every node together
- **Activation functions** after layers

Activation Functions

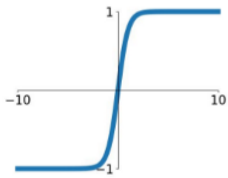
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



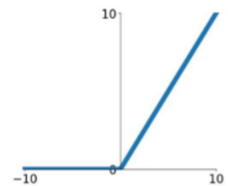
tanh

$\tanh(x)$



ReLU

$\max(0, x)$



Weights (+)

Weights (-)

Node / Unit

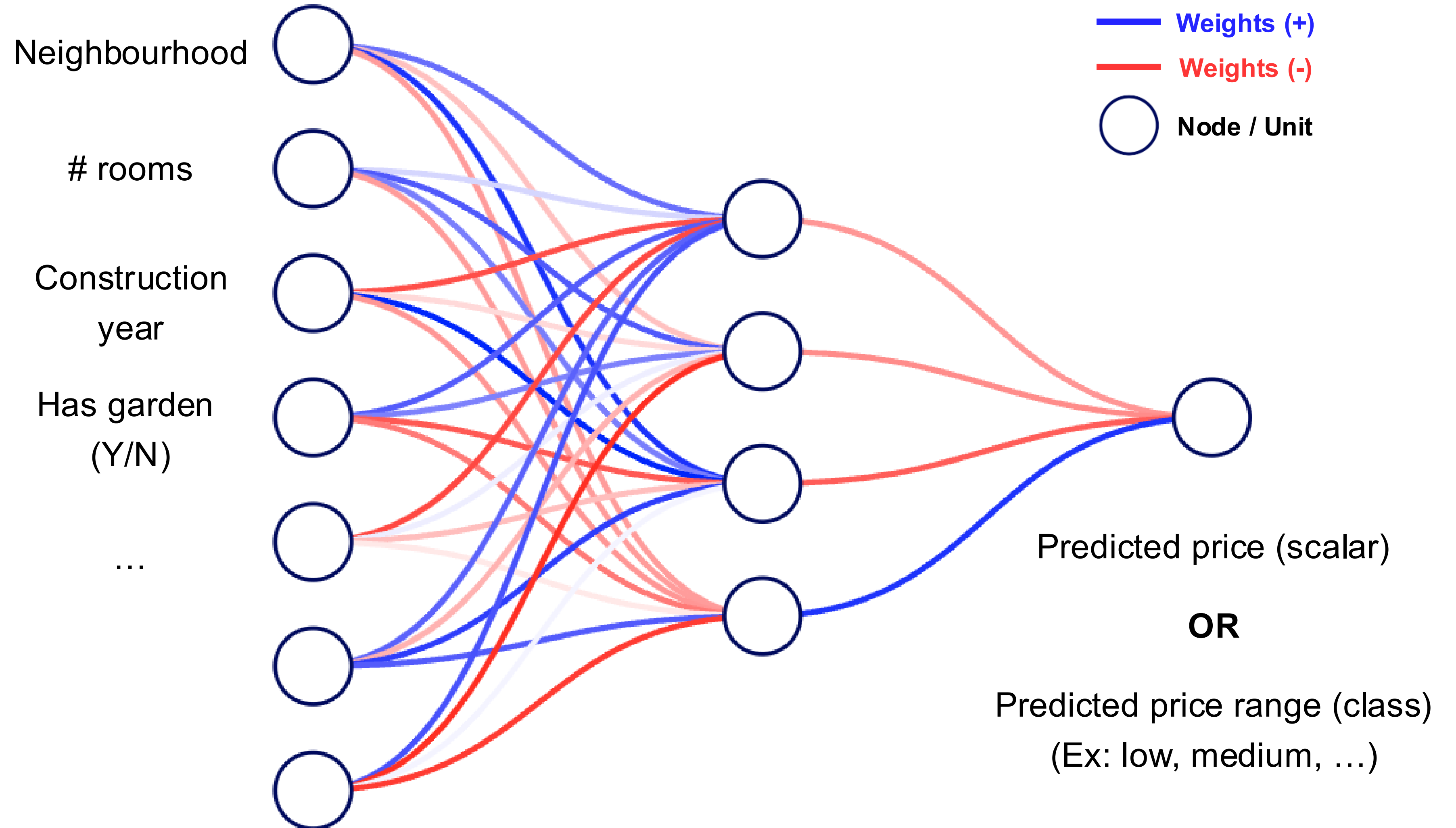
Input Layer $\in \mathbb{R}^7$ Hidden Layer $\in \mathbb{R}^4$ Output Layer $\in \mathbb{R}^1$

Recap: Feed-forward Networks (FFNs)

Use features to :
- **predict (scalar values)**
- **classify (labels)**

Ex:

Predict the price of a house based on features



Accelerating your models

- Yesterday we implemented a neural network using Python *for* loops.
- It is good for gaining an intuition for neural networks.
- But it is really not very efficient...

Accelerating your models

- Python *for* loops are extremely slow.
- For larger models we need to use more efficient implementations.
- Deep learning libraries with C++ backends provide this. For example:
  - PyTorch
  - TensorFlow
  - JAX + NumPy

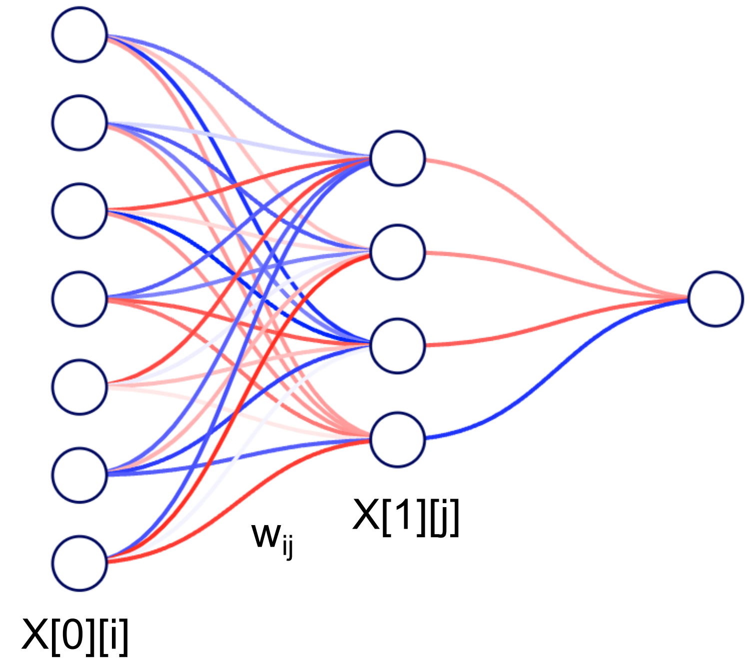
Accelerating your models

- Python *for* loops are extremely slow.
- For larger models we need to use more efficient implementations.
- Deep learning libraries with C++ backends provide this. For example:
    - PyTorch
    - TensorFlow
    - JAX + NumPy
- Today's goal: make few but large matrix operations so model training and data processing is more efficient.
- How?
    1. We will use **NumPy** so you understand the impact of matrix multiplications.
    2. And later the vectorization and automatic differentiation (Autograd) capabilties of **PyTorch**.

Using vectorized matrix operations to speed up processing

- Last time:

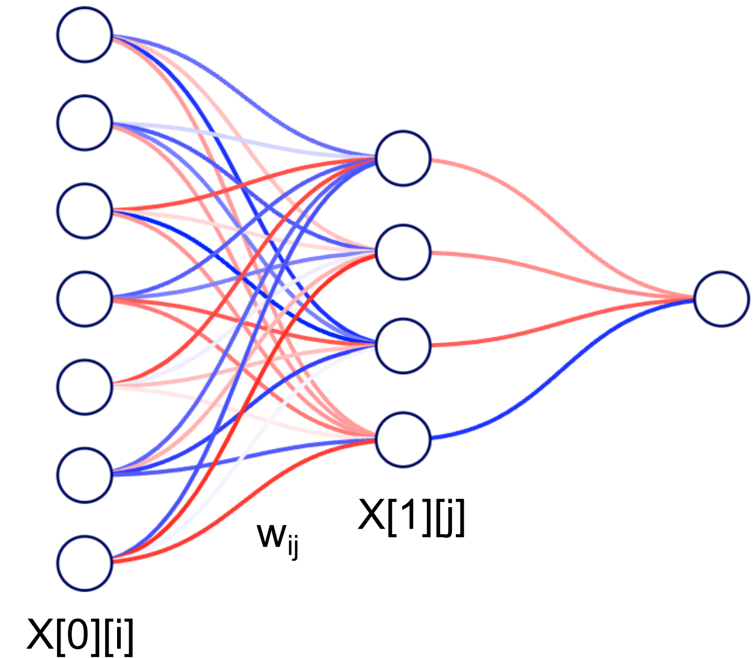
```
def forward(X):
    for j in range(hidden_layer_dim):
        z = 0.0
        for i in range(input_layer_dim+1):
            z += X[0][i]* w1[i,j]
        X[1][j] = sigmoid(z)
```



Using vectorized matrix operations to speed up processing

- Last time:

```
def forward(X):
        for j in range(hidden_layer_dim):
                z = 0.0
                for i in range(input_layer_dim+1):
                        z += X[0][i]* w1[i,j]
                X[1][j] = sigmoid(z)
```



Using vectorized matrix operations to speed up processing

- We can process an input in terms of matrix operations.

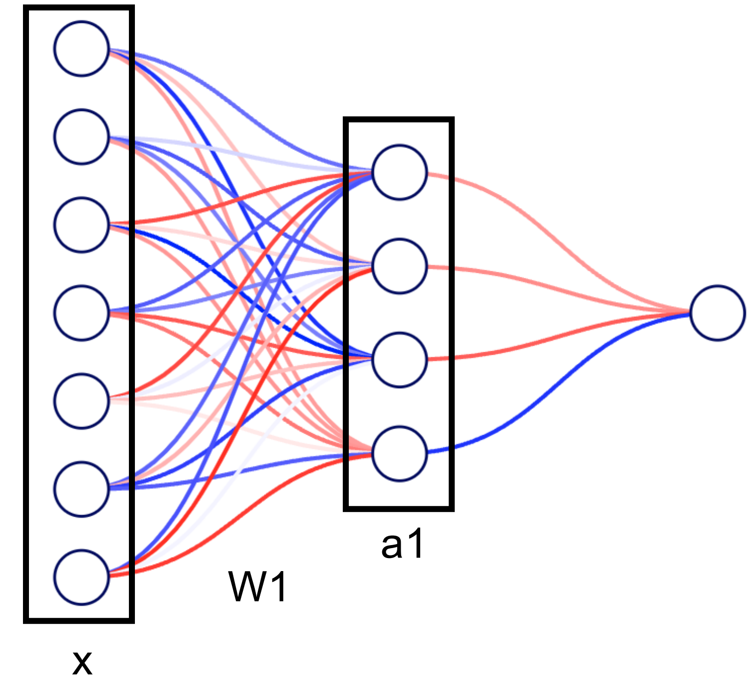
- Today:

NumPy

```
def forward(x):
# First layer
z1 = np.dot(x, W1)
a1 = activation(z1)
```

PyTorch

```
def forward(self, x):
# First layer
z1 = self.in_layer(x)
a1 = self.hidden_act(z1)
```



$$x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{bmatrix} \times W_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \\ w_{61} & w_{62} & w_{63} & w_{64} \\ w_{71} & w_{72} & w_{73} & w_{74} \end{bmatrix} = z_1 = \begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} \end{bmatrix} \Rightarrow a_1 = g(z_1)$$

(1 x 7) (7 x 4) (1 x 4) (1 x 4)

Batching data to speed up processing

- Last time:

```
predictions = []
for peptide_sequence in dataset:
  y_pred = forward(peptide_sequence)
  predictions.append(y_pred)
```

Batching data to speed up processing

- Last time:

```
predictions = []
for peptide_sequence in dataset:
  y_pred = forward(peptide_sequence)
  predictions.append(y_pred)
```

Batching data to speed up processing

- We can process batches of data at the same time.

- Today:

```
predictions = net(dataset)
```

Encoding of input data (peptides)

ALAKAAAAM
ALAKAAAAN
ALAKAAAAR
ALAKAAAAT
ALAKAAAAV
GMNERPILT
GILGFVFTM
TLNAWVKVV
KLNEPVLLL
AVVPFIVSV

0.9 0.05 0.05 …
0.9 0.05 0.05 ..

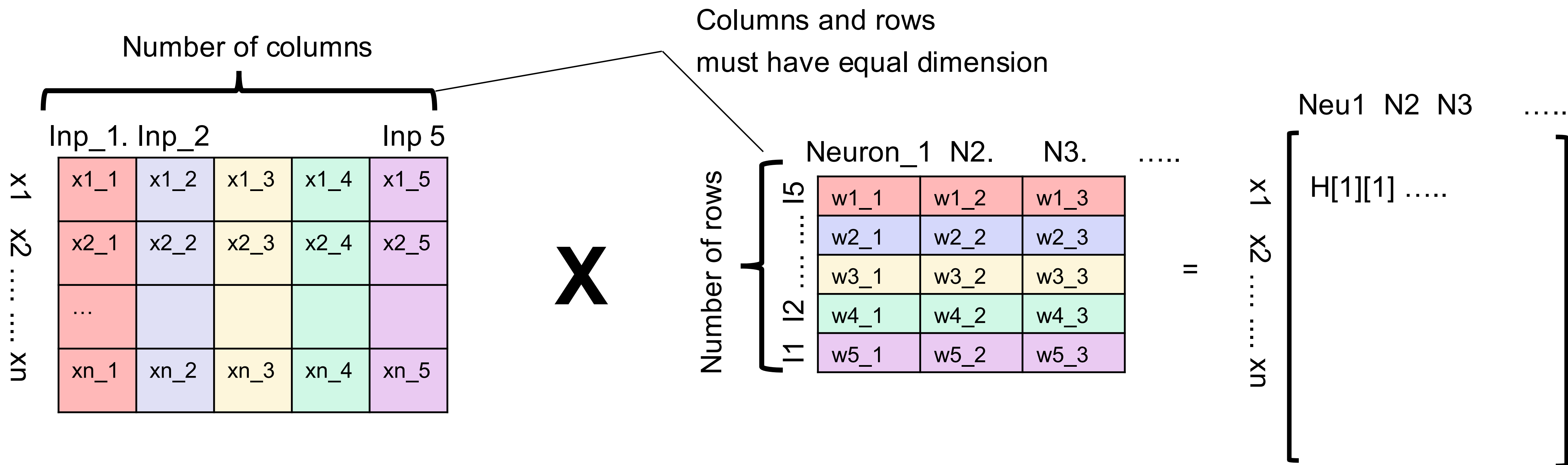
N_input

x1_1	x1_2	x1_3	x1_4	x1_5
x2_1	x2_2	x2_3	x2_4	x2_5
…				
xn_1	xn_2	xn_3	xn_4	xn_5

N_datasize/batch

Batching data to speed up processing

- We can process batches of data at the same time.
- We can apply the weights of all hidden neurons to the input in fewer operations.



Key takeaways

- In a **for loop**, each operation is executed **sequentially**, meaning one must finish before the next begins — this is **slow**
- When you **vectorize**, the operations involved in multiplying two matrices can be executed **in parallel.** This is especially effective on GPUs.
- Libraries like **NumPy** and **PyTorch** use highly optimized **C/C++** under the hood to perform these operations **much faster** than pure Python.

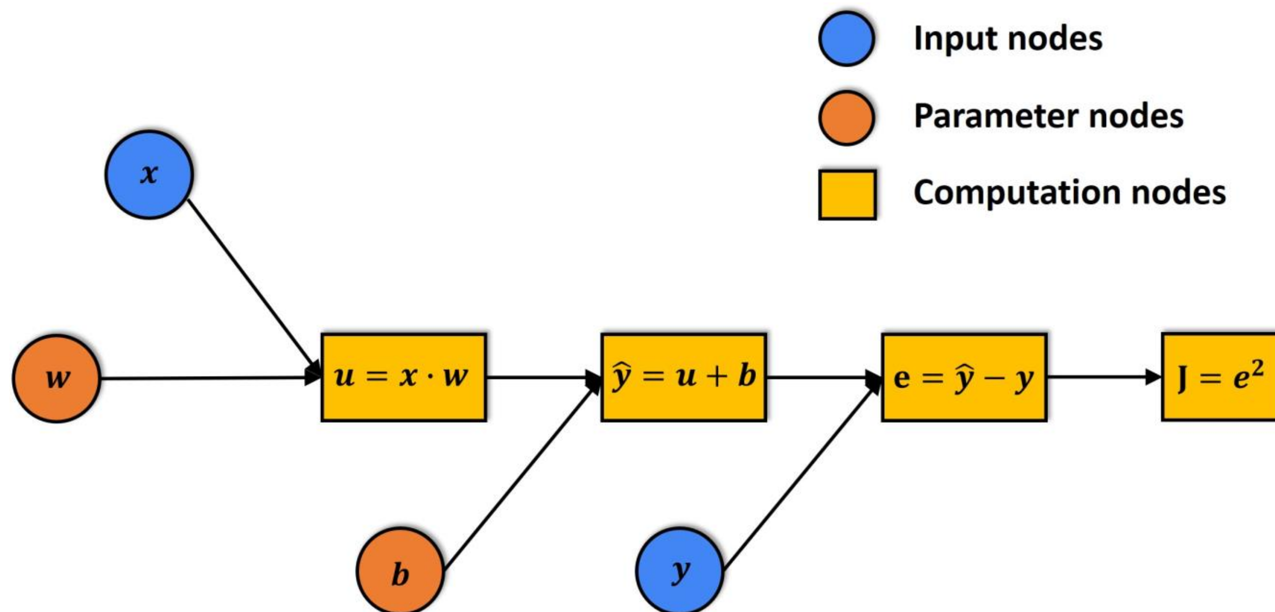
A B

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 3 & 4 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 5 & 1 & 1 & 1 \\ 6 & 7 & 1 & 1 & 1 \\ 1 & 8 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Multiply

Nice online matrix multiplication visualization tool:
https://matrixmultiplication.xyz/

Automatic differentiation - Autograd



In PyTorch

```
optimizer.zero_grad()
y_pred = net(x)
loss = criterion(y_pred, y)
loss.backward()
optimizer.step()
```

In practice, the backward pass is usually called on the loss: `loss.backward()`.

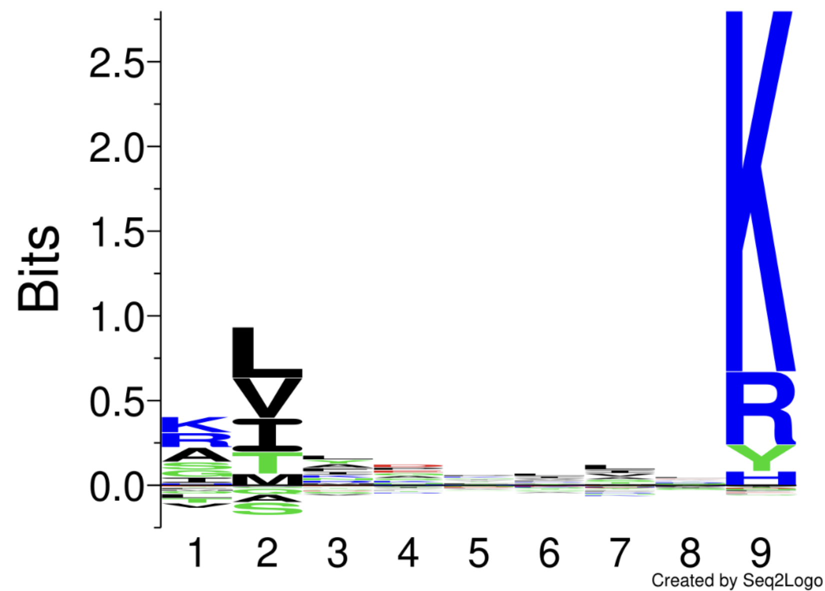
What about peptides which are not 9-mers?

- If all our peptides are 9-mers, FFNN works fine
- However, what about a 12-21mers?
- Need to pad the sequences to the maximum peptide length

Original	Padded
CAHHFWTK	CAHHFWTKXX
TINYTIFK	TINYTIFKXX
WMNSTGFTK	WMNSTGFTKX
YIFWIRTPR	YIFWIRTPRX
TTTIKPVSYK	TTTIKPVSYK
WLWGFLSRNK	WLWGFLSRNK

What about peptides which are not 9-mers?

- However, this padding creates a problem: The anchor residues are no longer aligned!



CAHHFWTKXX

TINYTIFKXX

WMNSTGFTKX

YIFWIRTPRX

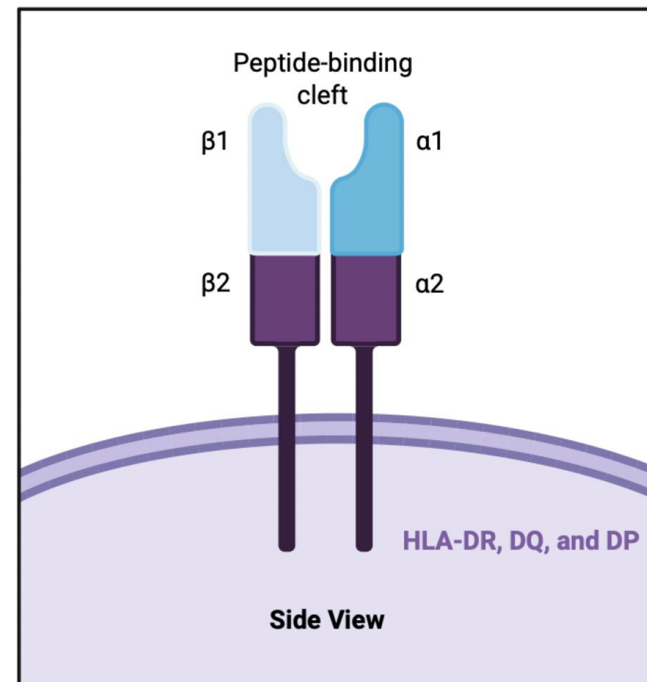
TTTIKPVSYK

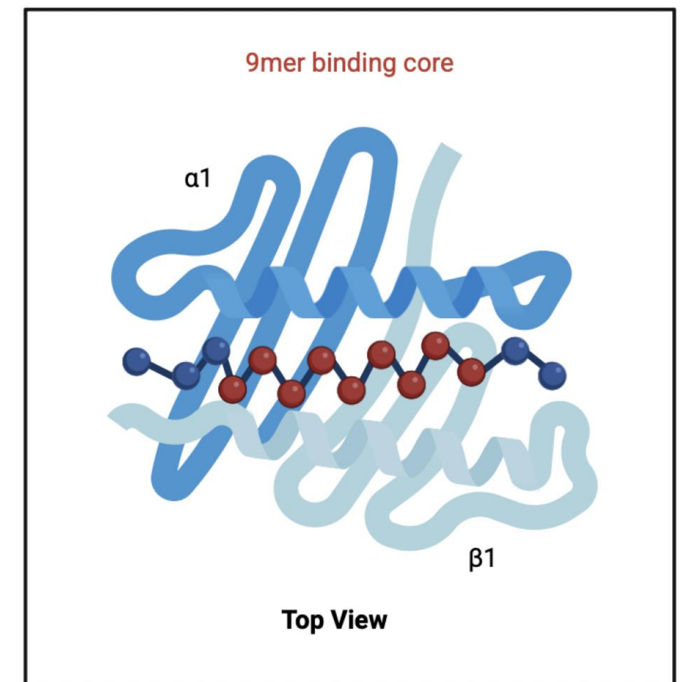
WLWGFLSRNK

What about peptides which are not 9-mers?

- MHC class II peptides are usually longer than 9 amino acids
- Binding is mainly driven by a 9-mer core within the peptide
- The 9-mer core can occur at different positions in the peptide

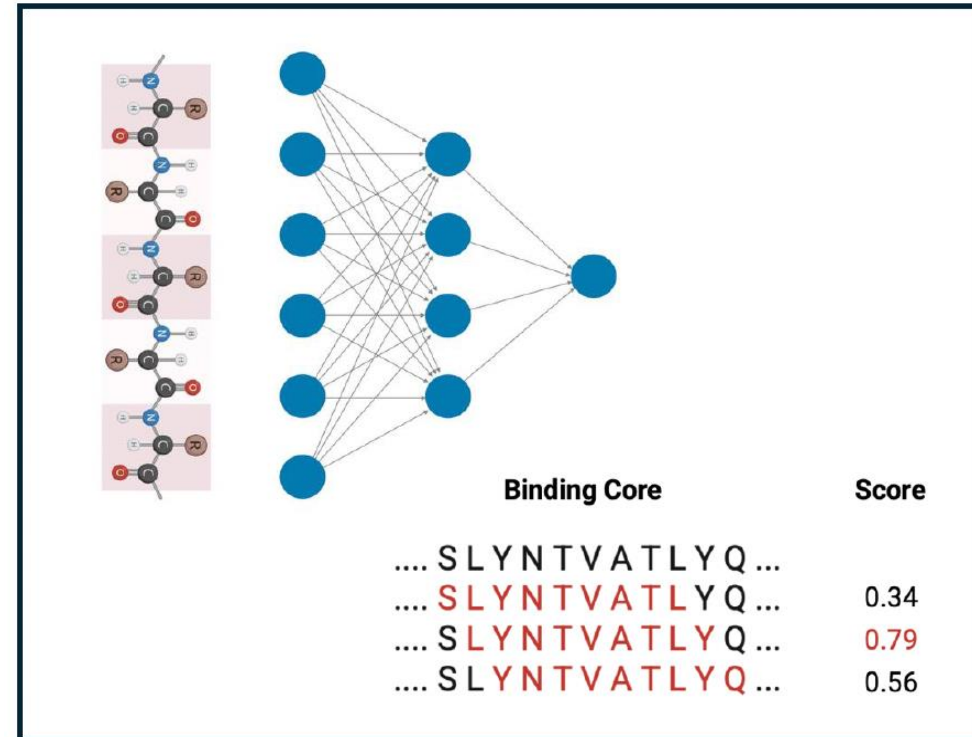
MHC Class II





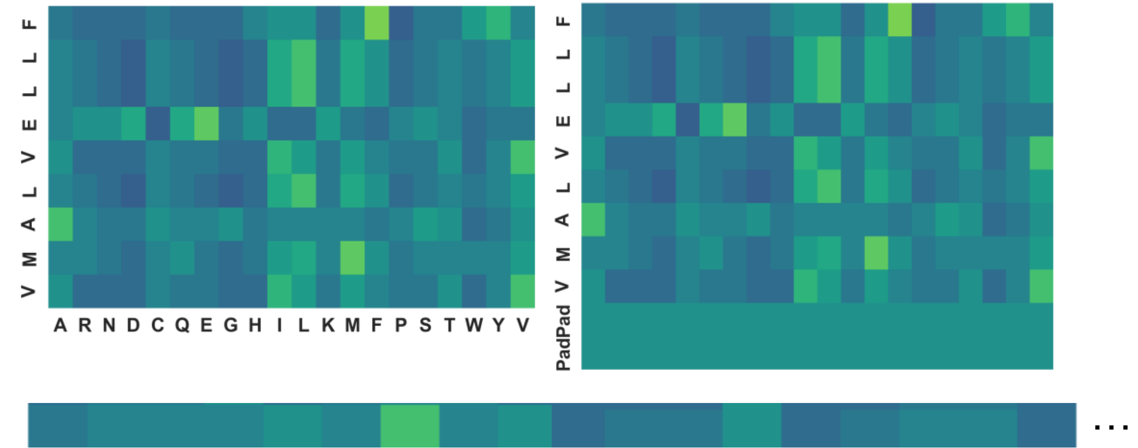
What about peptides which are not 9-mers?

- NNAlign evaluates all possible 9-mer binding cores within each MHC class II peptide during the forward pass
- The peptide score is defined as the highest score among these candidate cores, and backpropagation is performed through the best-scoring 9-mer core.

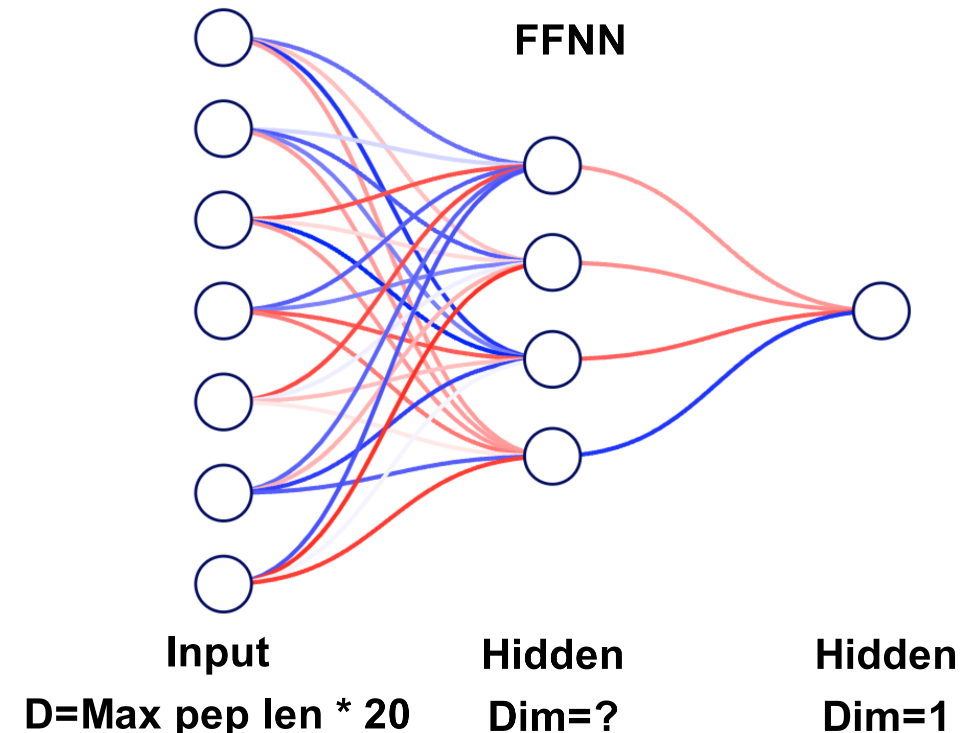


DTU

Part I - FFNN exercise

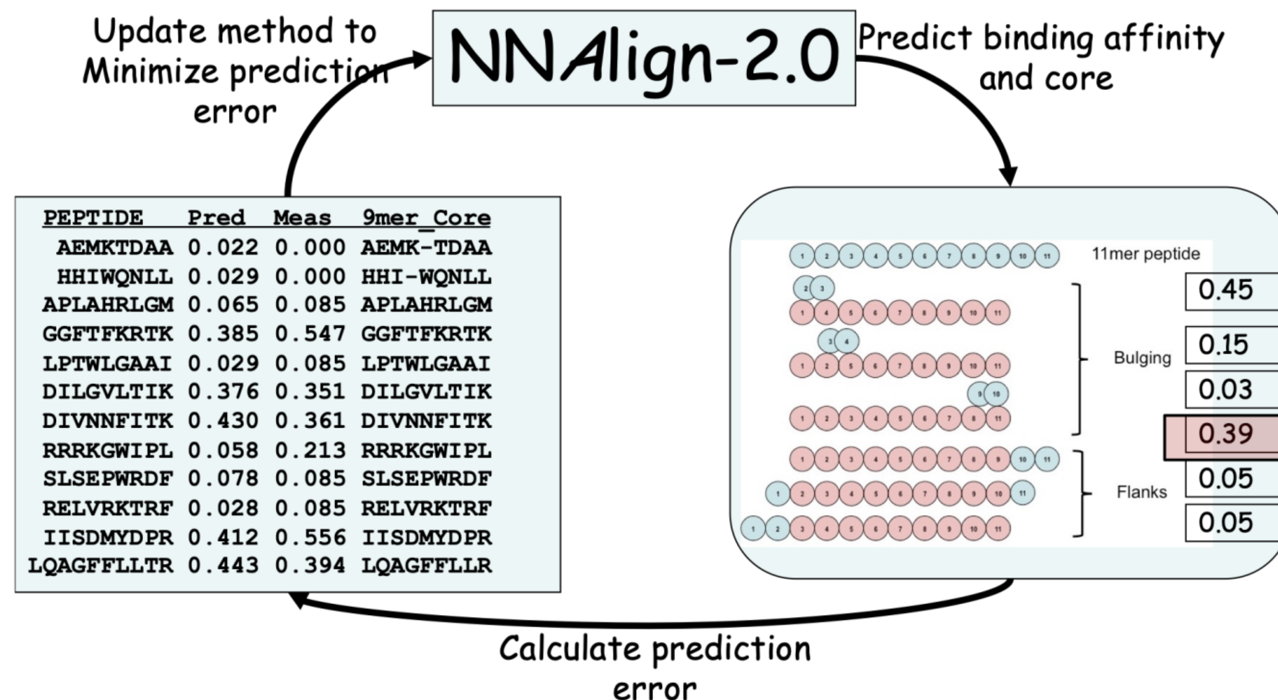


- Implement an FFNN using efficient matrix multiplication operations.
- Peptides may have different lengths. Shorter sequences must be padded. Why?
- Use a flattened array as input.
- Try to change the learning rate or number of neurons and see if it impacts performance.
- Compare your performance and computation speed with last times NN exercise.



Today's exercises: Part 2

- **Implement the forward pass of the NNAlign method**
- **Compare the performance of the NNAlign method vs standard Feed-forward neural network. Is NNAlign performing better?**
- **Aside from potential performance gains, what other advantages are there of the NNAlign method's predictions? (hint: interpretibility)**



PEPTIDE	Pred	Meas	9mer_Core
AEMKTDAA	0.022	0.000	AEMK-TDAA
HHIWQNLL	0.029	0.000	HHI-WQNLL
APLAHRLGM	0.065	0.085	APLAHRLGM
GGFTFKRTK	0.385	0.547	GGFTFKRTK
LPTWLGAAI	0.029	0.085	LPTWLGAAI
DILGVLTIK	0.376	0.351	DILGVLTIK
DIVNNFITK	0.430	0.361	DIVNNFITK
RRRKGWIPL	0.058	0.213	RRRKGWIPL
SLSEPWRDF	0.078	0.085	SLSEPWRDF
RELVRKTRF	0.028	0.085	RELVRKTRF
IISDMYDPR	0.412	0.556	IISDMYDPR
LQAGFFLLTR	0.443	0.394	LQAGFFLLR

We are here to take your questions.

(when in doubt: Print output of a given operation and their dimensions of a tensor x using x.shape or x.size)