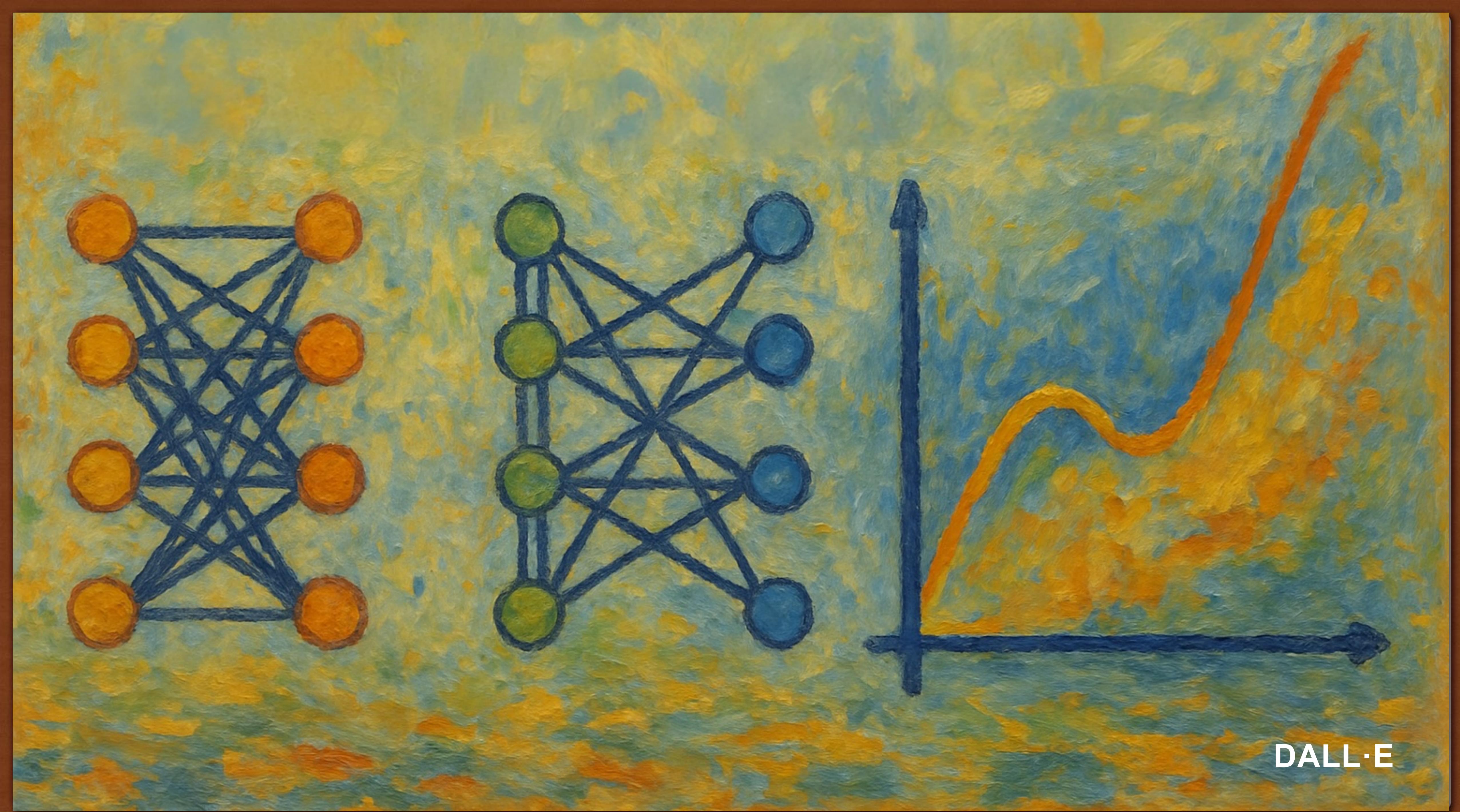


COM 3240

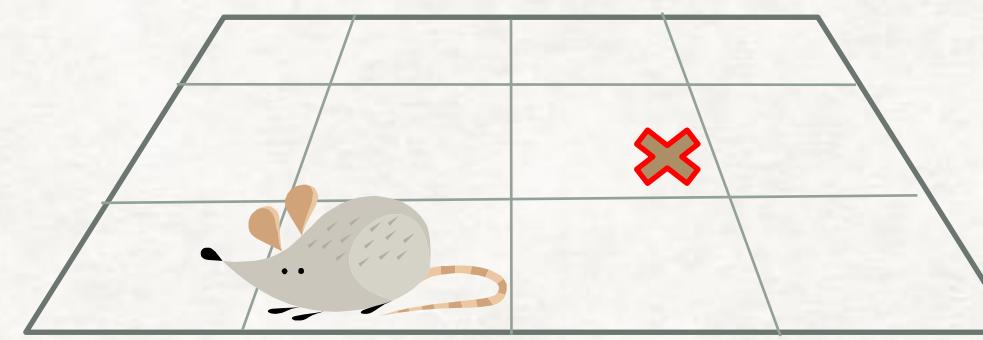
# REINFORCEMENT LEARNING



DALL·E

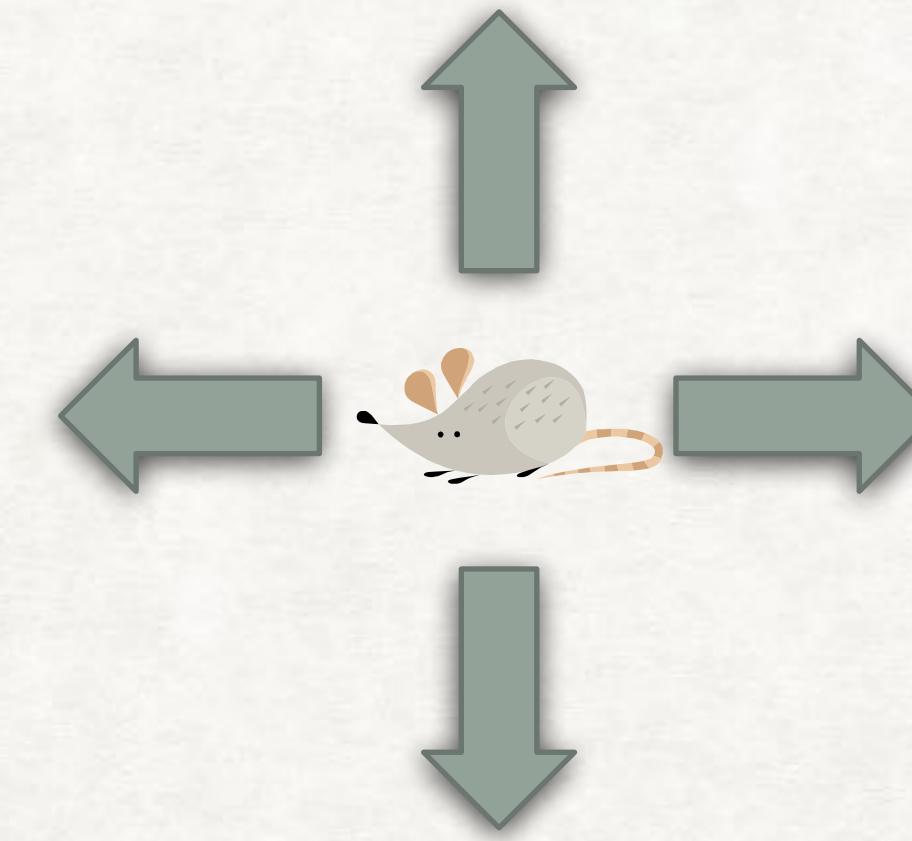
# FUTURE REWARDS

## Q-VALUES



Squares = State (in this case)

Q-values stored in a table

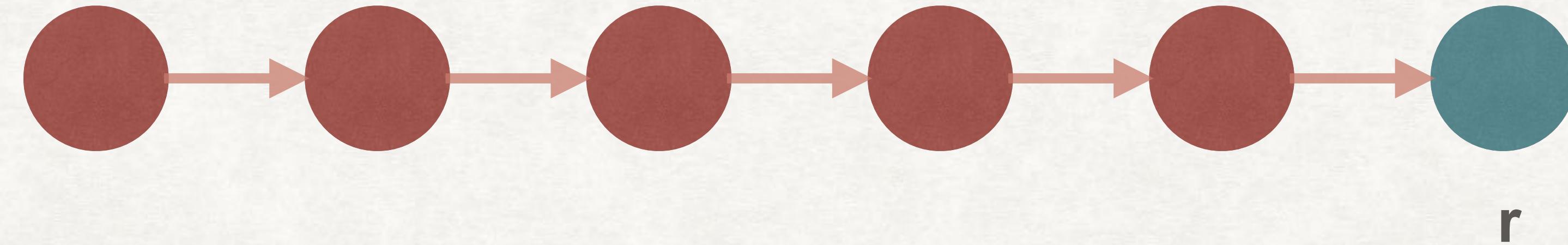


Actions

# FUTURE REWARDS

## SARSA

$$\Delta Q = \eta (r + \gamma Q(s', a') - Q(s, a))$$



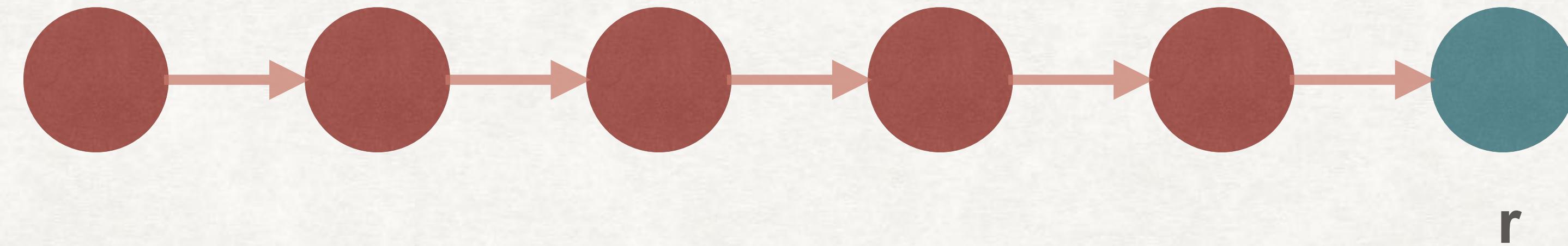
A robot walks this corridor and finds reward.

How many Q-values will be updated each time?

# FUTURE REWARDS

## SARSA

$$\Delta Q = \eta (r + \gamma Q(s', a') - Q(s, a))$$

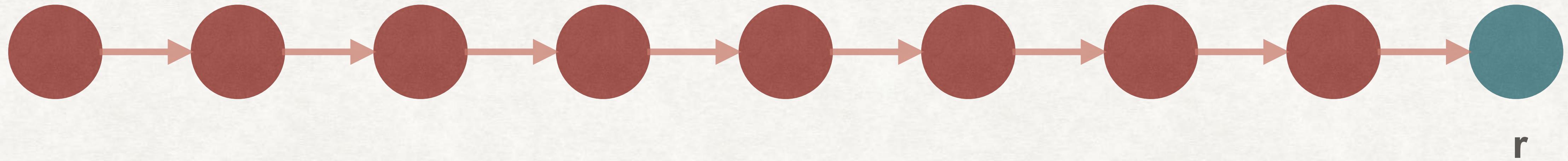


How large has to be the table storing the Q values?

# FUTURE REWARDS

## SARSA

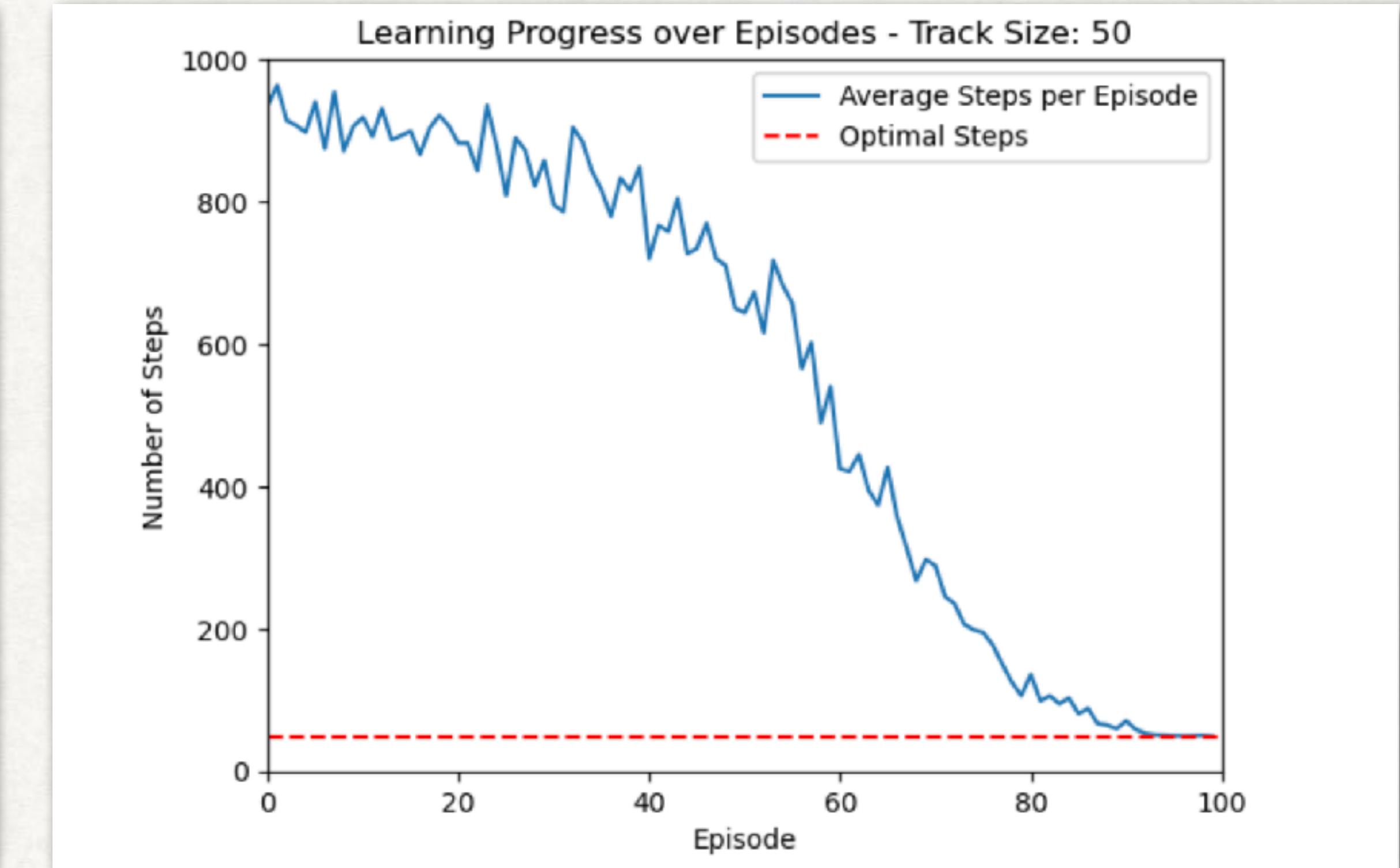
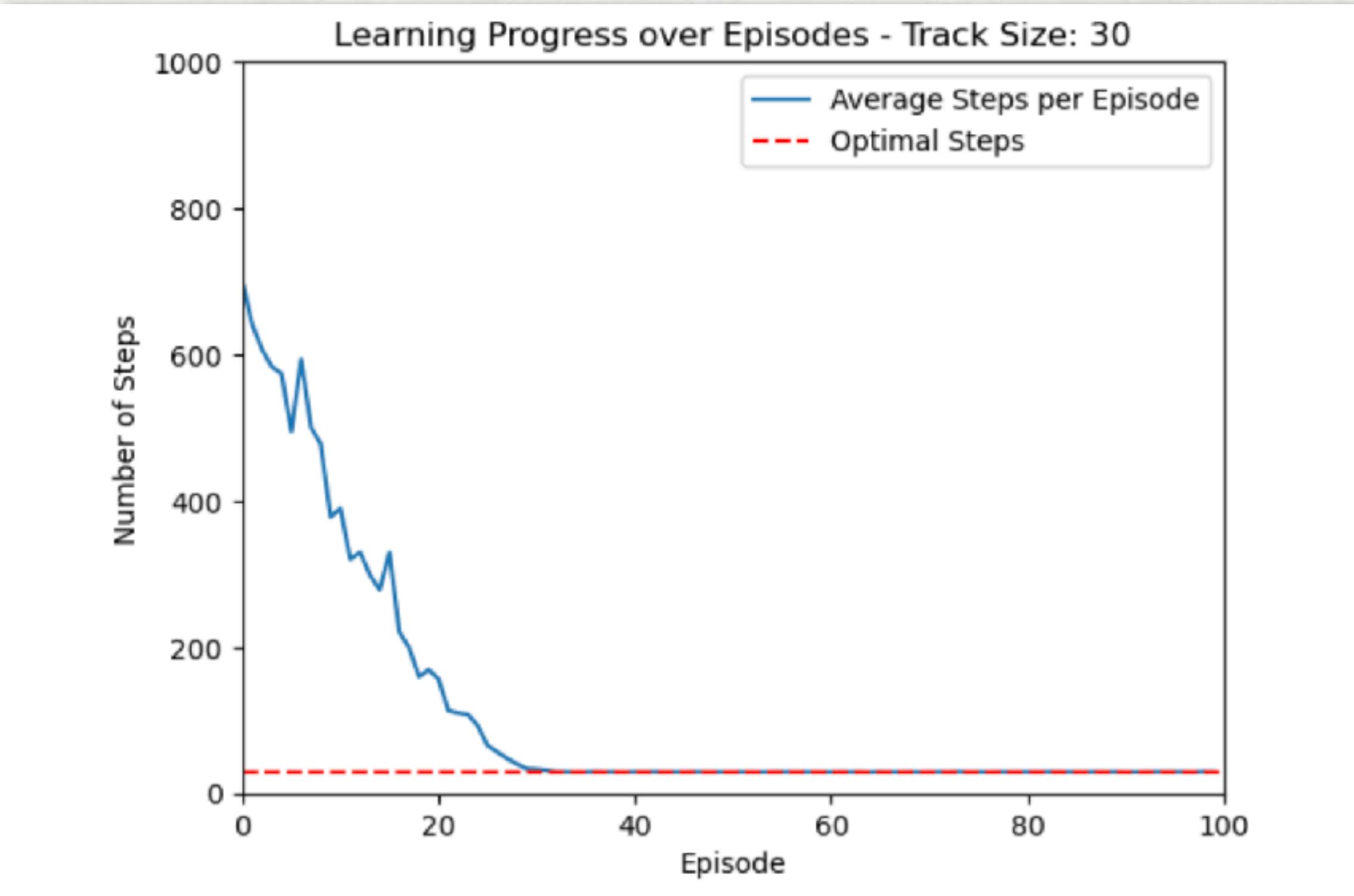
$$\Delta Q = \eta (r + \gamma Q(s', a') - Q(s, a))$$



We increase the number of states. How many Q-values we need to store?

# FUTURE REWARDS

## SARSA



# FUTURE REWARDS

## SARSA



Backward   Forward

[0.000, 0.002, ]  
[0.000, 0.012, ]  
[0.000, 0.069, ]  
[0.000, 0.261, ]  
[0.000, 0.651, ]  
[0.000, 0.000, ]

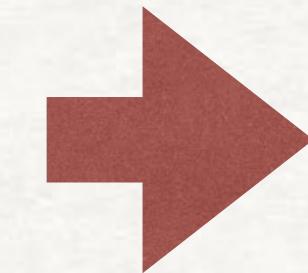
Has the algorithm converged?

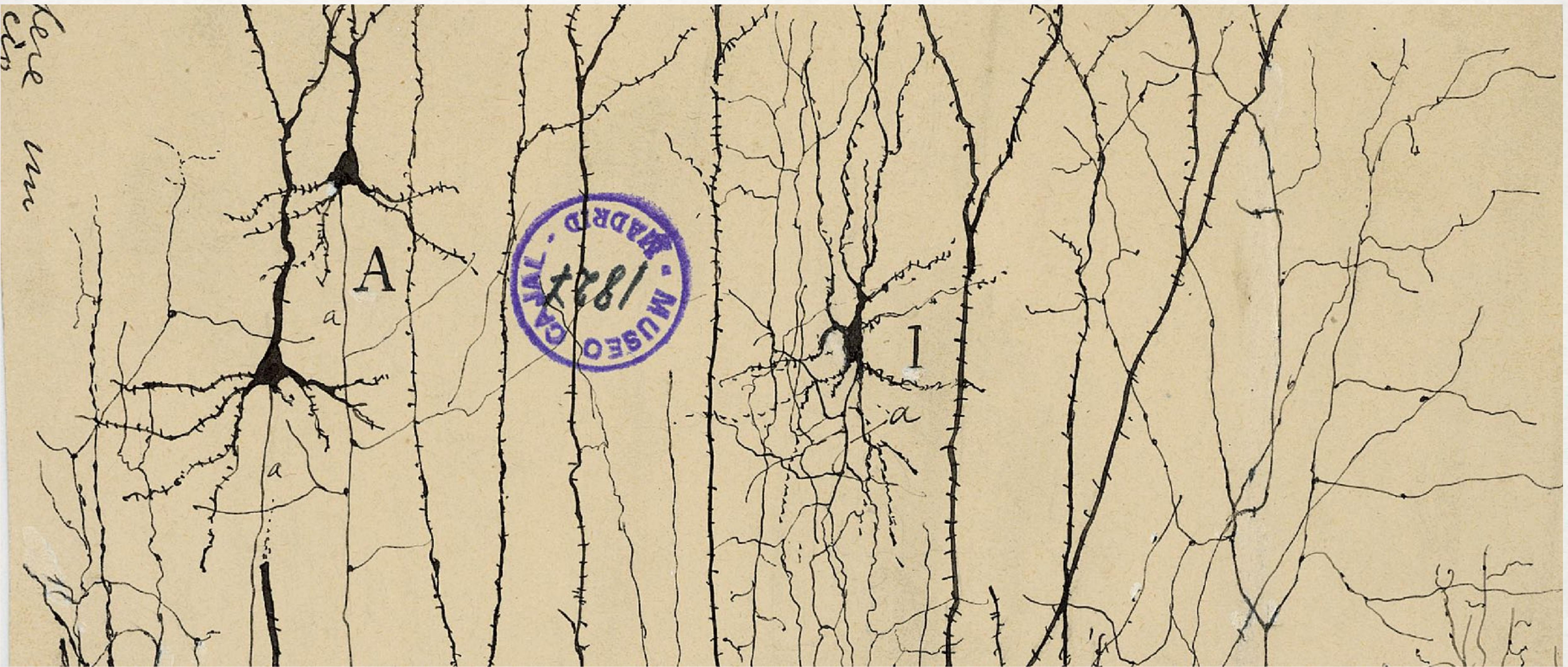


# LIMITATION OF TABULAR IMPLEMENTATIONS

## LEARNING THE VALUE OF ONE STATE DOES NOT INFORM OTHER STATES

```
[0.000, 0.002, ]  
[0.000, 0.012, ]  
[0.000, 0.069, ]  
[0.000, 0.261, ]  
[0.000, 0.651, ]  
[0.000, 0.000, ]
```





Ramon y Cahal

1mm<sup>3</sup>: 10,000 neurons 3km wires

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## SINGLE NEURON

$$\mathbf{y}^* = F^*(\mathbf{x})$$

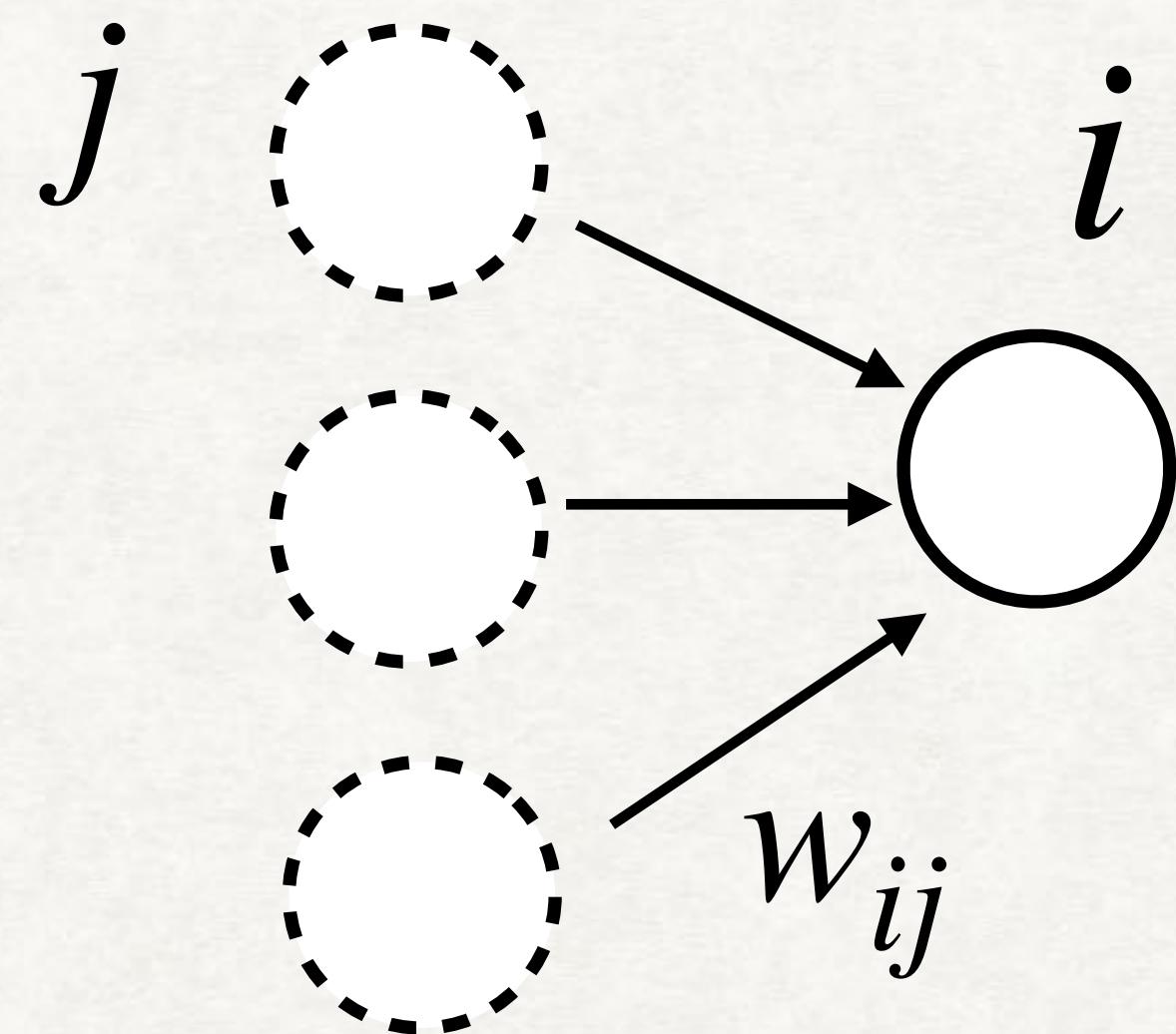
$$\mathbf{y} = F(\mathbf{x})$$

$$E = \frac{1}{2n} \sum_i \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x}; \mathbf{W}))^2$$

Samples from the input and output space of the function.

$$\mathbf{x}^{(1)} \ \mathbf{x}^{(2)} \ \mathbf{x}^{(3)} \dots \mathbf{x}^{(n)}$$

$$\mathbf{y}^{(1)} \ \mathbf{y}^{(2)} \ \mathbf{y}^{(3)} \dots \mathbf{y}^{(n)}$$



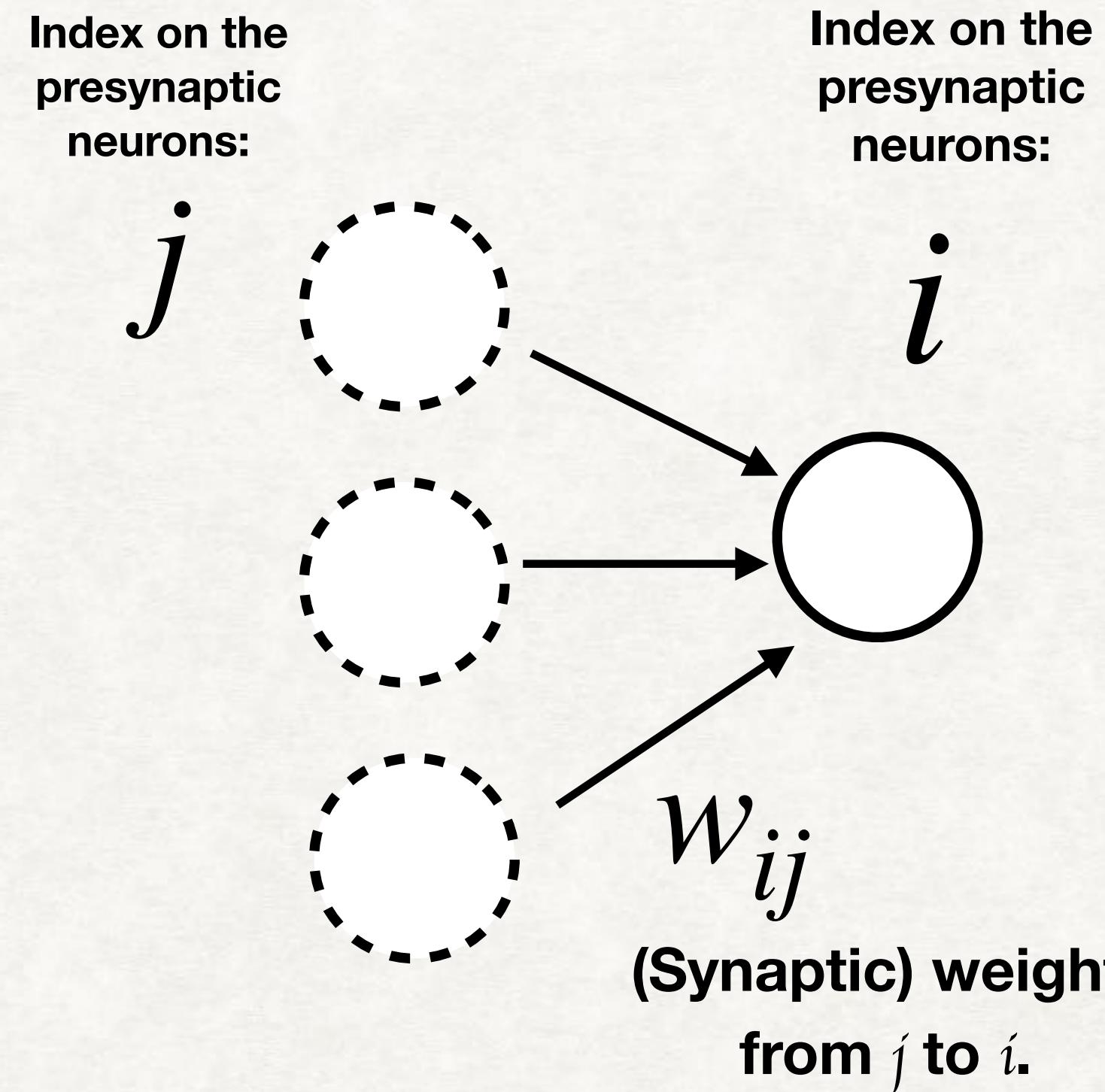
Find parameters (weights) that minimise the Loss function.

$$\Delta \mathbf{W}_{lk} \propto -\frac{\partial E}{\partial \mathbf{W}_{lk}}$$

$$\Delta \mathbf{W} \propto -\nabla_{\mathbf{W}} E$$

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## SINGLE NEURON



$$y = ax + b$$

$$h_i = \sum_j w_{ij}x_j + b_i$$

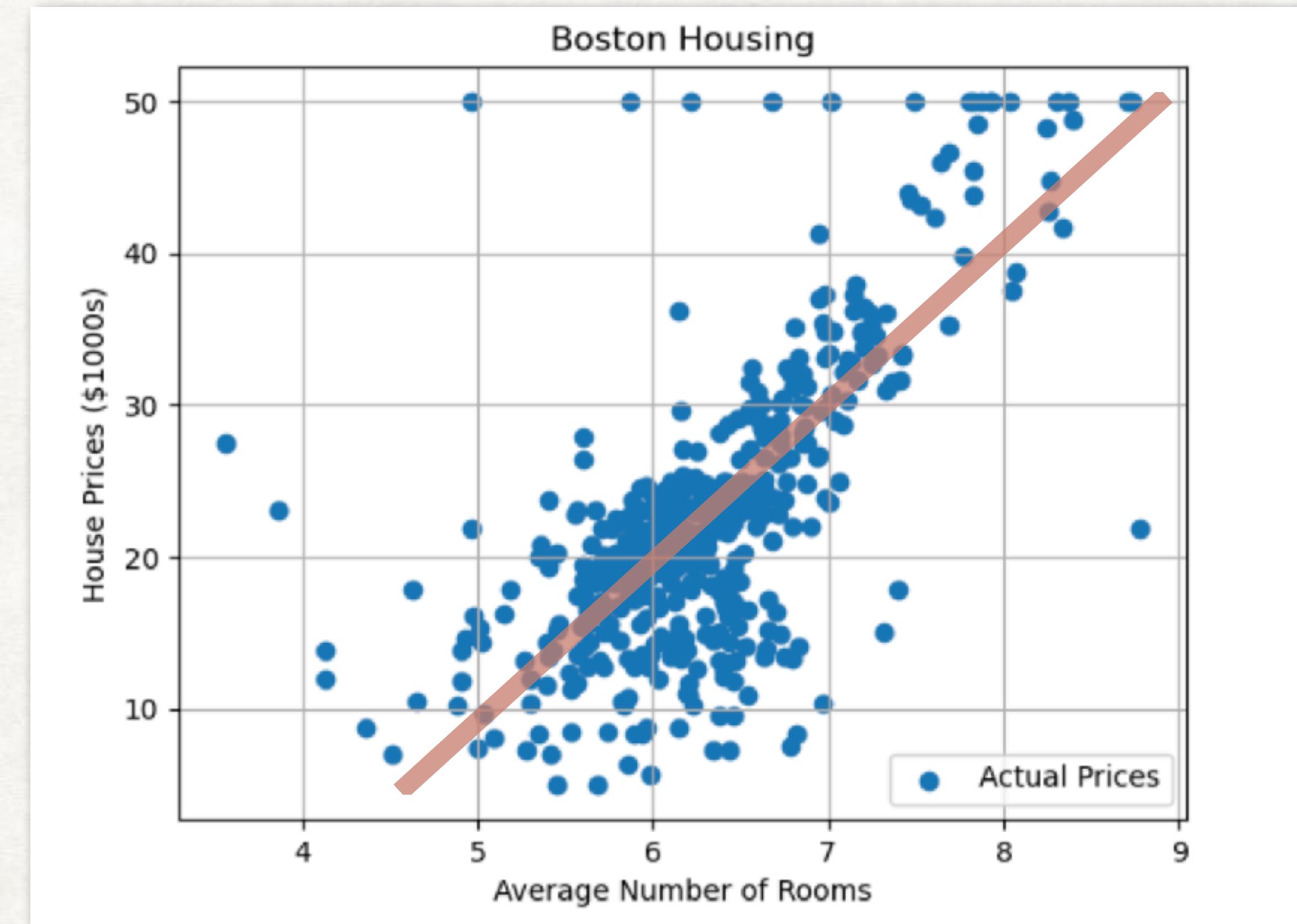
$$x_i = f(h_i)$$

Bias

Activation function  $f$ , sigmoidal or ReLU

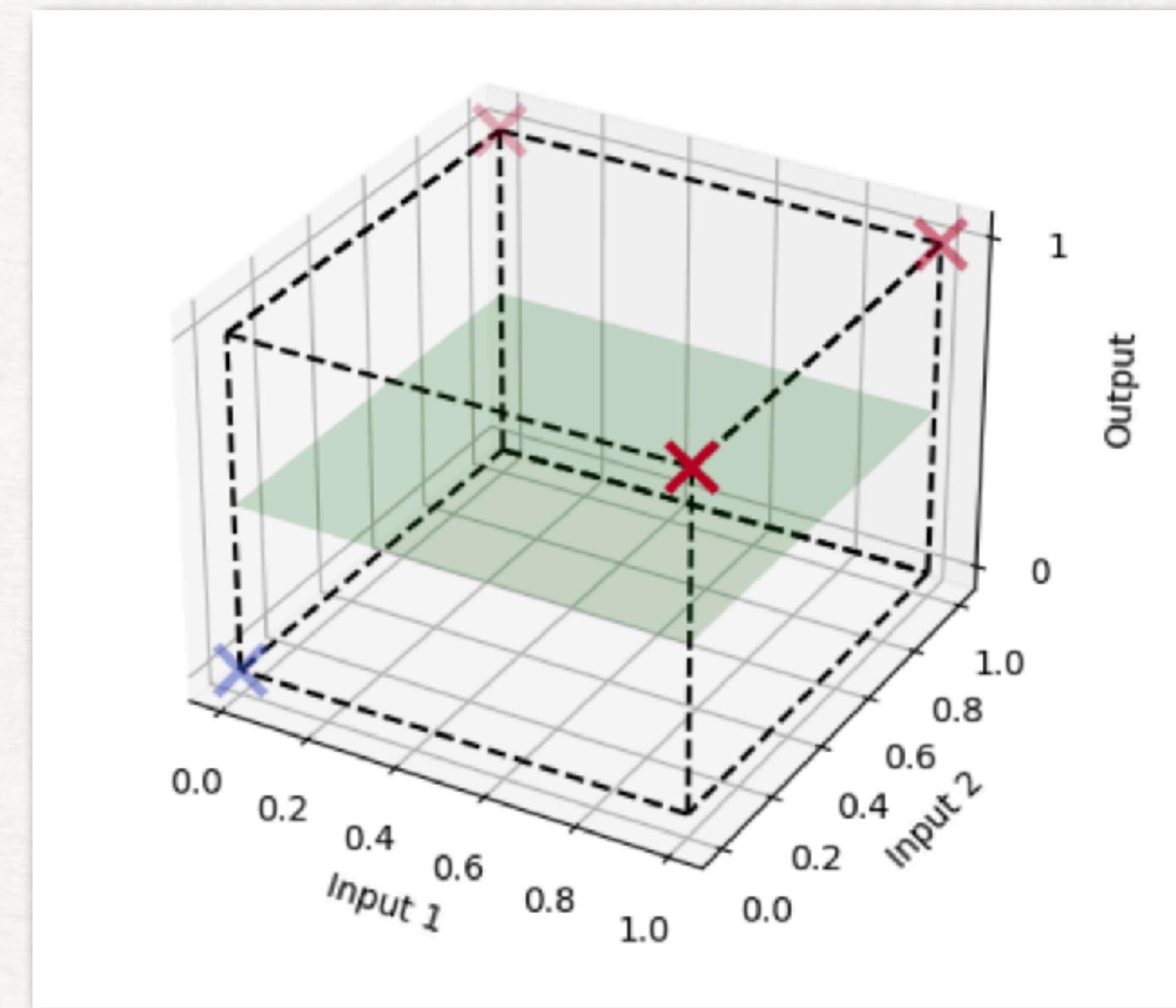
# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## LINEAR MODEL



# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## LINEAR SEPARABILITY



# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## GRADIENT DESCENT

$$L = \frac{1}{2n} \sum_i \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x}; \mathbf{W}))^2$$

$$\Delta w_{ij} = -\eta \frac{\partial L}{\partial w_{ij}}$$

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## GRADIENT DESCENT

$$L = \frac{1}{2n} \sum_i \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x}; \mathbf{W}))^2$$

$$\frac{\partial L}{\partial w_{ij}} = -\frac{1}{n} \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) \frac{\partial y_i(\mathbf{x})}{\partial w_{ij}}$$

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## GRADIENT DESCENT

$$L = \frac{1}{2n} \sum_i \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x}; \mathbf{W}))^2$$

$$\frac{\partial L}{\partial w_{ij}} = -\frac{1}{n} \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) \frac{\partial y_i(\mathbf{x})}{\partial w_{ij}}$$

$$y_i(\mathbf{x}) = f \left( \sum_j w_{ij} x_j + b_i \right)$$

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## GRADIENT DESCENT

$$L = \frac{1}{2n} \sum_i \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x}; \mathbf{W}))^2$$

$$\frac{\partial L}{\partial w_{ij}} = -\frac{1}{n} \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) \frac{\partial y_i(\mathbf{x})}{\partial w_{ij}}$$

$$\Delta w_{ij} = -\eta \frac{1}{n} \sum_{\mathbf{x}} (y_i(\mathbf{x}) - y_i^t(\mathbf{x})) y_i(\mathbf{x})(1 - y_i(\mathbf{x})) x_j$$

**f: sigmoid**

$$\Delta b_i = -\eta \frac{1}{n} \sum_{\mathbf{x}} (y_i(\mathbf{x}) - y_i^t(\mathbf{x})) y_i(\mathbf{x})(1 - y_i(\mathbf{x}))$$

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## GRADIENT DESCENT

$$L = \frac{1}{2n} \sum_i \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x}; \mathbf{W}))^2$$

$$\frac{\partial L}{\partial w_{ij}} = -\frac{1}{n} \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) \frac{\partial y_i(\mathbf{x})}{\partial w_{ij}}$$

$$\Delta w_{ij} = \eta \frac{1}{n} \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) \cdot \mathcal{H}(\sum_j w_{ij}x_j + b_i) \cdot x_j$$

**f: ReLU**

$$\Delta b_i = \eta \frac{1}{n} \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) \cdot \mathcal{H}(\sum_j w_{ij}x_j + b_i)$$

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## EFFICIENT IMPLEMENTATION

$$y_i(\mathbf{x}) = f \left( \sum_j w_{ij}x_j + b_i \right)$$

```
x = ... # Input data matrix, shape (batch_size, input_dim)
y_t = ... # Target values matrix, shape (batch_size, output_dim)
y = ... # Computed outputs matrix, shape (batch_size, output_dim)
W = ... # Weight matrix, shape (input_dim, output_dim)
b = ... # Bias vector, shape (output_dim,)
learning_rate = ... # Learning rate, a scalar
```

Always check DIMENSIONS

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## EFFICIENT IMPLEMENTATION

$$y_i(\mathbf{x}) = f \left( \sum_j w_{ij}x_j + b_i \right)$$

### Code Snippet:

```
# Compute the linear transformation using matrix multiplication
linear_output = X @ W + b

# X(batch_size * input_dim), W (input_dim * output_dim), b (output_dim,)
# Resulting matrix:(batch_size * output_dim)
```

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## EFFICIENT IMPLEMENTATION

$$y_i(\mathbf{x}) = f \left( \sum_j w_{ij}x_j + b_i \right)$$

### Code Snippet:

```
# Compute the linear transformation using matrix multiplication
linear_output = X @ W + b

# Apply activation function element-wise
y_i = activation_function(linear_output)
```

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## EFFICIENT IMPLEMENTATION

$$\Delta w_{ij} = \eta \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) y_i(\mathbf{x})(1 - y_i(\mathbf{x}))x_j,$$

### Code Snippet:

```
error_term = y_t - y # (batch_size x output_dim)
activation_derivative = y * (1 - y)
error_times_derivative = error_term * activation_derivative
```

# Compute updates for weights

```
update_weights = learning_rate * (X.T @ error_times_derivative) / X.shape[0]
```

*f: sigmoid*

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## EFFICIENT IMPLEMENTATION

$$\Delta w_{ij} = \eta \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) y_i(\mathbf{x})(1 - y_i(\mathbf{x}))x_j,$$

### Code Snippet:

```
# X.T: (input_dim * batch_size)
# error_times_derivative: (batch_size * output_dim)
# Resulting matrix: (input_dim * output_dim), matching shape of W

# Compute updates for weights

update_weights = learning_rate * (X.T @ error_times_derivative) / X.shape[0]
```

*f: sigmoid*

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## EFFICIENT IMPLEMENTATION

$$\Delta b_i = \eta \sum_{\mathbf{x}} (y_i^t(\mathbf{x}) - y_i(\mathbf{x})) y_i(\mathbf{x})(1 - y_i(\mathbf{x}))$$

### Code Snippet:

```
# Compute updates for biases
# Sum over the batch dimension (axis=0) to get individual updates for each
output neuron - in case there are more than one neurons

update_biases = learning_rate * np.sum(error_times_derivative, axis=0) /
x.shape[0]
```

*f: sigmoid*

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

## TRAINING, VALIDATION AND TESTING



parameters



hyper-parameters

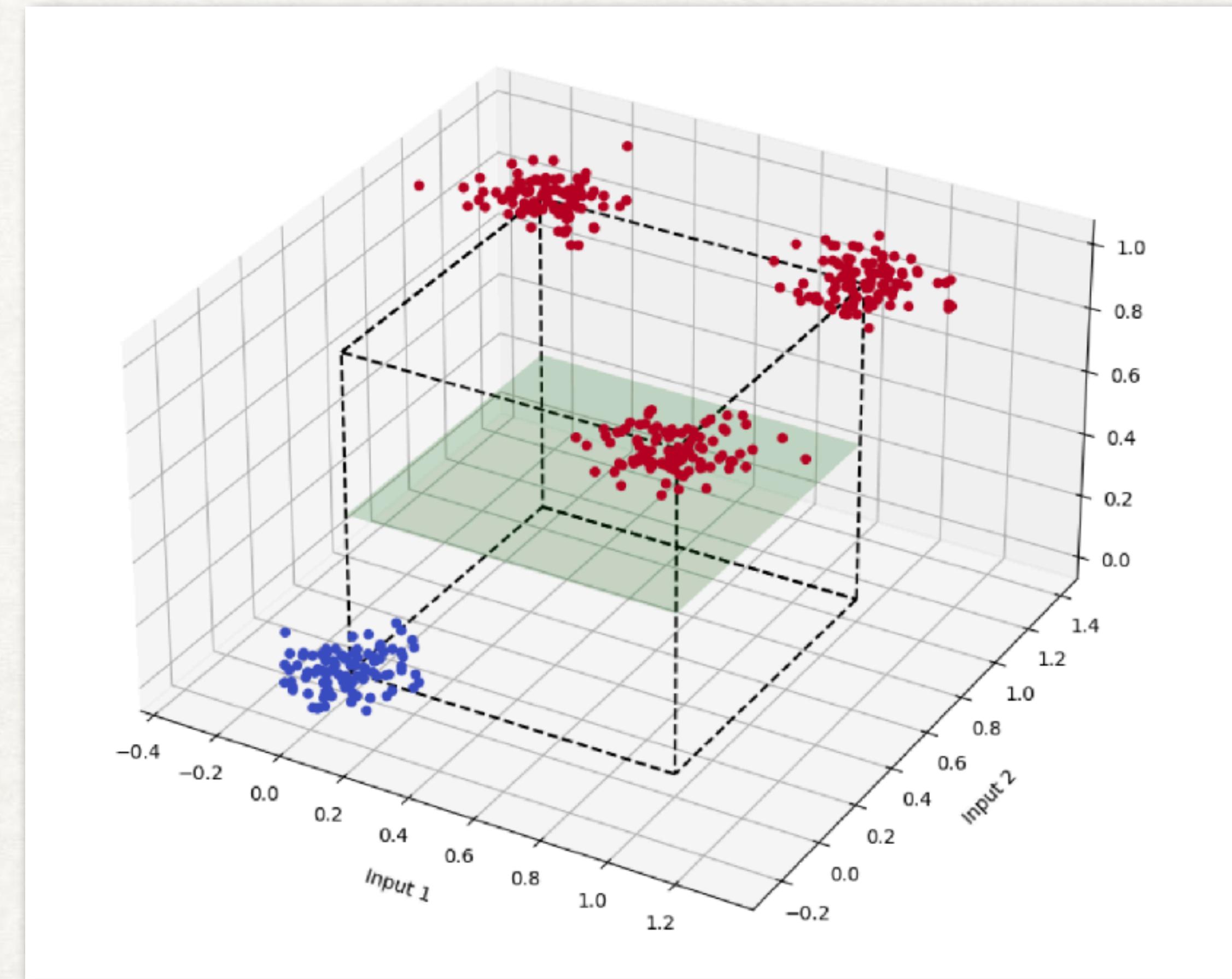


generalisation

Never check test data to optimise hyper parameters or parameters!  
It may lead to overfitting.

# ARTIFICIAL NEURAL NETWORKS AS FUNCTION APPROXIMATIONS

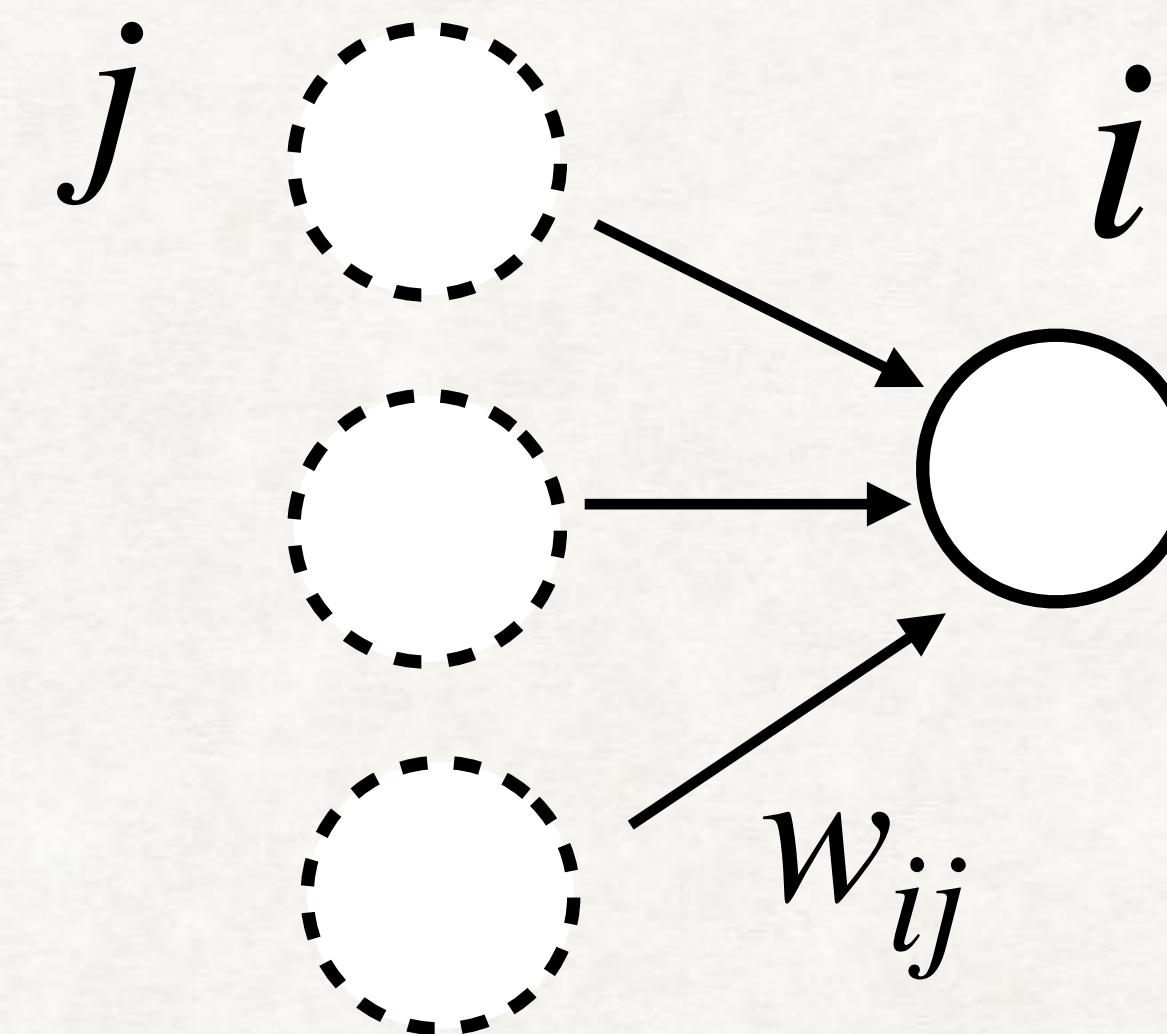
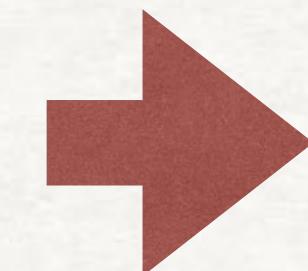
## EXAMPLE



# REPLACING TABLES WITH ANNS

## LEARNING THE VALUE OF ONE STATE INFORMS OTHER STATES

[0.000, 0.002, ]
[0.000, 0.012, ]
[0.000, 0.069, ]
[0.000, 0.261, ]
[0.000, 0.651, ]
[0.000, 0.000, ]



**THANK YOU!**