



Computer  
Science

# CSC380: Principles of Data Science

## Nonlinear Models 3

Xinchen Yu

- HW6 homework solution out on D2L -> Content.
- Final project has been out, due Dec 8, Friday by 11:59 pm.

- Basis Functions
- Support Vector Machine
- **Neural Networks**

Forms of NNs are used all over the place nowadays...



**FB Auto Tagging**

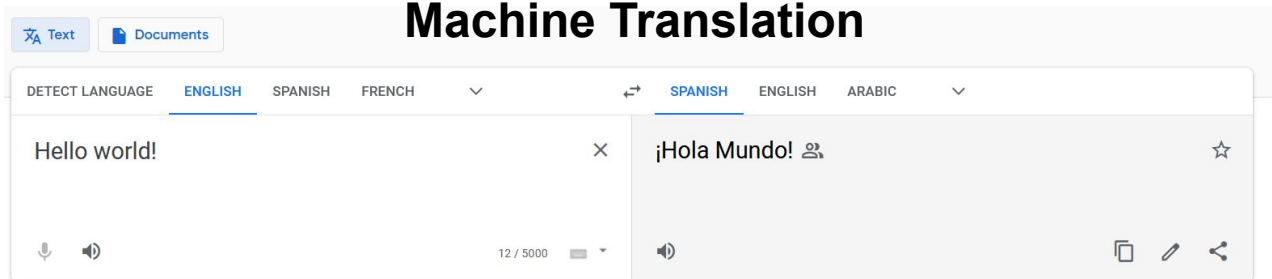


**Self-Driving Cars**



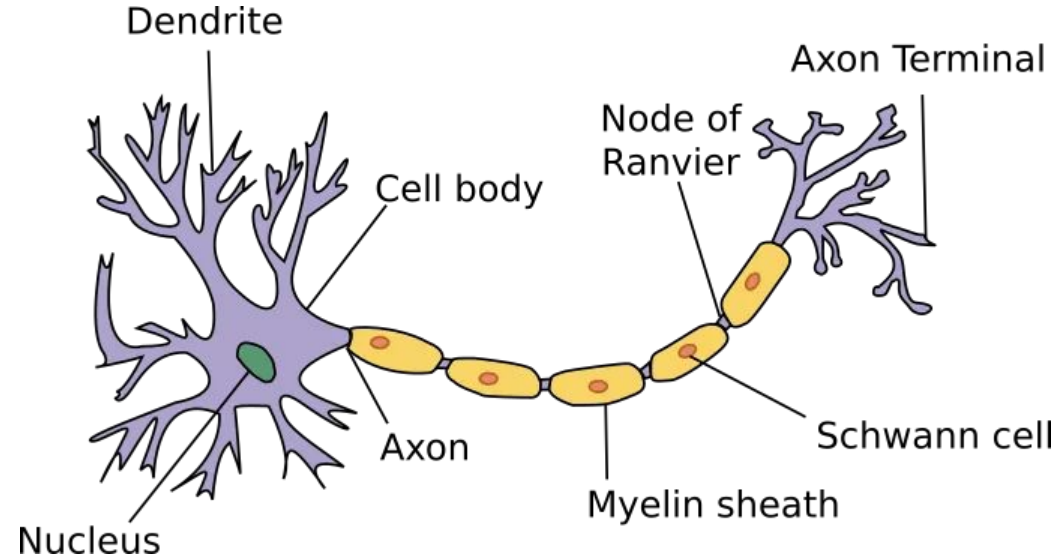
**Creepy Robots**

## Machine Translation



[Send feedback](#)

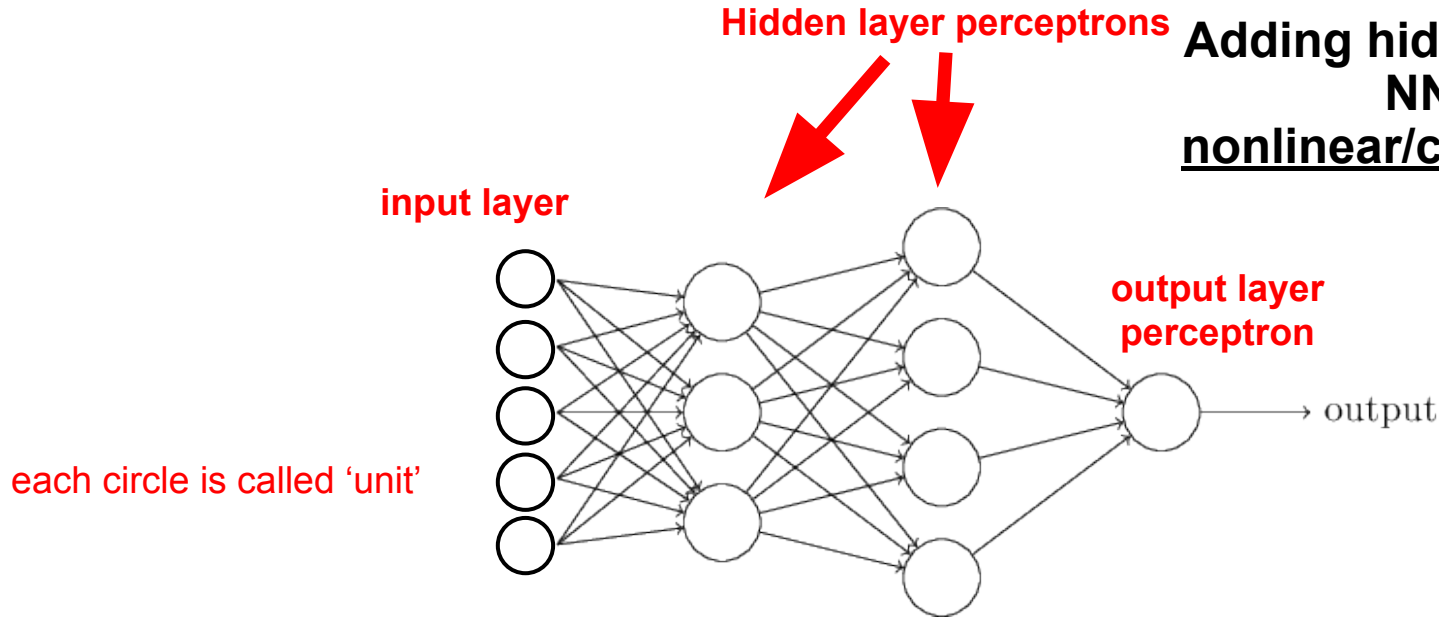
# Biological Neuron



1. It takes inputs from its dendrites;
2. A weighted sum of inputs is passed on to the axon hillock;
3. If weighted sum is larger than threshold limit, the neuron will fire. Otherwise, it stays at rest.
4. The state of our neuron (on or off) then propagates through its axon and is passed on to the other connected neurons via its synapses.

# Multilayer Perceptron

6



This is the quintessential (*Artificial*) *Neural Network*...

... the image above is a special case called *Feed Forward Neural Net*

feed forward: no backward connection

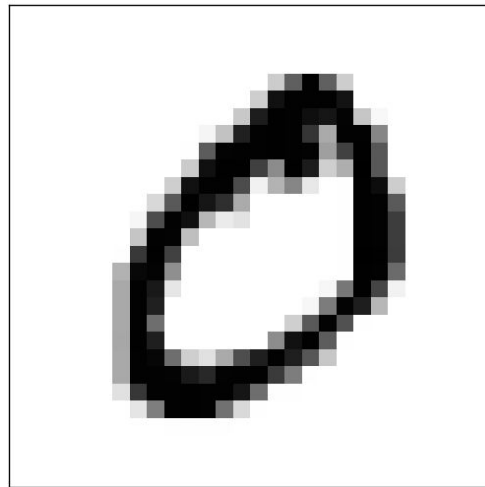
[ Source: <http://neuralnetworksanddeeplearning.com> ]

Classifying handwritten digits is the “Hello World” of NNs



Modified National Institute of Standards and Technology (MNIST) database contains 60k training and 10k test images

Each character is centered in a  $28 \times 28 = 784$  pixel grayscale image

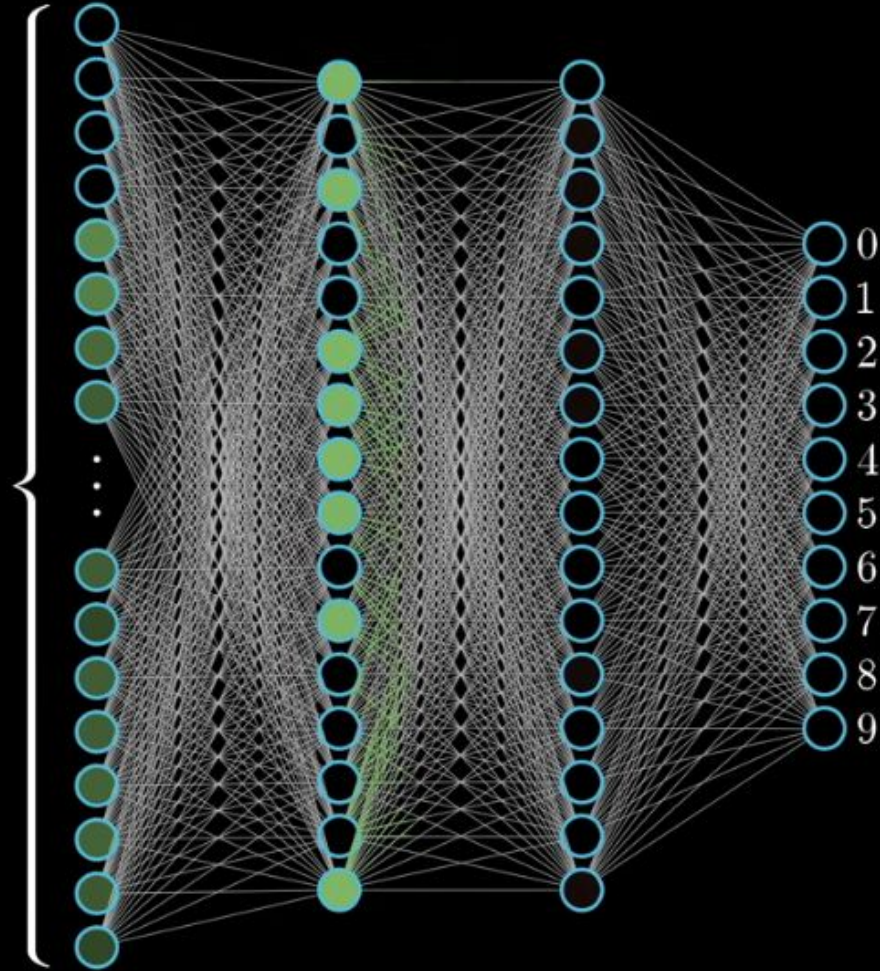


Fun fact: Kernel Ridge Regression with RBF kernel with no regularization gives 1.2% test error rate.

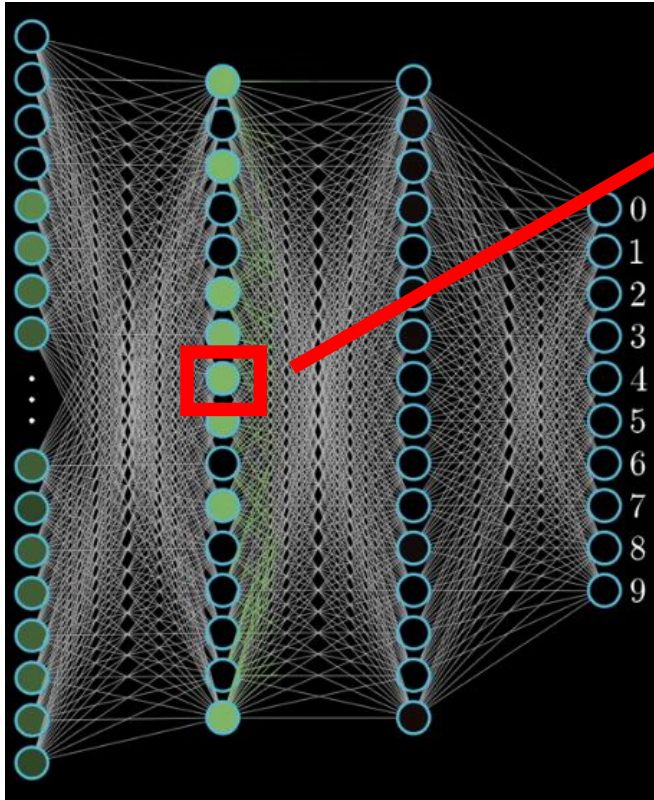


784

Each image pixel is a  
number in  $[0,1]$  indicated  
by highlighted color







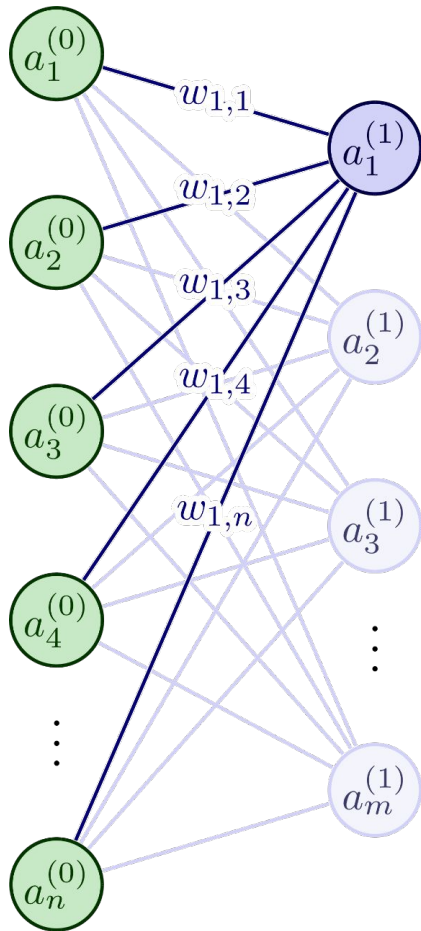
Each node computes a *weighted combination* of nodes at the previous layer...

$$w_1x_1 + w_2x_2 + \dots + w_nx_n$$

Then applies a *nonlinear function* to the result

$$\sigma(w_1x_1 + w_2x_2 + \dots + w_nx_n + b)$$

Usually, we also introduce a constant *bias* parameter (usually hidden when we visualize the network)



$$= \sigma \left( w_{1,0}a_0^{(0)} + w_{1,1}a_1^{(0)} + \dots + w_{1,n}a_n^{(0)} + b_1^{(0)} \right)$$

$$= \sigma \left( \sum_{i=1}^n w_{1,i}a_i^{(0)} + b_1^{(0)} \right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma \left[ \begin{pmatrix} w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ w_{2,0} & w_{2,1} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix} \right]$$

$$a^{(1)} = \sigma \left( \mathbf{W}^{(0)} a^{(0)} + \mathbf{b}^{(0)} \right)$$

Number of parameters in this example:  $m \times n + m$

We call this an *activation function* and typically write it in vector form,

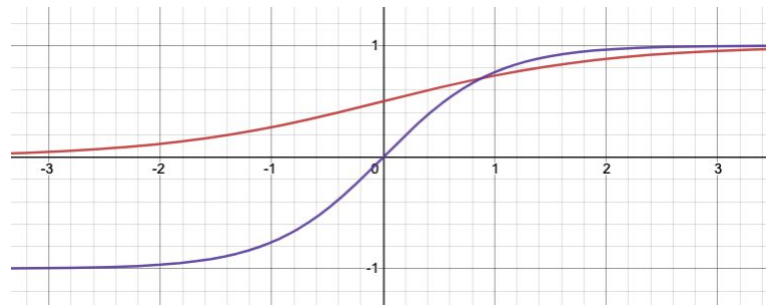
$$\sigma(w_1x_1 + w_2x_2 + \dots + w_nx_n + b) = \sigma(w^T x + b)$$



An early choice was the *logistic function*,

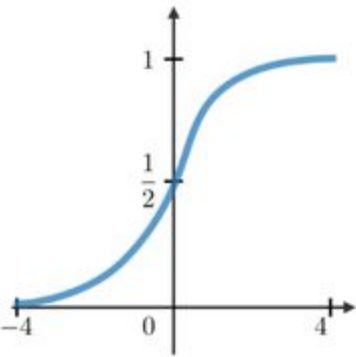
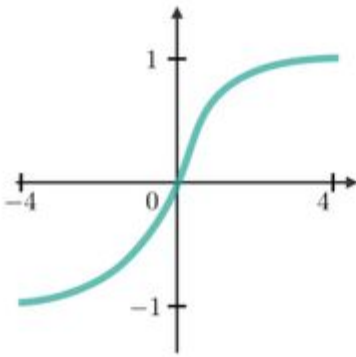
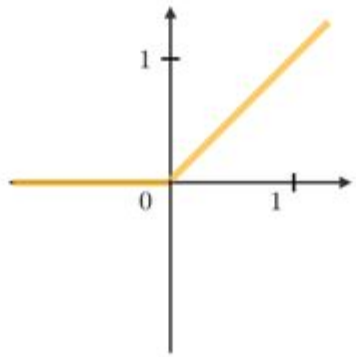
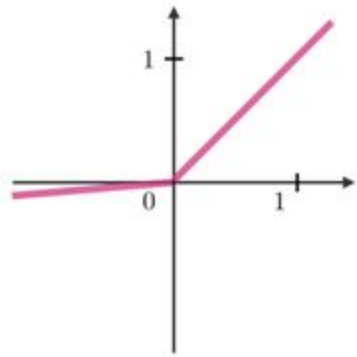
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

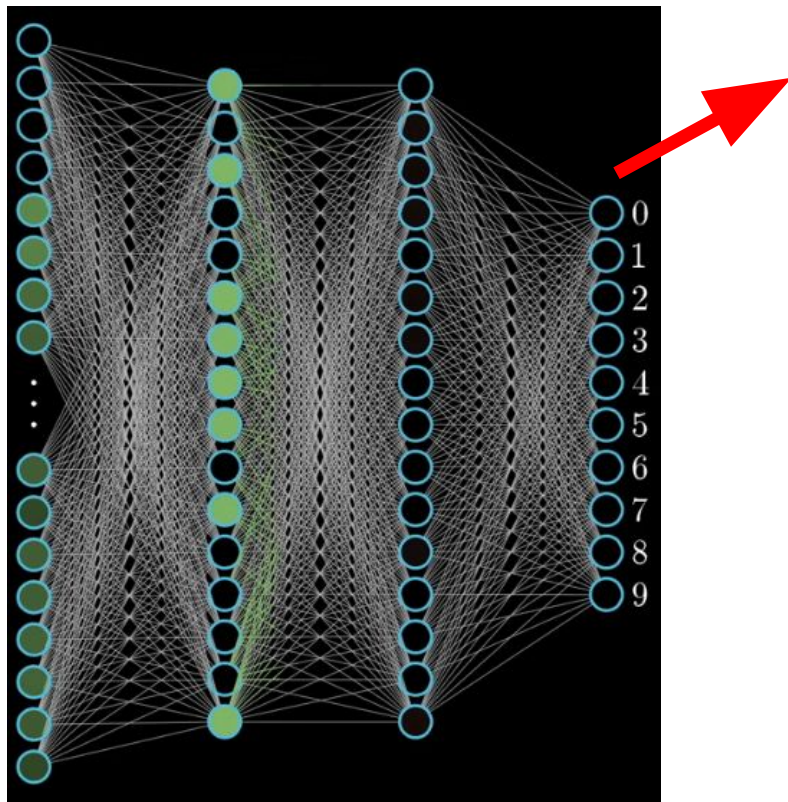
Later, people found that a scaled version called **tanh** trains faster (=converges faster)

$$\tanh(z) = 2\sigma(2z) - 1$$



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

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			



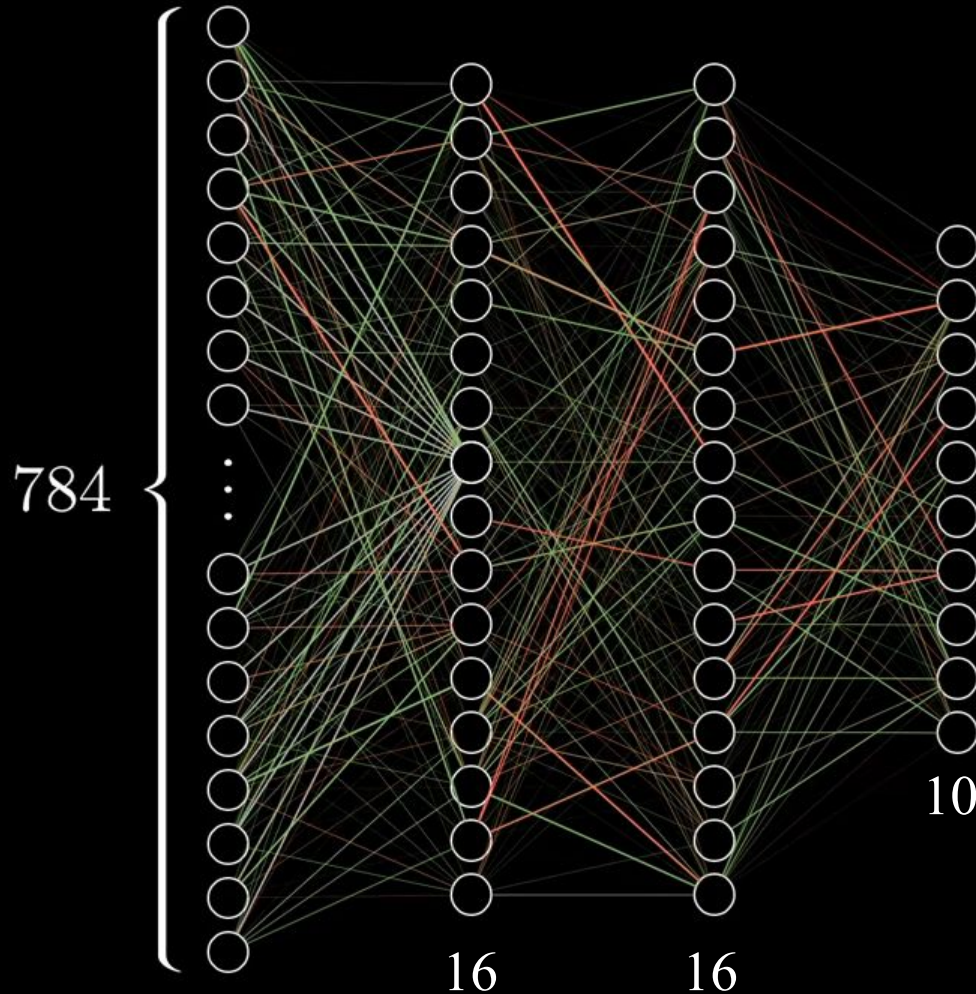
**Final layer** is a linear model...  
for classification this is a logistic regression

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

**Vector of activations from previous layer**

Note: we don't use ReLU for the last layer





$$784 \times 16 + 16 \times 16 + 16 \times 10$$

weights

$$16 + 16 + 10$$

biases

13,002

Each parameter has some impact  
on the output...need to train all  
these parameters simultaneously  
to have a good prediction  
accuracy

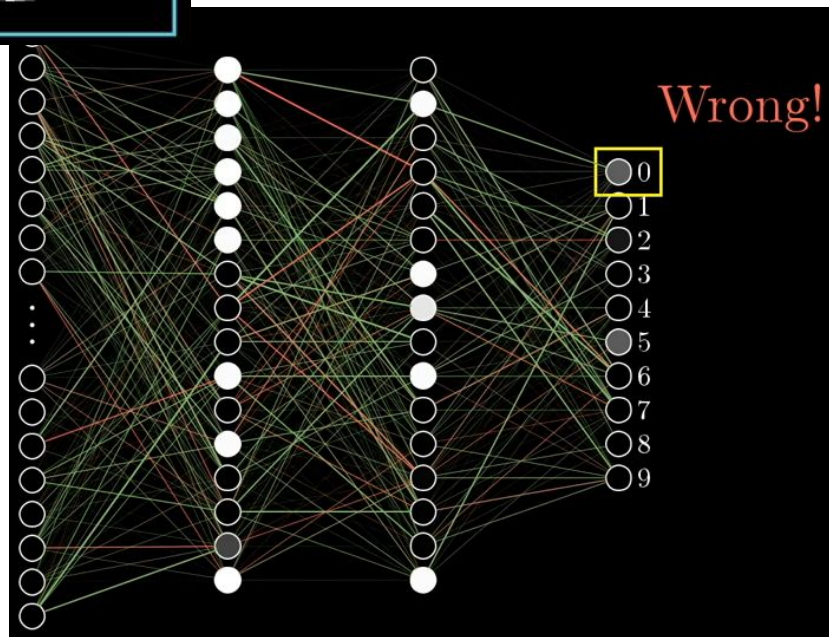
$$X^{\text{Train}} = \begin{matrix} \begin{matrix} 0 & 4 & 1 & 9 & 2 & 1 & 3 & 1 & 4 & 3 \\ 5 & 3 & 6 & 1 & 7 & 2 & 8 & 6 & 9 & 4 \\ 0 & 9 & 1 & 1 & 2 & 4 & 3 & 2 & 7 & 3 \\ 8 & 6 & 9 & 0 & 5 & 6 & 0 & 7 & 6 & 1 \\ 8 & 7 & 9 & 3 & 9 & 8 & 5 & 9 & 3 & 3 \\ 0 & 7 & 4 & 9 & 8 & 0 & 9 & 4 & 7 & 4 \\ 4 & 6 & 0 & 4 & 5 & 6 & 1 & 0 & 0 & 1 \\ 7 & 1 & 6 & 3 & 0 & 2 & 1 & 1 & 7 & 9 \\ 0 & 2 & 6 & 7 & 8 & 3 & 9 & 0 & 4 & 6 \\ 7 & 4 & 6 & 8 & 0 & 7 & 8 & 3 & 1 & 5 \end{matrix} \end{matrix}$$

$$Y^{\text{Train}} = \begin{pmatrix} 0 & 4 & 1 & \dots & 3 \\ 5 & 3 & 6 & \dots & 4 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 7 & 4 & 6 & \dots & 5 \end{pmatrix}$$

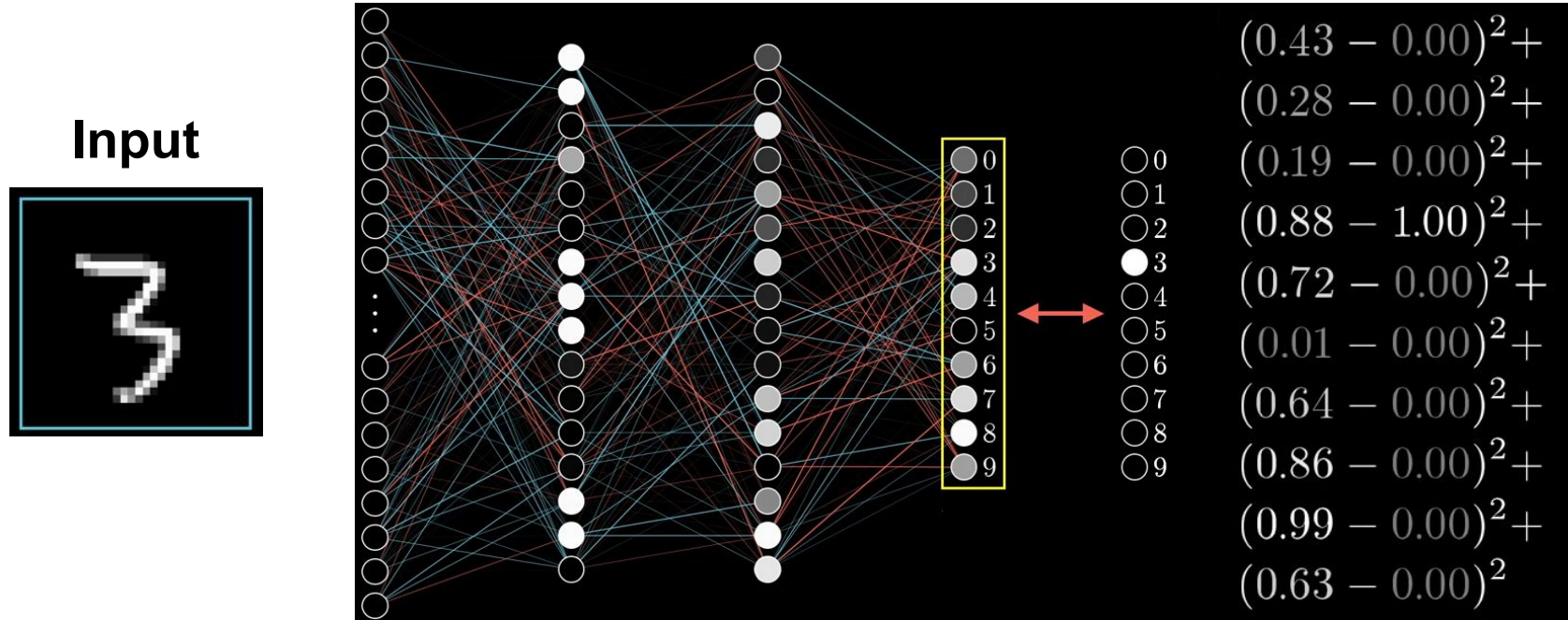
- How to score final layer output?
- How to adjust weights?



For each training example,  
predict label and adjust  
weights...



Score based on difference between final layer and one-hot vector of true class...





# Training Multilayer Perceptron

1  
7

Our cost function for  $i^{\text{th}}$  input is error in terms of weights / biases...

$$\text{Cost}_i(w_1, \dots, w_n, b_1, \dots, b_n)$$

13,002 Parameters in this network

...minimize cost over all training data...

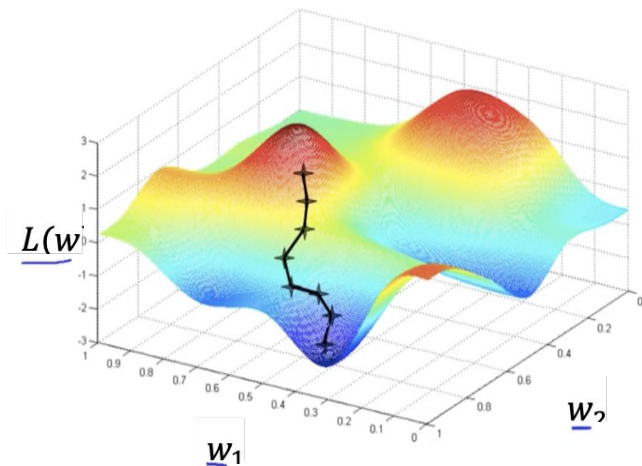
$$\min_{w,b} \mathcal{L}(w, b) = \sum_i \text{Cost}_i(w_1, \dots, w_n, b_1, \dots, b_n)$$

This is a super high-dimensional optimization (13,002 dimensions in this example)...how do we solve it?

**Gradient descent!**

How to minimize a function?

$$\arg \min_w L(w)$$



Randomly start from some  $w^{(1)} \in \mathbb{R}^d$

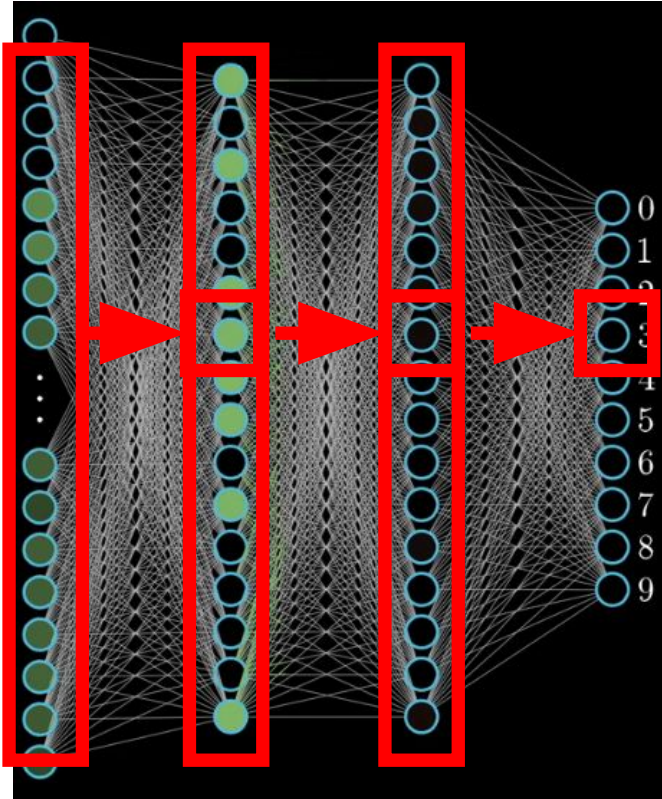
For  $t = 1, 2, \dots$

- Compute the gradient  $g_t \in \mathbb{R}^d$  at the location  $w^{(t)}$
- Move to that direction:  

$$w^{(t+1)} = w^{(t)} - \eta_t \cdot g^{(t)}$$
 where  $\eta_t > 0$  is a stepsize parameter.
- If  $L(w^{(t+1)}) \approx L(w^{(t)})$ , stop.

The choice of  $\eta_t$  matters! (default:  $\eta_t = 0.01$ )

[ Source : 3Blue1Brown : <https://www.youtube.com/watch?v=aircAruvnKk> ]



Randomly initialize  $\{w^{(u)}\}_{u \in \text{units in neural net}}$

For  $i \in \{1, \dots, n_{\text{epochs}}\}$

- For  $(x, y)$  in train set:
  - Forward pass:
    - evaluate the neural net output
    - measure the loss
  - Backward pass: compute the gradients.
  - Take the gradient step to update the weights  $\{w^{(u)}\}$

Dependencies between layers.

No dependencies between units at the same layer.

⇒ Many GPU supported libraries available.

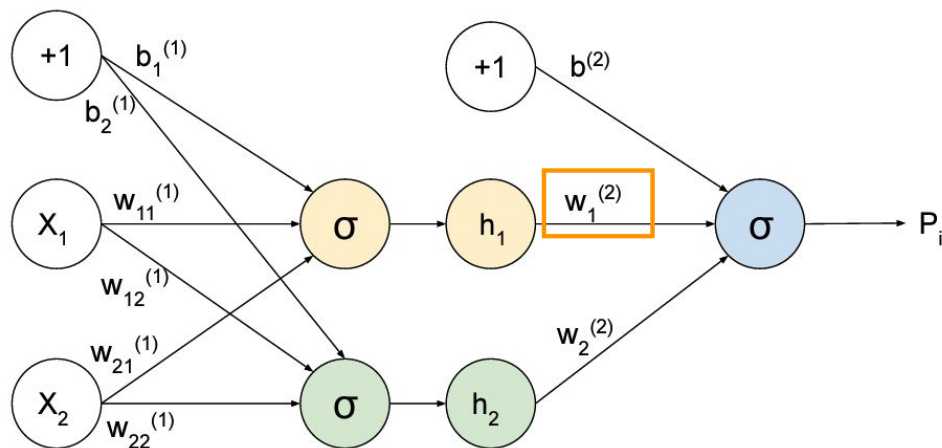
# Backpropagation: an example

$$\begin{pmatrix} z_1^{(1)} = w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + b_1^{(1)} \\ z_2^{(1)} = w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)} \end{pmatrix}$$

$$\begin{pmatrix} h_1 = \sigma(z_1^{(1)}) \\ h_2 = \sigma(z_2^{(1)}) \end{pmatrix}$$

$$(z^{(2)} = w_1^{(2)}h_1 + w_2^{(2)}h_2 + b^{(2)})$$

$$(p_i = \sigma(z^{(2)}))$$



Useful to know:

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

$$\text{Then } \frac{dp}{dx} = (1 + e^{-x})^{-2} e^{-x} = \frac{1}{1 + e^{-x}} \times \frac{e^{-x}}{1 + e^{-x}} = p(1 - p)$$

$$\text{Error: } L = \frac{1}{2} \|p - y\|^2 = \sum_i \frac{1}{2} (p_i - y_i)^2$$

$$\text{Compute: } \frac{\partial L}{\partial w_1^{(2)}}$$

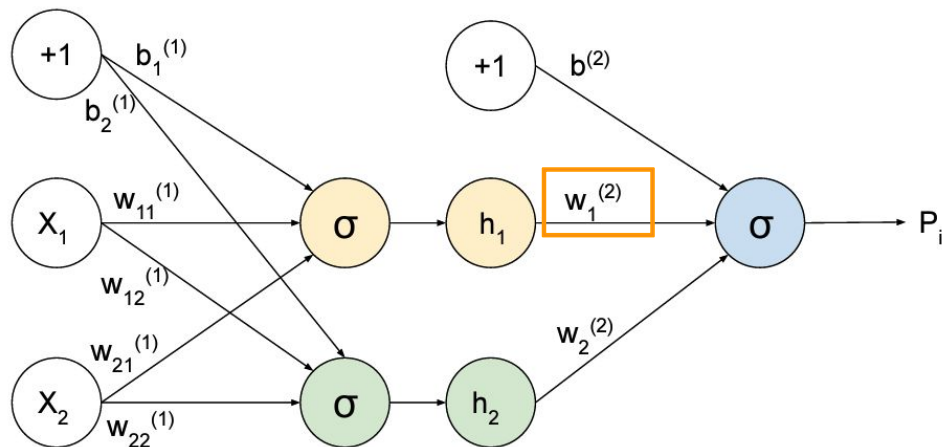
# Backpropagation: an example

$$\begin{pmatrix} z_1^{(1)} = w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + b_1^{(1)} \\ z_2^{(1)} = w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)} \end{pmatrix}$$

$$\begin{pmatrix} h_1 = \sigma(z_1^{(1)}) \\ h_2 = \sigma(z_2^{(1)}) \end{pmatrix}$$

$$(z^{(2)} = w_1^{(2)}h_1 + w_2^{(2)}h_2 + b^{(2)})$$

$$(p_i = \sigma(z^{(2)}))$$



$$\frac{\partial L}{\partial w_1^{(2)}} = \sum_i \frac{\partial L_i}{\partial w_1^{(2)}}$$

$$\text{Error: } L = \frac{1}{2} \|p - y\|^2 = \sum_i \frac{1}{2} (p_i - y_i)^2$$

$$\frac{\partial L_i}{\partial w_1^{(2)}} = \frac{\partial z^{(2)}}{\partial w_1^{(2)}} \times \frac{\partial p_i}{\partial z^{(2)}} \times \frac{\partial L_i}{\partial p_i} = h_1 \times p_i(1 - p_i) \times (p_i - y)$$

$$\text{Compute: } \frac{\partial L}{\partial w_1^{(2)}}$$

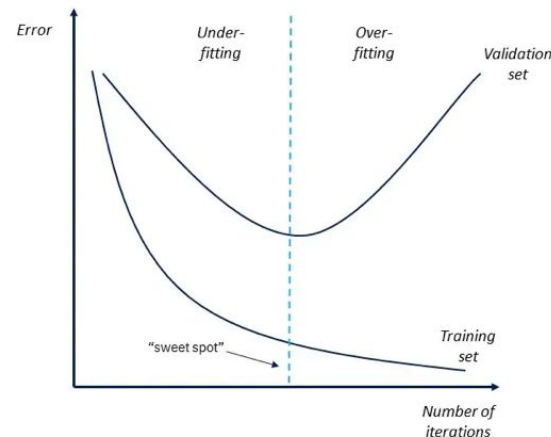
Chain rule

*With four parameters I can fit an elephant. With five I can make him wiggle his trunk. - John von Neumann*

$$w = \arg \min_w \text{Cost}(w) + \alpha \cdot \text{Regularizer}(\text{Model})$$

Our example model has 13,002 parameters...that's a lot of elephants!  
Regularization is critical to avoid overfitting...

...numerous regularization schemes are used in training neural networks.  
but the standard is of course,  $\sum_i w_i^2$



**hidden\_layer\_sizes** : *tuple, length = n\_layers - 2, default=(100,)*

The *i*th element represents the number of neurons in the *i*th hidden layer.

**activation** : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*

Activation function for the hidden layer.

**solver** : *{'lbfgs', 'sgd', 'adam'}, default='adam'*

The solver for weight optimization.

**alpha** : *float, default=0.0001*

L2 penalty (regularization term) parameter.

**learning\_rate** : *{'constant', 'invscaling', 'adaptive'}, default='constant'*

Learning rate schedule for weight updates.

**early\_stopping** : *bool, default=False*

Whether to use early stopping to terminate training when validation score is not improving. If set to true,



Fetch MNIST data from [www.openml.org](http://www.openml.org) :

```
X, y = fetch_openml("mnist_784", version=1, return_X_y=True)
X = X / 255.0
```

Train test split (60k / 10k),

```
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]
```

Create MLP classifier instance,

- Single hidden layer (50 nodes)
- Use stochastic gradient descent
- Maximum of 10 learning iterations
- Small L2 regularization  $\alpha=1e-4$

```
mlp = MLPClassifier(
    hidden_layer_sizes=(50,),
    max_iter=10,
    alpha=1e-4,
    solver="sgd",
    verbose=10,
    random_state=1,
    learning_rate_init=0.1,
)
```



Fit the MLP and print stuff...

```
mlp.fit(X_train, y_train)

print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
```

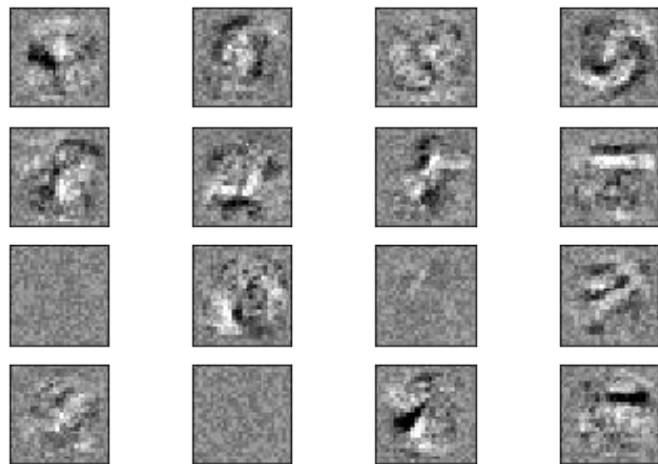
```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```

Visualize the weights for each node...

`coefs_[i]`:  $n_{\text{input}} \times n_{\text{output}}$  matrix of weights for layer  $i$

```
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray,
               |vmin=0.5 * vmin, vmax=0.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())
```

...magnitude of weights indicates which input features are important in prediction



## Many other NN architectures exist beyond MLP

- **Convolutional NN (CNN)** For image processing / computer viz.
- **Recurrent NN (RNN)** For sequence data (e.g. acoustic signals, video, etc.) , long short-term memory (LSTM) is popular
- **Generative Adversarial Nets (GANs)** For generating creepy deepfakes
- **Restricted Boltzmann Machine (RBM)** Another generative model

## Many open areas being researched

- More reliable uncertainty estimates
- Robustness to exploits
- Interpretability
- Better scalability



There are **tons** of excellent resources for learning about neural networks online...here are two quick ones:

3Blue1Brown Youtube channel has a nice four-part intro:  
<https://www.youtube.com/watch?v=aircAruvnKk>

Free book by Michael Nielson uses MNIST example in Python:  
<http://neuralnetworksanddeeplearning.com/>

Prof. Steven Bethard often teaches an excellent class:  
ISTA 457 / INFO 557

Play with a small multilayer perceptron on a  
binary classification task...

<https://playground.tensorflow.org/>