FSAN/ELEG815: Statistical Learning

Gonzalo R. Arce

Department of Electrical and Computer Engineering
University of Delaware

X: Neural Networks

What is Deep Learning?

Artificial Inteligence

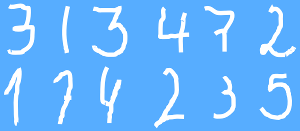
Any technique that enables computers to mimic human behavior

Machine Learning

Ability to learn without explicity being programed

Deep Learning

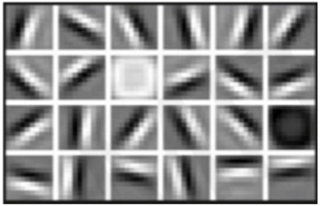
Extract patterns from data using neural networks



Why Deep Learning?

Hand engineered features are time consuming, brittle and not scalable in practice

Can we learn the **underlying features** directly from data?



Line and Edges

Eyes, Nose and Ears

Facial Structure

Why Now?

1952 Stochastic Gradient Descent

1958 Preceptron
- Learnable Weights

⋮

1986 Backpropagation
- Multi-layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

⋮

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Large Datasets
- Easier Collection and Storage

IMAGENET

Wikipedia The Free Encyclopedia

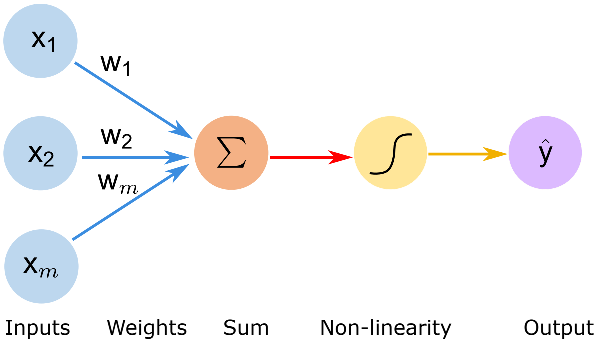
2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

PyTorch

The Perceptron: Forward Propagation



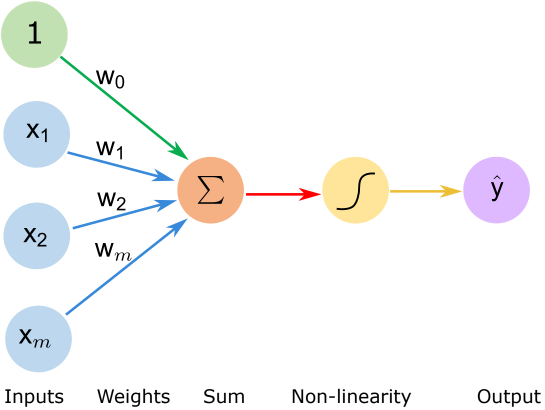
Output

Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{m} w_i x_i\right)$$

Non-linear activation function

The Perceptron: Forward Propagation



Output

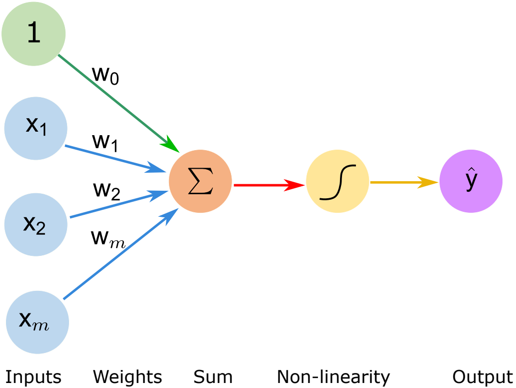
Bias

Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} w_i x_i\right)$$

Non-linear activation function

The Perceptron: Forward Propagation

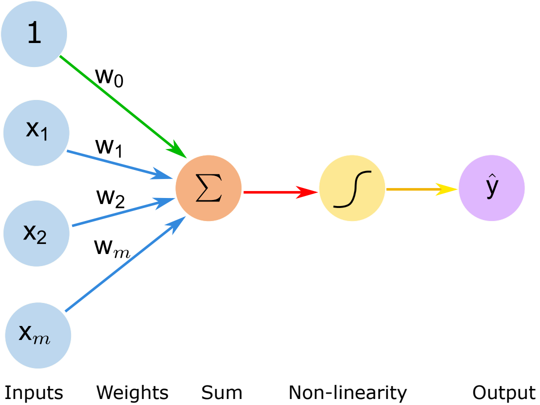


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} w_i x_i\right)$$

$$\hat{y} = g\left(w_0 + \mathbf{X}^\top \mathbf{W}\right)$$

where: $\mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

The Perceptron: Forward Propagation

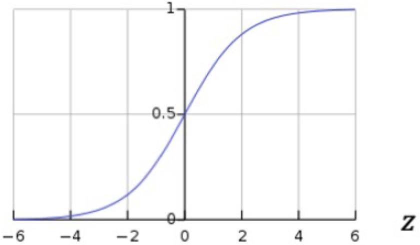


Activation Functions

$$\hat{y} = g\left(w_0 + \mathbf{X}^\top \mathbf{W}\right)$$

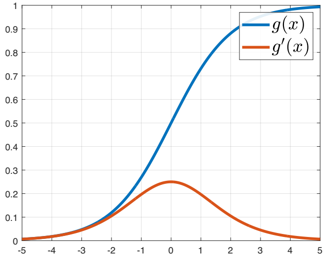
- Example: sigmoid function

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



Common Activation Functions

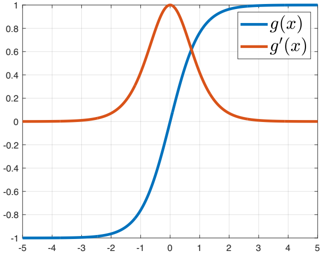
Sigmoid Function



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

$$g'(x) = g(x)(1-g(x))$$

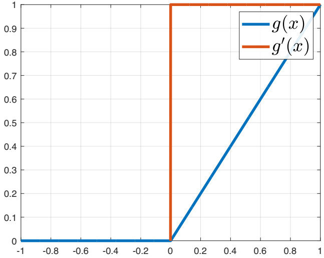
Hyperbolic Function



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

$$g'(x) = 1 - g(x)^2$$

Rectified Linear Unit (ReLU)

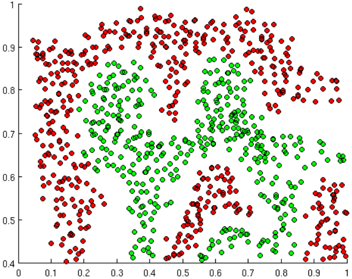


$$g(x) = \max(0, x)$$

$$g'(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The Importance of Activation Functions

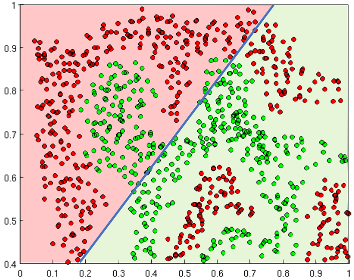
The purpose of the activation function is to **introduce non-linearities** into the network



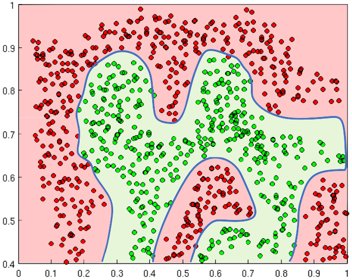
What if we want to build a neural network to distinguish green vs red points?

The Importance of Activation Functions

The purpose of the activation function is to **introduce non-linearities** into the network

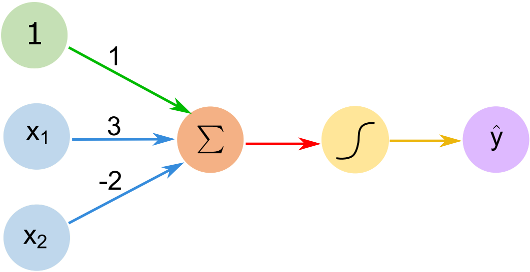


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example



We have $w_0 = 1$ and $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(w_0 + \mathbf{X}^\top \mathbf{W})$$

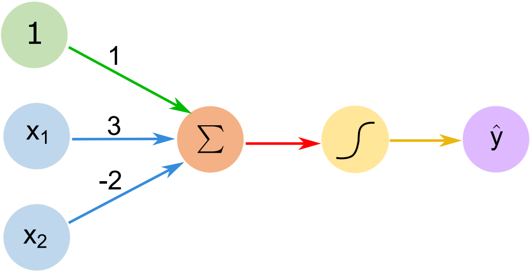
$$\hat{y} = g\left(w_0 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^\top \begin{bmatrix} 3 \\ -2 \end{bmatrix} \right)$$

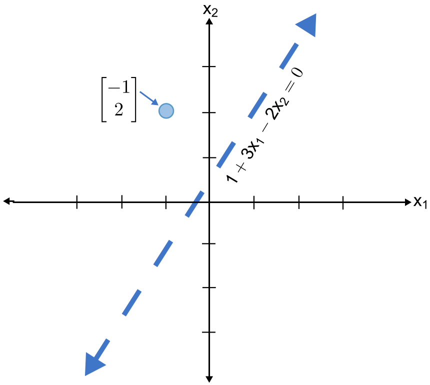
$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D}})$$

The Perceptron: Example

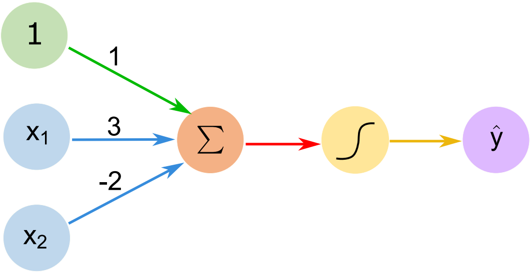
Assume $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$:

$$\hat{y} = g(1 + (3 \times -1) - (2 \times 2))$$
$$\hat{y} = g(-6) \approx 0.0002$$

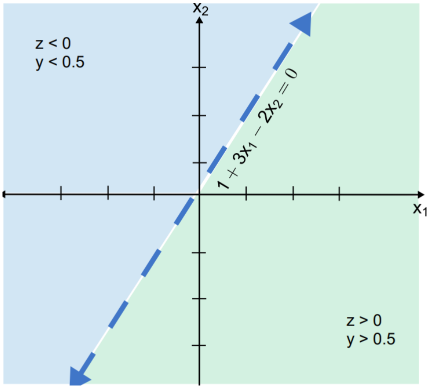




The Perceptron: Example

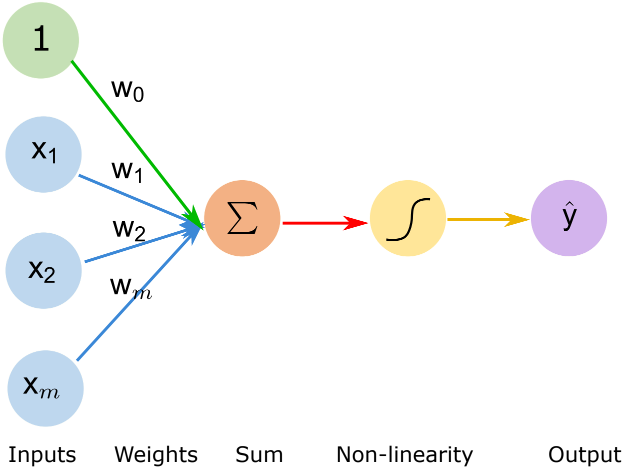


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

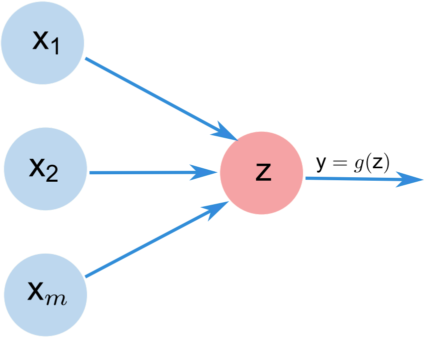


Building Neural Network with Perceptron

$$\hat{y} = g(w_0 + \mathbf{x}^\top \mathbf{w})$$



Building Neural Network with Perceptron



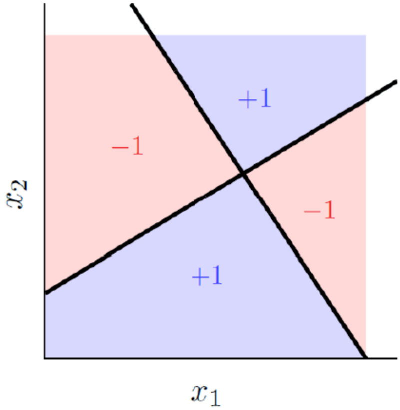
$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

Combining Perceptrons

Consider the target function in the figure which is a Boolean XOR function.

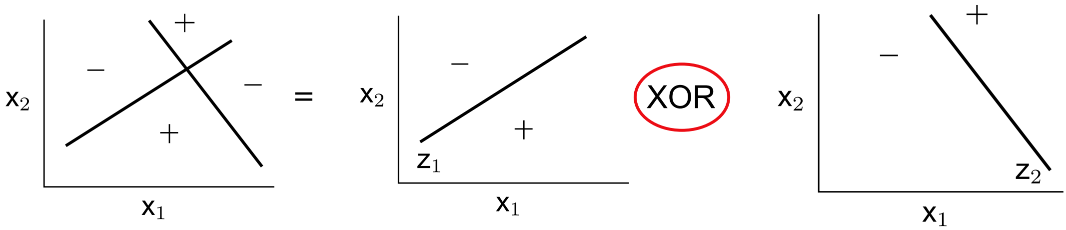
The perceptron cannot implement this classification.

Decompose the f into two perceptrons, corresponding to the lines in the figure.



Combining Perceptrons

$$f = \text{XOR}(z_1, z_2)$$



f $\qquad$ $g_1(\mathbf{x}) = \text{sign}(\mathbf{w}_1^\top \mathbf{x})$ $\qquad$ $g_2(\mathbf{x}) = \text{sign}(\mathbf{w}_2^\top \mathbf{x})$

Rewrite f using OR and AND operations:

$$f = g_1\overline{g_2} + \overline{g_1}g_2$$

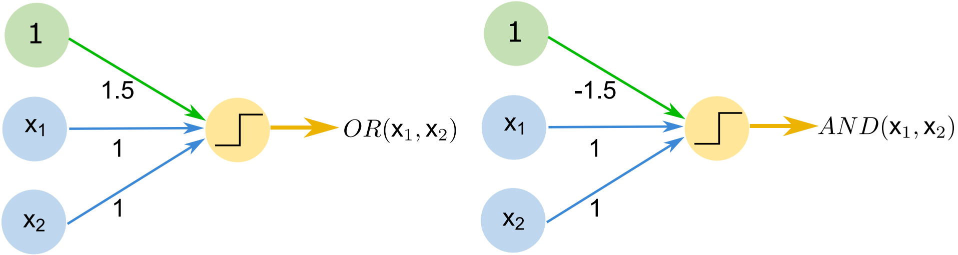
where AND is represented by multiplication, OR by addition and overbar for negation.

Combining Perceptrons

OR and AND can be implemented by the perceptron:

$$\mathrm{OR}(x_1, x_2) = sign(x_1 + x_2 + 1.5)$$

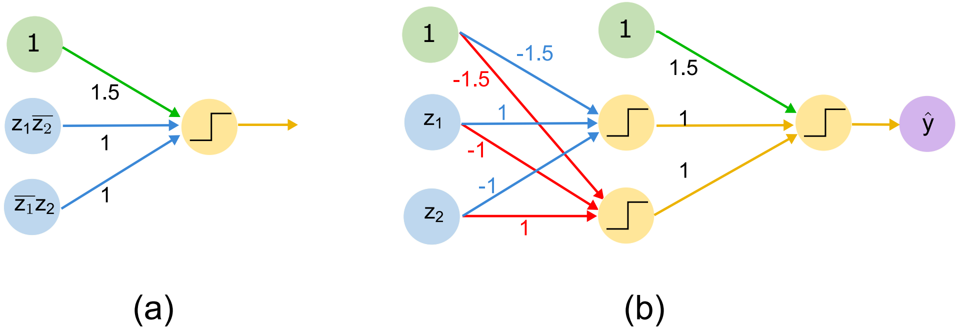
$$\mathrm{AND}(x_1, x_2) = sign(x_1 + x_2 - 1.5)$$



Everything coming to a node is summed and then transformed by $sign(\cdot)$ to get the final output

Creating Layers

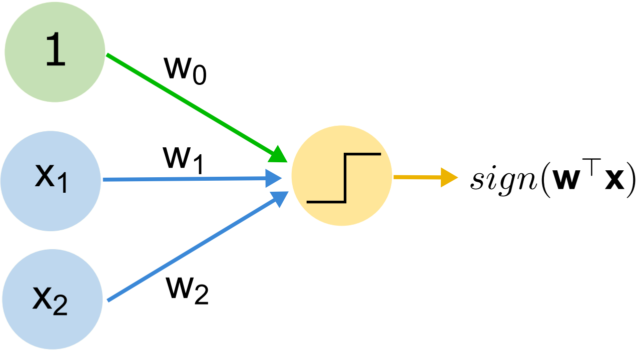
$$f = z_1\overline{z_2} + \overline{z_1}z_2$$



(a) OR of the inputs $z_1\overline{z_2}$ and $\overline{z_1}z_2$ (b) The blue and red weights simulate the required ANDS ($z_1\overline{z_2}$ and $\overline{z_1}z_2$). Negative in te weights handle negations.

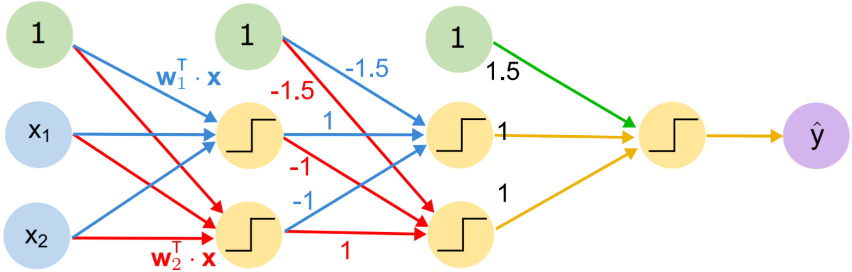
The Multilayer Perceptron

$z_1(\mathbf{x}) = \mathbf{w}_1^\top \mathbf{x}$ and $z_2(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{x}$ are perceptrons:



The Multilayer Perceptron

$z_1(\mathbf{x}) = \mathbf{w}_1^\top \mathbf{x}$ and $z_2(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{x}$ are perceptrons:

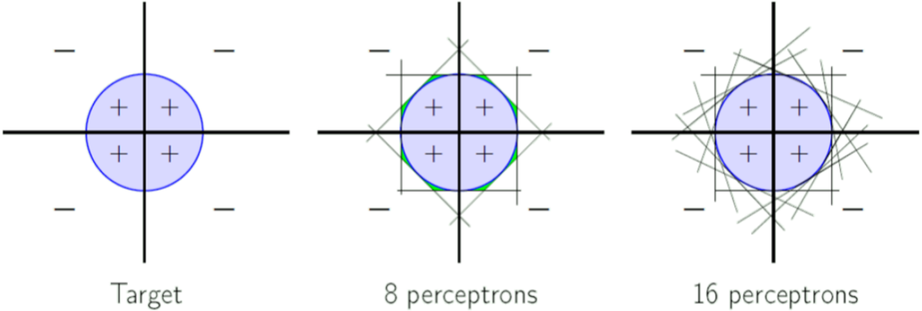


3 layers
compare to the perceptron
(one)

"Feedforward"
No backward pointing arrows and no jumps to other layers

A Powerful Model

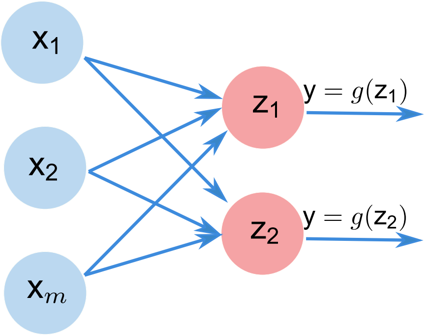
Let's consider now a dis target function:



We can model more complex functions by adding more nodes (hidden unites) in the hidden layers (more perceptrons in the decomposition of f)

Multi-output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers:

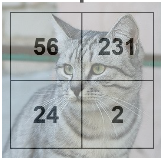


$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

Multi-output Perceptron

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

Stretch pixels into column



Input image

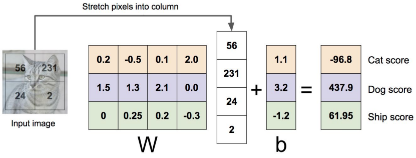
$$\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix}}_{W} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix} + \underbrace{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}}_{b} = \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix} \begin{matrix} \text{Cat score} \\ \text{Dog score} \\ \text{Ship score} \end{matrix}$$

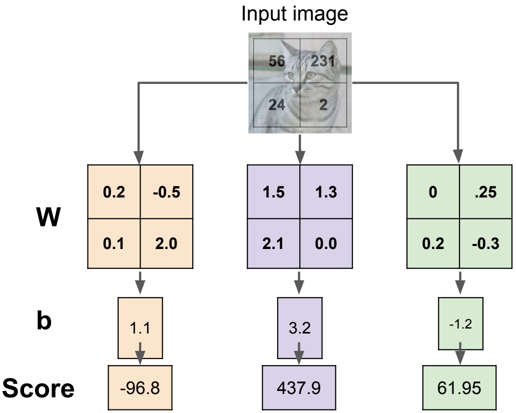
Multi-output Perceptron

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

Algebraic Viewpoint

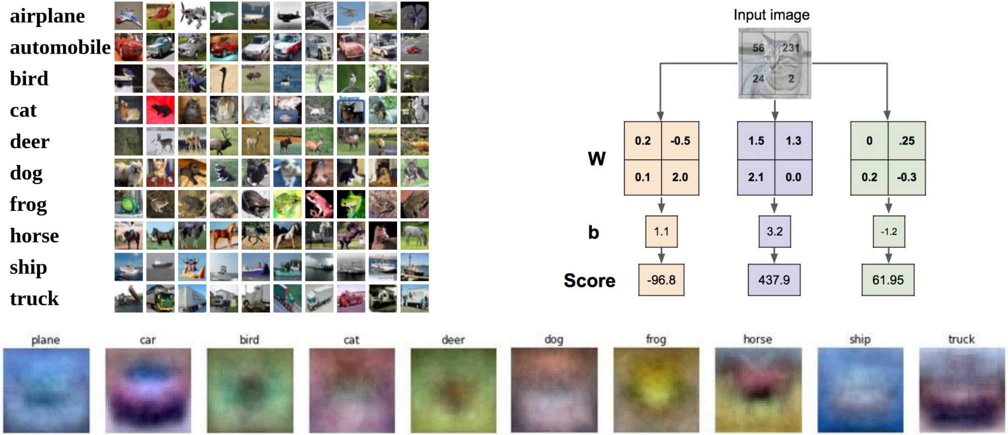
f(x,W) = Wx





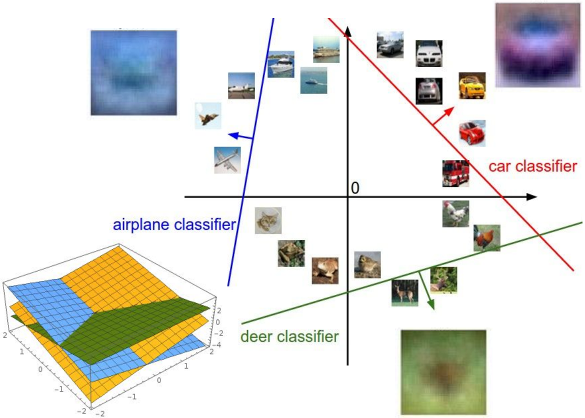
Multi-output Perceptron

Interpreting a Linear Classifier: Visual Viewpoint



Multi-output Perceptron

Interpreting a Linear Classifier: Geometric Viewpoint



Plot created using Wolfram Cloud

$$f(x,W) = Wx + b$$

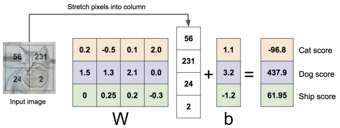
Array of **32x32x3** numbers
(3072 numbers total)

Multi-output Perceptron

Linear Classifier: Three Viewpoints

Algebraic Viewpoint

f(x,W) = Wx

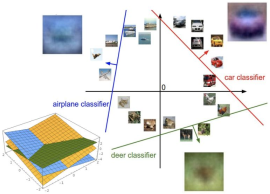


Visual Viewpoint

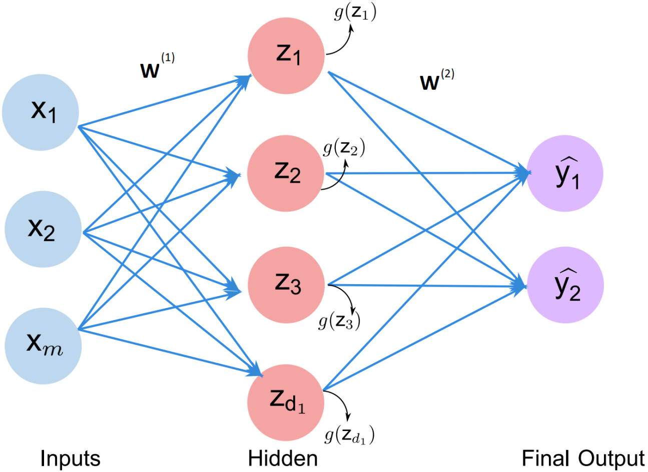
One template
per class

Geometric Viewpoint

Hyperplanes
cutting up space

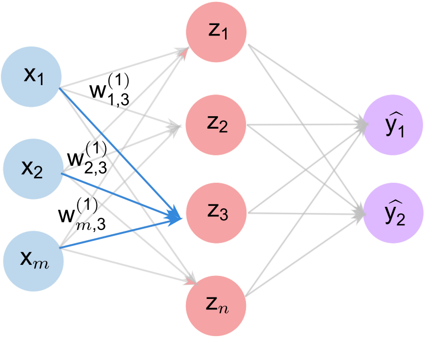


Multi-output Multi-layer perceptron



$$\mathbf{z}_i^{(1)} = \mathbf{w}_{0,i}^{(1)} + \sum_{j=1}^{m} \mathbf{x}_j^{(0)} \mathbf{w}_{j,i}^{(1)} \qquad \hat{\mathbf{y}}_i = g(\mathbf{w}_{0,i}^{(2)} + \sum_{j=1}^{m} \mathbf{x}_j^{(1)} \mathbf{w}_{j,i}^{(2)}) \qquad \mathbf{x}_j^{(1)} = g(\mathbf{z}_j^{(1)})$$

Multi-output Multi-layer perceptron



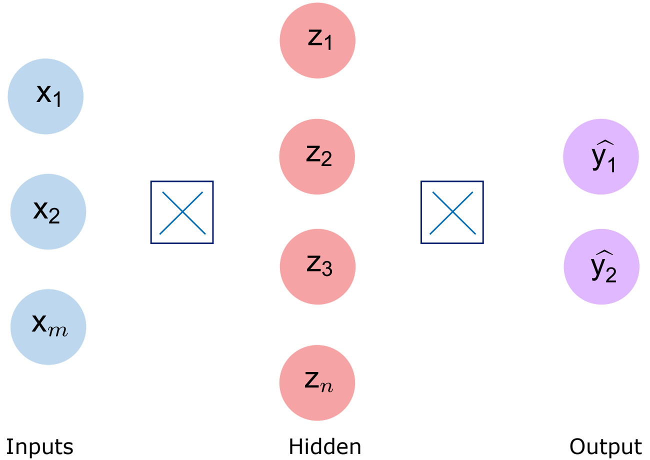
$$z_3 = w^{(1)}_{0,3} + \sum_{j=1}^{m} x_j w^{(1)}_{j,3}$$

$$z_3 = w^{(1)}_{0,3} + x_1 w^{(1)}_{1,3} + x_2 w^{(1)}_{2,3} + x_3 w^{(1)}_{3,3}$$

Generally

$$\mathbf{z} = \mathbf{W}^{(1)\top} \mathbf{x}^{(0)} + w^{(1)}_0$$

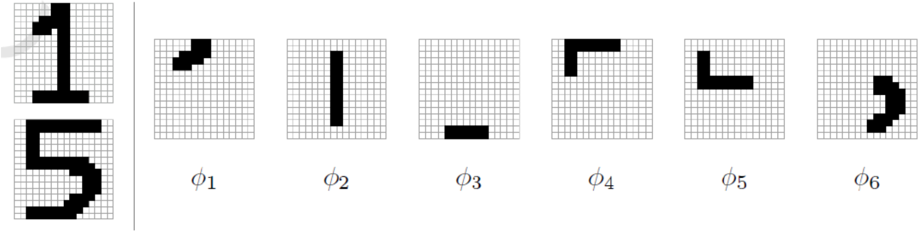
Single Layer Neural Network



Network with many layers - Example

Classify "1" vs "5". Decompose the two digits into basic components:

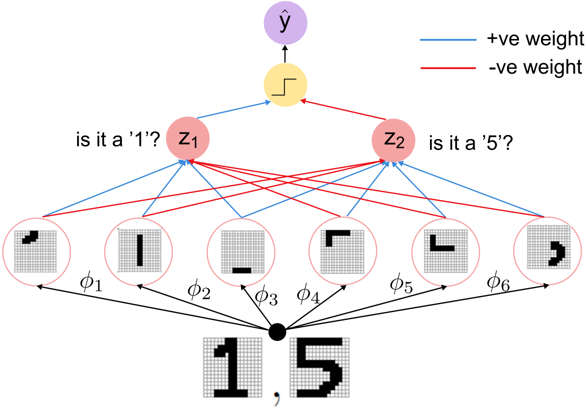
- Every "1" should contain ϕ_1, ϕ_2 and ϕ_3
- Every "5" should contain ϕ_3, ϕ_4, ϕ_5 and ϕ_6, perhaps a little of ϕ_1



These shapes are *features* of the input. We want ϕ_1 to be large (close to 1) if the corresponding feature is in the input image and small (close to -1) if not.

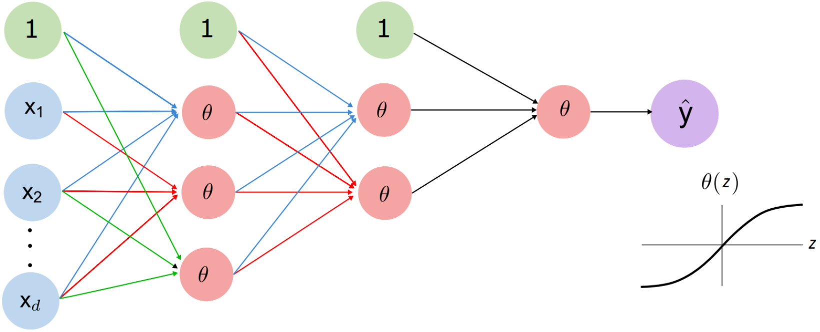
Network with many layers - Example

ϕ_1 is feature function which computes the presence (+1) and absence (-1) of the corresponding feature.



If we feed in "1", ϕ_1, ϕ_2 and ϕ_3 compute $+1$ and ϕ_4, ϕ_5 and ϕ_6 compute -1. Combining with the signs of the weights, z_1 will be positive and z_2 will be negative.

Deep Neural Network

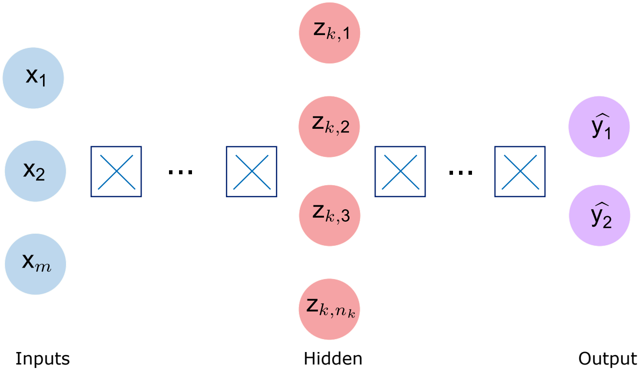


input $\mathbf{x}$ hidden layers $1 \leq l \leq L$ output layer $l = L$

When f is not strictly decomposable into perceptrons, but the decision boundary is smooth (θ), then a multilayer perceptron can be close to implementing f.

Deep Neural Network

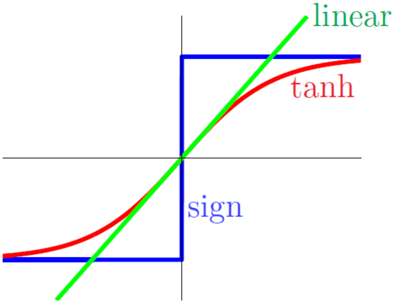
$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{m} x_j^{(k-1)} w_{j,i}^{(k)}$$



Combining Perceptrons

Consider a simple perceptron $z(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$:

- $\theta(z) = \text{sign}(z)$: Learning the weights is hard combinatorial problem (not smooth)
- $\theta(z) = \tanh(z)$: differentiable approximation to sign$(\cdot)$ that allows analytic methods for learning

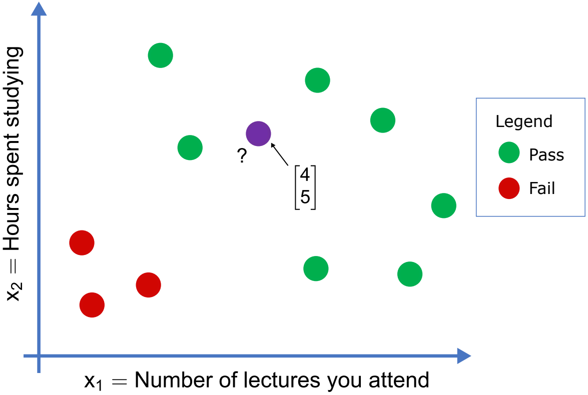


$$\theta(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

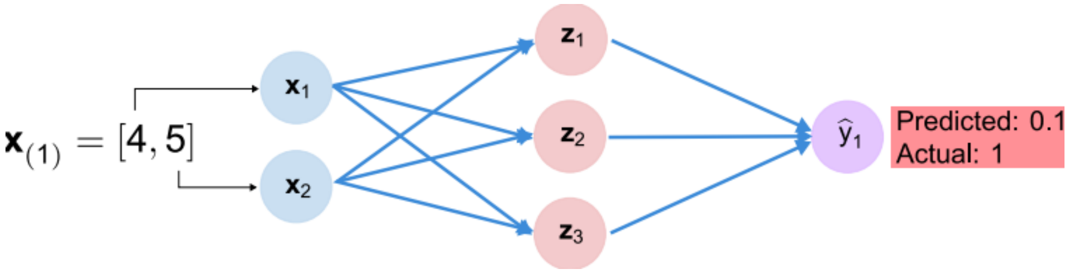
Example problem

Will I pass this class? Let's start with a simple two feature model:

- x_1 Number of lectures you attend
- x_2 Hours spent studying

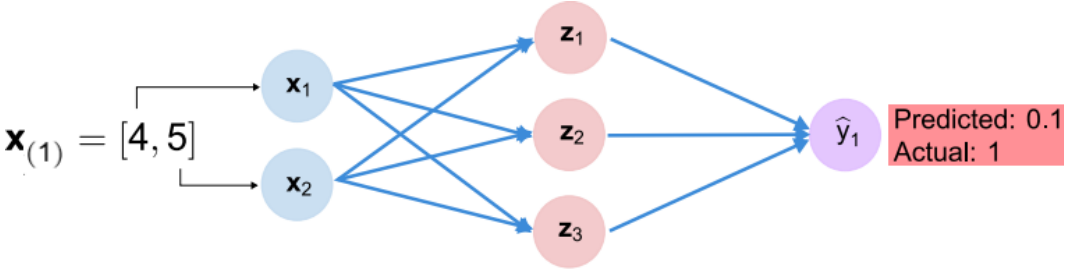


Example problem: Will I pass this class?



Quantifying Loss

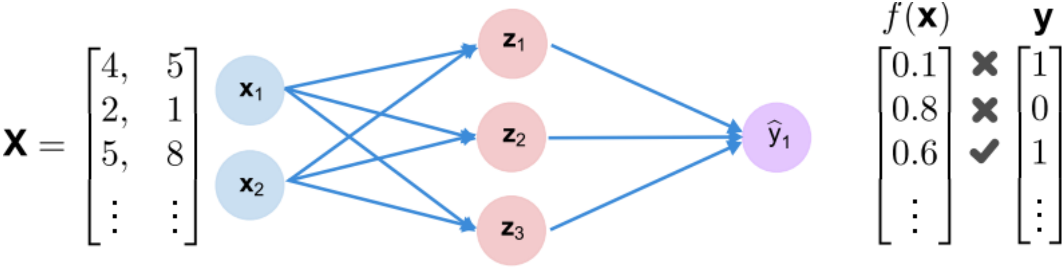
The **loss** of a network measures the cost incurred from incorrect predictions



$$\mathcal{L}(\underbrace{f(\mathbf{x}_{(i)};\mathbf{W})}_{\text{Predicted}}, \underbrace{\mathbf{y}_{(i)}}_{\text{Actual}})$$

Empirical Loss

The **empirical loss** measures the total loss over the entire dataset



Also Known as:

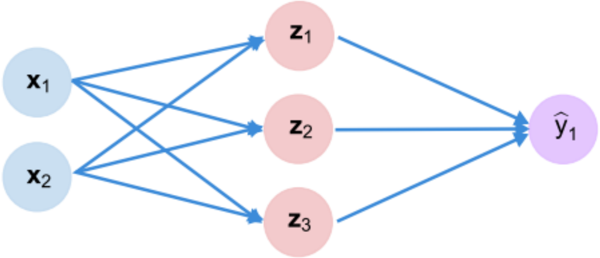
- Objective function
- Cost function
- Empirical Risk

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\underbrace{f(\mathbf{x}_{(i)};\mathbf{W})}_{\text{Predicted}}, \underbrace{\mathbf{y}_{(i)}}_{\text{Actual}})$$

Binary Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1

$$\mathbf{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



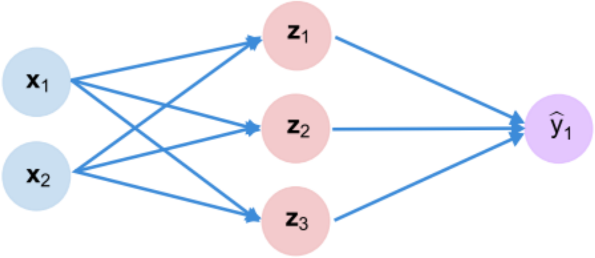
$$\begin{array}{ccc} f(\mathbf{x}) & & \mathbf{y} \\ \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} & \begin{matrix} ✗ \\ ✗ \\ ✓ \\ \end{matrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix} \end{array}$$

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{\mathbf{y}_{(i)}}_{\text{Actual}} \log(\underbrace{f(\mathbf{x}_{(i)};\mathbf{W})}_{\text{Predicted}}) + (1 - \underbrace{\mathbf{y}_{(i)}}_{\text{Actual}}) \log(1 - \underbrace{f(\mathbf{x}_{(i)};\mathbf{W})}_{\text{Predicted}})$$

Mean Square Error Loss

Mean square error loss can be used with models that output a continuous real numbers

$$\mathbf{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$

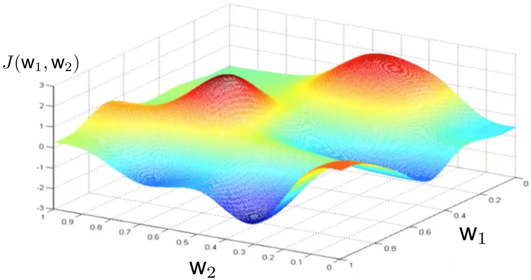


$$f(\mathbf{x}) \quad \begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} \qquad \mathbf{y} \quad \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$$

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^{n} \Big(\underbrace{\mathbf{y}_{(i)}}_{\text{Actual}} - \underbrace{f(\mathbf{x}_{(i)}; \mathbf{W})}_{\text{Predicted}}\Big)^2$$

Training Neural Networks: Loss Optimization

Find the network weights that **achieve the lowest loss**



$$\mathbf{W}^* = \underset{\mathbf{W}}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f(\mathbf{x}_i; \mathbf{W}), \mathrm{y}_{(i)}\right)$$

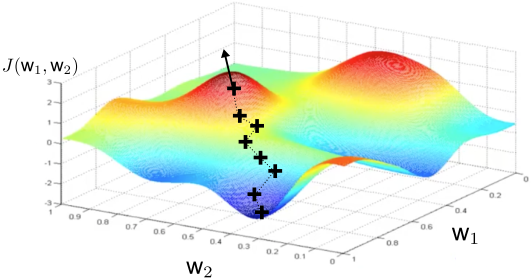
$$\mathbf{W}^* = \underset{\mathbf{W}}{\arg\min} J(\mathbf{W})$$

Remember:

$$\mathbf{W} = \left\{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \ldots\right\}$$

The loss function is a function of the network weights

Training Neural Networks: Loss Optimization

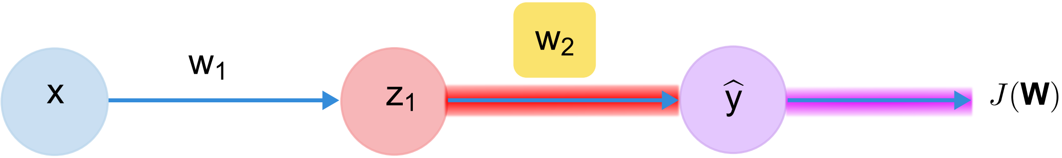


1. Randomly pick an initial (w_1, w_2)
2. Compute the gradient:

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

3. Take a small step in the opposite direction of the gradient
4. Repeat until convergence

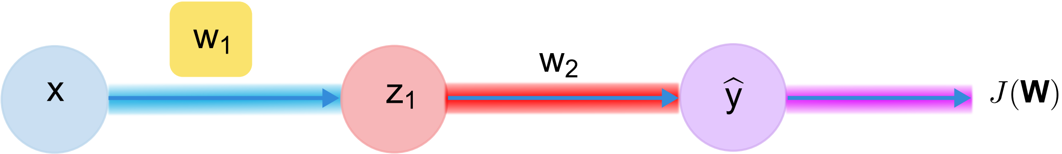
Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \text{Use the chain rule!}$$

$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}\ \underline{\frac{\partial \hat{y}}{\partial w_2}}$$

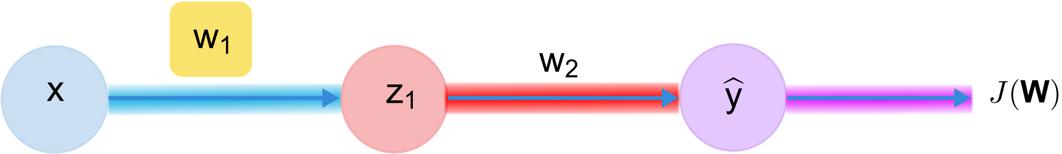
Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1}$$

$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

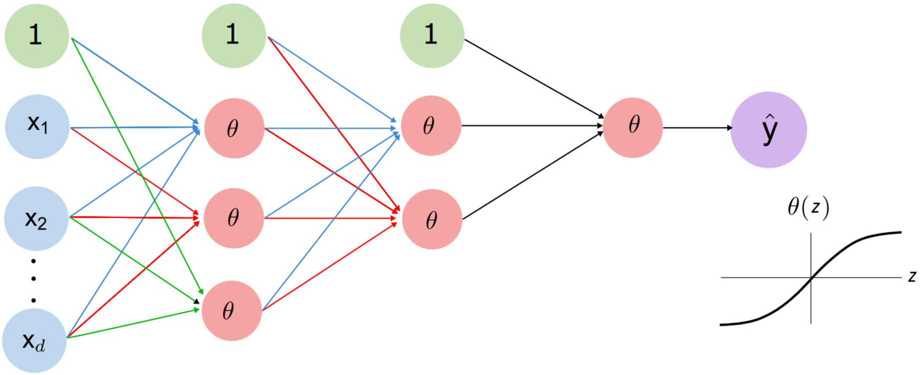
Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial w_1}$$

$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underline{\frac{\partial J(\mathbf{W})}{\partial \widehat{y}}}\ \underline{\frac{\partial \widehat{y}}{\partial z_1}}\ \underline{\frac{\partial z_1}{\partial w_1}}$$

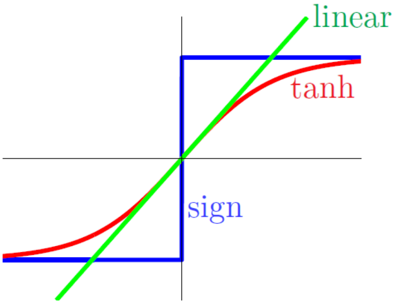
How the network operates



input $\mathbf{x}$ hidden layers $1 \leq l \leq L$ output layer $l = L$

How the network operates

$$\mathrm{w}_{ij}^{(l)} = \begin{cases} 1 \leq l \leq L & \text{Layers} \\ 0 \leq i \leq d^{(l-1)} & \text{Layers} \\ 1 \leq j \leq d^{(l)} & \text{Layers} \end{cases}$$



$$\mathrm{x}_j^{(l)} = \theta(z_j^{(l)}) = \theta(\textstyle\sum_{i=0}^{d^{(l-1)}} \mathrm{w}_{ij}^{l} \mathrm{x}_i^{(l-1)})$$

Apply $\mathbf{x}$ to $\mathrm{x}_1^{(0)} \dots \mathrm{x}_{d^{(0)}}^{(0)} \rightarrow \mathrm{x}_1^{(L)} = h(\mathbf{x})$

$$\theta(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Applying SGD

- All the weights $\mathbf{w} = \left\{ w_{ij}^{(l)} \right\}$ determine $f(\mathbf{x})$
- Error on example $(\mathbf{x}_n, y_n)$ is

$$J(f(\mathbf{x}_n), y_n)) = J(\mathbf{w})$$

- To implement SGD, we need the gradient

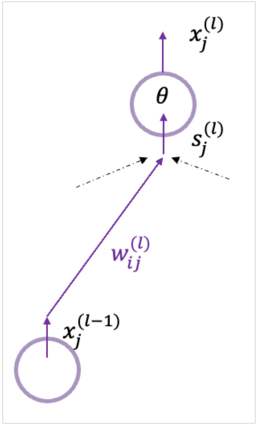
$$\nabla \mathbf{J}(\mathbf{w}) = \frac{\partial J(\mathbf{w})}{\partial w_{ij}^{(l)}}, \quad \text{for all } i, j, l$$

Compute $\nabla J(\mathbf{w})$

- We can evaluate $\nabla \mathbf{J}(\mathbf{w}) = \frac{\partial J(\mathbf{w})}{\partial w_{ij}^{(l)}}$ one by one: analytically or numerically
- A trick for efficient computation:

$$\frac{\partial J(\mathbf{w})}{\partial w_{ij}^{(l)}} = \frac{\partial J(\mathbf{w})}{\partial z_j^{(l)}} \times \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}$$

- We have $\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$
- We only need: $\frac{\partial J(\mathrm{w})}{\partial z_j^{(l)}} = \delta_j^{(l-1)}$



δ for the final layer

For the final layer $l = L$ and $j = 1$:

$$\delta_1^{(L)} = \frac{\partial J(\mathbf{w})}{\partial z_1^{(L)}}$$

$$J(\mathbf{w}) = \left(x_1^{(L)} - y_n\right)^2$$

$$x_1^{(L)} = \theta\left(z_1^{(L)}\right)$$

Since $\theta'(z) = 1 - \theta^2(z)$ for the tanh

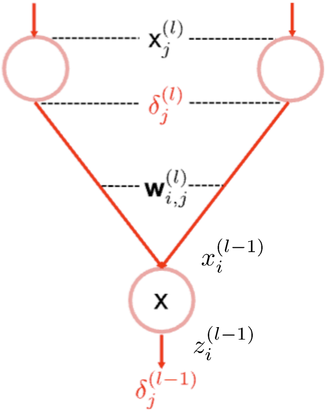
$$\frac{\partial J(\mathbf{w})}{\partial z_1^{(L)}} = 2\left(\theta\left(z_1^{(L)}\right) - y_n\right)\left(1 - \theta^2(z_1^{(L)})\right)$$

$$\frac{\partial J(\mathbf{w})}{\partial z_1^{(L)}} = 2\left(x_1^{(L)} - y_n\right)\left(1 - \left(x_1^{(L)}\right)^2\right)$$

Backpropagation of δ

$$
\begin{aligned}
\delta_i^{(l-1)} &= \frac{\partial J(\mathbf{w})}{\partial z_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \frac{\partial J(\mathbf{w})}{\partial z_j^{(l)}} \times \frac{\partial z_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial z_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'\left(z_i^{(l-1)}\right)
\end{aligned}
$$

$$
\delta_j^{(l-1)} = \left(1 - \left(x_i^{(l-1)}\right)^2\right) \sum_{j=1}^{d^{(l)}} w_{ij}^{(l)} \delta_j^{(l)}
$$



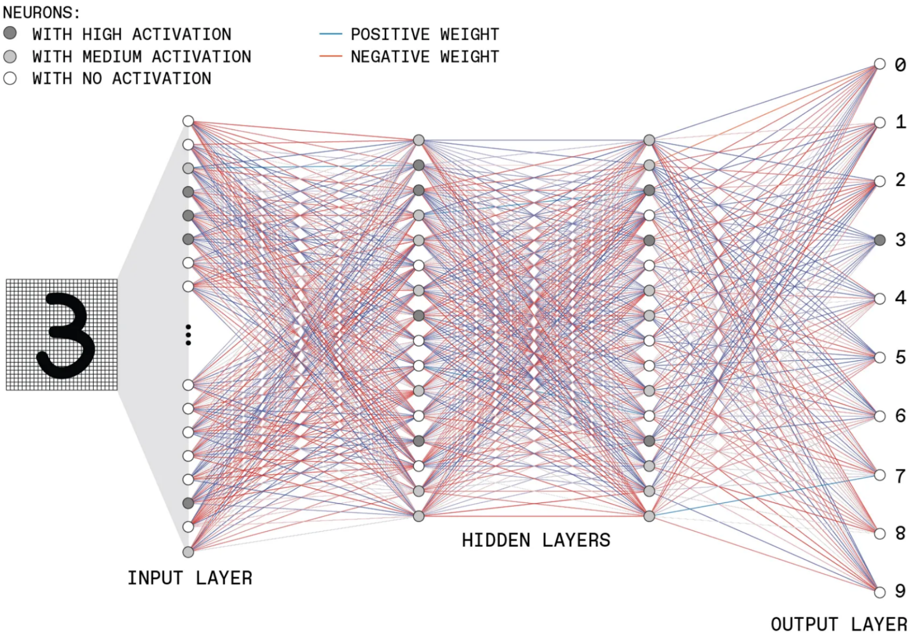
Backpropagation Algorithm

- Initialize all weights $w_{ij}^{(l)}$ at small values chosen **at random**
- Pick one sample $n \in \{1, 2, \cdots, N\}$ uniformly at random
- **Forward Part:** Compute all $x_j^{(l)}$
- **Backward Part:** Compute all $\delta_j^{(l)}$
- Update the weights:

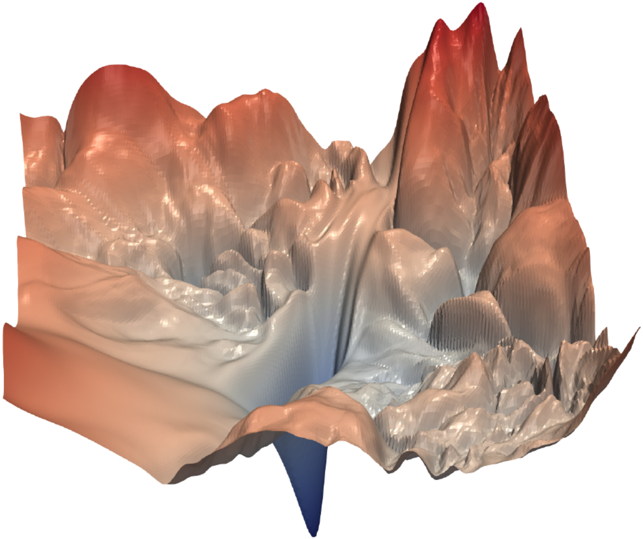
$$
\begin{aligned}
w_{ij}^{(l)} &\leftarrow w_{ij}^{(l)} - \eta \frac{\partial \mathrm{J}(\mathbf{w})}{\partial\, w_{ij}^{(l)}} \\
w_{ij}^{(l)} &\leftarrow w_{ij}^{(l)} - \eta x_i^{(l-1)} \delta_j^{(l)}
\end{aligned}
$$

- Iterate until it is time to stop.

When is the best time to stop?
Considering a marginal error improvement, a maximum error and also a maximum number of iterations is a good practice.


NEURONS:
WITH HIGH ACTIVATION
WITH MEDIUM ACTIVATION
WITH NO ACTIVATION
POSITIVE WEIGHT
NEGATIVE WEIGHT
0
1
2
3
4
5
6
7
8
9
INPUT LAYER
HIDDEN LAYERS
OUTPUT LAYER

Training Neural Networks



Li, H. *et al.* Visualizing the loss landscape of neural nets (2017).

Loss Functions Can Be Difficult to Optimize

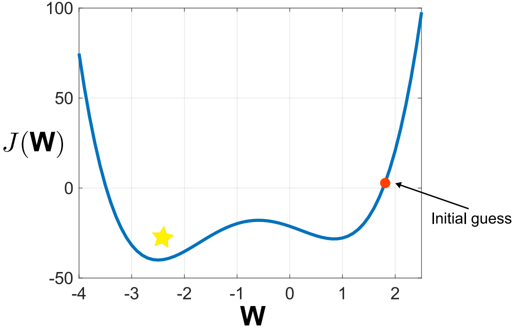
Remember: Optimization through gradient descent:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

$\eta \rightarrow$ How can we set the learning rate?

Training Neural Networks: Loss Optimization

- **Small learning rate** converges slowly and gets stuck in false local minima
- **Large learning rates** overshoot, become unstable and diverge
- **Stable learning rates** converge smoothly and avoid local minima



How to select the learning rate

Idea 1:

Try a lots of different learning rates and see what works "just right"

Idea 2:

Do something smarter!

Design an adaptive learning rate that "adapts" to the landscape

Adaptive Learning Rates

- Learning Rates are not longer fixed
- Can be made larger or smaller depending on:
  - how large the gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

Neural Networks in Practice: Mini-batches

Gradient Descent:
Compute:

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} \rightarrow \text{ Can be very } \textbf{computational intensive} \text{ to compute!}$$

Stochastic Gradient Descent:
Pick a single point i and compute:

$$\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}} \rightarrow \text{Easy to compute but } \textbf{very noisy} \text{ (stochastic)!}$$

Neural Networks in Practice: Mini-batches

Mini-batches:

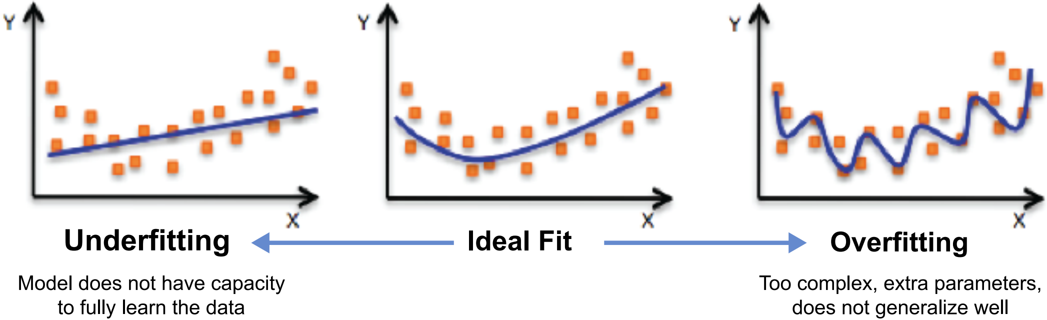
Pick B pints and compute:

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$$

- More accurate estimation of the gradient
  - Smoother convergence
  - Allows for large learning rates
- Lead to fast training
  - Can parallelize
  - Achieve significant speed increases on GPU's

Neural Networks in Practice: Overfitting

The overfitting problem:



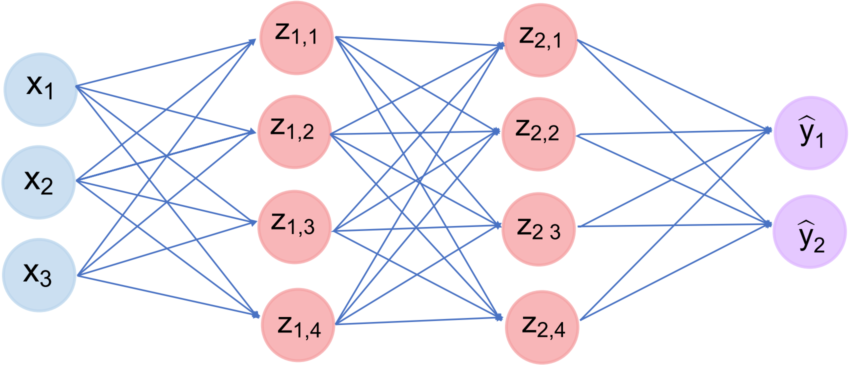
Neural Networks in Practice: Regularization

What is it?
Technique that constrains our optimization problem to discourage complex models

Why we need it?
Improve generalization of our models on unseen data

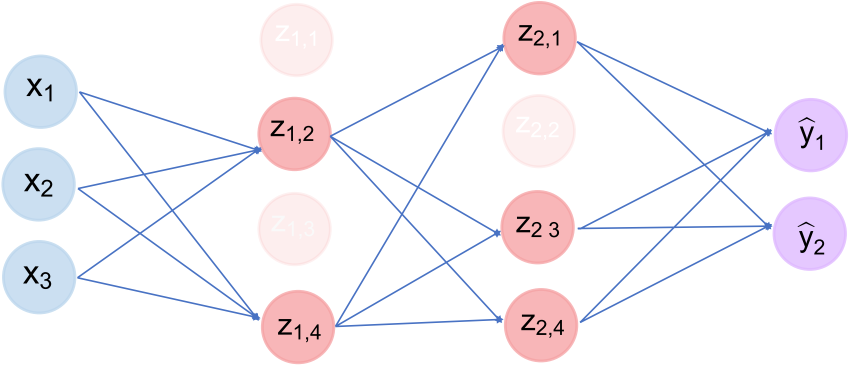
Neural Networks in Practice: Regularization 1

Dropout:
During training, at each iteration, randomly set some activations to 0



Neural Networks in Practice: Regularization 1

Dropout:
During training, at each iteration, randomly set some activations to 0

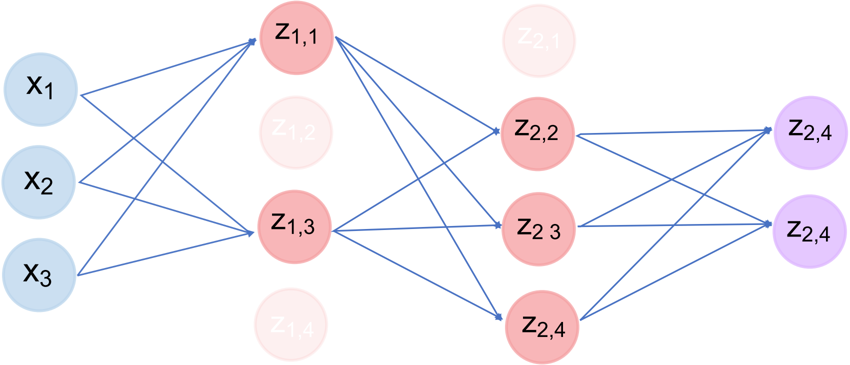


Neural Networks in Practice: Regularization 1

Dropout:

During training, at each iteration, randomly set some activations to 0

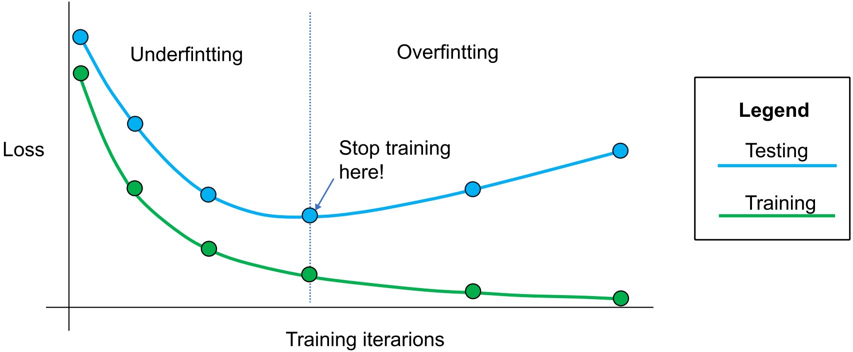
- Typically "drop" 50% of activations in layer
- Forces network to not rely on any node



Neural Networks in Practice: Regularization 2

Early Stopping

Stop training before we have a chance to overfit



Working with Real Data Sets: The Iris Data set

$\mathbf{x}$ is a $150 \times 4 \rightarrow$ it has 4 features and we have 150 samples

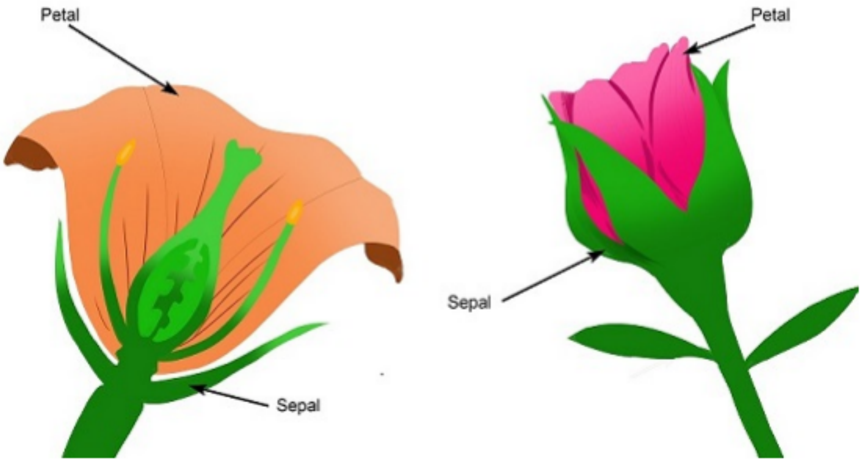
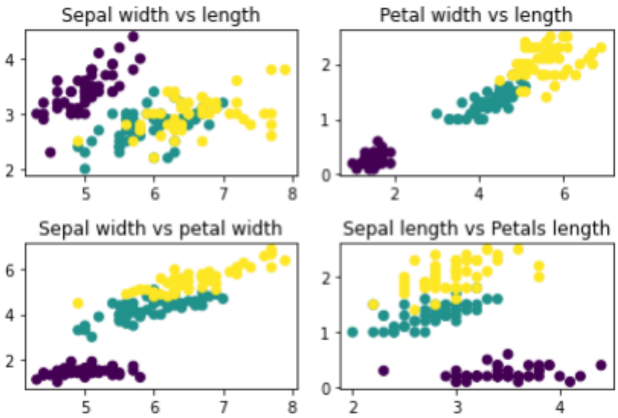


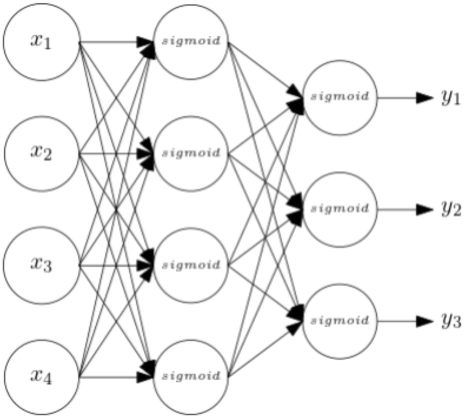
Image Features

1. Width
   - Petal
   - Sepal

2. Length
   - Petal
   - Sepal

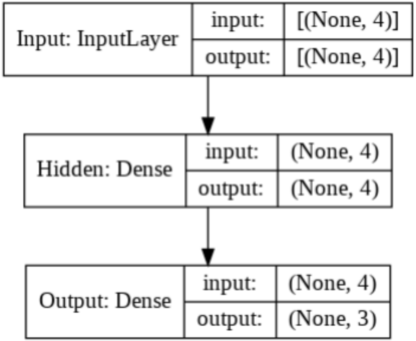


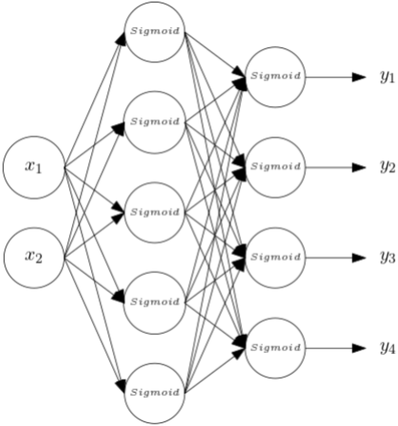
Approach



Generate fully connected layers with multiple activation functions to solve a multi feature classification problem.

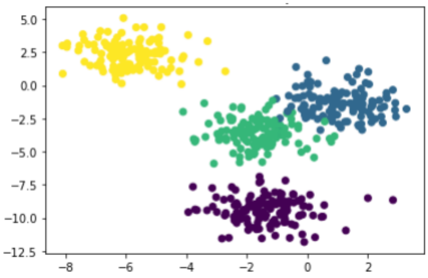
An efficient coding method: Keras

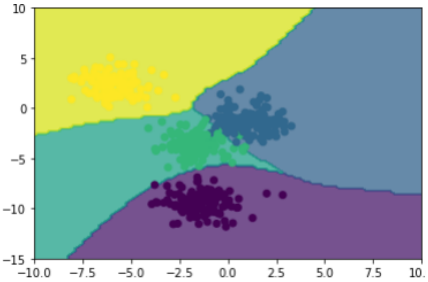




Solutions

Accurate boundary lines for Classification





Code

```
input = keras.layers.Input(4,name="Input")
hidden = keras.layers.Dense(4,activation='sigmoid',name="Hidden")(input)
output = keras.layers.Dense(3,activation='sigmoid', name="Output")(hidden)

modelIRIS = keras.Model(input,output)

modelIRIS.layers[1].set_weights([w1,b1])
modelIRIS.layers[2].set_weights([w2,b2])

keras.utils.plot_model(modelIRIS,show_shapes=True)

# print(modelIRIS.predict(np.array([0,1,1,2],ndmin=2)))
```

```
X,y = datasets.make_blobs(centers = 4,random_state=2,n_samples = 500)

plt.scatter(X[:,0],X[:,1],c=y)
plt.title("Hand-made Example")
plt.show()
```

```
w1 = np.array([[-1.5132886 ,  0.78692156, -0.59150934,  1.3457433 ,  0.11323632],
        [-0.9712865 , -1.5561647 , -1.578427  , -0.7123345 ,  0.75948274]])
b1 = np.array([-2.8332472,  2.0457697, -3.736685 , -0.8151084,  4.7346272])

w2 = np.array([[ 1.3780868 , -4.0084467 ,  0.0883714 ,  2.5190408 ],
        [ 0.37726903,  0.76979834,  1.5470964 , -5.40783   ],
        [ 1.3585504 , -1.799717  ,  2.6277127 , -1.8256785 ],
        [ 2.7409966 ,  1.6214521 , -2.7639852 , -3.9925196 ],
        [-5.5055437 ,  2.8043787 ,  1.1856915 ,  3.2509704 ]])
b2 = np.array([-0.54020935,  0.57120425, -0.5757609 ,  0.8131391 ])
```

Acknowledgement

Alexander Amini and Ava Soleimanym, MIT 6.S191: Introduction to Deep Learning, IntroToDeepLearning.com