DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 8 | Part 1

Recap

Where have we been?

- We started with **nearest neighbor rules**.
  - Capable of learning **non-linear patterns**.
  - **Did not learn** feature importance.
  - Computationally-expensive.
    - $\Theta(n)$ memory and prediction time.

Where have we been?

- In response, we developed **empirical risk minimization** (ERM).
- Step 1: choose a **hypothesis class**
- Step 2: choose a **loss function**
- Step 3: minimize **expected loss (empirical risk)**

Where have we been?

- For first hypothesis class, we chose **linear models**.

$$H(\vec{x}) = w_0 + w_1 x_1 + \ldots + w_d x_d$$

- $\Theta(d)$ memory and prediction time.

Where have we been?

- We saw different loss functions:
  - square, absolute, perceptron, hinge

- To train a linear model, pick loss L and minimize risk:

$$\arg\min_{\vec{w}} R(\vec{w}) = \arg\min_{\vec{w}} \frac{1}{n} \left[\sum_{i=1}^{n} L(\vec{x}^{(i)}, y_i, \vec{w}) \right]$$

Where have we been?

- We saw how to control the complexity of the learned model with **regularization**.

$$\arg\min_{\vec{w}} \tilde{R}(\vec{w}) = \arg\min_{\vec{w}} \frac{1}{n}\left[\sum_{i=1}^{n} L(\vec{x}^{(i)}, y_i, \vec{w})\right] + \rho(\vec{w})$$

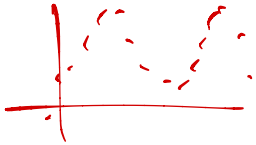
Where have we been?

- Some ERM problems have direct solutions.
  - Least squares, ridge regression.

- We saw most others do not, and must be solved iteratively with, e.g., **(stochastic) (sub)gradient descent**.

Linear Model Zoo

Name	Loss Function	Regularizer	Direct Solution
Least Squares	square	-	yes
Ridge Regression	square	$\|\vec{w}\|^2$	yes
LASSO	square	$\|\vec{w}\|_1$	no
Perceptron	perceptron	-	no
Soft-SVM	hinge	$\|\vec{w}\|^2$	no



Non-Linear Patterns

- We saw two ways of learning non-linear patterns with linear models:

1. Explicit mapping to feature space with **basis functions**.
   - E.g., learn $H(\vec{x}) = w_0 + w_1\phi_1(\vec{x}) + \ldots + w_k\phi_k(\vec{x})$
2. Implicit mapping with **kernel methods**.

- Each has downsides.

Basis Functions

- **Idea:** choose a mapping $\vec{\phi}$ that transforms data; train linear model in feature space.

- Downsides:
  - Must choose a good mapping. How?
  - Feature space is often very high-dimensional (**costly**).

Kernels

- **Idea:** implicitly map to high-dimensional space with **kernel trick**.

- Downsides:
  - Since prediction is sum over training points, $\Theta(n)$ in memory and time

Where are we now?

- A new hypothesis class, beyond linear models.

190 / 140B

DSC 140A

Probabilistic Modeling & Machine Learning

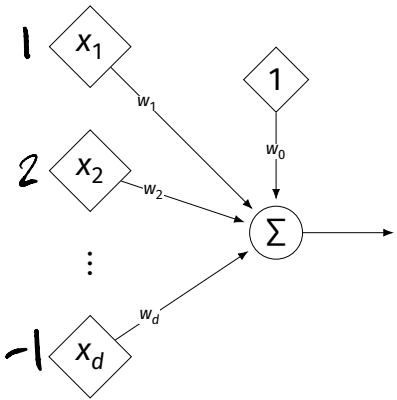
Lecture 8 | Part 2

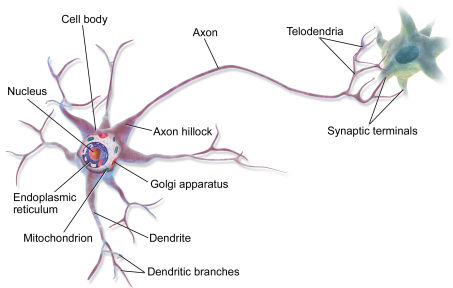
Neural Networks

Linear Models

$\vec{x} = (1, 2, -1)$

$$H(\vec{x}) = w_0 + w_1 x_1 + \ldots + w_d x_d$$



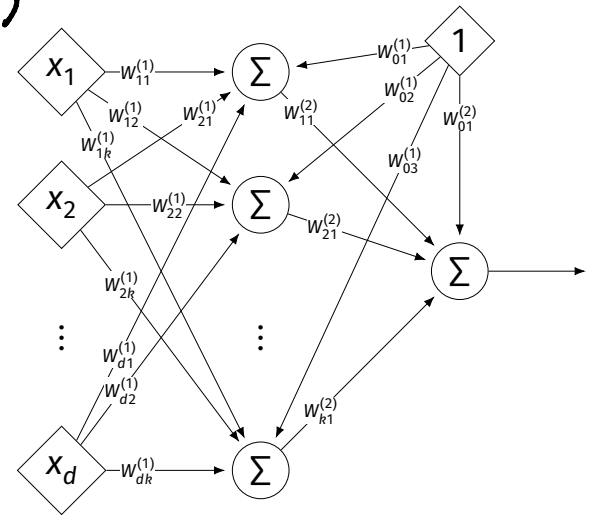


Generalizing Linear Models

- The brain is a **network** of neurons.
- The output of a neuron is used as an input to another.
- **Idea:** chain together multiple “neurons” into a **neural network**.

Neural Network[1] (One Hidden Layer)

$\vec{x} = (x_1, \ldots, x_d)^T$



[1]Specifically, a fully-connected, feed-forward neural network

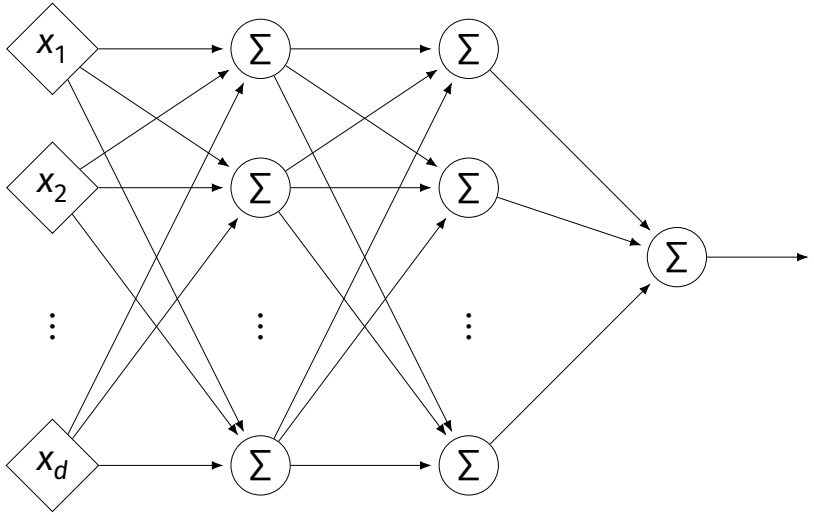
Architecture

- Neurons are organized into **layers**.
  - **Input layer**, **output layer**, and **hidden layers**.
- Number of cells in input layer determined by dimensionality of input feature vectors.
- Number of cells in hidden layer(s) is determined by you.
- Output layer can have >1 neuron.

Architecture

- Can have more than one hidden layer.
  - A network is “**deep**” if it has >1 hidden layer.
- Hidden layers can have different number of neurons.

Neural Network (Two Hidden Layers)



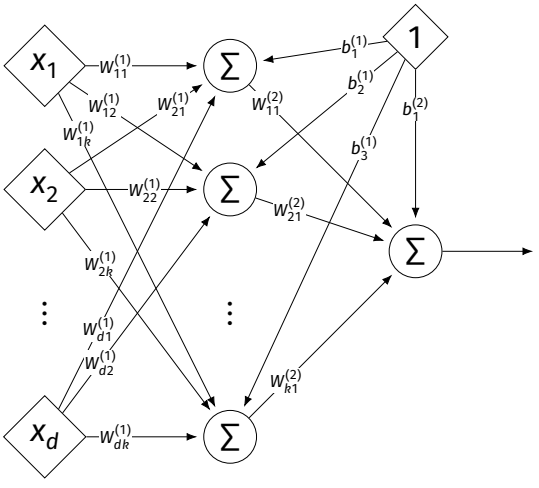
$H(\vec{x}) = \vec{w} \cdot A_{\text{NN}}(\vec{x})$

Network Weights

- A neural network is a type of function.
- Like a linear model, a NN is **totally determined** by its weights.
- But there are often many more weights to learn!

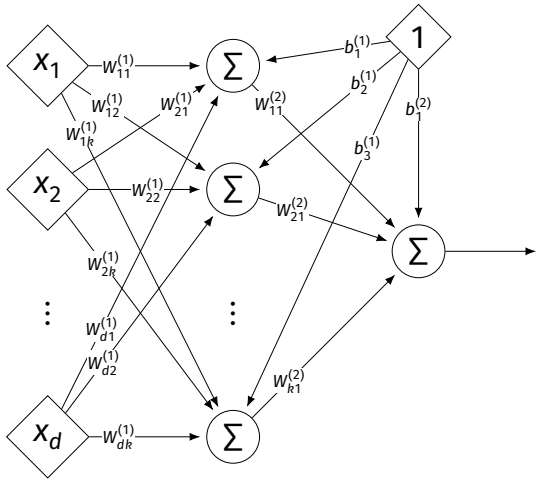
Notation

- Input is layer #0.
- $W_{jk}^{(i)}$ denotes weight of connection between neuron j in layer $(i - 1)$ and neuron k in layer i
- Layer weights are 2-d arrays.



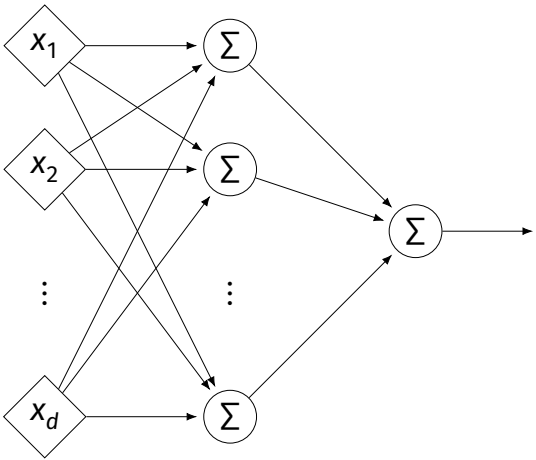
Notation

- Each hidden/output neuron gets a "dummy" input of 1.
- jth node in ith layer assigned a bias weight of $b_j^{(i)}$
- Biases for layer are a vector: $\vec{b}^{(i)}$

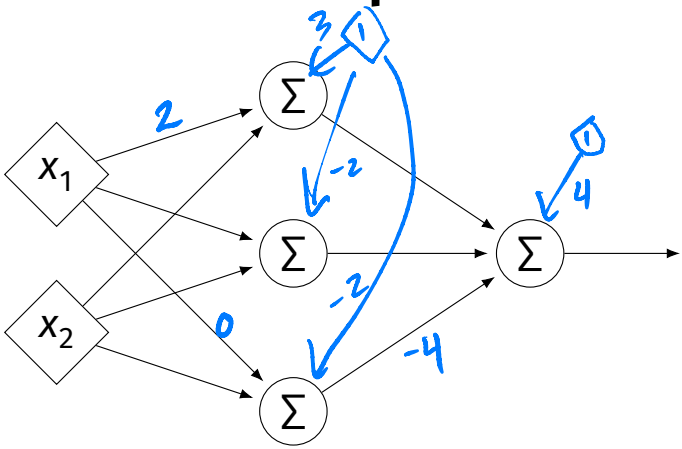


Notation

- Typically, we will not draw the weights.
- We will not draw the dummy input, too, but it is there.



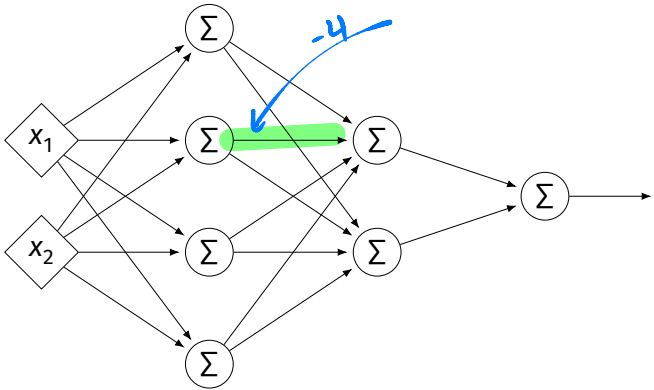
Example



$$W^{(1)} = \begin{pmatrix} 2 & -1 & 0 \\ 4 & 5 & 2 \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} 3 \\ 2 \\ -4 \end{pmatrix}$$

$$\vec{b}^{(1)} = (3, -2, -2)^T \quad \vec{b}^{(2)} = (-4)^T$$

Example



$$W^{(1)} = \begin{pmatrix} 2 & -1 & -3 & 0 \\ 4 & 5 & -7 & 2 \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} 1 & 2 \\ -4 & 3 \\ -6 & -2 \\ 3 & 4 \end{pmatrix} \quad W^{(3)} = \begin{pmatrix} -1 & 5 \end{pmatrix}$$

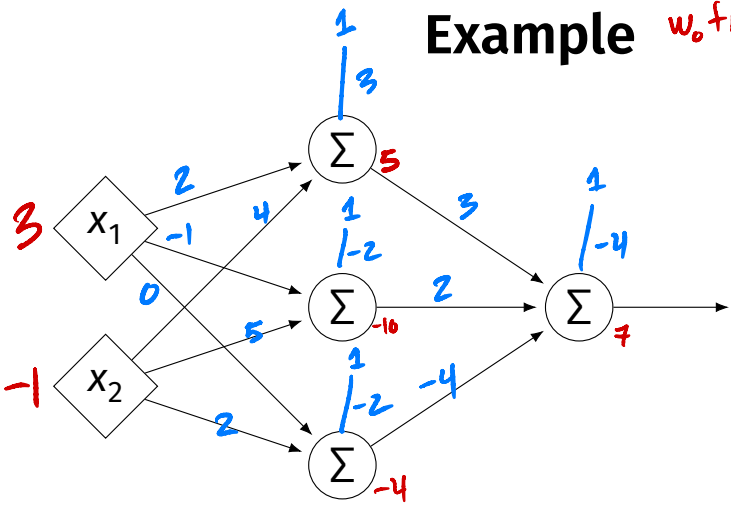
$$\vec{b}^{(1)} = (3, 6, -2, -2)^T \quad \vec{b}^{(2)} = (-4, 0)^T \quad \vec{b}^{(3)} = (1)^T$$

Evaluation

- These are "**fully-connected, feed-forward**" networks with one output.
- They are functions $H(\vec{x}) : \mathbb{R}^d \to \mathbb{R}^1$
- To evaluate $H(\vec{x})$, compute result of layer i, use as inputs for layer $i + 1$.

Example

$w_0 + w_1 x_1 + w_2 x_2$



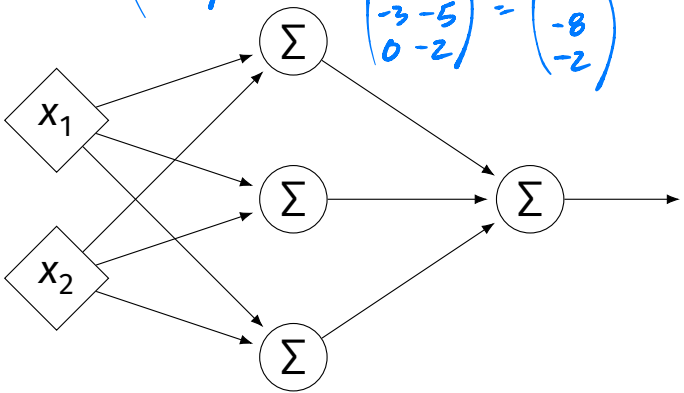
- $\vec{x} = (3, -1)^T$
- $z_1^{(1)} = 3 + (2)(3) + (4)(-1)$ $3 + 6 - 4 = 5$
- $z_2^{(1)} = -2 + (3)(-1) + (-1)(5)$ $-2 - 3 - 5 = -10$
- $z_3^{(1)} = -4$
- $z_1^{(2)} = -4 + (5)(3) + (-10)(2) + (-4)(-4)$ $= -4 + 15 - 20 + 16 = 7$

$$W^{(1)} = \begin{pmatrix} 2 & -1 & 0 \\ 4 & 5 & 2 \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} 3 \\ 2 \\ -4 \end{pmatrix} \quad \vec{b}^{(1)} = (3, -2, -2)^T \quad \vec{b}^{(2)} = (-4)^T$$

Evaluation as Matrix Multiplication

- Let $z_j^{(i)}$ be the output of node j in layer i.
- Make a vector of these outputs: $\vec{z}^{(i)} = (z_1^{(i)}, z_2^{(i)}, \ldots)^T$
- Observe that $\vec{z}^{(i)} = \left[W^{(i)}\right]^T \vec{z}^{(i-1)} + \vec{b}^{(i)}$

Example



- $\vec{x} = (3, -1)^T$ =
- $z_1^{(1)}$ =
- $z_2^{(1)}$ =
- $z_3^{(1)}$ =
- $z_1^{(2)}$ =

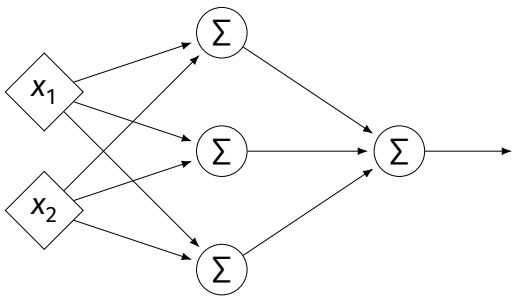
$W^{(1)} = \begin{pmatrix} 2 & -1 & 0 \\ 4 & 5 & 2 \end{pmatrix}$ $W^{(2)} = \begin{pmatrix} 3 \\ 2 \\ -4 \end{pmatrix}$ $\vec{b}^{(1)} = (3, -2, -2)^T$ $\vec{b}^{(2)} = (-4)^T$

Each Layer is a Function

- We can think of each layer as a function mapping a vector to a vector.

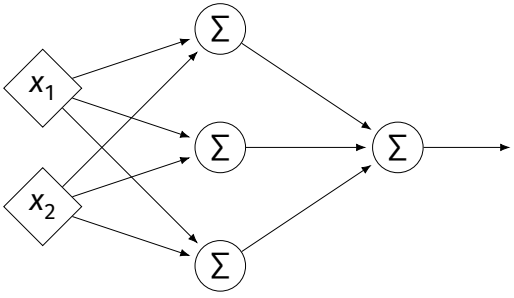
- $H^{(1)}(\vec{z}) = \left[W^{(1)}\right]^T \vec{z} + \vec{b}^{(1)}$
  - $H^{(1)} : \mathbb{R}^2 \to \mathbb{R}^3$

- $H^{(2)}(\vec{z}) = \left[W^{(2)}\right]^T \vec{z} + \vec{b}^{(2)}$
  - $H^{(2)} : \mathbb{R}^3 \to \mathbb{R}^1$



NNs as Function Composition

- The full NN is a composition of layer functions.



$$H(\vec{x}) = H^{(2)}(H^{(1)}(\vec{x})) = \left[W^{(2)}\right]^T \underbrace{\left(\left[W^{(1)}\right]^T \vec{x} + \vec{b}^{(1)}\right)}_{\vec{z}^{(1)}} + \vec{b}^{(2)}$$

NNs as Function Composition

- In general, if there k hidden layers:

$$H(\vec{x}) = H^{(k+1)}\left(\cdots H^{(3)}\left(H^{(2)}\left(H^{(1)}(\vec{x})\right)\right)\cdots\right)$$

$W^{(1)} = 2 \times 3 \quad W^{(2)} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$

Exercise

Show that:

$$H(\vec{x}) = \left[W^{(2)}\right]^T \left(\left[W^{(1)}\right]^T \vec{x} + \vec{b}^{(1)}\right) + \vec{b}^{(2)} = \vec{w} \cdot \text{Aug}(\vec{x})$$

for some appropriately-defined vector $\vec{w}$.

$\left[W^{(2)}\right]^T \left[W^{(1)}\right]^T \vec{x} + \left[W^{(2)}\right] \vec{b}^{(1)} + \vec{b}^{(2)}$

$1\times3 \quad 3\times2 \quad 2\times1 \quad 1\times3 \quad 3\times1 \quad 1$

1×2

Result

- The composition of linear functions is again a linear function.
- The NNs we have seen so far are all equivalent to linear models!
- For NNs to be more useful, we will need to add **non-linearity**.

Activations

- So far, the output of a neuron has been a linear function of its inputs:

$$g(w_0 + w_1 x_1 + w_2 x_2 + \ldots)$$

- Can be arbitrarily large or small.

- But real neurons are **activated** non-linearly.
  - E.g., saturation.

Idea

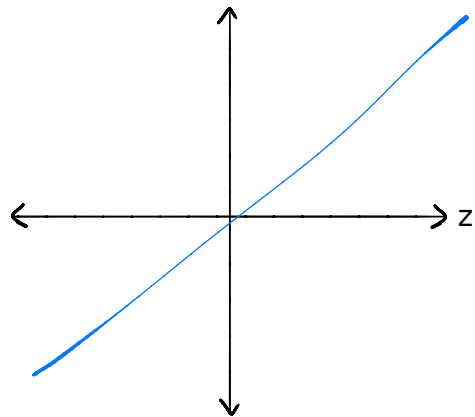
- To add nonlinearity, we will apply a non-linear **activation function** g to the output of **each** hidden neuron (and sometimes the output neuron).

Linear Activation

- The **linear** activation is what we've been using.

$z = w_0 + w_1 x_1 + \ldots$

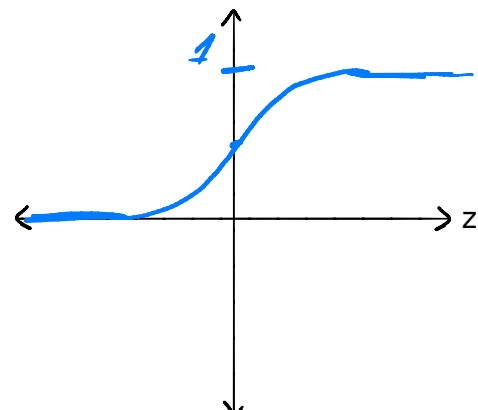
$\sigma(z) = z$



Sigmoid Activation

- The **sigmoid** models saturation in many natural processes.

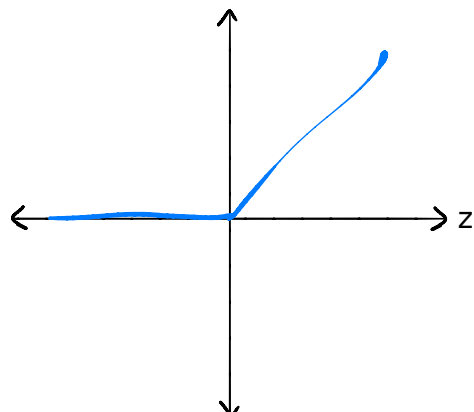
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



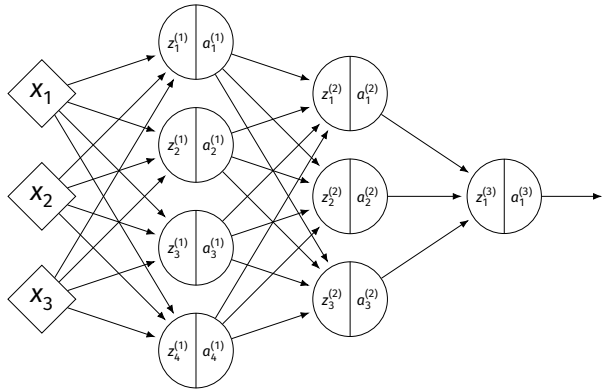
ReLU Activation

- The **Rectified Linear Unit (ReLU)** tends to work better in practice.

$g(z) = \max\{0, z\}$

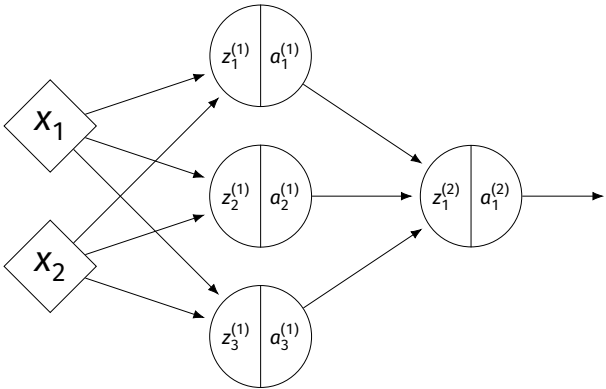


Notation



- $z_j^{(i)}$ is the linear activation before g is applied.
- $a_j^{(i)} = g(z^{(i)})$ is the actual output of the neuron.

Example



- g = ReLU
- Linear output
- $\vec{x} = (3, -1)^T$
- $z_1^{(1)} =$
- $a_1^{(1)} =$
- $z_2^{(1)} =$
- $a_2^{(1)} =$
- $z_3^{(1)} =$
- $a_3^{(1)} =$
- $z_1^{(2)} =$

$$W^{(1)} = \begin{pmatrix} 2 & -1 & 0 \\ 4 & 5 & 2 \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} 3 \\ 2 \\ -4 \end{pmatrix} \quad \vec{b}^{(1)} = (3, -2, -2)^T \quad \vec{b}^{(2)} = (-4)^T$$

Output Activations

- The activation of the output neuron(s) can be different than the activation of the hidden neurons.
- In classification, **sigmoid** activation makes sense.
- In regression, **linear** activation makes sense.

Main Idea

A neural network with linear activations is a linear model. If non-linear activations are used, the model is made non-linear.

DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 8 | Part 3

Training Neural Networks

Training Neural Networks

- As with linear models, we can use ERM.
- Step 1: choose a **hypothesis class**
  - Choose a neural network architecture (depth / width), activation.
- Step 2: choose a **loss function**
- Step 3: minimize **expected loss (empirical risk)**

Parameter Vectors

- A neural network is totally determined by its parameters.
- We can package all of the parameter arrays $W^{(1)}, W^{(2)}, \ldots$, as well as the biases $\vec{b}^{(1)}, \vec{b}^{(2)}, \ldots$ into a single **parameter vector** $\vec{w}$.

Square Loss for NNs

- The square loss can be used to train a neural network.

$$L(\vec{x}, y, \vec{w}) = (H(\vec{x}; \vec{w}) - y)^2$$

Cross-Entropy Loss for NNs

- When using sigmoid output activation for classification, we often use the **cross-entropy**.

- Assume labels are 1 and 0. Then:

$$L(\vec{x}, y, \vec{w}) = -\begin{cases} \log H(\vec{x}; \vec{w}), & \text{if } y = 1 \\ \log\left[1 - H(\vec{x}; \vec{w})\right], & \text{if } y = 0 \end{cases}$$

Minimizing Risk

- Having chosen a loss, we next minimize empirical risk:[2]

$$\arg\min_{\vec{w}} R(\vec{w}) = \arg\min_{\vec{w}} \frac{1}{n}\left[\sum_{i=1}^{n} L(\vec{x}^{(i)}, y_i, \vec{w})\right]$$

- Except for special cases, there is no direct solution for the minimizer.

- Use iterative methods: e.g., SGD.

[2] Can also add a regularizer.

Gradient Descent for NNs

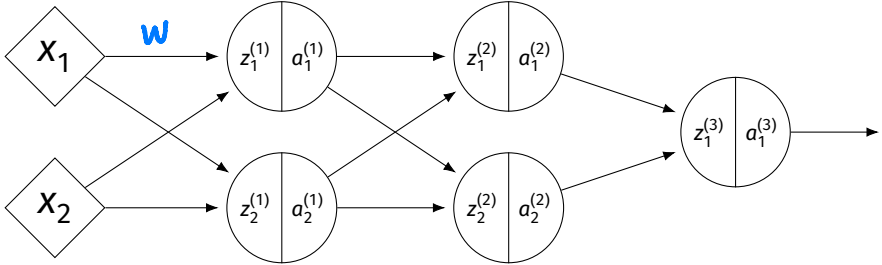
- To perform SGD, we must compute ∇R.
- E.g., using square loss:

$$\nabla R = \frac{2}{n} \sum_{i=1}^{n} (H(\vec{x}^{(i)}; \vec{w}) - y_i) \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})$$

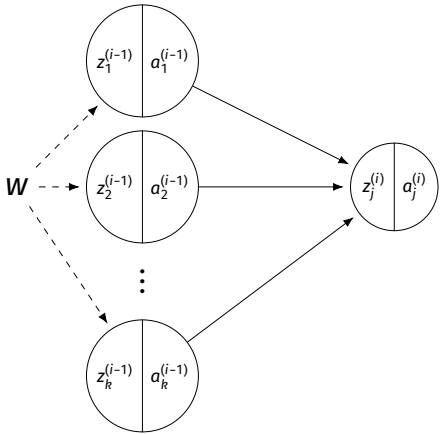
- We must compute $\nabla_{\vec{w}} H$; the gradient of the network with respect to the parameter vector, $\vec{w}$.

Gradient of a Network

- The gradient of H w.r.t., parameter vector $\vec{w}$ can be computed using the chain rule.



Example



$$z_j^{(i)} = b_j^{(i)} + \sum_{\ell=1}^{k} W_{\ell j}^{(i)} a_\ell^{(i-1)}$$

$$\begin{aligned}
\frac{\partial a_j^{(i)}}{\partial w} &= \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial z_j^{(i)}}{\partial w} \\
&= \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \left[\sum_{\ell=1}^{k} \frac{\partial z_j}{\partial a_\ell^{(i-1)}} \right] \\
&= \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \left[\sum_{\ell=1}^{k} W_{\ell j}^{(i)} \frac{\partial a_\ell^{(i-1)}}{\partial w} \right]
\end{aligned}$$

Gradient of a Network

- We found:

$$\frac{\partial a_j^{(i)}}{\partial w} = \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \left[\sum_{\ell=1}^{k} W_{\ell j}^{(i)} \frac{\partial a_\ell^{(i-1)}}{\partial w} \right]$$

- Recalling that $a_j^{(i)} = g(z_j^{(i)})$, we can simplify a little:

$$\frac{\partial a_j^{(i)}}{\partial w} = g'(z_j^{(i)}) \left[\sum_{\ell=1}^{k} W_{\ell j}^{(i)} \frac{\partial a_\ell^{(i-1)}}{\partial w} \right]$$

Gradient of a Network

$$\frac{\partial a_j^{(i)}}{\partial w} = g'(z_j^{(i)}) \left[\sum_{\ell=1}^{k} W_{\ell j}^{(i)} \frac{\partial a_\ell^{(i-1)}}{\partial w} \right]$$

- We can apply the above formula recursively to compute $\partial H / \partial w$ for any parameter w.
- Efficient algorithm: **backpropagation**.
- Outside of the scope of DSC 140A (take DSC 140B).

Implications

$$\frac{\partial a_j^{(i)}}{\partial w} = g'(z_j^{(i)}) \left[\sum_{\ell=1}^{k} W_{\ell j}^{(i)} \frac{\partial a_\ell^{(i-1)}}{\partial w} \right]$$

- **Vanishing gradients**: the deeper the network, the "weaker" the gradients; harder to train.
- Prefer activations with stronger gradients.
  - ReLU > sigmoid

Convex Risk?

- When training linear models with convex losses, the risk was a convex function of $\vec{w}$.
  - Because it is composed of convex functions.

- E.g., with the square loss:

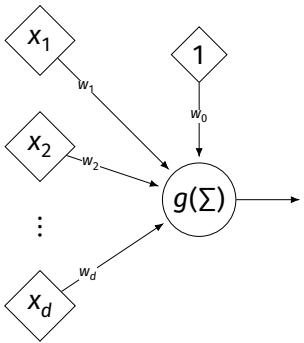
$$\frac{1}{n}\sum_{i=1}^{n}\left(\vec{w}\cdot\vec{x}^{(i)} - y_i\right)^2$$

- We like this because it was **easy** to optimize.

Convex Risk?

- What about with NNs? Is the risk still convex?

- Consider a very simple network with zero hidden layers, representing $H(\vec{x}; \vec{w}) = g(\vec{w} \cdot \text{Aug}(\vec{x}))$.



Convex Risk?

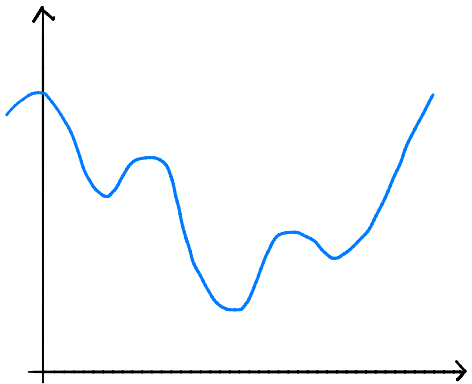
- The risk w.r.t. the square loss is:

$$\frac{1}{n}\sum_{i=1}^{n}\left(g(\vec{w}\cdot \text{Aug}(\vec{x}^{(i)})) - y_i\right)^2$$

- If g is non-convex, the risk is in general no longer convex!

Non-Convexity

- In general, NNs with non-linear activations have (highly) **non-convex** risk functions.



Training NNs

- SGD is still used to train neural networks, even though the risk is **non-convex**.
- May get stuck in local minima; depends heavily on starting position.

Downsides of NNs

- NNs tend to be harder to train than linear models.
- How do we choose the architecture?
- Engineering challenges with large networks.

DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 8 | Part 4

Demo

Feature Map

- We have seen how to fit non-linear patterns with linear models via **basis functions** (i.e., a feature map).

$$H(\vec{x}) = w_0 + w_1\phi_1(\vec{x}) + \ldots + w_k\phi_k(\vec{x})$$

- These basis functions are fixed **before** learning.

- **Downside:** we have to choose $\vec{\phi}$ somehow.

Learning a Feature Map

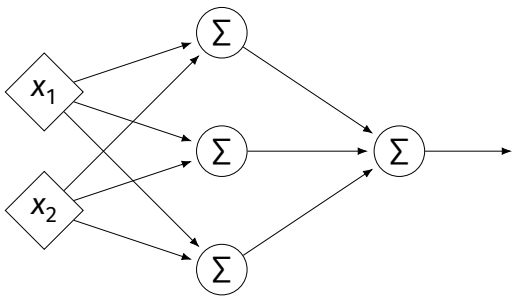
- **Interpretation:** The hidden layers of a neural network **learn** a feature map.

Each Layer is a Function

- We can think of each layer as a function mapping a vector to a vector.

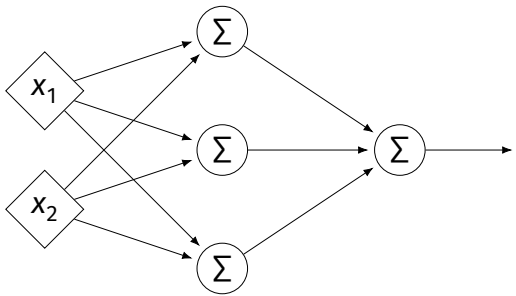
- $H^{(1)}(\vec{z}) = \left[W^{(1)}\right]^T \vec{z} + \vec{b}^{(1)}$
  - $H^{(1)} : \mathbb{R}^2 \to \mathbb{R}^3$

- $H^{(2)}(\vec{z}) = \left[W^{(2)}\right]^T \vec{z} + \vec{b}^{(2)}$
  - $H^{(2)} : \mathbb{R}^3 \to \mathbb{R}^1$



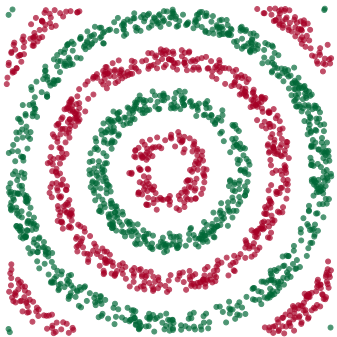
Each Layer is a Function

- The hidden layer performs a feature map from $\mathbb{R}^2$ to $\mathbb{R}^3$.
- The output layer makes a prediction in $\mathbb{R}^3$.
- **Intuition:** The feature map is learned so as to make the output layer's job "easier".



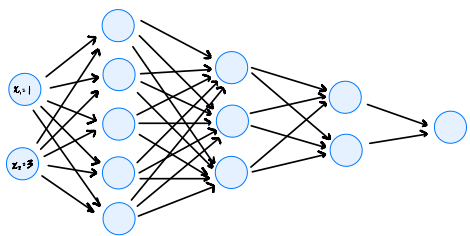
Demo

- Train a deep network to classify the data below.
- Hidden layers will learn a new feature map that makes the data linearly separable.

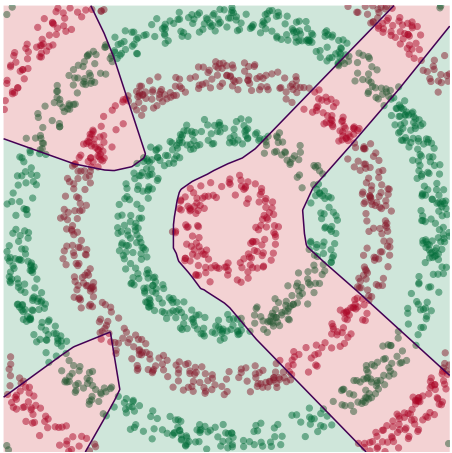


Demo

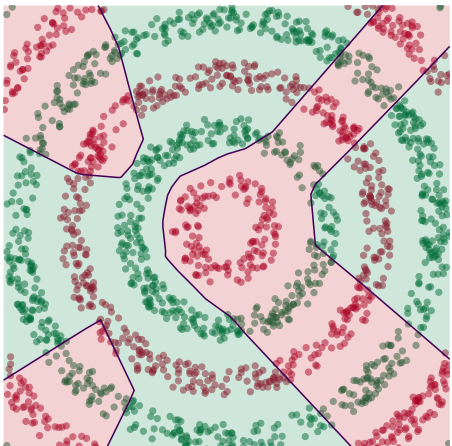
- We'll use three hidden layers, with last having two neurons.
- We can see this new representation!
- Plug in $\vec{x}$ and see activations of last hidden layer.



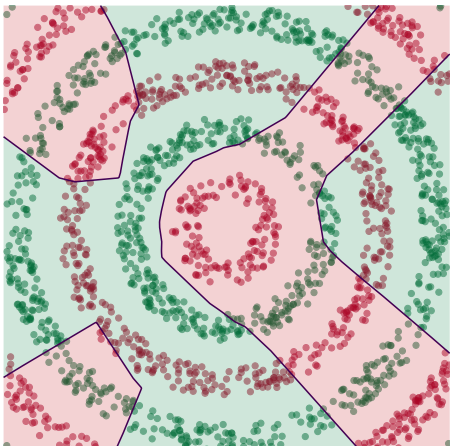
Learning a New Representation



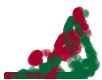
Learning a New Representation

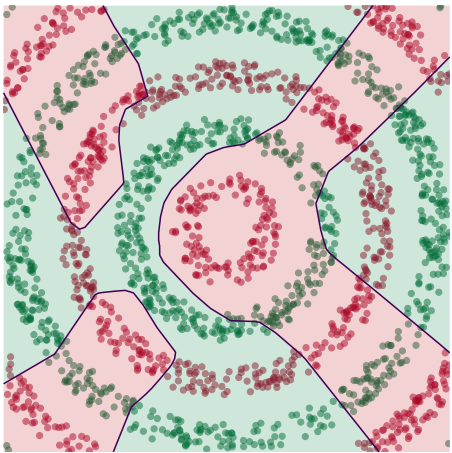


Learning a New Representation

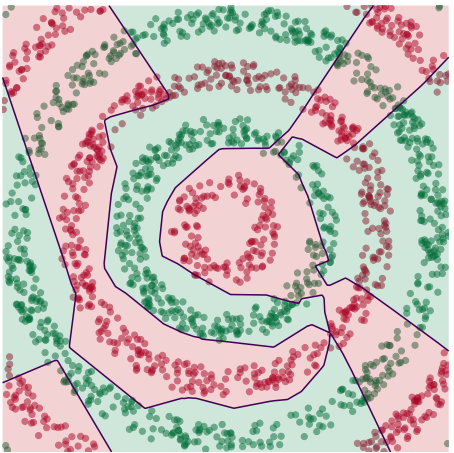


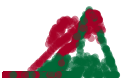
Learning a New Representation



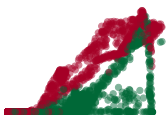


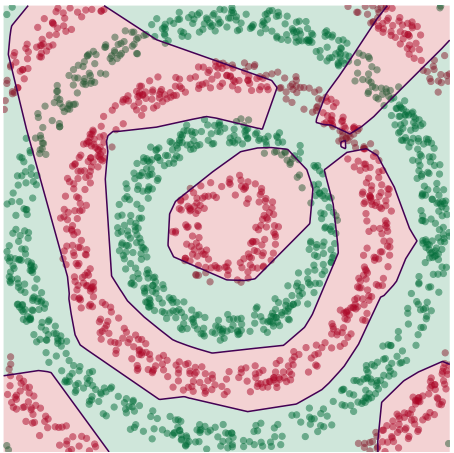
Learning a New Representation



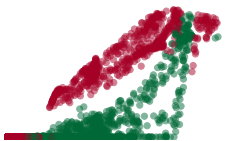


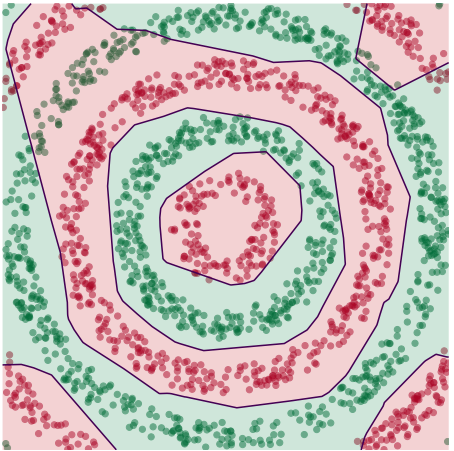
Learning a New Representation



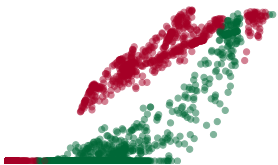


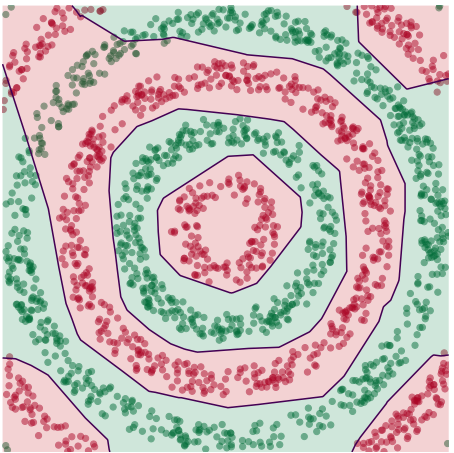
Learning a New Representation



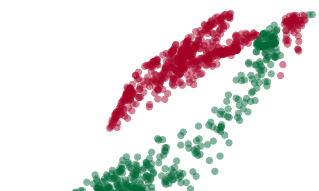


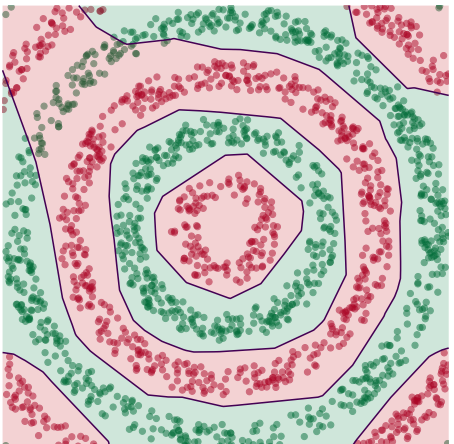
Learning a New Representation



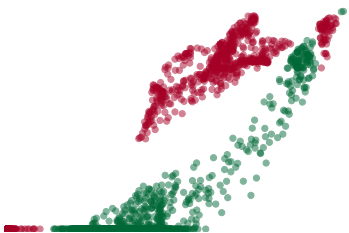


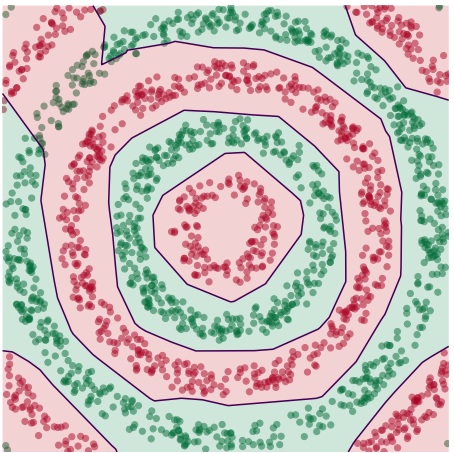
Learning a New Representation



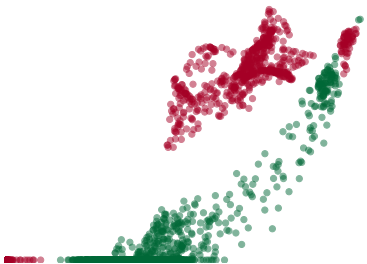


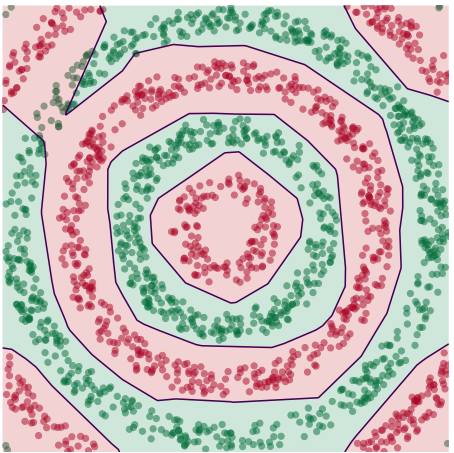
Learning a New Representation



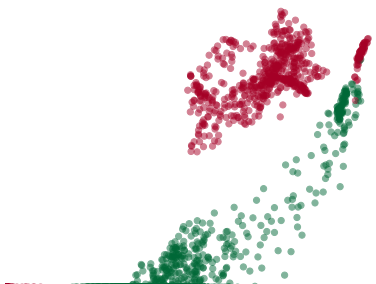


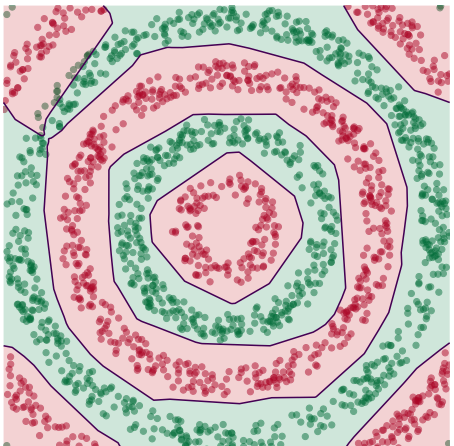
Learning a New Representation



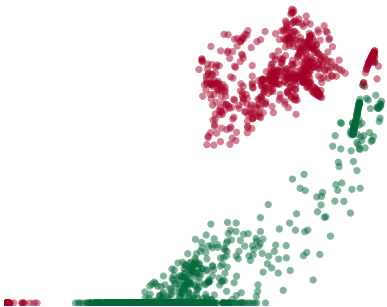


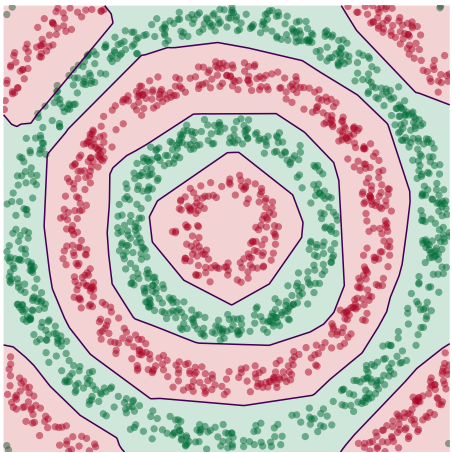
Learning a New Representation



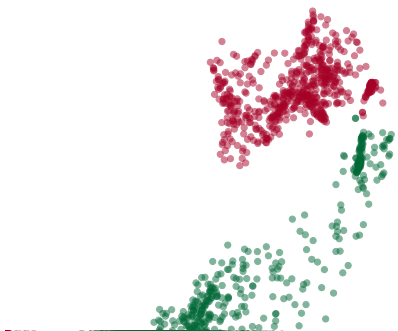


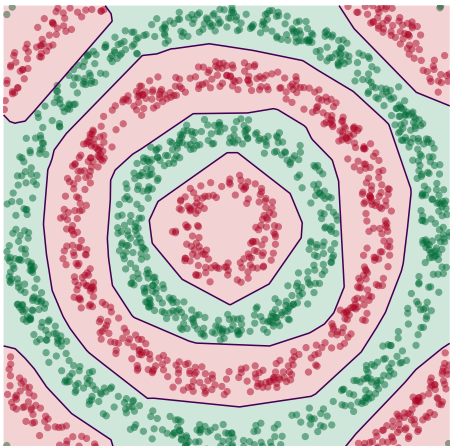
Learning a New Representation



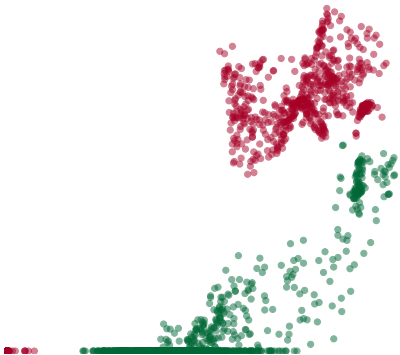


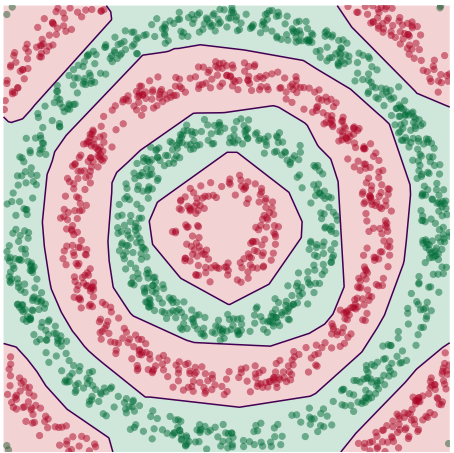
Learning a New Representation



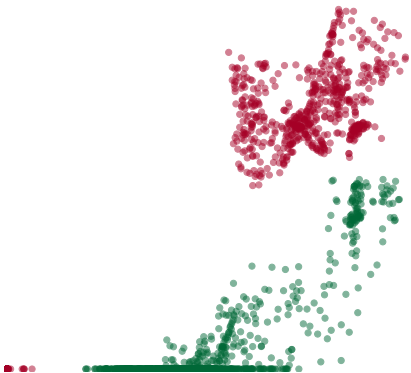


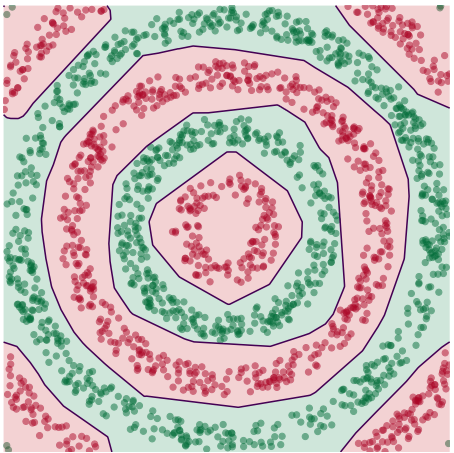
Learning a New Representation



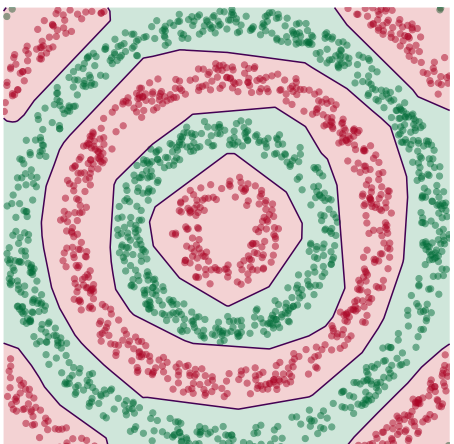


Learning a New Representation

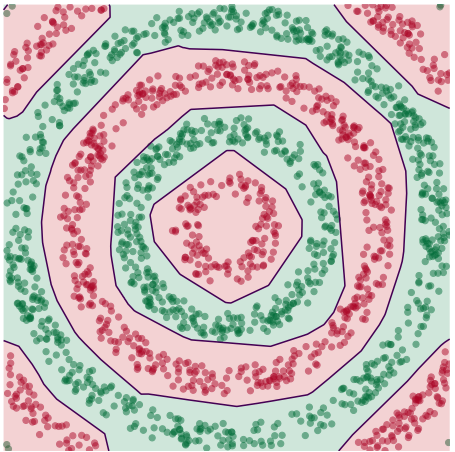




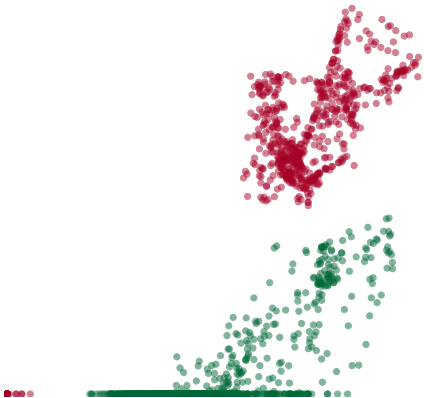
Learning a New Representation

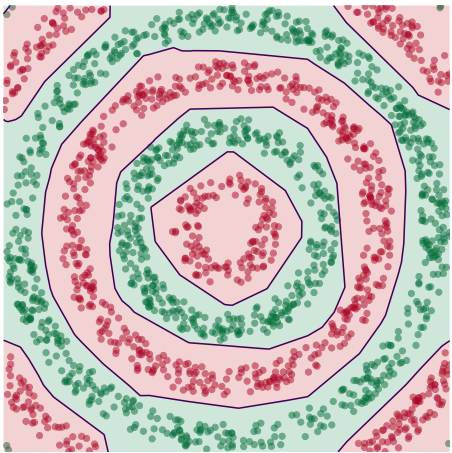


Learning a New Representation

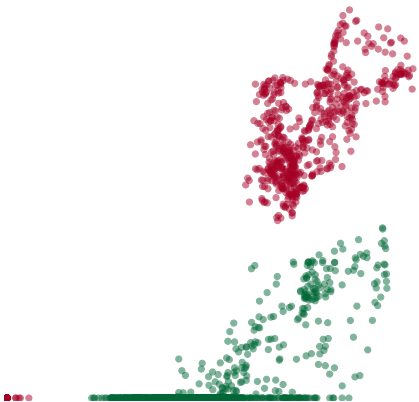


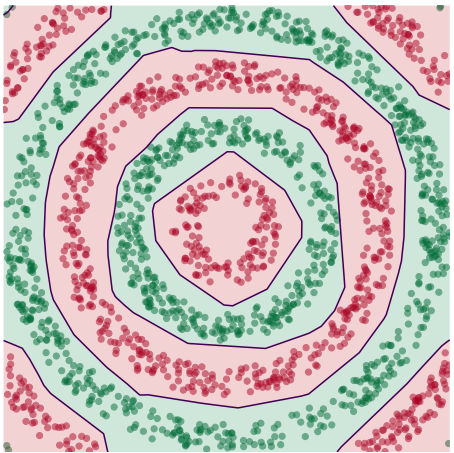
Learning a New Representation



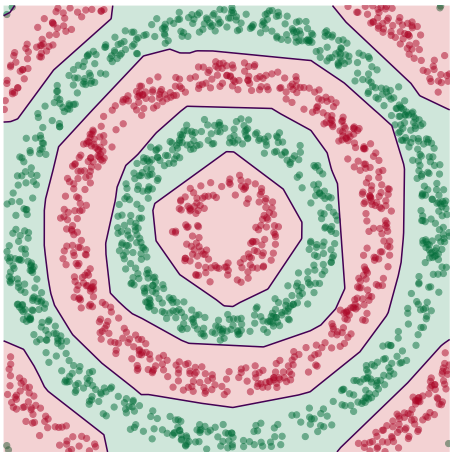


Learning a New Representation

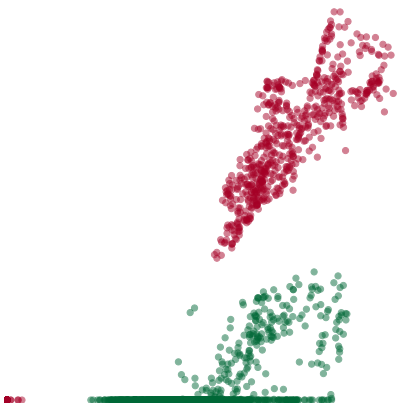


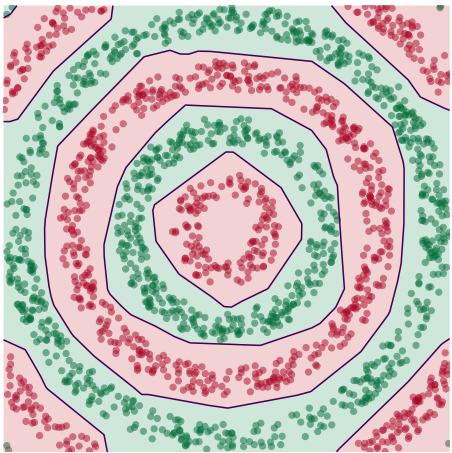


Learning a New Representation

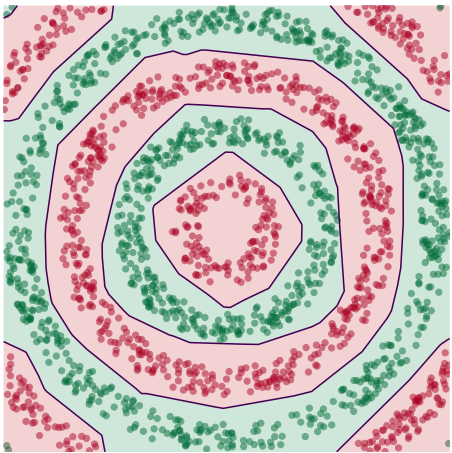


Learning a New Representation

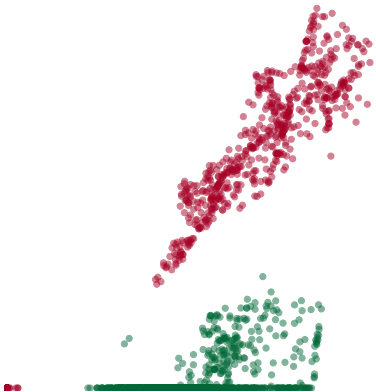


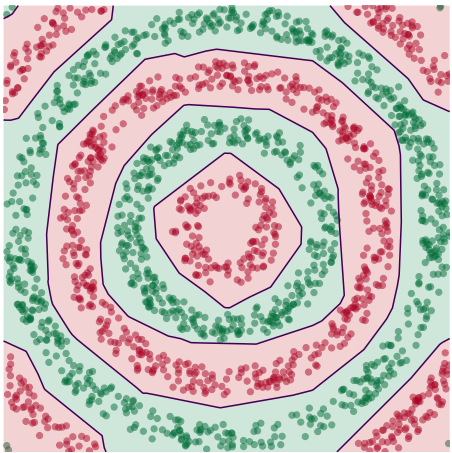


Learning a New Representation

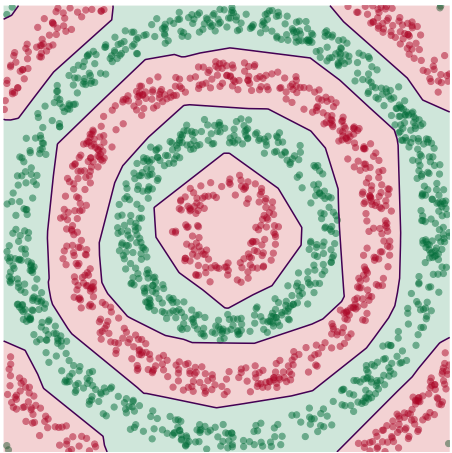


Learning a New Representation

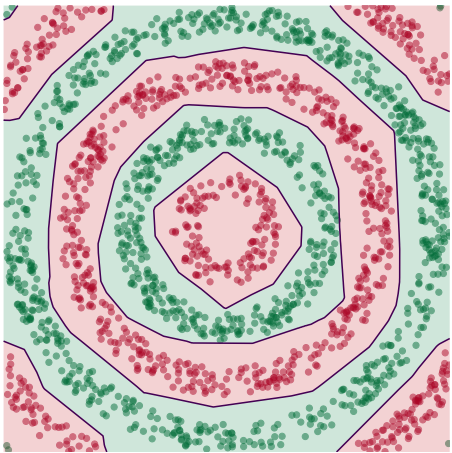




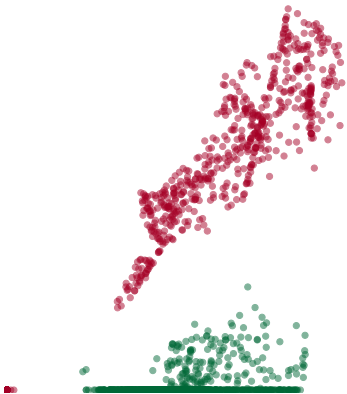
Learning a New Representation

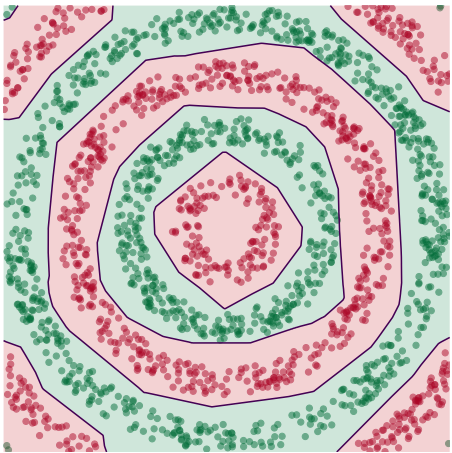


Learning a New Representation

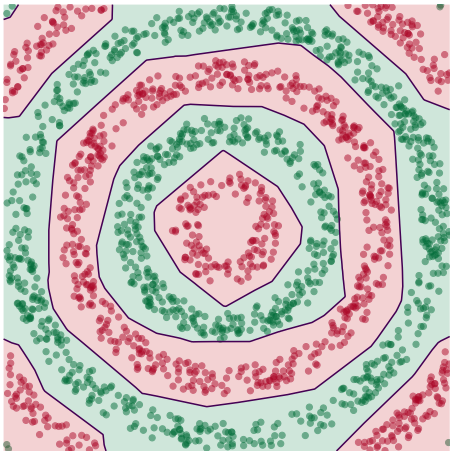


Learning a New Representation

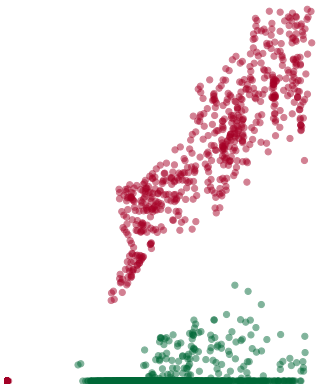


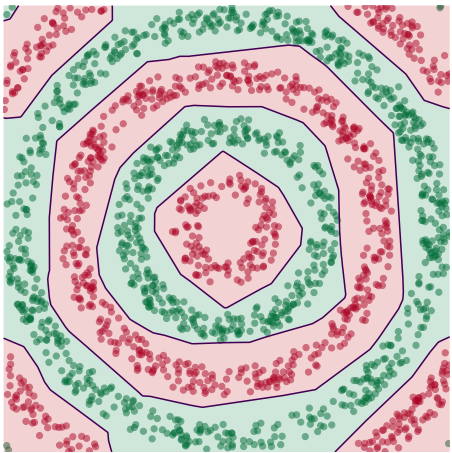


Learning a New Representation

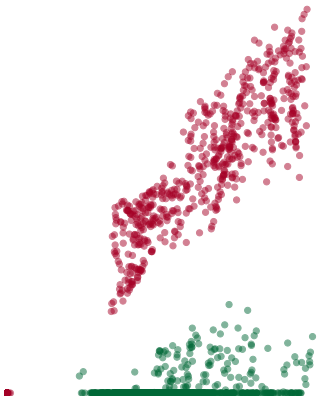


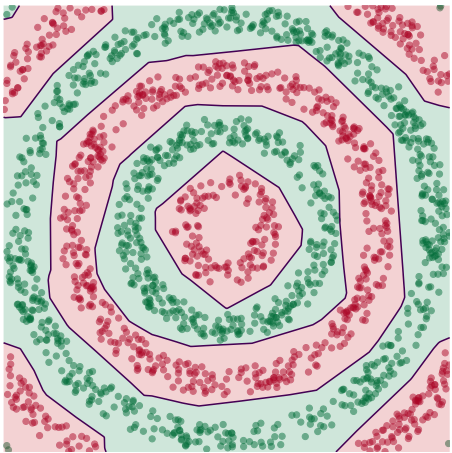
Learning a New Representation





Learning a New Representation





Deep Learning

- The NN has learned a new **representation** in which the data is easily classified.

- For more: **DSC 140B - Representation Learning**.