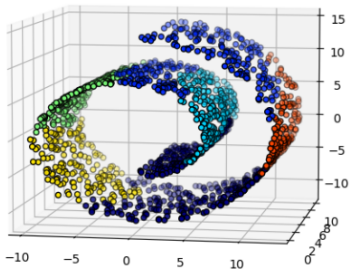
Rethinking Dimensionality

- Informally: data expressed with d dimensions, but its *really* confined to k-dimensional region
- This region is called a **manifold**
- d is the **ambient** dimension
- k is the **intrinsic** dimension

Example

- Ambient dimension: 3
- Intrinsic dimension: 2



The Graph Laplacian

- Define $L = D - W$
  - D is the degree matrix
  - W is the similarity matrix (weighted adjacency)
- L is called the **Graph Laplacian** matrix.
- It is a very useful object

Very Important Fact

- Claim:

$$\text{Cost}(\vec{f}) = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(f_i - f_j)^2 = \vec{f}^T L \vec{f}$$

- Proof: expand both sides [1]

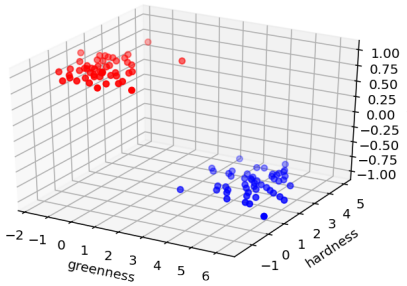
[1] Note that there was originally a $\frac{1}{2}$ in front of $\vec{f}^T L \vec{f}$, but this was not correct as written. See Problem 06 in the Midterm 02 practice for a longer explanation.

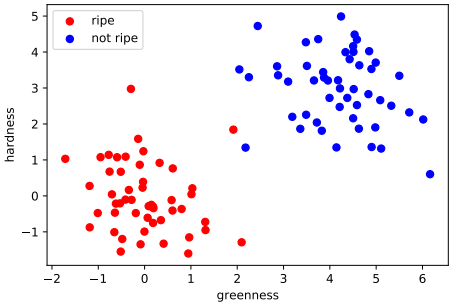
Surface

- The **surface** of a prediction function H is the surface made by plotting $H(\vec{x})$ for all $\vec{x}$.

- If H is a linear prediction function, and[4]
  - $\vec{x} \in R^1$, then $H(x)$ is a straight line.
  - $\vec{x} \in \mathbb{R}^2$, the surface is a plane.
  - $\vec{x} \in \mathbb{R}^d$, the surface is a d-dimensional **hyperplane**.

[4] when plotted in the original feature coordinate space!


greenness
hardness
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


ripe
not ripe
hardness
greenness

Least Squares and Outliers

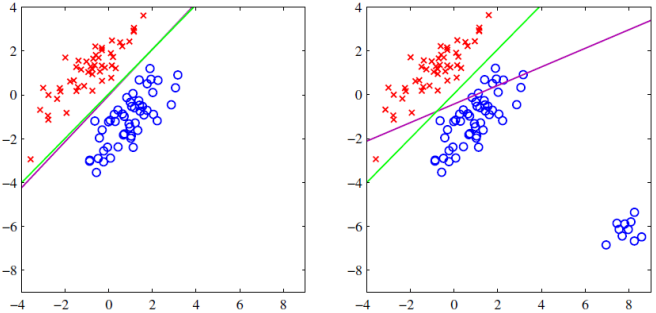


Figure 4.4 The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve) and also by the logistic regression model (green curve), which is discussed later in Section 4.3.2. The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom left of the diagram, showing that least squares is highly sensitive to outliers, unlike logistic regression.

[6] Bishop, *Pattern Recognition and Machine Learning*

Basis Functions

- We will transform:
  - the time, x_1, to $|x_1 - \text{Noon}|$
  - the temperature, x_2, to $|x_2 - 72^\circ|$

- Formally, we've designed non-linear **basis functions**:

$$\varphi_1(x_1, x_2) = |x_1 - \text{Noon}|$$
$$\varphi_2(x_1, x_2) = |x_2 - 72^\circ|$$

- In general a basis function φ maps $\mathbb{R}^d \to \mathbb{R}$

Training

- Map each training example $\vec{x}^{(i)}$ to feature space, creating new training data:

$$\vec{z}^{(1)} = \vec{\varphi}(\vec{x}^{(1)}), \quad \vec{z}^{(2)} = \vec{\varphi}(\vec{x}^{(2)}), \quad \ldots, \quad \vec{z}^{(n)} = \vec{\varphi}(\vec{x}^{(n)})$$

- Fit linear prediction function H in usual way:

$$H_f(\vec{z}) = w_0 + w_1 z_1 + w_2 z_2 + \ldots + w_d z_d$$

Prediction

- If we have $\vec{z}$ in feature space, prediction is:

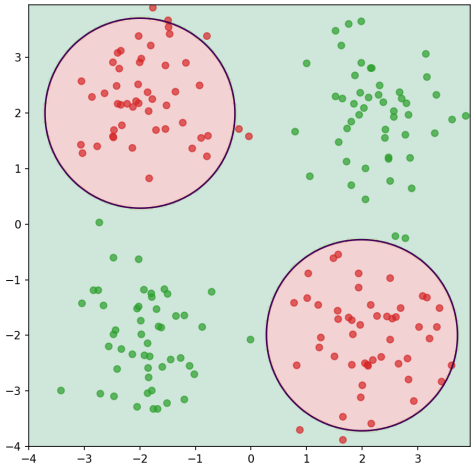
$$H_f(\vec{z}) = w_0 + w_1 z_1 + w_2 z_2 + \ldots + w_d z_d$$

Prediction

- But if we have $\vec{x}$ from original space, we must "convert" $\vec{x}$ to feature space first:

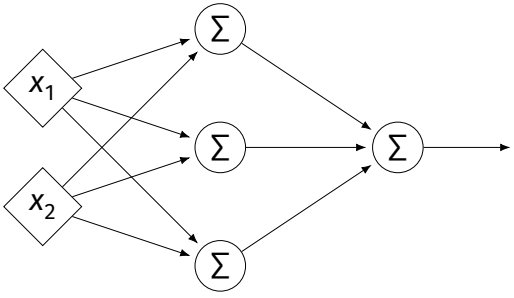
$$\begin{aligned} H(\vec{x}) &= H_f(\vec{\varphi}(\vec{x})) \\ &= H_f(\,(\varphi_1(\vec{x}),\, \varphi_2(\vec{x}),\, \ldots,\, \varphi_d(\vec{x}))^T\,) \\ &= w_0 + w_1\varphi_1(\vec{x}) + w_2\varphi_2(\vec{x}) + \ldots + w_d\varphi_d(\vec{x}) \end{aligned}$$

Decision Boundary



NNs as Function Composition

- The full NN is a composition of layer functions.



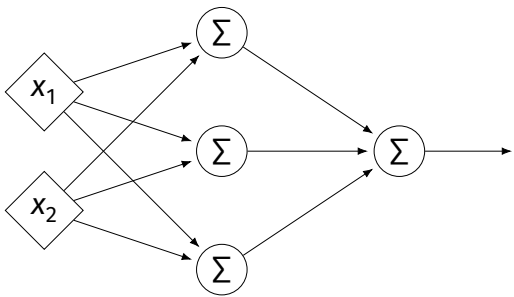
$$H(\vec{x}) = H^{(2)}(H^{(1)}(\vec{x})) = \left[W^{(2)}\right]^T \underbrace{\left(\left[W^{(1)}\right]^T \vec{x} + \vec{b}^{(1)}\right)}_{\vec{z}^{(1)}} + \vec{b}^{(2)}$$

Each Layer is a Function

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

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

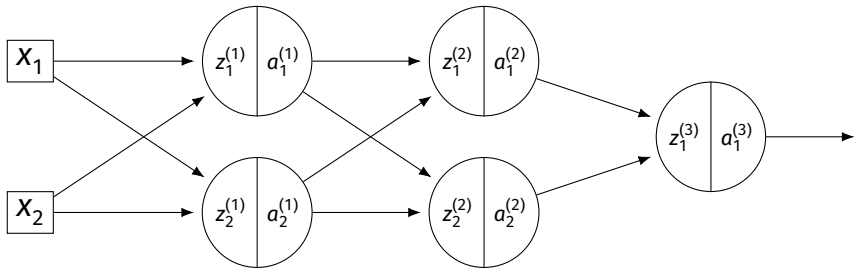
- $H^{(2)}(\vec{z}) = \left[W^{(2)}\right]^T \vec{z} + \vec{b}^{(2)}$
  - $H^{(2)} : \mathbb{R}^3 \to \mathbb{R}^1$



Example

Compute the entries of the gradient given:

$$W^{(1)} = \begin{pmatrix} 2 & -3 \\ 2 & 1 \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \quad W^{(3)} = \begin{pmatrix} 3 \\ -2 \end{pmatrix} \quad \vec{x} = (2, 1)^T \quad g(z) = \text{ReLU}$$



$$\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)} \qquad \frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^{\ell}) \qquad \frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}} a_i^{(\ell-1)}$$

DSC 140B

Representation Learning

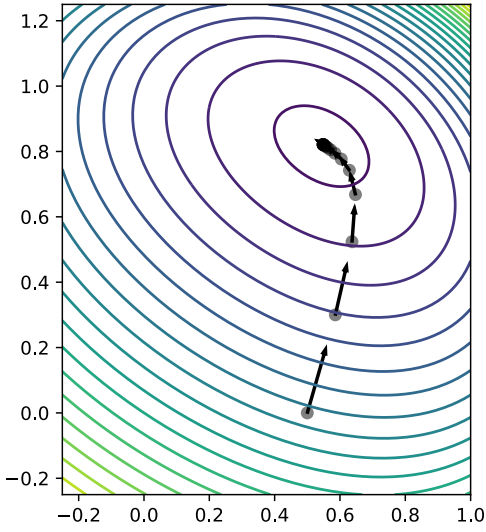
Lecture 24 | Part 1

Gradient Descent for NN Training

Gradient Descent

- Pick arbitrary starting point $\vec{x}^{(0)}$, **learning rate** parameter $\eta > 0$.
- Until convergence, repeat:
  - Compute gradient of f at $\vec{x}^{(i)}$; that is, compute $\vec{\nabla} f(\vec{x}^{(i)})$.
  - Update $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \eta \vec{\nabla} f(\vec{x}^{(i)})$.
- When do we stop?
  - When difference between $\vec{x}^{(i)}$ and $\vec{x}^{(i+1)}$ is negligible.
  - I.e., when $\|\vec{x}^{(i)} - \vec{x}^{(i+1)}\|$ is small.

```
def gradient_descent(
        gradient, x, learning_rate=.01,
        threshold=.1e-4
):
    while True:
        x_new = x - learning_rate * gradient(x)
        if np.linalg.norm(x - x_new) < threshold:
            break
        x = x_new
    return x
```


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0

Backprop Revisited

- The weights of a neural network can be trained using **gradient descent**.
- This requires the gradient to be calculated repeatedly; this is where **backprop** enters.
- Sometimes people use “backprop” to mean “backprop + SGD”, but this is not strictly correct.

Backprop Revisited

- Consider training a NN using the square loss:

$$\begin{aligned}\nabla_{\vec{w}} R(\vec{w}) &= \frac{1}{n}\sum_{i=1}^{n} \frac{\partial \ell}{\partial H} \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w}) \\ &= \frac{2}{n}\sum_{i=1}^{n} (H(\vec{x}^{(i)}) - y_i)\, \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})\end{aligned}$$

Backprop Revisited

- For any node in any neural network[1], we have the following recursive formulas:
  - $\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)}$
  - $\frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^{\ell})$
  - $\frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}} a_i^{(\ell-1)}$
  - $\frac{\partial H}{\partial b_j^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}}$

[1] Fully-connected, feedforward network

Backprop Revisited

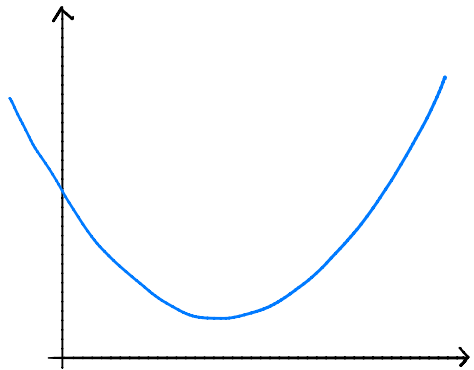
- Interpretation:

$$\nabla_{\vec{w}} R(\vec{w}) = \frac{2}{n} \sum_{i=1}^{n} \underbrace{(H(\vec{x}^{(i)}) - y_i)}_{\text{Error}} \underbrace{\nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})}_{\text{Blame}}$$

- When used in SGD, backprop “propagates error backward” in order to update weights.

Difficulty of Training NNs

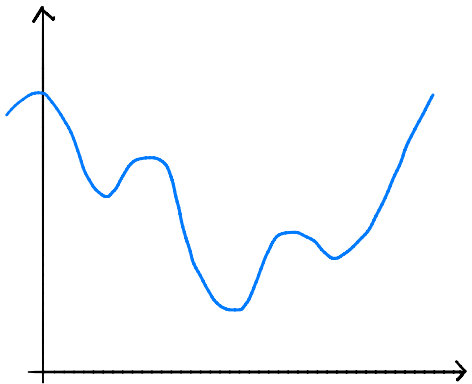
- Gradient descent is guaranteed to find optimum when objective function is **convex**.[2]



[2] Assuming it is properly initialized

Difficulty of Training NNs

- When activations are non-linear, neural network risk is **highly non-convex**:



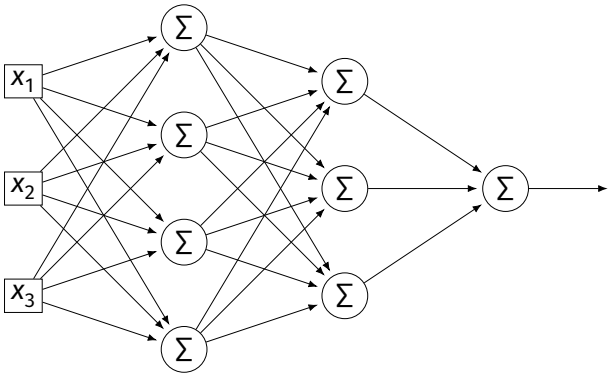
Non-Convexity

- When R is non-convex, GD can get "stuck" in local minima.
  - Solution depends on initialization.

- More sophisticated optimizers, using momentum, adaptation, better initialization, etc.
  - Adagrad, RMSprop, Adam, etc.

Difficulty of Training (Deep) NNs

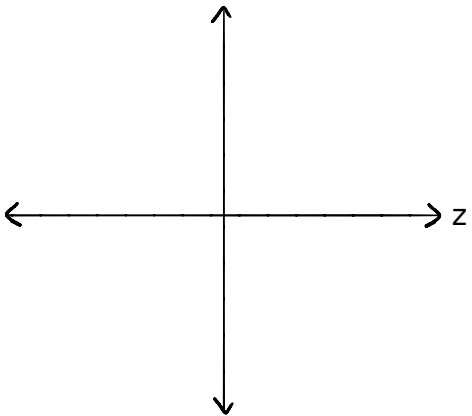
- Deep networks can suffer from the problem of **vanishing gradients**: if w is a weight at the "front" of the network, $\partial H / \partial w$ can be very small



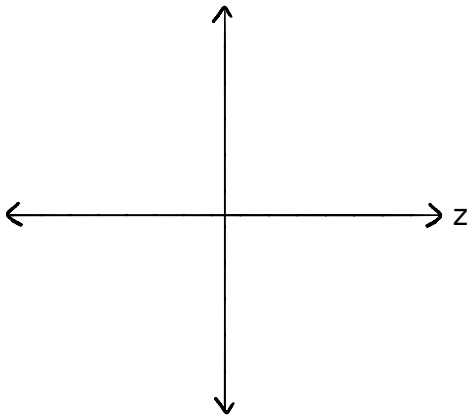
Vanishing Gradients

- If $\partial H / \partial w$ is always close to zero, w is updated **very slowly** by gradient descent.
- In short: early layers are slower to train.
- One mitigation: use ReLU instead of sigmoid.

Vanishing Gradients



Sigmoid



ReLU

DSC 140B

Representation Learning

Lecture 24 | Part 2

Stochastic Gradient Descent

Gradient Descent for Minimizing Risk

- In ML, we often want to minimize a **risk function**:

$$R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

Observation

- The gradient of the risk function is a sum of gradients:

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

- One term for each point in training data.

Problem

- In machine learning, the number of training points n can be **very large**.
- Computing the gradient can be **expensive** when n is large.
- Therefore, each step of gradient descent can be **expensive**.

Idea

- The (full) gradient of the risk uses all of the training data:

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

- It is an average of n gradients.

- **Idea:** instead of using all n points, randomly choose $\ll n$.

Stochastic Gradient

- Choose a random subset (**mini-batch**) B of the training data.
- Compute a **stochastic gradient**:

$$\nabla R(\vec{w}) \approx \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

Stochastic Gradient

$$\nabla R(\vec{w}) \approx \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- **Good:** if $|B| \ll n$, this is much faster to compute.
- **Bad:** it is a (random) approximation of the full gradient, noisy.

Stochastic Gradient Descent (SGD) for ERM

- Pick arbitrary starting point $\vec{x}^{(0)}$, **learning rate** parameter $\eta > 0$, batch size $m \ll n$.

- Until convergence, repeat:
  - Randomly sample a batch B of m training data points (on each iteration).
  - Compute stochastic gradient of f at $\vec{x}^{(i)}$:

$$\vec{g} = \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

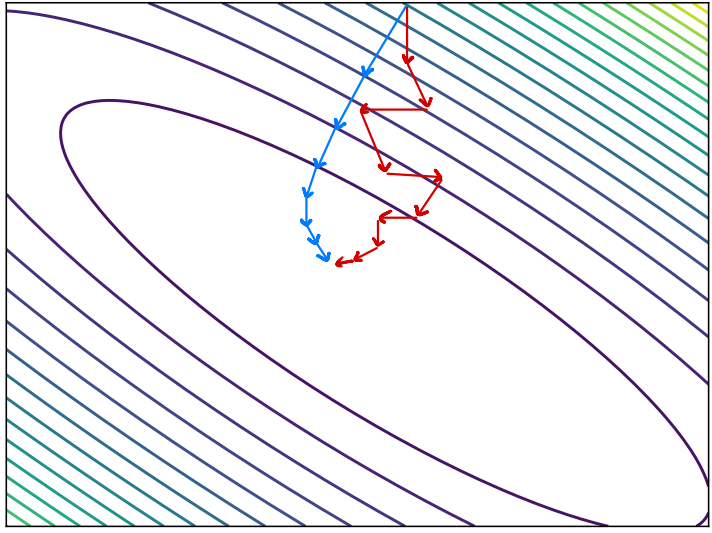
  - Update $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \eta \vec{g}$

Idea

- In practice, a stochastic gradient often works well enough.
- It is better to take many noisy steps quickly than few exact steps slowly.

Batch Size

- Batch size m is a parameter of the algorithm.
- The larger m, the more reliable the stochastic gradient, but the more time it takes to compute.
- Extreme case when $m = 1$ will still work.



Usefulness of SGD

- SGD allows learning on **massive** data sets.
- Useful even when exact solutions available.
  - E.g., least squares regression / classification.

Training NNs in Practice

- There are several Python packages for training NNs:
  - PyTorch
  - Tensorflow / Keras