Introduction to Modern AI
Week 2: Supervised Learning I - Differentiable Parametric Models

Gavin Hartnett

PRGS, Winter Quarter 2022

Topics we will cover this week

- In this week we explore more complex models of the form $f(\boldsymbol{x}; \boldsymbol{\theta})$
- Mainly in supervised learning context, but many results will apply to other learning paradigms
- As we move through the course $f(\boldsymbol{x}; \boldsymbol{\theta})$ will become more complicated (expressive), but it will retain two important properties
  - it will be described in terms of parameters $\boldsymbol{\theta}$
  - it will be differentiable, i.e. $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}; \boldsymbol{\theta})$ will exist and be computable
- Main goal of this week is to introduce neural networks and gradient descent
- First, we will warm-up with the Perceptron
- Next week we will look at non-parametric models such as k-nearest neighbors and decision trees

The Perceptron

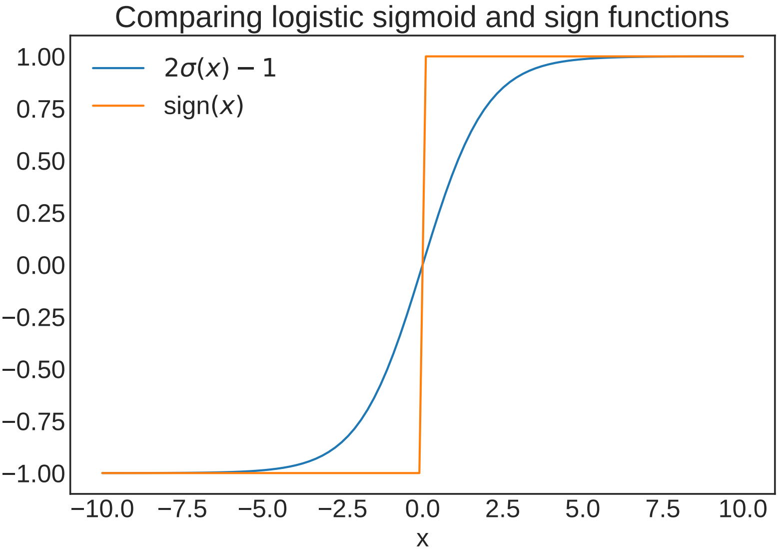
The Perceptron

- Historically significant model + algorithm for binary classification
- Useful to formulate it using the convention that $y \in \{-1, +1\}$
- Precursor to the sophisticated neural networks we use today
- Similar to logistic regression
- Linear model (absorbing bias into $\boldsymbol{w}$):

$$f(\boldsymbol{x}; \boldsymbol{w}) = \text{sign}(\boldsymbol{w}^T \boldsymbol{x}) = \begin{cases} -1\,, & \boldsymbol{w}^T \boldsymbol{x} < 0 \\ 1\,, & \boldsymbol{w}^T \boldsymbol{x} > 0 \end{cases}$$

- Unlike logistic regression, the perceptron just predicts a class, not a probability

The Perceptron



The Perceptron

Algorithm 1 Perceptron Learning Algorithm

initialize weights $\boldsymbol{w}$
while not converged **do**
    for $i = 1, ..., N$ **do**
        compute model prediction $\hat{y}_i = f(\boldsymbol{x}_i; \boldsymbol{w})$
        $\boldsymbol{w} \mathrel{-}= \frac{\lambda}{2}(\hat{y}_i - y_i)\boldsymbol{x}_i$
    end for
end while

Interpretation:

- if prediction is correct, don't change weight vector
- if prediction is wrong, increase/decrease weight vector proportional to $\boldsymbol{x}_i$

The Perceptron

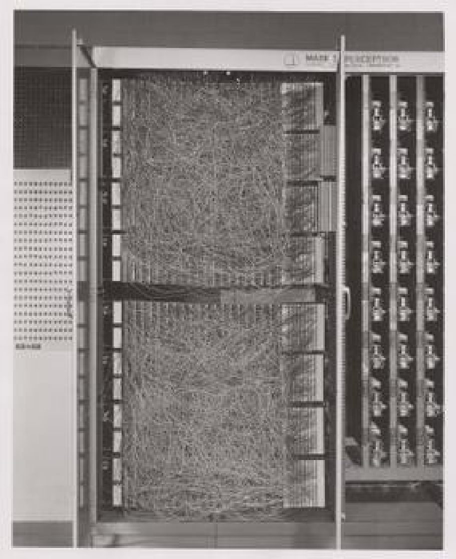
Show Example

The Perceptron

- The Perceptron learning algorithm is trying to find a separating hyper-plane
- If the data is not linearly separable, then the algorithm will not converge
- There are many solutions when the data is separable, and the algorithm doesn't necessarily pick the best one

The Perceptron

- The Perceptron was actually built as a piece of specialized hardware (Mark I Perceptron)
- That's cool... but why are you telling us this?
  - Non-linear generalizations of Perceptrons are the simplest Artificial Neural Networks (ANNs)
  - The learning algorithm is actually stochastic gradient descent
  - They are closely related to a powerful set of models called Support Vector Machines (SVMs)



(Artificial) Neural Networks

The Need For More Expressive Models

- Both linear regression and logistic regression are *linear* (or log-linear) models
- This property severely constrains the types of functions they can fit
- It would be desirable to have models capable of learning arbitrarily complicated functions
  - Requires flexible families of models (i.e., NNs)
  - Requires general learning algorithm/framework (i.e., gradient descent)

Basis Expansions

- Recall linear regression model:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{w}^T \boldsymbol{x} + b$$

- How can this be improved to incorporate non-linearities?
  - Suppose we knew that the probability explicitly depended on $(x_4)^2$, in addition to a linear dependence on $\boldsymbol{x}$?
  - Just add it in!:

$$\boldsymbol{x}' = (x_1, x_2, ..., x_p, (x_4)^2)$$

$$\boldsymbol{w}' = (w_1, w_2, ..., w_p, w_{p+1})$$

  - Model remains linear in enlarged $\boldsymbol{x}'$ space!
  - Suppose it also depends on $\sin(x_2)$?
  - Just add it in!:

$$\boldsymbol{x}' = (x_1, x_2, ..., x_p, (x_4)^2, \sin(x_2))$$

$$\boldsymbol{w}' = (w_1, w_2, ..., w_p, w_{p+1}, w_{p+2})$$

Basis Expansions

- This basic idea can be extended to incorporate arbitrary non-linearities in a systematic fashion
- Suppose we identify a set of M useful transformations of the original input (or feature) space $\boldsymbol{x}$: $h_m(\boldsymbol{x}) : \mathbb{R}^p \rightarrow \mathbb{R}$, for $m = 1, ..., M$
- Simply build a linear model in the transformed feature space:

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \sum_{m=1}^{M} w_m h_m(\boldsymbol{x}) + b$$

- Issues with this approach
  - Poor scaling: p^d independent terms (and parameters) for d-degree polynomial
  - Inefficient way to make model more flexible unless we have special knowledge of the non-linearities

Neural Networks

- Basic idea is very simple: build expressive model $f(\boldsymbol{x}; \boldsymbol{\theta})$ using simpler modular components called layers:

$$f = f^L \circ f^{L-1} \circ \ldots \circ f^1$$

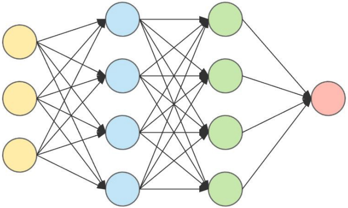
- here $\circ$ means "compose", i.e. $(f \circ g)(x) = f(g(x))$
- the output of layer $f^{(\ell)}$ becomes the input to layer $f^{(\ell+1)}$
- As we will see, f has the structure of a bunch of simple processing units connected through a network - loose inspiration from (real) neural networks
- This lecture: simple, "vanilla" neural network, aka multi-layer perceptron (MLP)
- In later lectures we will learn about many sophisticated variants and extensions

Neural Networks

- Neural networks are mathematical models, $f(\boldsymbol{x}; \boldsymbol{\theta})$, where

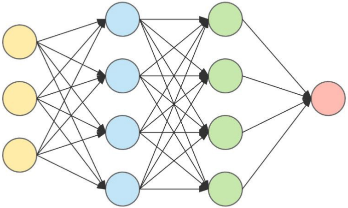
$$f = f^L \circ f^{L-1} \circ \ldots \circ f^1$$

- Can also be described graphically
- Circles represent both variables and a computation, such as $y = g(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$
- Computation involves add/multiply and a possible non-linearity, called the activation function g
- Lines represent flow of information
- Network is organized into layers, input, hidden ($\times 2$), and output



Neural Networks

Here is how to "translate" the diagram



- Input (yellow dots):

$$\boldsymbol{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)})$$

- Hidden Layer 1 (blue dots):

$$\boldsymbol{x}^{(1)} = (x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, x_4^{(1)}) = g^{(1)}\left(\boldsymbol{w}^{(1)}\boldsymbol{x}^{(0)} + \boldsymbol{b}^{(1)}\right)$$

- Hidden Layer 2 (green dots):

$$\boldsymbol{x}^{(2)} = (x_1^{(2)}, x_2^{(2)}, x_3^{(2)}, x_4^{(2)}) = g^{(2)}\left(\boldsymbol{w}^{(2)}\boldsymbol{x}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

- Output Layer (red dot):

$$\boldsymbol{x}^{(3)} = (x_1^{(3)}) = g^{(3)}\left(\boldsymbol{w}^{(3)}\boldsymbol{x}^{(2)} + \boldsymbol{b}^{(3)}\right)$$

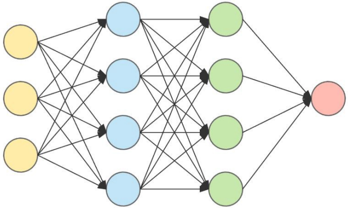
Neural Networks

Putting it all together: $y = x_1^{(3)} = f(\boldsymbol{x};\boldsymbol{\theta})$, with

$$x_1^{(3)} = g^{(3)}\left(\boldsymbol{w}^{(3)}\left(g^{(2)}\left(\boldsymbol{w}^{(2)}\left(g^{(1)}\left(\boldsymbol{w}^{(1)}\boldsymbol{x}^{(0)} + \boldsymbol{b}^{(1)}\right)\right) + \boldsymbol{b}^{(2)}\right)\right) + \boldsymbol{b}^{(3)}\right)$$

Some comments

- The input and output dimensions are fixed by the problem
- Hidden layer dimensions are unconstrained
- Number of hidden layers is unconstrained
- Each layer can have a different activation function



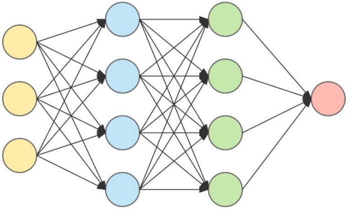
Neural Networks

Putting it all together: $y = x_1^{(3)} = f(\boldsymbol{x}; \boldsymbol{\theta})$, with

$$x_1^{(3)} = g^{(3)}\left(\boldsymbol{w}^{(3)}\left(g^{(2)}\left(\boldsymbol{w}^{(2)}\left(g^{(1)}\left(\boldsymbol{w}^{(1)}\boldsymbol{x}^{(0)} + \boldsymbol{b}^{(1)}\right)\right) + \boldsymbol{b}^{(2)}\right)\right) + \boldsymbol{b}^{(3)}\right)$$

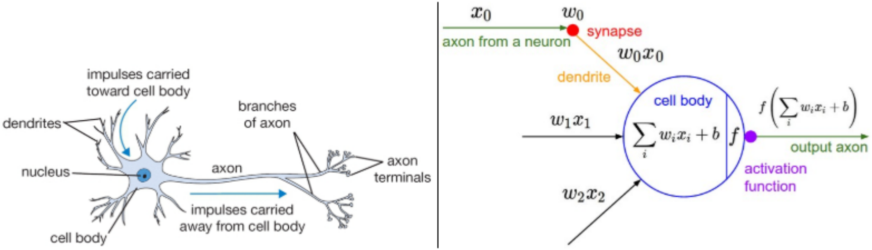
How many parameters does the network contain?

- bias *vectors*
  - $\dim(\boldsymbol{b}_1) : N_{h_1}$
  - $\dim(\boldsymbol{b}_2) : N_{h_2}$
  - $\dim(\boldsymbol{b}_3) : 1$
- weight *matrices*
  - $\dim(\boldsymbol{w}^{(1)})$: $N_{h_1} \times p$
  - $\dim(\boldsymbol{w}^{(2)})$: $N_{h_2} \times N_{h_1}$
  - $\dim(\boldsymbol{w}^{(3)})$: $1 \times N_{h_2}$
- The model can be made arbitrarily large by increasing the number of units in either hidden layer



Activation Functions

- Non-linearities come entirely from the activation functions
- Some common choices for g:
  - Logistic sigmoid: $g(x) = \sigma(x)$
  - ReLU: $g(x) = \max(0, x)$
  - tanh: $g(x) = \tanh(x)$
  - GELU: $g(x) = \frac{x}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Figure: Source: cs231n Stanford Class

Universal Approximation Theorem

What class of functions can NNs represent?

- By adding more hidden layers (increasing depth) or adding more units to existing layers (increasing width) we can make the model more expressive (contain more parameters)
- More parameters loosely implies a capacity to approximate a larger class of functions
- The universal approximation theorems formally guarantee that, with enough parameters, NNs are capable of approximating any function with arbitrary accuracy
- There are many theorems for different activation functions and assumptions about model architecture (arbitrary width or depth)
- Theorems are not prescriptive - they do not tell you have many units/layers you need to achieve a certain accuracy
- In the worst-case, you often need an exponentially large network
- How to train a NN is an entirely separate matter

Gradient Descent

Gradient Descent

- We've already seen that more complex models can be harder to train (fit) then simpler ones
  - The best-fit parameters for linear regression can be solved for in closed-form
  - Fitting logistic regression requires numerical optimization
  - No surprise: NNs will also require numerical optimization
- There are many optimization algorithms, which one(s) should we use?
  - Want a flexible method
  - No assumptions on geometry of loss function (i.e., cannot assume convexity)
  - Want it to be relatively fast/efficient
- Gradient Descent (GD) and variations are the dominant method used for training NNs

Gradient Descent

- GD is a very simple algorithm that only requires local information:
  - Starting at a given point, find the direction of steepest descent (given by the gradient vector)
  - Then take a small step in this direction
  - Repeat
- Think of a blind man trying to get to the bottom of a hill

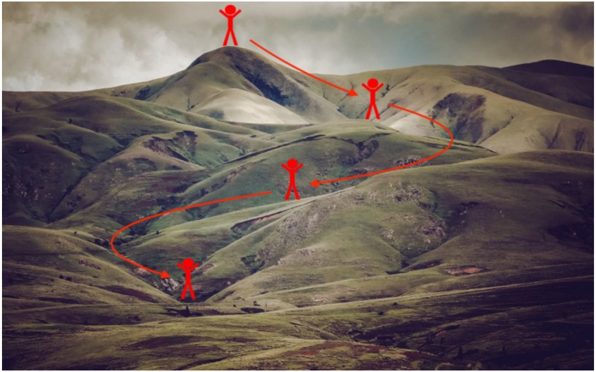


Figure: Source: https://towardsdatascience.com/understanding-gradient-descent-and-adam-optimization-472ae8a78c10

Gradient Descent Algorithm

Algorithm 2 Gradient Descent

function to be minimized: $f(\boldsymbol{x})$
set learning rate α
initialize variable $\boldsymbol{x}$
while not converged **do**
    $\boldsymbol{x} \leftarrow \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$
end while

Gradient Descent Algorithm

- Example: Rosenbrock (banana) function

$$f(x, y) = (1 - x)^2 + 10(y - x^2)^2 \,,$$

- The gradient is

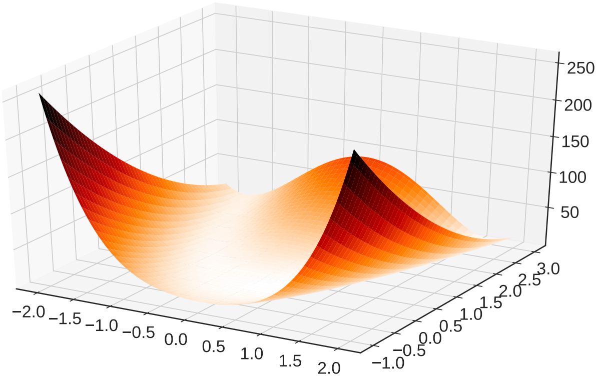
$$\nabla f = \begin{bmatrix} -2(1-x) - 40x(y - x^2) \\ 20(y - x^2) \end{bmatrix} .$$

- The gradient vanishes for the single point $(x, y) = (1, 1)$, which is also the global minimum. At this point the function takes on the value $f(1, 1) = 0$.

Gradient Descent Algorithm

- Example: Rosenbrock function

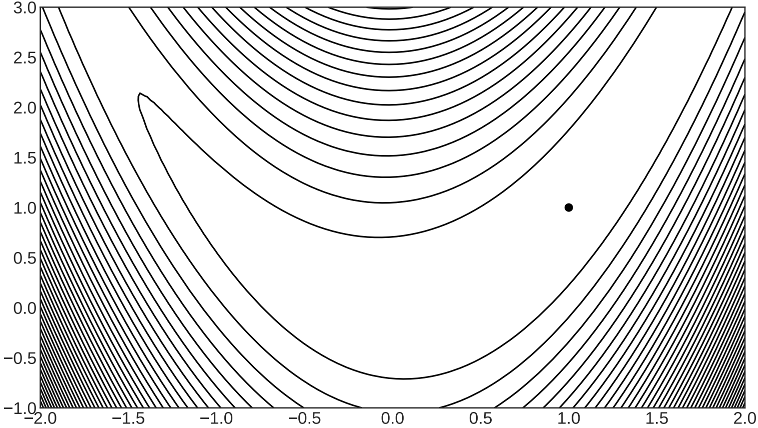
$$f(x, y) = (1 - x)^2 + 10(y - x^2)^2 \, ,$$



Gradient Descent Algorithm

- Example: Rosenbrock function

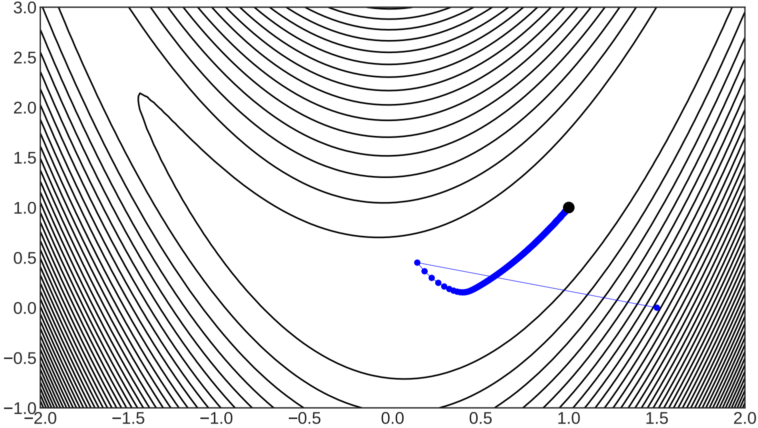
$$f(x, y) = (1 - x)^2 + 10(y - x^2)^2 \,,$$



Gradient Descent Algorithm

- Example: Rosenbrock function

$$f(x, y) = (1 - x)^2 + 10(y - x^2)^2 \,,$$



Gradient Descent Algorithm

Some issues to keep in mind

- The algorithm only converges if the learning rate α is small enough
- "Small enough" is problem dependent
- GD can get stuck in local minima
- Not well-suited for problems with widely-varying length scales (i.e., some very steep directions and some very shallow directions)

Example: Quadratic Loss

- To better understand GD, let's examine a simple optimization problem that we can solve exactly
- Consider a quadratic loss function:

$$L(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T \boldsymbol{A}\boldsymbol{w} - \boldsymbol{b}^T \boldsymbol{w}$$

- $\boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^d$, and $\boldsymbol{A} \in \mathbb{R}^{d \times d}$. also assume $\boldsymbol{A}$ is symmetric and invertible, meaning $\boldsymbol{A}^{-1}$ exists
- Gradient is:

$$\nabla_{\boldsymbol{w}} L = \boldsymbol{A}\boldsymbol{w} - \boldsymbol{b}$$

- Optimal solution is then $\boldsymbol{w}^* = \boldsymbol{A}^{-1}\boldsymbol{b}$

Example: Quadratic Loss

- Quadratic loss function: $L(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{A}\boldsymbol{w} - \boldsymbol{b}^T\boldsymbol{w}$
- Gradient: $\nabla_{\boldsymbol{w}} L = \boldsymbol{A}\boldsymbol{w} - \boldsymbol{b}$
- Optimal solution: $\boldsymbol{w}^* = \boldsymbol{A}^{-1}\boldsymbol{b}$
- Diagonalize:

$$\boldsymbol{A} = \boldsymbol{O}^{-1}\boldsymbol{\Lambda}\boldsymbol{O}\,, \qquad \boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \lambda_2, ..., \lambda_d)\,.$$

$$\boldsymbol{b}' = \boldsymbol{O}\boldsymbol{b}\,, \qquad \boldsymbol{x} = \boldsymbol{O}\boldsymbol{w}\,.$$

- Gradient descent update rule:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \alpha\nabla_{\boldsymbol{w}} L(\boldsymbol{w}^t) = \boldsymbol{w}^t - \alpha\left(\boldsymbol{A}\boldsymbol{w}^t - \boldsymbol{b}\right)$$

- Becomes:

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \alpha\left(\boldsymbol{\Lambda}\boldsymbol{x}^t - \boldsymbol{b}'\right)$$

- In terms of components:

$$x_a^{t+1} = (1 - \alpha\lambda_a)\,x_a^t + \alpha b'_a$$

Example: Quadratic Loss

- GD update rule in $\boldsymbol{x}$ variable:

$$x_a^{t+1} = (1 - \alpha\lambda_a)\, x_a^t + \alpha b_a'$$

- Can "unroll":

$$\begin{aligned} x_a^{t+1} &= (1 - \alpha\lambda_a)\left((1 - \alpha\lambda_a)\, x_a^{t-1} + \alpha b_a'\right) + \alpha b_a' \\ &= (1 - \alpha\lambda_a)^2\, x_a^{t-1} + (1 + (1 - \alpha\lambda_a))\, \alpha b_a' \end{aligned}$$

- Continuing:

$$x_a^{t+1} = (1 - \alpha\lambda_a)^{t+1}\, x_a^0 + \alpha b_a' \sum_{k=0}^{t} (1 - \alpha\lambda_a)^k$$

- For this to converge as $t \to \infty$, require $|1 - \alpha\lambda_a| < 1$. Then first term vanishes and second term is a geometric series, can be summed to give $\boldsymbol{x}^* = \boldsymbol{O}\boldsymbol{w}^* = \boldsymbol{\Lambda}^{-1}\boldsymbol{b}$:

$$x_a^* = \lim_{t\to\infty} x_a^t = \alpha b_a' \left(\frac{1}{1 - (1 - \alpha\lambda_a)}\right) = \frac{b_a'}{\lambda_a}$$

Example: Quadratic Loss

To summarize

- Optimization problem for d-dim vector $\boldsymbol{w}$ decomposes into d separate 1-dim optimization problems if we work in the eigenbasis
- Convergence requires $|1 - \alpha\lambda_a| < 1$
- Corresponds to $0 < \alpha\lambda_a < 2$
  - requires all eigenvalues to have same sign - positive for a (global) minimum
  - then $\alpha > 0$
  - $\alpha < 0$, $\lambda_a < 0$ corresponds to gradient ascent
- Convergence in this case is *exponential*
- But convergence rate is different for each eigen-direction, overall rate is limited by worst eigenvalue

Stochastic Gradient Descent Algorithm

Let's use gradient descent to minimize a loss function

- Loss function is averaged over training set:

$$\text{loss}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta})$$

- ℓ could be the sum of squared errors (SSE), or binary cross entropy (BCE), or really any loss
- Gradient of loss will also be averaged:

$$\nabla_{\boldsymbol{\theta}}\text{loss}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta})$$

Stochastic Gradient Descent Algorithm

$$\nabla_{\boldsymbol{\theta}} \text{loss}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta})$$

- This expression is not always convenient to work with
- GD requires many iterations, and each iteration requires that the gradient be computed N times
- N is often large (and we want N to be as large as possible)
- Idea: let's cut a corner and instead of averaging the gradient over all N data points, let's average it over a "mini-batch" of $N_B \ll N$ data points
- Each GD update can use a different, randomly chosen batch
- Note on terminology:
  - $N_B = N$: gradient descent
  - $1 < N_B < N$: mini-batch stochastic gradient descent
  - $N_B = 1$: stochastic gradient descent (aka on-line learning)

Introduction to Backpropagation

- To use SGD, we must be able to compute the gradient of the loss function $\nabla_{\boldsymbol{\theta}} \ell$
- Let's consider the MSE loss for a simple NN with 1 hidden layer:

$$\ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) = (y_i - f(\boldsymbol{x}; \boldsymbol{\theta}))^2 \ ,$$

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = g^{(2)}\left(\boldsymbol{w}^{(2)}\left(g^{(1)}\left(\boldsymbol{w}^{(1)}\boldsymbol{x}^{(0)} + \boldsymbol{b}^{(1)}\right)\right) + \boldsymbol{b}^{(2)}\right)$$

  - Need to compute $\nabla_{\boldsymbol{w}^{(1)}} f$, $\nabla_{\boldsymbol{w}^{(2}} f$, $\nabla_{\boldsymbol{b}^{(1)}} f$, $\nabla_{\boldsymbol{b}^{(2)}} f$
  - This is do-able, but a bit of a pain
  - What if we later want to tweak the network? What if we want to consider a large model with 10's or 100's of layers?
  - It's desirable to have an automated procedure to compute the gradient for us

Introduction to Backpropagation

- Suppose we have functions f, g, and the composition $h = g \circ f$
- The chain rule let's use calculate the derivative h':

$$h' = (g \circ f)' = (g' \circ f)f'$$

or,

$$h'(x) = g'(f(x))f'(x)$$

- Example:

$$\frac{d}{dx}\sin(x)^2 = 2\sin x \cos x$$

  - $f(x) = \sin(x)$, $g(x) = x^2$
  - $f'(x) = \cos(x)$, $g'(x) = 2x$
  - $h'(x) = (2\sin x)\cos x$

Introduction to Backpropagation

- What about functions built using more than 1 composition?

$$f = f^{(L)} \circ f^{(L-1)} \circ \ldots \circ f^{(1)}$$

- Apply the chain rule iteratively:

$$f = f^{(L)} \circ F^{(L-1)}, \qquad F^{(L-1)} = f^{(L-1)} \circ \ldots \circ f^{(1)}$$

$$f' = (f^{(L)\prime} \circ F^{(L-1)}) F^{(L-1)\prime}$$

- and so on for $F^{(L-1)\prime}$, and then $F^{(L-2)\prime}, \ldots$
- This "algorithm" can be used to compute the gradient of an arbitrarily complicated neural net
- Q: why is it called backpropagation?
- In week 4 we'll discuss how this is implemented in modern deep learning software libraries using automatic differentiation

Recap

- Vanilla, Feed-Forward Neural Networks
- (Stochastic) Gradient Descent
- Backpropagation