11. Classification

Last time

- Automatic differentiation
- Newton method for finding roots

Goals for today

- Differentiation in higher dimensions
- Newton in higher dimensions
- Machine learning: Classification
- Introduction to neural networks

Automatic differentiation in higher dimensions

Automatic differentiation (reminder)

- Recall: Can differentiate a function $f : \mathbb{R} \to \mathbb{R}$ automatically
- Define dual number `Dual(a, b)` corresponding to $a + b\epsilon$
- Math: calculate $f(a + b\epsilon)$ – coefficient of ϵ gives $f'(a)$
- Julia: calculate `f(Dual(a, b))` – derivative part of result gives $f'(a)$

Higher dimensions

- What happens for higher-dimensional functions
- E.g. $f(x, y) = x^2 + y^2 - 1$
- Generalise approach from 1D
- Pass in dual numbers with *same* ϵ:
- Set $x = a + c\epsilon$ and $y = b + d\epsilon$
- Calculate

$$f(a + c\epsilon, b + d\epsilon)$$

Partial derivatives

- Calculate

$$f(a+v_1\epsilon, b+v_2\epsilon)$$

- Expand:

$$f(a,b)+v_1\epsilon\frac{\partial f}{\partial x}(a,b)+v_2\epsilon\frac{\partial f}{\partial y}(a,b)+O(\epsilon^2)$$

- Coefficient of ϵ is

$$\frac{\partial f}{\partial x}v_1+\frac{\partial f}{\partial y}v_2$$

- Derivatives evaluated at (a,b)

Directional derivative

- Recall: Gradient $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$
- We have

$$\frac{\partial f}{\partial x} v_1 + \frac{\partial f}{\partial y} v_2$$

$$= \nabla f(a, b) \cdot \mathbf{v}$$

- **Directional derivative** in direction **v** (rate of change in that direction)

Calculating directional derivatives

- `f(Dual(a, v₁), Dual(b, v₂))` calculates $\nabla f(a,b) \cdot \mathbf{v}$!
- $\mathbf{v} = (1,0)$ gives $\partial f / \partial x$
- $\mathbf{v} = (0,1)$ gives $\partial f / \partial y$

Functions $\mathbb{R}^2 \to \mathbb{R}^2$

- For $f : \mathbb{R}^2 \to \mathbb{R}^2$, have $f = (f_1, f_2)$ with $f_i : \mathbb{R}^2 \to \mathbb{R}$
- So can find directional derivatives:

$$f_i(\mathbf{a} + \epsilon\mathbf{v}) = f_i(\mathbf{a}) + \epsilon\nabla f_i(\mathbf{a}) \cdot \mathbf{v}$$

- Think of $\nabla f_i(\mathbf{a})$ as *row* vectors

Jacobian II

- Put ∇f_i together into *matrix*:

$$f(\mathbf{a} + \epsilon \mathbf{v}) = f(\mathbf{a}) + \epsilon \, Df(\mathbf{a}) \cdot \mathbf{v}$$

- $Df(\mathbf{a})$ is **Jacobian matrix** – matrix of partial derivatives $\frac{\partial f_i}{\partial x_j}$
- Coefficient of ϵ is Jacobian–vector product
- Directional derivative for function $f : \mathbb{R}^2 \to \mathbb{R}^2$

Newton in higher dimensions

- Generalise argument for Newton method from 1D to higher dimensions:
- Look for root of $f(\mathbf{x}) = \mathbf{0}$.
- Initial guess $\mathbf{x}_0$
- Let $\mathbf{x}_1 = \mathbf{x}_0 + \boldsymbol{\delta}$; want to find $\boldsymbol{\delta}$

Newton II

- Want $f(\mathbf{x}_1) = f(\mathbf{x}_0 + \boldsymbol{\delta}) = 0$
- Approximate:

$$f(\mathbf{x}_0) + \mathsf{J} \cdot \boldsymbol{\delta} \simeq \mathbf{0}$$

- Where $\mathsf{J} := Df(\mathbf{a})$
- So need to solve *system of linear equations*

$$\mathsf{J} \cdot \boldsymbol{\delta} = -f(\mathbf{a})$$

Linear algebra in Julia

- Given matrix A and vector $\mathbf{b}$
- Solve $\mathsf{A} \cdot \mathbf{x} = \mathbf{b}$ for vector $\mathbf{x}$:

```
x = A \ b
```

- `\` is a kind of “division”

Machine learning: Classification

How can we classify?

- What is **classification**?
- Given input, e.g. photo, **classify** it
- Assign it to one of several **classes** / sets that are distinguished in some way
- E.g.: Classify medical image as damaged vs. healthy tissue
- Output: Integer **label**: 0 or 1 (healthy / damaged)
- How express classification mathematically?

Mathematical description

- Input: Convert matrix (image) to vector of numbers
- *Function* from input vector $\mathbf{x} \in \mathbb{R}^n$ to output
- More useful: *continuous* output $y \in [0, 1]$
- If y closer to 0, output 0
- So classification task is function $f : \mathbb{R}^n \to \mathbb{R}$

Learning from data: Supervised learning

- Traditional: Expert assigns classification based on hand-picked **features**
- Modern: Machine *learns* features and classification from *data*
- I.e. have flexible *model* = **function with parameters**
- Machine **learns** (finds) parameters of model that best fit *data*
- **Supervised learning**: Need **training data** pre-labelled (by human)

Simple case

- Input is one real number, e.g. average color of photo
- Output is 0 for apple, 1 for banana
- Provide:
  - Vector of x_i = colors as input
  - Vector of 0s and 1s as desired output
- How model this?

Artificial neurons

- Artificial “neuron” models real neurons
- Receives inputs and has an output
- How combine inputs x_i?
- Simplest: affine function $\sum_i w_i x_i + b$
- **Weights** w_i; **bias** b
- Need to squash real output to $[0, 1]$

Sigmoid function

- Traditional solution: **sigmoid** ("s-shaped") or **logistic** function
- $\sigma(x) := \frac{1}{1+\mathrm{exp}(-x)}$
- Smooth transition from 0 to 1
- Threshold at $x = 0$
- How move transition point and make more abrupt?

Sigmoid function II

- $\sigma(x; w, b) := \sigma(w * (x - b))$
- Increasing w makes jump narrower
- Transition point at b

Artificial neurons

- **Neuron**: Several inputs, one output
- Simple model:

$$f(\mathbf{x}; \mathbf{w}, b) = \sigma\left(\sum_i w_i x_i + b\right) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

- Often put $x_{n+1} = 1$ and $w_{n+1} = b$
- Get $\sigma(\mathbf{w} \cdot \mathbf{x})$
- A neuron is just a particular function with parameters!
- How *learn* parameter values?

Loss function

- Define a **loss function** $\mathcal{L}$
- Distance of predictions $\hat{y}_i := f(\mathbf{x}_i)$ from true labelled result y_i
- Optimize loss function with respect to parameters
- I.e. *learn* parameters of model that best fits data

More classes

- With 2 classes, only need single scalar output
- With n classes, need way to distinguish between n outputs
- How could we do this?

One-hot vectors

- Usual solution: **one-hot** vectors
- Like Euclidean basis vectors
- $(\mathbf{e}_i)_j = 1$ if $j = i$ and 0 otherwise
- E.g. apple = $(1, 0, 0)$, banana = $(0, 1, 0)$, grape = $(0, 0, 1)$
- Output probability vector, e.g. $(0.4, 0.5, 0.1)$ classified as banana

Neural network layer

- Each neuron has single output
- Need m outputs, so need m neurons
- **Layer**: maps input vector $\mathbf{x}_i \in \mathbb{R}^n$ to m outputs
- $f_i(\mathbf{x}) = \sigma(\mathbf{w}_i \cdot \mathbf{x})$ – neuron i has own weight vector $\mathbf{w}_i = (w_{ij})_{j=1}^n$
- A neural network layer is just a particular kind of function!

What does single layer do?

- Each neuron is independent
- $f_i(\mathbf{x})$ measures **distance from hyperplane**
- Neuron i classifies inputs on opposite sides of **separating hyperplane**

$$\mathbf{w_i} \cdot \mathbf{x} + b_i = 0.5$$

- How obtain function that can classify with a nonlinear separating set?

One layer as a matrix

- Layer is a *function* $\mathbb{R}^n \to \mathbb{R}^m$:

$$f(\mathbf{x}) = \sigma.(\mathsf{W}\,\mathbf{x})$$

- Used Julia "dot notation": σ *is applied to each component*
- W is a matrix; W $\mathbf{x}$ is matrix–vector multiplication
- Each layer: linear transformation W followed by nonlinear transformation σ

Feed-forward neural networks

- “Multi-layer perceptron”: combine (compose) several layers
- E.g. 2 layers with input $\mathbf{x}_0$:

$$\mathbf{x}_1 = \sigma.(\mathsf{W}_1\, \mathbf{x}_0)$$

$$\mathbf{x}_2 = \sigma.(\mathsf{W}_2\, \mathbf{x}_1)$$

- How convert output to probability vector?

Converting to a probability vectory: softmax

- Output is *vector* $\hat{\mathbf{y}}_i$ for input $\mathbf{x}_0$
- Want *probability* to be in each class
- Need to compress vector of outputs to vector of probabilities
- Generalize σ to **softmax**:

$$\mathrm{softmax}(\mathbf{z})_i := \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

Feed-forward neural network

- Put it all together:

$$\hat{\mathbf{y}} = f(\mathbf{x})$$

- Where

$$\mathbf{x}_1 = \sigma.(\mathsf{W}_1 \cdot \mathbf{x}_0)$$

$$\mathbf{x}_2 = \sigma.(\mathsf{W}_2 \cdot \mathbf{x}_1)$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{x}_2)$$

- *A neural network is just a particular kind of function with*

Training

- Gradient descent – gradient of loss function with respect to all parameters
- If large number N of data, calculating gradient is expensive
- Loss function is mean of partial loss functions for each data point

$$\mathcal{L}_i := (y_i - f_{\mathsf{W}}(\mathbf{x}))^2$$

- Instead calculate gradient of only 1 or a few (“batch”) random data points in each step
- **Stochastic gradient descent**

Train–test split

- Use most of data for training
- Retain some to *test* how well model *generalizes* to unseen data
- “Train–test split”

Review

- Higher-dimensional automatic differentiation
- Supervised learning: Classification
- Neurons
- Neural networks