12. Neural networks

Last time

- Automatic differentiation and Newton in higher dimensions
- Classification
- Neurons

Goals for today

- Neural networks
- Stochastic gradient descent
- Training and testing

Recall: Supervised learning

- Goal of **supervised learning**:
- Learn mapping from given inputs and output
- E.g. inputs = images; outputs = labelled categories
- Idea: **Predict** output when given new data
- I.e. should be able to *generalize*
- Example: map image of handwritten digit to correct answer

Supervised learning II

- Inputs: vectors $\mathbf{x}_i$ in $\mathbb{R}^n$
- Desired outputs: numbers or vectors $\mathbf{y}_i$
- Learn function that maps each $\mathbf{x}_i$ to $\mathbf{y}_i$ as closely as possible
- Need **parametrized functions**
- Learn parameter values giving **best fit**

Recall: Artificial neurons

- **Neuron**: Element (function) mapping n inputs to one output:

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

- σ is nonlinear **activation function**, e.g. $\sigma(x) = \frac{1}{1+\exp(-x)}$
- Put $x_{n+1} = 1$ and $w_{n+1} = b$ so $f(\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w} \cdot \mathbf{x})$
- So neuron is function $f : \mathbb{R}^n \to \mathbb{R}$
- Classifies data using hyperplane

Defining a neuron in Julia

- Want to treat neuron as a *function*
- But natural to make a new *type* `Neuron`
- Define `n = Neuron()`
- Now want to call `n` *as if it were a function*, acting on input data `x`:
- `n(x)` should give output of neuron for input vector `x`

Defining a neuron II

- Combine a type and a function: make a type that is **callable**:

```
struct Neuron
    w
    b
end

(n::Neuron)(x) = n.w * x + n.b

n = Neuron(3, 4)
n(5)
```

- Note that this is *different* from a constructor, which is a function with same name as type

Neural networks

- One neuron gives relatively simple function
- Useful once *couple many of them together* into a network with more complex behaviour
- Can show: suitable network structure gives **universal function approximator**
- Any function $\mathbb{R}^n \to \mathbb{R}$ can be closely approximated by a neural network

Loss function

- **Partial loss function $\mathcal{L}_i$:**
- Measures distance of single prediction $\hat{y}_i := f(\mathbf{x}_i)$ from desired result y_i
- E.g. mean-squared error:

$$\mathcal{L}_i(\mathbf{w}) := (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2 = [f(\mathbf{x}_i; \mathbf{w}) - \mathbf{y}_i]^2$$

- Define (total) **loss function** over *all* data:

$$\mathcal{L}(\mathbf{w}) := \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i$$

Minimize!

- We want *best* fit to data
- So *minimize* loss function (distance of prediction from data)
- With respect to *parameter* values $\mathbf{w}_j$ of neuron j for all j
- How should we minimize?

Training

- This is often called **training** a neural network
- Process of “learning” from data
- Run optimization algorithm using data to push network closer and closer to desired results
- Think of as a dynamic process

Optimization algorithms

- There are many optimization algorithms – e.g. book *Algorithms for Optimization* by Kochenderfer & Wheeler
- We will use simplest: **gradient descent**:
- Take step "downhill" by moving all weights – $\mathbf{w}^t$ at time t
- Update weights by small step in direction opposite gradient:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla \mathcal{L}(\mathbf{w}^t)$$

- Will move towards local minimum (hopefully)
- η is **learning rate**: leave fixed or allow to change over time.

Calculating gradient

- Need gradient of $\mathcal{L}$ with respect to *all* parameters
- This is expensive
- Use forward-mode automatic differentiation in problem set 5
- But really should use **backpropagation** = reverse-mode automatic differentiation
- Backpropagation calculates gradient with respect to all parameters in constant multiple of time to calculate function itself!
- How can we reduce the cost of taking gradient of $\mathcal{L}$?

Stochastic gradient descent

- Often have huge data sets with large value of N
- Too expensive to calculate full gradient $\nabla \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \mathcal{L}_i(\mathbf{w})$
- What could we do instead?

Stochastic gradient descent II

- Idea: Don't calculate gradient of full $\mathcal{L}$
- Only use a piece of it
- E.g. calculate $\nabla \mathcal{L}_i$ using *single* data point
- Or use mean over a few data points: a **batch**
- **Stochastic gradient descent**: stochastic estimate of full gradient

Stochastic gradient descent III

- So move **w** in direction that decreases error for one or few data points
- But may increase loss function (total error) over all
- This may actually *help*, e.g. to escape local minima / saddle points

Classifying with >2 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; $\mathsf{W}\,\mathbf{x}$ 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**:

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

Feed-forward neural network

- Put it all together:

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

- Output of one layer is input of next layer:

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

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

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

Train–test split

- Use most of data for training
- Retain some to *test* how well model *generalizes* to unseen data
- "Train–test split"
- Information about how well network is learning:
- Calculate total loss over training samples, and total loss over test samples

Review

- Neural networks
- Stochastic gradient descent
- Training and testing