COMP30027 Machine Learning
“Deep” Learning: Part I

Semester 1, 2018

Jeremy Nicholson & Tim Baldwin & Karin Verspoor

THE UNIVERSITY OF MELBOURNE

Lecture Outline

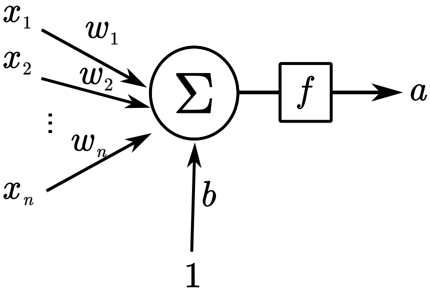
Introduction I

- At the heart of **deep learning** are **neural networks**, which are composed of **neurons**
- A neuron is defined as follows:
  - input = a vector $\mathbf{x_i}$ of numeric inputs ($\langle x_{i1}, x_{i2}, ...x_{in}\rangle \in \mathbb{R}^n$)
  - output = a scalar $a_i \in \mathbb{R}$
  - hyper-parameter: an **activation function** f
  - parameters
    - a vector of weights $\mathbf{w} = \langle b, w_1, w_2, ...w_n\rangle \in \mathbb{R}^{n+1}$, one for each input plus a bias term ($b \equiv w_0$)
- Mathematically:

$$a_i = f\left(\left[\sum_j w_j x_{ij}\right] + b\right) = f(\mathbf{w} \cdot \mathbf{x}_i + b)$$

Introduction II

- Graphically:



Lecture Outline

Training Neural Networks: Perceptron I

- Let's start out with the simple example of a single-neuron neural network (aka a "perceptron"), and the case of a binary classifier, with the activation function:

$$f(\mathbf{w} \cdot \mathbf{x}_i + b) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x}_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Training Neural Networks: Perceptron II

- Training a neural network entails identifying the weights $\boldsymbol{w}$ which minimise the errors on the training data
- The "classic" way to train a neural network is with the **perceptron algorithm**, within which each iteration is termed an **epoch**

Training Neural Networks: Perceptron III

Let $D = \{(\mathbf{x}_i, y_i) | i = 1, 2, ..., N\}$ be the set of training instances

Initialise the weight vector $\mathbf{w}$ randomly

repeat

    for each training instance $(\mathbf{x}_i, y_i) \in D$ **do**

        compute $\hat{y}_i^{(k)} = f(w \cdot \mathbf{x}_i + b)$

        for each each weight w_j **do**

            update $w_j \leftarrow w_j + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}$

        end for

    end for

until stopping condition is met

Perceptron Example I

- Training instances:

$\langle x_{i1}, x_{i2} \rangle$	y_i
$\langle 1, 1 \rangle$	1
$\langle 1, 2 \rangle$	1
$\langle 0, 0 \rangle$	0
$\langle -1, 0 \rangle$	0

- Learning rate $\lambda = 1$

Perceptron Example II

- $w = \langle 0, 0, 0 \rangle$
- Epoch 1:

$\langle x_1, x_2 \rangle$	$b + w_1 \cdot x_1 + w_2 \cdot x_2$	$\hat{y}_i^{(1)}$	y_i
$\langle 1, 1 \rangle$	$0 + 1 \times 0 + 1 \times 0 = 0$	1	1
$\langle 1, 2 \rangle$	$0 + 1 \times 0 + 2 \times 0 = 0$	1	1
$\langle 0, 0 \rangle$	$0 + 0 \times 0 + 0 \times 0 = 0$	1	0
Update to $w = \langle -1, 0, 0 \rangle$			
$\langle -1, 0 \rangle$	$-1 + -1 \times 0 + 0 \times 0 = -1$	0	0

Perceptron Example III

- $w = \langle -1, 0, 0 \rangle$
- Epoch 2:

$\langle x_1, x_2 \rangle$	$b + w \cdot x$	$\hat{y}_i^{(2)}$	y_i
$\langle 1, 1 \rangle$	$-1 + 1 \times 0 + 1 \times 0 = -1$	0	1
	Update to $w = \langle 0, 1, 1 \rangle$		
$\langle 1, 2 \rangle$	$0 + 1 \times 1 + 2 \times 1 = 3$	1	1
$\langle 0, 0 \rangle$	$0 + 0 \times 1 + 0 \times 1 = 0$	1	0
	Update to $w = \langle -1, 1, 1 \rangle$		
$\langle -1, 0 \rangle$	$-1 + -1 \times 1 + 0 \times 1 = -2$	0	0

Perceptron Example IV

- $w = \langle -1, 1, 1 \rangle$
- Epoch 3:

$\langle x_1, x_2 \rangle$	$b + w \cdot x$	$\hat{y}_i^{(3)}$	y_i
$\langle 1, 1 \rangle$	$-1 + 1 \times 1 + 1 \times 1 = 1$	1	1
$\langle 1, 2 \rangle$	$-1 + 1 \times 1 + 2 \times 1 = 2$	1	1
$\langle 0, 0 \rangle$	$-1 + 0 \times 1 + 0 \times 1 = -1$	0	0
$\langle -1, 0 \rangle$	$-1 + -1 \times 1 + 0 \times 1 = -2$	0	0

- Convergence, as no updates throughout epoch

Perceptron: Properties

- The Perceptron algorithm is guaranteed to converge for linearly-separable data, but:
  - the convergence point will depend on the initialisation
  - the convergence point will depend on the learning rate
  - (no guarantee of the margin being maximised)
- No guarantee of convergence over non-linearly separable data
- Possible to extend to multiclass classification problems in a similar manner to logistic regression

Common Activation Functions

- In practice, we use non-linear activation functions (we'll see why in a second), with common such examples being:
  - (logistic) sigmoid ("σ"):

    $$f(x) = \frac{1}{1 + e^{-x}}$$

  - hyperbolic tan ("tanh"):

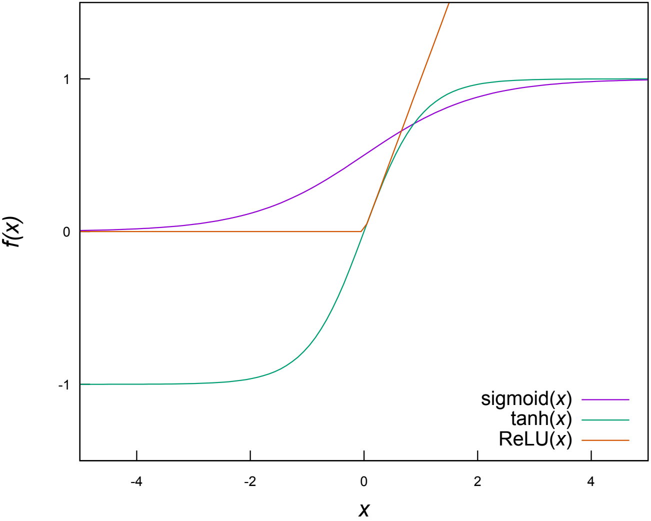
    $$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

  - rectified linear unit ("ReLU"):

    $$f(x) = \max(0, x)$$

    note not differentiable at $x = 0$

Geometry of Activations



Neural Networks and Logistic Regression

- In fact, a neural network with a single neuron and a sigmoid activation (and a binary step function to generate a binary classifier) is equivalent to logistic regression

$$a_i = \frac{1}{1 + e^{-(w \cdot x + b)}}$$

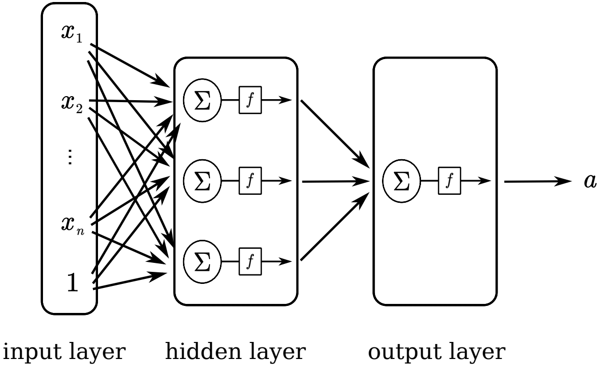
- ... So what's all the fuss about?

Lecture Outline

The Power of Neural Nets: Stacking Neurons I

- The power of neural nets comes from "stacking" multiple neurons together in different ways:
  - **layers** of (parallel) neurons of varying sizes
  - feeding layers into **hidden layers** of varying sizes
- For example, a **fully-connected feed-forward neural network** takes the following form:
  - the **input layer** is made up of the individual features
  - each **hidden layer** is made up an arbitrary number of neurons, each of which is connected to all neurons in the preceding layer, and all neurons in the following layer
  - the **output layer** combines the inputs from the preceding layer into the output
- **Multi-layer perceptron** ("MLP") = full-connected feed-forward neural network with at least one hidden layer

The Power of Neural Nets: Stacking Neurons II



Common Output Layer Activation Functions

- In case of two-class classification:
  - one neuron, with step function, as before
- In case of multiclass classification:
  - multiple perceptrons, with **softmax** (somewhat obnoxious)
  - one perceptron, multiple (binary) output neurons (possibly with softmax)
- In case of regression:
  - identity function (assuming at least one hidden layer ... why?)
  - sigmoid or tanh (possibly with some linear scaling out the other end)

Representational Power of Neural Nets

- The **universal approximation theorem** states that a feed-forward neural network with a single hidden layer (and finite neurons) is able to approximate any continuous function on $\mathbb{R}^n$
- That is, it is possible for a feed-forward neural net with non-linear activation functions to learn any continuous (linear or otherwise) basis function *dynamically*, unlike SVMs e.g., where the kernel is a hyperparameter
- Note that the activation functions **must** by non-linear, as without this, the model is simply a (complex) linear model

How to Train a NN with Hidden Layers I

- All good to here, but the perceptron algorithm can't be used to train neural nets with hidden layers, as we can't directly observe the labels
- Instead, train neural nets with **back propagation**, the details of which are beyond this subject, but intuitively:
  - compute errors at the output layer wrt each weight using partial differentiation
  - propagate those errors back to each of the input layers
- Essentially just gradient descent, but using the chain rule to make the calculations more efficient
- Still have a **learning rate**

How to Train a NN with Hidden Layers II

- Many choices for objective function, of which mean-squared error or residual squared error are the obvious ones

$$
\begin{aligned}
RSS(y, \hat{y}) &= \sum_i (y_i - \hat{y}_i)^2 \\
MSE(y, \hat{y}) &= \frac{1}{N} \sum_i (y_i - \hat{y}_i)^2
\end{aligned}
$$

Lecture Outline

1. Introduction
2. Perceptrons
3. Multi-layer Perceptrons
4. Neural Networks in Practice
5. Summary

Regularisation

- Neural nets are prone to chronic overfitting, due to the large number of parameters, meaning that regularisation is critical
- Common approaches include some combination of:
  - L1/L2 regularisation over weights
  - **early stopping** — stop training when performance plateaus on the dev data
  - **dropout** — randomly drop out a certain proportion of units (inputs and hidden layers) for each instance based on a fixed dropout rate

Theoretical Properties of Neural Networks

- Can be applied to either classification (multiclass or otherwise) or regression problems
- Relies on **continuous** features (but **nominal** features can be trivially binarised)
- Assuming at least one hidden layer (i.e. MLP), can model arbitrary basis functions
- Arbitrarily complex to train, but produces relatively compact models (and reasonably fast at test time)

Practical Properties of Neural Networks

- Large number of parameters, meaning need to be trained over very large amounts of data, in turn meaning slow training times
- Feature engineering less critical ... but architecture engineering much more critical
- Tend to chronically overfit when trained over small datasets
- Contain a number of random variables (initialisation of weights, "mini-batching" in SGD, etc.), with implications for the determinism/stability of the model; in empirical work, tend to average results over multiple runs to ensure robustness
- Very hard to interpret a trained neural net model

But what about "Deep" Learning?

- Deep learning is the combination of "deep" models (i.e. having hidden layers) with sufficient data to train the models
- Deep learning is intimately related to **representation learning**, which will be the topic of the next lecture

Lecture Outline

Summary

- What is a neuron?
- What is the perceptron algorithm?
- What activation functions are commonly used in neural networks?
- What is a (fully-connected) feed-forward neural network?
- What is a hidden layer in a neural network?
- What activation functions are commonly used in the output layer of a neural network?
- What is the universal approximation theorem, and what is its relevance to neural nets?
- How are neural nets trained?