Today’s Lecture

- linear regression
- stochastic gradient descent (SGD)
- backpropagation
- a simple neural network

Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$

Model: $h_\theta(x) = \theta_0 + \theta_1 x$
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$
Model: $h_\theta(x) = \theta_0 + \theta_1 x$

5 10 15 20
Population
0
5
10
15
20
25
Profit

Data

R. Sennrich MT – 2018 – 02
3/21
Linear Regression

Parameters: $\theta = [\theta_0 \ \theta_1]$  
Model: $h_\theta(x) = \theta_0 + \theta_1 x$

The cost (or loss) function

We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \to \mathbb{R}$$
$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$$

Mean Square Error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x(i)) - y(i))^2$$

Mean Square Error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x(i)) - y(i))^2$$

$$= \frac{1}{2m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x(i) - y(i))^2$$
The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:
  \[
  J : \mathbb{R}^2 \to \mathbb{R}, \\
  \hat{\theta} = \arg\min_{\theta \in \mathbb{R}^2} J(\theta)
  \]

- Mean Square Error:
  \[
  J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2
  = \frac{1}{2m} \sum_{i=1}^{m} \left( \theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2
  \]
  where $m$ is the number of data points in the training set.

\[J(\begin{bmatrix} -5.00 \\ 1.50 \end{bmatrix}) = 6.1561\]

\[J(\begin{bmatrix} -6.00 \\ 2.00 \end{bmatrix}) = 19.3401\]

\[J(\begin{bmatrix} -2.50 \\ 1.00 \end{bmatrix}) = 4.7692\]
The cost (or loss) function

So, how do we find \( \hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta) \) computationally?

\[
J(\begin{bmatrix} -3.90 \\ 1.19 \end{bmatrix}) = 4.4775
\]

(Stochastic) gradient descent

\[
\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j
\]

\[
\theta := \begin{bmatrix} -3.90 \\ 1.19 \end{bmatrix}
\]
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 0, \( \alpha = 0.01 \)

\[ J(\theta) \]

\[ \theta_0 \]

\[ \theta_1 \]

\[ 0, 1, 2, 3, 4 \]

\[ 0, 500, 1000, 1500, 2000 \]

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 1, \( \alpha = 0.01 \)

\[ J(\theta) \]

\[ \theta_0 \]

\[ \theta_1 \]

\[ 0, 1, 2, 3, 4 \]

\[ 0, 500, 1000, 1500, 2000 \]

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 20, \( \alpha = 0.01 \)

\[ J(\theta) \]

\[ \theta_0 \]

\[ \theta_1 \]

\[ 0, 1, 2, 3, 4 \]

\[ 0, 500, 1000, 1500, 2000 \]

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 200, \( \alpha = 0.01 \)

\[ J(\theta) \]

\[ \theta_0 \]

\[ \theta_1 \]

\[ 0, 1, 2, 3, 4 \]

\[ 0, 500, 1000, 1500, 2000 \]
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10000, \( \alpha = 0.01 \)

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10000, \( \alpha = 0.005 \)

(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10000, \( \alpha = 0.02 \)

(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10, \( \alpha = 0.025 \)
Backpropagation

How do we calculate $\frac{\partial}{\partial \theta_j} J(\theta)$?

In other words:
how sensitive is the loss function to the change of a parameter $\theta_j$?

why backpropagation?
we could do this by hand for linear regression...
but what about complex functions?
→ propagate error backward
(special case of automatic differentiation)

Computation Graphs

applying chain rule:
$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5$

next, let's use dynamic programming
to avoid re-computing intermediate results...

Christopher Olah
http://colah.github.io/posts/2015-08-Backprop/
applying chain rule:
\[
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5
\]
next, let's use dynamic programming to avoid re-computing intermediate results...

forward-mode differentiation lets us compute partial derivatives \( \frac{\partial x}{\partial b} \) for all nodes \( x \) → still inefficient if you have many inputs

backward-mode differentiation lets us efficiently compute \( \frac{\partial e}{\partial x} \) for all inputs \( x \) in one pass → also known as error backpropagation

To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;
- the gradient(s) of the cost function (if required by the optimization algorithm).
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;
- the gradient(s) of the cost function (if required by the optimization algorithm).
To summarize what we have learned

When approaching a machine learning problem, we need:
- a suitable model; *(here: a linear model)*
- a suitable cost (or loss) function; *(here: mean square error)*
- an optimization algorithm; *(here: a variant of SGD)*
- the gradient(s) of the cost function (if required by the optimization algorithm).

What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, ...)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices
- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components

What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, ...)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices
- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components
  - more complex architectures with *hidden* units (neither input nor output)
  - neural networks typically use non-linear activation functions
An Artificial Neuron

- $x$ is a vector input, $y$ is a scalar output
- $w$ and $b$ are the parameters ($b$ is a bias term)
- $g$ is a (non-linear) activation function

$$y = g(w \cdot x + b)$$

Why Non-linearity?

Functions like XOR cannot be separated by a linear function

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Why Non-linearity?

Functions like XOR cannot be separated by a linear function

XOR

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)

Activation functions

- desirable:
  - differentiable (for gradient-based training)
  - monotonic (for better training stability)
  - non-linear (for better expressivity)

A Simple Neural Network: Maths

we can use linear algebra to formalize our neural network:

the network

\[
    w_1 = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \end{bmatrix}, \quad h_1 = \begin{bmatrix} A \\ B \\ C \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}
\]

\[
    w_2 = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} D \end{bmatrix}
\]

calculation of $x \mapsto y$

\[
    h_1 = \varphi(xw_1)
\]

\[
    y = \varphi(h_1w_2)
\]

A Simple Neural Network: Python Code

```python
import numpy as np

#activation function
def phi(x):
    return np.greater_equal(x,1).astype(int)

def nn(x, w1, w2):
    h1 = phi(np.dot(x, w1))
    y = phi(np.dot(h1, w2))
    return y

c1 = np.array([[1, 0, 0], [0, 0, 1]])
c2 = np.array([[1], [-2], [1]])
x = np.array([1, 0])
print nn(x, c1, c2)
```

R. Sennrich  MT – 2018 – 02 14/21

R. Sennrich  MT – 2018 – 02 15/21

R. Sennrich  MT – 2018 – 02 16/21

R. Sennrich  MT – 2018 – 02 17/21
To address the problem of varying sentence lengths, the DCNN we take the embedding matrix \( w \in \mathbb{R}^d \) of a word in the sentence:

\[
\begin{bmatrix}
w_1 & \ldots & w_s
\end{bmatrix}
\]  

As described in Sect. 2.2. The resulting matrix \( c \) has dimensions \( d \times (s + m - 1) \).

More Complex Architectures

- **Convolutional**
  - Folded
  - Wide convolution \((s = 7)\)
  - Wide convolution \((m = 3)\)
  - k-max pooling \((k = f(s) = 5)\)

- **Recurrent**
  - Hidden layer
  - Output layer

![Diagram of DCNN](http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Practical Considerations

- **efficiency:**
  - GPU acceleration of BLAS operations
  - Perform SGD in mini-batches
- **hyperparameters:**
  - Number and size of layers
  - Minibatch size
  - Learning rate
  - ...  
- **initialisation of weight matrices**
- **stopping criterion**
- **regularization (dropout)**
- **bias units (always-on input)**

Further Reading

- Further reading on backpropagation:

Toolkits for Neural Networks

- **Torch**
  - [http://torch.ch/](http://torch.ch/)
- **TensorFlow**
  - [https://www.tensorflow.org/](https://www.tensorflow.org/)
- **Theano**
  - [http://deeplearning.net/software/theano/](http://deeplearning.net/software/theano/)

There are many more!
some slides borrowed from:

- Sennrich, Birch, and Junczys-Dowmunt (2016): Advances in Neural Machine Translation
- Sennrich and Haddow (2017): Practical Neural Machine Translation