Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons.

![Figure: The basic computational unit of the brain: Neuron](http://cs231n.github.io/neural-networks-1/)

[Pic credit: http://cs231n.github.io/neural-networks-1/]
Inspiration: The Brain

- Neurons receive input signals and accumulate voltage. After some threshold they will fire spiking responses.

[Pic credit: www.moleculardevices.com]
Inspiration: The Brain

- For neural nets, we use a much simpler model neuron, or **unit**:

  \[ y = \sigma(w^\top x + b) \]

- Compare with logistic regression: \( y = \sigma(w^\top x + b) \)

- By throwing together lots of these incredibly simplistic neuron-like processing units, we can do some powerful computations!
Multilayer Perceptrons

- We can connect lots of units together into a directed acyclic graph.
- Typically, units are grouped together into layers.
- This gives a feed-forward neural network. That’s in contrast to recurrent neural networks, which can have cycles.
Multilayer Perceptrons

- Each layer connects \( N \) input units to \( M \) output units.
- In the simplest case, all input units are connected to all output units. We call this a **fully connected layer**. We’ll consider other layer types later.
- Note: the inputs and outputs for a layer are distinct from the inputs and outputs to the network.
- We need to compute \( M \) outputs from \( N \) inputs. We can do so in parallel using matrix multiplication. This means we’ll be using a \( M \times N \) matrix.
- The output units are a function of the input units:
  \[
  y = f(x) = \phi(Wx + b)
  \]
- A multilayer network consisting of fully connected layers is called a **multilayer perceptron**. Despite the name, it has nothing to do with perceptrons!
Multilayer Perceptrons

Some activation functions:

Identity
\[ y = z \]

Rectified Linear Unit (ReLU)
\[ y = \max(0, z) \]

Soft ReLU
\[ y = \log(1 + e^z) \]
Multilayer Perceptrons

Some activation functions:

- **Hard Threshold**
  \[ y = \begin{cases} 
  1 & \text{if } z > 0 \\
  0 & \text{if } z \leq 0 
  \end{cases} \]

- **Logistic**
  \[ y = \frac{1}{1 + e^{-z}} \]

- **Hyperbolic Tangent (tanh)**
  \[ y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]
Each layer computes a function, so the network computes a composition of functions:

\[ h^{(1)} = f^{(1)}(x) = \phi(W^{(1)}x + b^{(1)}) \]
\[ h^{(2)} = f^{(2)}(h^{(1)}) = \phi(W^{(2)}h^{(1)} + b^{(2)}) \]
\[ \vdots \]
\[ y = f^{(L)}(h^{(L-1)}) \]

Or more simply:

\[ y = f^{(L)} \circ \ldots \circ f^{(1)}(x). \]

Neural nets provide modularity: we can implement each layer’s computations as a black box.
Feature Learning

- If task is regression: choose $y = f^{(L)}(h^{(L-1)}) = (w^{(L)})^T h^{(L-1)} + b^{(L)}$
- If task is binary classification: choose $y = f^{(L)}(h^{(L-1)}) = \sigma((w^{(L)})^T h^{(L-1)} + b^{(L)})$
- Neural nets can be viewed as a way of learning features:

![Diagram showing feature learning process.](image)

- The goal:
Feature Learning

- Suppose we’re trying to classify images of handwritten digits. Each image is represented as a vector of $28 \times 28 = 784$ pixel values.
- Each first-layer hidden unit computes $\phi(w_i^T x)$. It acts as a feature detector.
- We can visualize $w$ by reshaping it into an image. Here’s an example that responds to a diagonal stroke.
Feature Learning

Here are some of the features learned by the first hidden layer of a handwritten digit classifier:
Expressive Power

- We’ve seen that there are some functions that linear classifiers can’t represent. Are deep networks any better?

- Suppose a layer’s activation function was the identity, so the layer just computes an affine transformation of the input.
  - We call this a linear layer.

- Any sequence of *linear* layers can be equivalently represented with a single linear layer.

  $\mathbf{y} = \mathbf{W}^{(3)} \mathbf{W}^{(2)} \mathbf{W}^{(1)} \mathbf{x}$

  $\triangleq \mathbf{w}'$

- Deep linear networks are no more expressive than linear regression.
- Linear layers do have their uses.
Expressive Power

- Multilayer feed-forward neural nets with *nonlinear* activation functions are **universal function approximators**: they can approximate any function arbitrarily well.

- This has been shown for various activation functions (thresholds, logistic, ReLU, etc.)
  - Even though ReLU is “almost” linear, it’s nonlinear enough.
Designing a network to classify XOR:

Assume hard threshold activation function
Multilayer Perceptrons

- $h_1$ computes $\mathbb{I}[x_1 + x_2 - 0.5 > 0]$
  - i.e. $x_1$ OR $x_2$
- $h_2$ computes $\mathbb{I}[x_1 + x_2 - 1.5 > 0]$
  - i.e. $x_1$ AND $x_2$
- $y$ computes $\mathbb{I}[h_1 - h_2 - 0.5 > 0] \equiv \mathbb{I}[h_1 + (1 - h_2) - 1.5 > 0]$
  - i.e. $h_1$ AND (NOT $h_2$)
Expressive Power

Universality for binary inputs and targets:

- Hard threshold hidden units, linear output
- Strategy: $2^D$ hidden units, each of which responds to one particular input configuration

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>:</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>:</td>
</tr>
</tbody>
</table>

Only requires one hidden layer, though it needs to be extremely wide.
What about the logistic activation function?

You can approximate a hard threshold by scaling up the weights and biases:

\[ y = \sigma(x) \]

\[ y = \sigma(5x) \]

This is good: logistic units are differentiable, so we can train them with gradient descent.
Expressive Power

- Limits of universality
  - You may need to represent an exponentially large network.
  - How can you find the appropriate weights to represent a given function?
  - If you can learn any function, you’ll just overfit.
  - Really, we desire a *compact* representation.
Training neural networks with backpropagation
Recall: gradient descent moves opposite the gradient (the direction of steepest descent)

Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in *all* the layers

Conceptually, not any different from what we’ve seen so far — just higher dimensional and harder to visualize!

We want to compute the cost gradient \( \frac{dJ}{dw} \), which is the vector of partial derivatives.

This is the average of \( \frac{d\mathcal{L}}{dw} \) over all the training examples, so in this lecture we focus on computing \( \frac{d\mathcal{L}}{dw} \).
We’ve already been using the univariate Chain Rule.

Recall: if $f(x)$ and $x(t)$ are univariate functions, then

$$\frac{d}{dt} f(x(t)) = \frac{df}{dx} \frac{dx}{dt}.$$
Recall: Univariate logistic least squares model

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2}(y - t)^2 \]

Let’s compute the loss derivatives \( \frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b} \)
Univariate Chain Rule

How you would have done it in calculus class

\[ \mathcal{L} = \frac{1}{2} (\sigma(wx + b) - t)^2 \]

\[ \frac{\partial \mathcal{L}}{\partial w} = \frac{\partial}{\partial w} \left[ \frac{1}{2} (\sigma(wx + b) - t)^2 \right] \]

\[ = \frac{1}{2} \frac{\partial}{\partial w} (\sigma(wx + b) - t)^2 \]

\[ = (\sigma(wx + b) - t) \frac{\partial}{\partial w} (\sigma(wx + b) - t) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial w} (wx + b) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b)x \]

\[ \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial b} \left[ \frac{1}{2} (\sigma(wx + b) - t)^2 \right] \]

\[ = \frac{1}{2} \frac{\partial}{\partial b} (\sigma(wx + b) - t)^2 \]

\[ = (\sigma(wx + b) - t) \frac{\partial}{\partial b} (\sigma(wx + b) - t) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial b} (wx + b) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b) \]

What are the disadvantages of this approach?

Univariate Chain Rule

A more structured way to do it

Computing the loss:

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2}(y - t)^2 \]

Computing the derivatives:

\[
\begin{align*}
\frac{d\mathcal{L}}{dy} &= y - t \\
\frac{d\mathcal{L}}{dz} &= \frac{d\mathcal{L}}{dy} \frac{dy}{dz} = \frac{d\mathcal{L}}{dy} \sigma'(z) \\
\frac{\partial \mathcal{L}}{\partial w} &= \frac{d\mathcal{L}}{dz} \frac{dz}{dw} = \frac{d\mathcal{L}}{dz} x \\
\frac{\partial \mathcal{L}}{\partial b} &= \frac{d\mathcal{L}}{dz} \frac{dz}{db} = \frac{d\mathcal{L}}{dz} 
\end{align*}
\]

Remember, the goal isn’t to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.
Univariate Chain Rule

- We can diagram out the computations using a **computation graph**.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.

**Computing the loss:**

\[
\begin{align*}
  z &= wx + b \\
  y &= \sigma(z) \\
  L &= \frac{1}{2}(y - t)^2
\end{align*}
\]
A slightly more convenient notation:

- Use \( \bar{y} \) to denote the derivative \( \frac{d\mathcal{L}}{dy} \), sometimes called the **error signal**.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).

**Computing the loss:**

\[
\begin{align*}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2
\end{align*}
\]

**Computing the derivatives:**

\[
\begin{align*}
\bar{y} &= y - t \\
\bar{z} &= \bar{y} \sigma'(z) \\
\bar{w} &= \bar{z} x \\
\bar{b} &= \bar{z}
\end{align*}
\]
**Problem:** what if the computation graph has **fan-out** \( > 1 \)? This requires the **multivariate Chain Rule**!

### \(L_2\)-Regularized regression

\[
\begin{align*}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2} (y - t)^2 \\
\mathcal{R} &= \frac{1}{2} w^2 \\
\mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda \mathcal{R}
\end{align*}
\]

### Softmax regression

\[
\begin{align*}
z_\ell &= \sum_j w_{\ell j} x_j + b_\ell \\
y_k &= \frac{e^{z_k}}{\sum_\ell e^{z_\ell}} \\
\mathcal{L} &= -\sum_k t_k \log y_k
\end{align*}
\]
Multivariate Chain Rule

- Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$. (All the variables here are scalar-valued.) Then

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

- Example:

$$f(x, y) = y + e^{xy}$$

$x(t) = \cos t$

$y(t) = t^2$

- Plug in to Chain Rule:

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

$$= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t$$
Multivariable Chain Rule

- In the context of backpropagation:

\[
\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}
\]

- In our notation:

\[
\bar{t} = \bar{x} \frac{dx}{dt} + \bar{y} \frac{dy}{dt}
\]
Backpropagation

Full backpropagation algorithm:

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

$v_N$ denotes the variable we’re trying to compute derivatives of (e.g. loss).

Forward pass

For $i = 1, \ldots, N$

Compute $v_i$ as a function of $\text{Pa}(v_i)$

$\overline{v_N} = 1$

Backward pass

For $i = N - 1, \ldots, 1$

$\overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$
Backpropagation

**Example:** univariate logistic least squares regression

Forward pass:

- \( z = wx + b \)
- \( y = \sigma(z) \)
- \( \mathcal{L} = \frac{1}{2}(y - t)^2 \)
- \( \mathcal{R} = \frac{1}{2}w^2 \)
- \( \mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R} \)

Backward pass:

- \( \mathcal{L}_{\text{reg}} = 1 \)
- \( \overline{\mathcal{R}} = \mathcal{L}_{\text{reg}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} = \mathcal{L}_{\text{reg}} \lambda \)
- \( \overline{\mathcal{L}} = \mathcal{L}_{\text{reg}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} = \mathcal{L}_{\text{reg}} \)
- \( \overline{y} = \overline{\mathcal{L}} \frac{d\mathcal{L}}{dy} = \overline{\mathcal{L}} (y - t) \)
- \( \overline{w} = \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{d\mathcal{R}}{dw} = \overline{z} x + \overline{\mathcal{R}} w \)
- \( \overline{b} = \overline{z} \frac{\partial z}{\partial b} = \overline{z} \)
Backpropagation

**Multilayer Perceptron** (multiple outputs):

**Forward pass:**

\[
\begin{align*}
    z_i &= \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \\
    h_i &= \sigma(z_i) \\
    y_k &= \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \\
    \mathcal{L} &= \frac{1}{2} \sum_k (y_k - t_k)^2
\end{align*}
\]

**Backward pass:**

\[
\begin{align*}
    \overline{\mathcal{L}} &= 1 \\
    \overline{y_k} &= \overline{\mathcal{L}} (y_k - t_k) \\
    \overline{w_{ki}^{(2)}} &= \overline{y_k} h_i \\
    \overline{b_k^{(2)}} &= \overline{y_k} \\
    \overline{h_i} &= \sum_k \overline{y_k w_{ki}^{(2)}} \\
    \overline{z_i} &= \overline{h_i} \sigma'(z_i) \\
    \overline{w_{ij}^{(1)}} &= \overline{z_i} x_j \\
    \overline{b_i^{(1)}} &= \overline{z_i}
\end{align*}
\]
Backpropagation

In vectorized form:

Forward pass:

\[ z = W^{(1)}x + b^{(1)} \]
\[ h = \sigma(z) \]
\[ y = W^{(2)}h + b^{(2)} \]
\[ \mathcal{L} = \frac{1}{2} \| t - y \|^2 \]

Backward pass:

\[ \overline{\mathcal{L}} = 1 \]
\[ \overline{y} = \overline{\mathcal{L}} (y - t) \]
\[ \overline{W}^{(2)} = \overline{y}h^\top \]
\[ \overline{b}^{(2)} = \overline{y} \]
\[ \overline{h} = W^{(2)\top} \overline{y} \]
\[ \overline{z} = \overline{h} \circ \sigma'(z) \]
\[ \overline{W}^{(1)} = \overline{zx}^\top \]
\[ \overline{b}^{(1)} = \overline{z} \]
Computational Cost

- Computational cost of forward pass: one *add-multiply operation* per weight
  \[ z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \]

- Computational cost of backward pass: two add-multiply operations per weight
  \[ w_{ki}^{(2)} = y_k h_i \]
  \[ h_i = \sum_k y_k w_{ki}^{(2)} \]

- Rule of thumb: the backward pass is about as expensive as two forward passes.

- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.
Backprop is used to train the overwhelming majority of neural nets today.

- Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.

Despite its practical success, backprop is believed to be neurally implausible.

- No evidence for biological signals analogous to error derivatives.
- Forward & backward weights are tied in backprop.
- Backprop requires synchronous update (1 forward followed by 1 backward).
- All the biologically plausible alternatives we know about learn much more slowly (on computers).