Introduction to Neural Networks

Demonstrating Some Intelligence

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

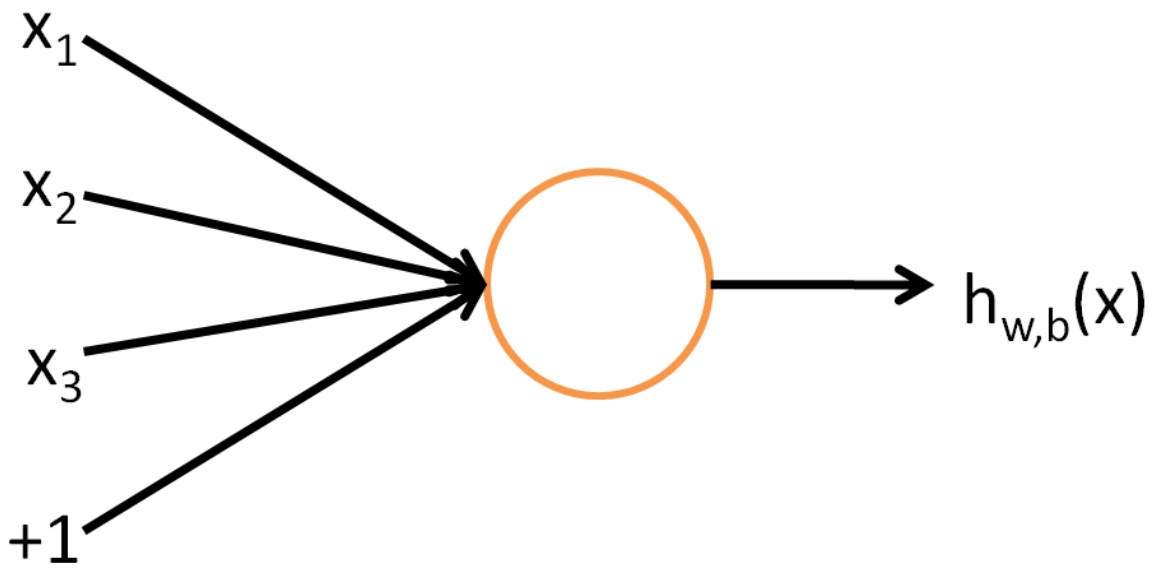
"Mastering the game of Go with Deep Neural Networks and Tree Search", Nature 529, Jan 28, 2016.

Basic Unit

- Consider a supervised learning problem where we have access to labeled training examples $\left(x^{(i)}, y^{(i)}\right)$.
- Neural networks give a way of defining a complex, non-linear form of hypotheses $h_{W,b}(x)$, with parameters W, b that we can fit to our data.

Basic Unit

- We will describe neural network by describing the simplest possible neural network, one which comprises a single “neuron.” The following diagram is used to denote a single neuron:



Basic Unit

- This "neuron" is a computational unit that takes as input x_1, x_2, x_3 (and a +1 intercept term), and outputs $h_{W,b}(x) = f(W^T x) = f\left(\sum_{i=1}^{3} W_i x_i + b\right)$, where $f: \Re \mapsto \Re$ is called the **activation function**
- We will choose $f(\cdot)$ to be the sigmoid function:

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

Basic Unit

- Our single neuron corresponds exactly to the input-output mapping defined by logistic regression.
- Although we will use the sigmoid function, it is worth noting that another common choice for f is the hyperbolic tangent, or tanh, function:

$$f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}.$$

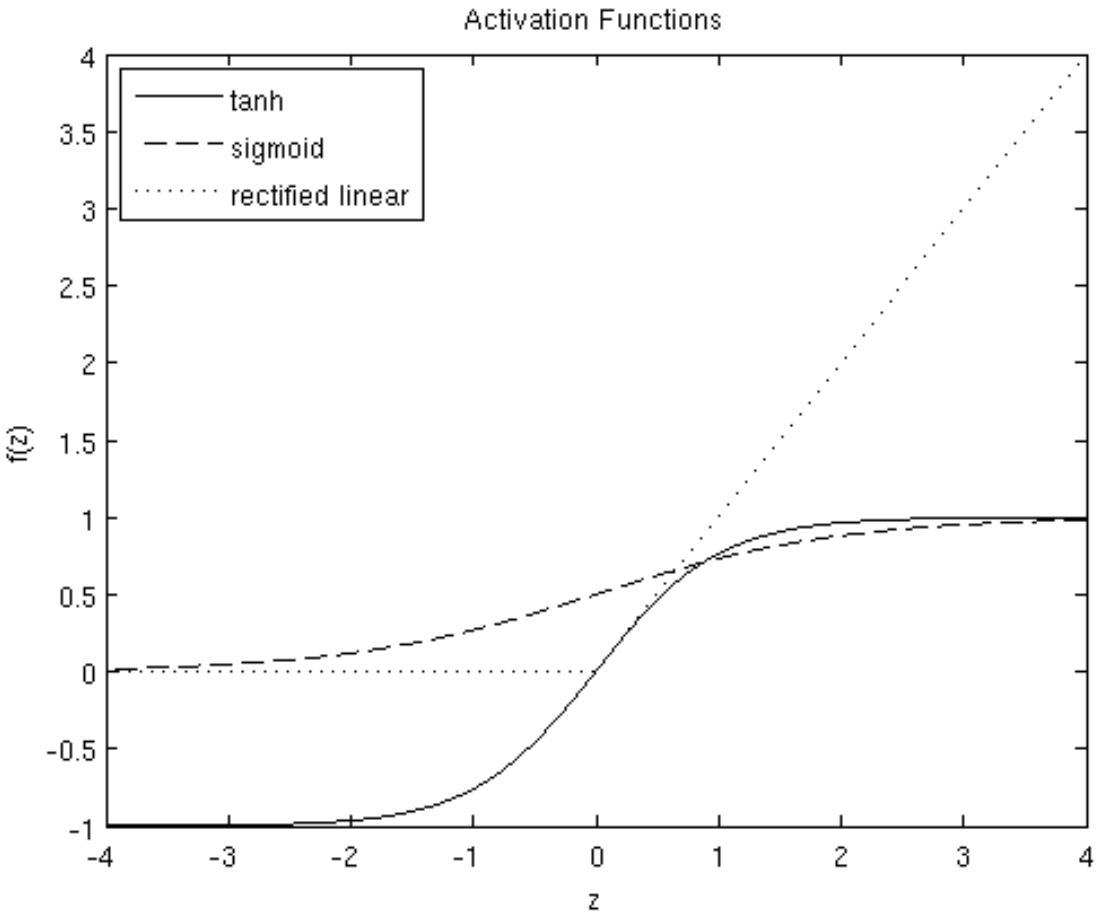
Basic Unit

- Recent research has found a different activation function, the rectified linear function, often works better in practice for deep neural networks.
- This activation function is different from sigmoid and tanh because it is not bounded or continuously differentiable.
- The rectified linear activation function is given by:

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

Basic Unit

The following diagram are plots of the sigmoid, tanh and rectified linear functions.



Basic Unit

- The $\tanh(z)$ function is a rescaled version of the sigmoid, and its output range is $[-1,1]$ instead of $[0,1]$.
- The rectified linear function is piece-wise linear and saturates at exactly 0 whenever the input z is less than 0.

Basic Unit

One identity that'll be useful later:

- If $f(z) = 1/(1 + \exp(-z))$ is the sigmoid function, then its derivative is given by:

$$f'(z) = f(z)\big(1 - f(z)\big).$$

- If f is the tanh function, then its derivative is given by:

$$f'(z) = 1 - \big(f(z)\big)^2$$

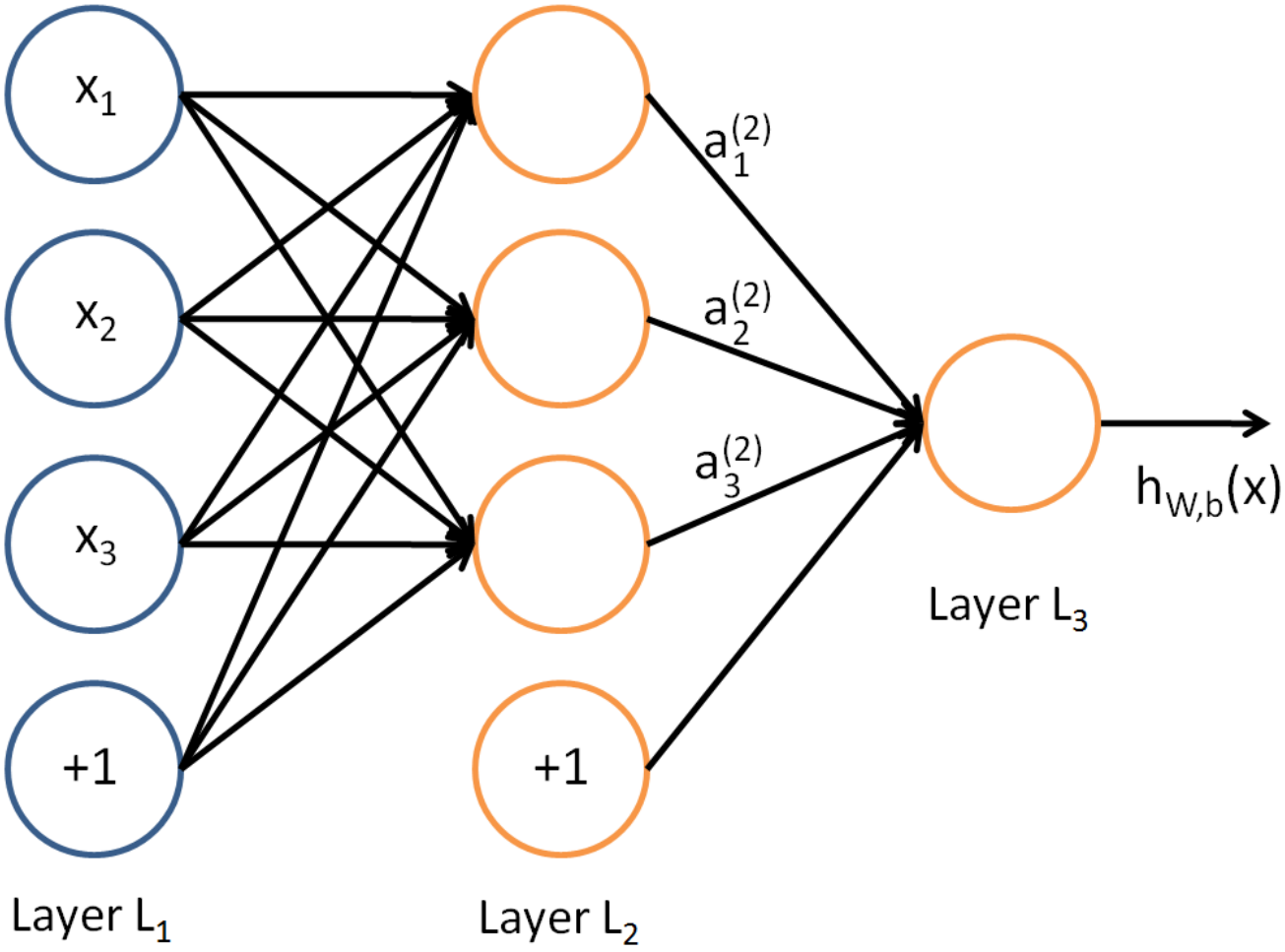
Basic Unit

- The rectified linear function has gradient 0 when $z \leq 0$ and 1 otherwise.
- The gradient is undefined at $z = 0$, though this doesn't cause problems in practice because we average the gradient over many training examples during optimization.

Network Model

- A neural network is put together by hooking together many of our simple “neurons,” so that the output of a neuron can be the input of another.
- For example, the following diagram is a small neural network.

Network Model



Network Model

- Circles denote the inputs to the network. The circles labeled “+1” are called **bias units**, and correspond to the intercept term.
- The leftmost layer of the network is called the **input layer**, and the rightmost layer the **output layer** (In this example has only one node).

Network Model

- The middle layer of nodes is called the **hidden layer**, because its values are not observed in the training set.
- Our example neural network has 3 **input units** (not counting the bias unit), 3 **hidden units**, and 1 **output unit**.

Network Model

- Let n_l denote the number of layers in our network; thus $n_l = 3$ in our example.
- Label layer l as L_l, so layer L_1 is the input layer, and layer L_{n_l} the output layer.

Network Model

- The neural network has parameters $(W, b) = \left(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\right)$, where we write $W_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l + 1$. (Note the order of the indices.)

Network Model

- $b_i^{(l)}$ is the bias associated with unit i in layer $l + 1$. Thus, in our example, we have $W^{(1)} \in \Re^{3x3}$, and $W^{(2)} \in \Re^{3x3}$.
- Note that bias units don't have inputs or connections going into them, since they always output the value +1.
- We also let s_l denote the number of nodes in layer l (not counting the bias unit).

Network Model

- $a_i^{(l)}$ denotes the activation (meaning output value) of unit i in layer l.
- For $l = 1$, we use $a_i^{(l)} = x_i$ to denote the i-th input.
- Given a fixed setting of the parameters W, b, our neural network defines a hypothesis $h_{W,b}(x)$ that outputs a real number.

Network Model

- Specifically, the computation that the previous neural network represents is given by:

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

Network Model

- In the sequel, $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term (e.g., $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}$), so that $a_i^{(l)} = f\left(z_i^{(l)}\right)$.
- Note that this easily lends itself to a more compact notation.

Network Model

- Specifically, if we extend the activation function $f(\cdot)$ to apply to vectors in an element-wise fashion (i.e., $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$), then we can write the previous equations more compactly as follows.

$$z^{(2)} = W^{(1)}x + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$
$$h_{W,b}(x) = a^{(3)} = f(z^{(3)})$$

- We call this step **forward propagation**.

Network Model

- Recalling that we also use $a^{(1)} = x$ to denote the values from the input layer, then given layer l's activations a^{l}, we can compute layer $l + 1$'s activations $a^{(l+1)}$ as:

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$

$$a^{(l+1)} = f(z^{(l+1)})$$

- By organizing our parameters in matrices and using matrix-vector operations, we can take advantage of fast linear algebra routines to quickly perform calculations in our network.

Network Model

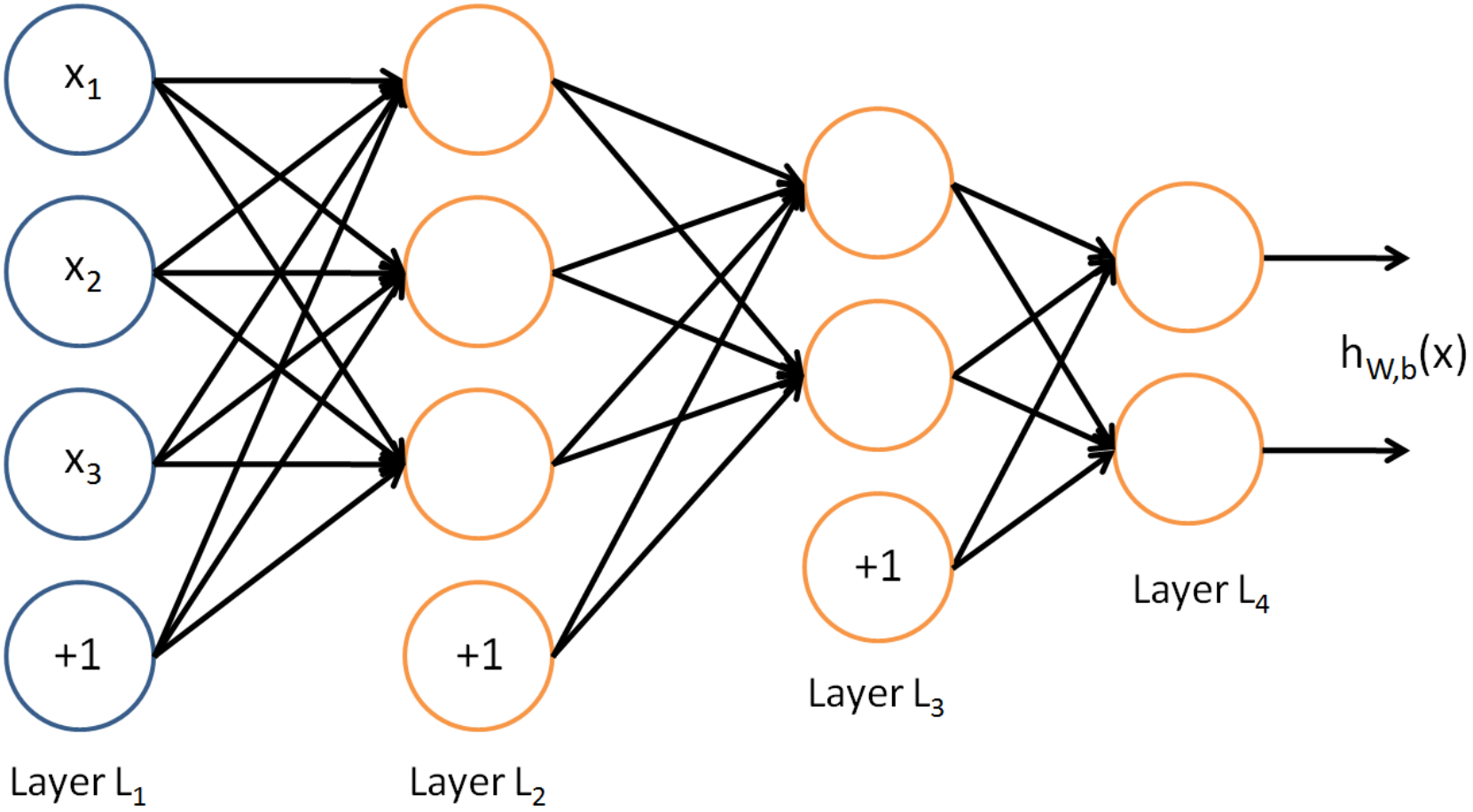
- We focused on one example neural network, but one can also build neural networks with other **architectures** (meaning patterns of connectivity between neurons), including ones with multiple hidden layers.
- The most common choice is a n_l-layered network where layer 1 is the input layer, layer n_l is the output layer, and each layer l is densely connected to layer $l + 1$.

Network Model

- In this setting, to compute the output of the network, we can successively compute all the activations in layer L_2, then layer L_3, and so on, up to layer L_{nl}, using the forward propagation step.
- This is one example of a **feedforward** neural network, since the connectivity graph does not have any directed loops or cycles.

Network Model

Neural networks can also have multiple output units.



Network Model

- To train this network, we need training examples $\left(x^{(i)}, y^{(i)}\right)$ where $y^{(i)} \in \Re^2$.
- This sort of network is useful if there're multiple outputs that you're interested in predicting.
- For example, in a medical diagnosis application, the vector x might give the input features of a patient, and the different outputs y_i's might indicate presence or absence of different diseases.

Backpropagation Algorithm

- Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples. We can train our neural network using batch gradient descent.
- For a single training example(x, y), we define the cost function with respect to that single example to be:

$$J(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2.$$

Backpropagation Algorithm

- This is a (one-half) squared-error cost function. Given a training set of m examples, we then define the overall cost function to be:

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

$$= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}\left(x^{(i)}\right) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

Backpropagation Algorithm

- The first term in the definition of $J(W, b)$ is an average sum-of-squares error term.
- The second term is a regularization term (called a **weight decay** term) that tends to decrease the magnitude of the weights, and helps prevent overfitting.
- Usually weight decay is not applied to the bias terms $b_i^{(l)}$, as reflected in our definition for $J(W, b)$.

Backpropagation Algorithm

- Applying weight decay to the bias units usually makes only a small difference to the final network.
- The **weight decay parameter λ** controls the relative importance of the two terms.
- Note also the slightly overloaded notation: $J(W, b; x, y)$ is the squared error cost with respect to a single example; $J(W, b)$ is the overall cost function, which includes the weight decay term.

Backpropagation Algorithm

- This cost function above is often used both for classification and for regression problems.
- For classification, let $y = 0$ or 1 represent the two class labels (the sigmoid activation function outputs values in $[0,1]$; if we were using a tanh activation function, we would instead use -1 and +1 to denote the labels).

Backpropagation Algorithm

- For regression problems, first scale our outputs to ensure that they lie in the $[0,1]$ range (or if we were using a tanh activation function, then the $[-1,1]$ range).

Backpropagation Algorithm

- Our goal is to minimize $J(W, b)$ as a function of W and b.
- To train our neural network, initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero (say according to a $Normal(0, \epsilon^2)$ distribution for some small ϵ, say 0.01), and then apply an optimization algorithm such as batch gradient descent.

Backpropagation Algorithm

- Since $J(W, b)$ is a non-convex function, gradient descent is susceptible to local optima.
- However, in practice gradient descent usually works fairly well.
- Note that it is important to initialize the parameters randomly, rather than to all 0's.

Backpropagation Algorithm

- If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input (more formally, $W_{ij}^{(1)}$ will be the same for all values of i, so that $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \cdots$ for any input x).
- The random initialization serves the purpose of **symmetry breaking**.

Backpropagation Algorithm

- One iteration of gradient descent updates the parameters W, b as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)$$

where α is the learning rate.

Backpropagation Algorithm

- The key step is computing the partial derivatives .
- We will describe the **backpropagation** algorithm, which gives an efficient way to compute these partial derivatives.
- First describe how backpropagation can be used to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, u)$, the partial derivatives of the cost function $J(W, b; x, y)$ defined with respect to a single example (x, y).

Backpropagation Algorithm

- Once we compute these, we see that the derivative of the overall cost function $J(W,b)$ can be computed as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) \right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_{i}^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_{i}^{(l)}} J(W,b;x^{(i)},y^{(i)})$$

The two lines above differ slightly because weight decay is applied to W but not b.

Backpropagation Algorithm

- The intuition behind the backpropagation algorithm is as follows.
- Given a training example (x, y), we run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$.
- For each node i in layer l, we would compute an "error term" $\delta_i^{(l)}$ that measures how much node was "responsible" for any errors in output.

Backpropagation Algorithm

- For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer).
- For hidden units, we compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

Backpropagation Algorithm

- Here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

Backpropagation Algorithm

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_l+1} W_{ji}^{(l)} \delta_j^{(l+1)}\right) f'(z_i^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}.$$

Backpropagation Algorithm

- We can also re-write the algorithm using matrix-vectorial notation.
- We will use ” •” to denote the element-wise product operator (denoted .* in Matlab or Octave, and also called the Hadamard product), so that if $a = b \bullet$ c, then $a_i = b_i c_i$.
- Similar to how we extended the definition of $f(\cdot)$ to apply element-wise to vectors, we do the same for $f'(\cdot)$. ($\therefore f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$)

Backpropagation Algorithm

- The algorithm can be written:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps
2. For output layer (layer n_l), set

$$\delta_i^{(n_l)} = -\left(y - a^{(n_l)}\right) \bullet f'(z^{(n_l)})$$

Backpropagation Algorithm

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta_i^{(l)} = \left(\left(W^{(l)} \right)^T \delta^{(l+1)} \right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \left(a^{(l)} \right)^T,$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}.$$

Backpropagation Algorithm

- In steps 2 and 3, we need to compute $f'(z_i^{(l)})$ for each value of i.
- Assuming $f(z)$ is the sigmoid activation function, we would already have $a_i^{(l)}$ stored away from the forward pass through the network.
- Using the expression that we worked out earlier for $f'(z)$, we can compute this as $f'\left(z_i^{(l)}\right) = a_i^{(l)}\left(1 - a_i^{(l)}\right)$.

Backpropagation Algorithm

- We are ready to describe the full gradient descent algorithm.
- In the following pseudo-code, $\Delta W^{(l)}$ is a matrix (of the same dimension as $W^{(l)}$), and $\Delta b^{(l)}$ is a vector (of the same dimension as $b^{(l)}$).
- In this notation, "$\Delta W^{(l)}$" is a matrix, and in particular it is not "Δ times $W^{(l)}$."

Backpropagation Algorithm

- We implement one iteration of batch gradient descent as follows:

1. Set $\Delta W^{(l)} := 0, \Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.
2. For $i = 1$ to m,
   1. Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.
   2. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$
   3. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$

Backpropagation Algorithm

3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

- To train our neural network, we can repeatedly take steps of gradient descent to reduce our cost function $J(W, b)$.