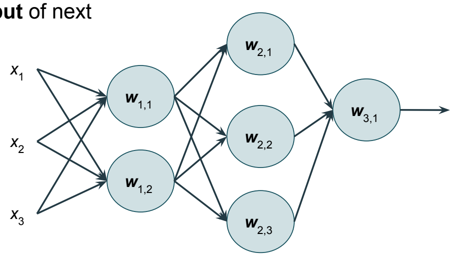
Neural Networks

Supervised Classification so far

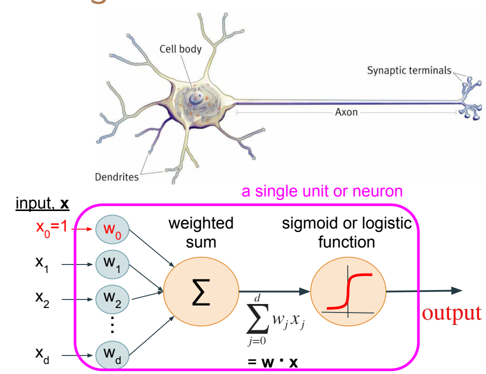
- Linear Classifiers
  - E.g., perceptron, logistic regression
  - Fail when data is not linearly separable
- Non-Linear Classifiers
  - E.g., *k*NN, decision trees or random forests
  - For large datasets: slow testing time (*k*NN), slow training time (decision trees)
- Can we extend linear classifiers to non-linear?

What is an *artificial* neural network?

- Stacked layers of linear classifiers
  - **Output** of each layer is **input** of next
- **Training**
  - Given labeled data and an architecture, learn the weight parameters
- **Prediction**
  - Given all the weight parameters, compute final output (predicted class label)

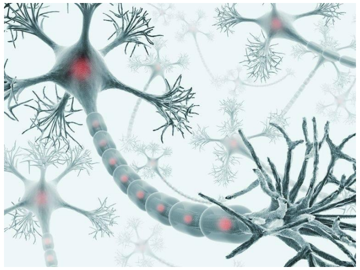


Logistic Regression



$$\text{output} = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}$$

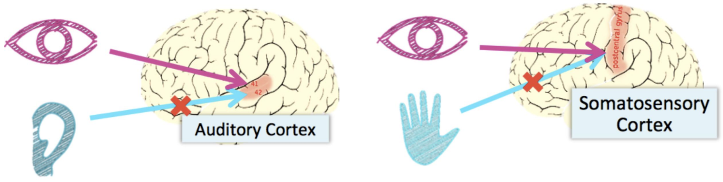
Biological Motivation



- Inspired by brains: each neuron takes inputs from other neurons, passes output to others
- Neurons "learn" from inputs over time

Biological Motivation

- "One Learning Algorithm" hypothesis
  - Any neural network in our brain can learn any functionality



Auditory cortex learns to see
[Roe et al., 1992]

Somatosensory cortex learns to see
[Metin & Frost, 1989]

Biological Motivation

	Computer	Human Brain
Computation Units	10^9 gates	10^{11} neurons
Storage Units	10^9 bits RAM, 10^{12} bits disk	10^{11} neurons, 10^{14} synapses
Cycle Time	10^{-9} seconds	10^{-3} seconds
Bandwidth	10^9 bits/second	10^{14} bits/second

- Computer >> Brain for speed
- Brain >> Computer for parallelism

History of Neural Networks

- McCulloch and Pitts (1943): devise neural networks, invent the perceptron learning algorithm (perceptron = single neuron)
- Widrow and Hoff (1962): simple learning algorithm for neural networks with one hidden layer
- 1986: backpropagation to learn arbitrary network weights
- Late 1980s to late 2000s: research on NNs pauses
  - Slow to train
  - Requires lots of data to prevent overfitting
- Late 2000s: computing power ⬆, data ⬆, training time ⬇, large networks show high prediction accuracies. Rebranded as *deep learning*

Democratization of Deep Learning

Libraries freely available

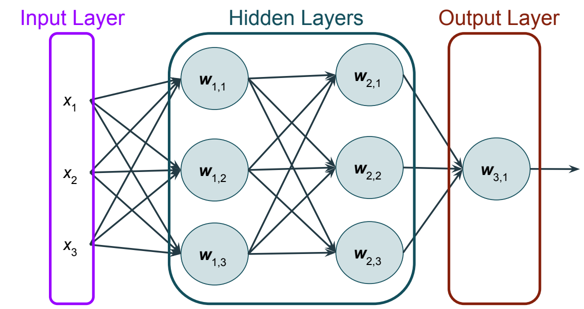
- TensorFlow
- Torch
- Theano
- Caffe
- Keras
- CNTK
- Deeplearning4j

User supplies network architecture and data.

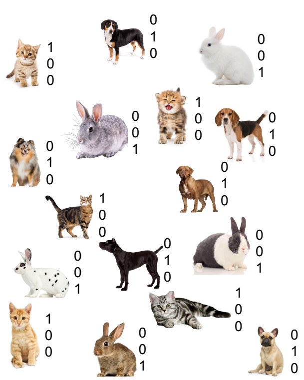
Library performs training with automatic gradient computations.

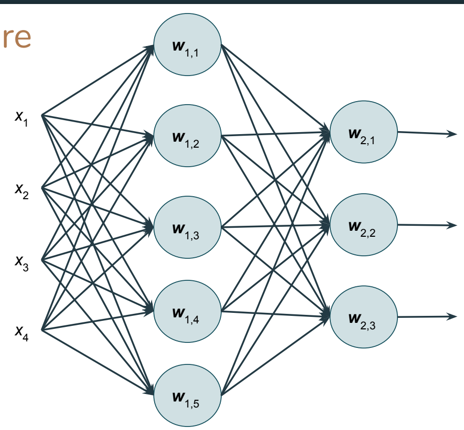
Large networks may require 100s of computers with GPUs and take weeks to train.

Network Architecture



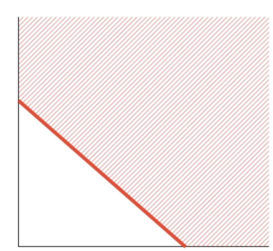
Network Architecture





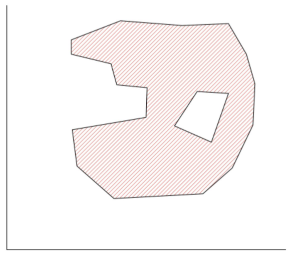
What can neural networks compute?

- Single layer: hyperplane
  (or collection of hyperplanes for multi-class)

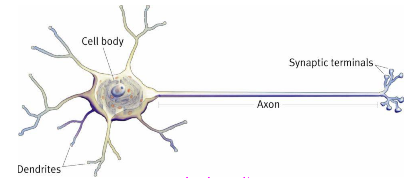


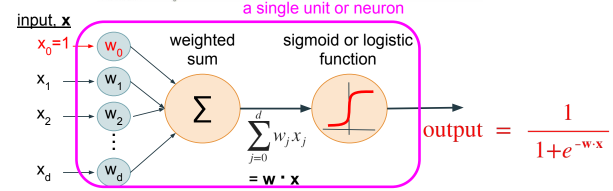
What can neural networks compute?

- More than one layer: anything!
- Two-layer networks = universal function approximators



Logistic Regression

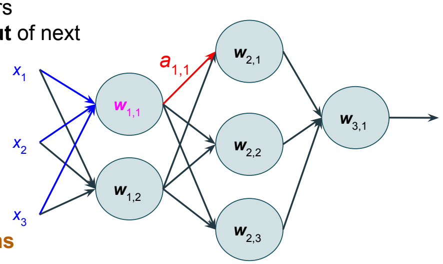




Forward Propagation

$$a_{1,1} = g(\boldsymbol{w}_{1,1} \cdot \boldsymbol{x})$$

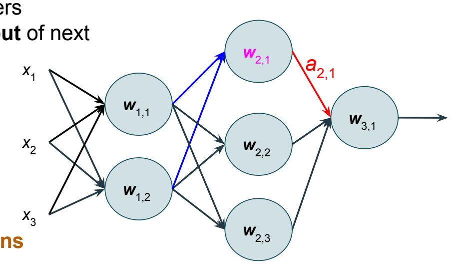
- Stacked layers of linear classifiers
  - **Output** of each layer is **input** of next
- Every output is the result of an activation function applied to the dot product of the weight parameters and the inputs
- Examples of activation functions
  - Sigmoid
  - Tanh
  - Rectified Linear Unit



Forward Propagation

$$a_{2,1} = g(\boldsymbol{w}_{2,1} \cdot \boldsymbol{a}_1)$$

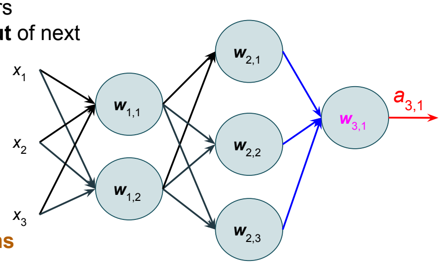
- Stacked layers of linear classifiers
  - **Output** of each layer is **input** of next
- Every output is the result of an activation function applied to the dot product of the weight parameters and the inputs
- Examples of activation functions
  - Sigmoid
  - Tanh
  - Rectified Linear Unit



Forward Propagation

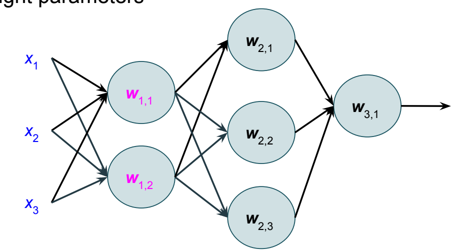
$a_{3,1} = g(\boldsymbol{w}_{3,1} \cdot \boldsymbol{a}_2)$

- Stacked layers of linear classifiers
  - **Output** of each layer is **input** of next
- Every output is the result of an **activation function** applied to the dot product of the weight parameters and the inputs
- Examples of **activation functions**
  - Sigmoid
  - Tanh
  - Rectified Linear Unit



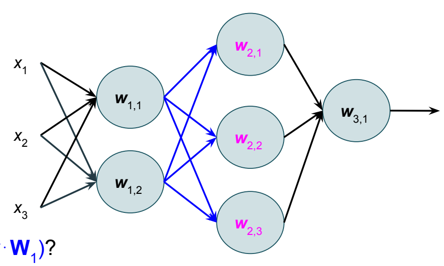
Forward Propagation: Implementation

- Let $\mathbf{W}_1$ be a matrix of all the weight parameters from units in hidden layer 1
- Each column of $\mathbf{W}_1$ corresponds to the weight parameters for one unit
- What is the dimensionality of $\boldsymbol{x}$? Of $\mathbf{W}_1$? Of $\boldsymbol{x} \cdot \mathbf{W}_1$?
- First layer output is $g(\boldsymbol{x} \cdot \mathbf{W}_1)$ where the activation function g is applied to each element in $\boldsymbol{x} \cdot \mathbf{W}_1$



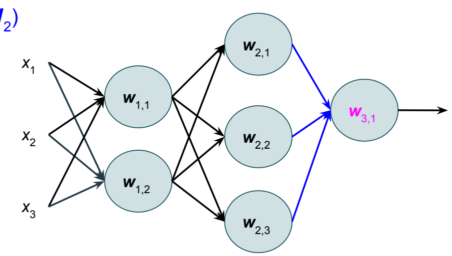
Forward Propagation: Implementation

- First layer output is $g(\boldsymbol{x} \cdot \mathbf{W}_1)$
- Second layer input is $g(\boldsymbol{x} \cdot \mathbf{W}_1)$
- Let $\mathbf{W}_2$ be a matrix of all the weight parameters from units in hidden layer 2
- Each column of $\mathbf{W}_2$ corresponds to the weight parameters for one unit
- What is the dimensionality of $g(\boldsymbol{x} \cdot \mathbf{W}_1)$? Of $\mathbf{W}_2$? Of $g(\boldsymbol{x} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2$?
- Second layer output is $g(g(\boldsymbol{x} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2)$



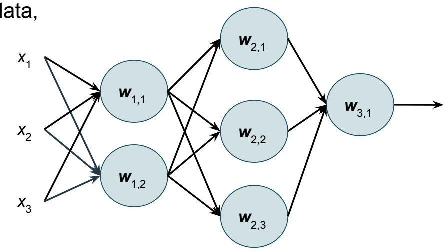
Forward Propagation: Implementation

- Second layer output is $g(g(\boldsymbol{x} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2)$
- Final layer input is $g(g(\boldsymbol{x} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2)$
- Let $\mathbf{W}_3$ be a matrix of all the weight parameters from units in hidden layer 3
- Each column of $\mathbf{W}_3$ corresponds to the weight parameters for one unit
- What is the dimensionality of $g(g(\boldsymbol{x} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2)$? Of $\mathbf{W}_3$? Of $g(g(\boldsymbol{x} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2) \cdot \mathbf{W}_3$?
- Final output is $g(g(g(\boldsymbol{x} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2) \cdot \mathbf{W}_3)$



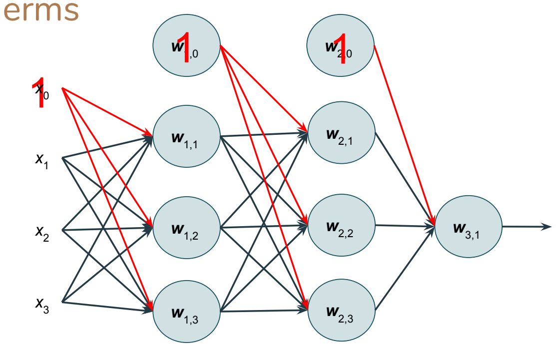
Forward Propagation for a Batch of Data

- Typically, we want to process several data points at once
- Let **X** be an $n \times d$ matrix of input data, where each row is a data point (n data points, d features)
- What is the dimensionality of the first hidden layer output $g(\mathbf{X} \cdot \mathbf{W}_1)$?
- What is the dimensionality of the second hidden layer output $g(g(\mathbf{X} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2)$?



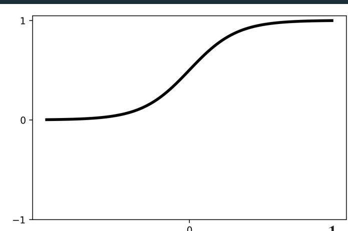
- What is the dimensionality of the final output $g(g(g(\mathbf{X} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2) \cdot \mathbf{W}_3)$?

Bias Terms



Activation Functions

- Sigmoid



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

PROS:

★ Units are analogous to logistic regression

★ 0 to 1 range is biologically nice (neuron either fires or not)

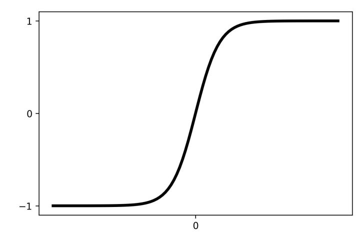
CONS:

➢ Outputs are always positive

➢ Gradient at lower and upper end is almost 0.
When gradients are 0, gradient-based training doesn't progress.

Activation Functions

- Tanh



$$g(x) = \tanh(x)$$

PROS:

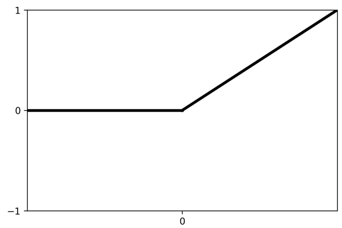
★ Like sigmoid, but outputs can also be negative

CONS:

➢ Gradient at lower and upper end is almost 0.
When gradients are 0, gradient-based training doesn't progress.

Activation Functions

- Rectified Linear Unit



$$g(x) \;=\; \max(0,\; x)$$

PROS:

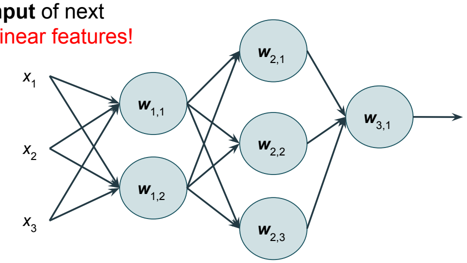
- ★ Easier to compute than sigmoid and tanh (e.g., no exponentiation)
- ★ Gradient does not become 0 at large values

CONS:

- ➢ Outputs are always non-negative, like sigmoid
- ➢ Gradient is 0 at negative values

Neural Network Learning

- Stacked layers of classifiers
  - **Output** of each layer is **input** of next
  - NNs learn their own non-linear features!
- **Training**
  - Given labeled data and an architecture, learn the weight parameters
- **Prediction**
  - Given all the weight parameters, compute final output (predicted class label)



Cost Function

Linear Regression:

$$J(w) \;=\; -\frac{1}{2n}\left[\sum_{i=1}^{n}(h(x^{(i)})-y^{(i)})^2\right]$$

Logistic Regression:

$$J(w) \;=\; -\frac{1}{n}\left[\sum_{i=1}^{n}(y^{(i)}\log(h(x^{(i)}))+(1-y^{(i)})\log(1-h(x^{(i)})))\right]$$

Neural Networks:

$$J(\mathbf{W}) \;=\; -\frac{1}{n}\left[\sum_{i=1}^{n} y^{(i)}\log(h(x^{(i)})) \;+(1-y^{(i)})\log(1-(h(x^{(i)})) \;)\right]$$

Cost Function

Linear Regression:

$$J(w) \;=\; -\frac{1}{2n}\left[\sum_{i=1}^{n}(h(x^{(i)})-y^{(i)})^2\right]$$

Logistic Regression:

$$J(w) \;=\; -\frac{1}{n}\left[\sum_{i=1}^{n}(y^{(i)}\log(h(x^{(i)}))+(1-y^{(i)})\log(1-h(x^{(i)})))\right]$$

Neural Networks (multiclass):

$$J(\mathbf{W}) \;=\; -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)}\log(h(x^{(i)}))_k+(1-y_k^{(i)})\log(1-(h(x^{(i)}))_k)\right]$$

Cost Function with Regularization

Linear Regression:

$$J(w) = -\frac{1}{2n}\left[\sum_{i=1}^{n}(h(x^{(i)})-y^{(i)})^2\right]+\frac{\lambda}{2n}\sum_{j=1}^{d}w_j^2$$

Logistic Regression:

$$J(w) = -\frac{1}{n}\left[\sum_{i=1}^{n}(y^{(i)}\log(h(x^{(i)}))+(1-y^{(i)})\log(1-h(x^{(i)})))\right]+\frac{\lambda}{2n}\sum_{j=1}^{d}w_j^2$$

Neural Networks (multiclass):

$$J(\mathbf{W}) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{k=1}^{K}y_k^{(i)}\log(h(x^{(i)}))_k+(1-y_k^{(i)})\log(1-(h(x^{(i)}))_k)\right]+\frac{\lambda}{2n}\sum_{\text{layers}}\sum_{\substack{\text{units}\\ \text{in layer}}}\sum_{\text{inputs}}w^2$$

Training

We want to find model parameters, i.e., weights for units in our network, that minimize our cost function $J(\mathbf{W})$ on the training data

Gradient Descent:

- Initialize weights to different random values close to 0
- Iteratively update weights in order to reduce the cost

Gradient descent needs to know the gradients, i.e., the partial derivatives of the cost function with respect to the weight parameters.

We use *backpropagation* for this!

Backpropagation

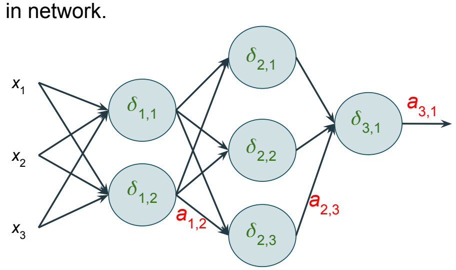
Compute "error" δ for each unit in network.

For example:

$\delta_{3,1}$ is error for $a_{3,1}$

$\delta_{2,3}$ is error for $a_{2,3}$

$\delta_{1,2}$ is error for $a_{1,2}$

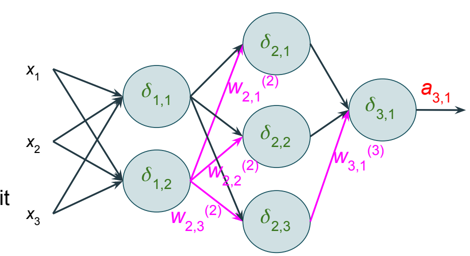


Backpropagation

Compute "error" δ for each unit in network.

For a given training example, first use forward propagation to compute the output a of each unit

Then use backpropagation to compute the error δ of each unit



$\delta_{3,1} = y - a_{3,1}$ $\quad$ $\delta_{2,3} = w_{3,1}^{(3)} \cdot \delta_{3,1}$ $\quad$ $\delta_{1,2} = w_{2,1}^{(2)} \cdot \delta_{2,1} + w_{2,2}^{(2)} \cdot \delta_{2,2} + w_{2,3}^{(2)} \cdot \delta_{2,3}$

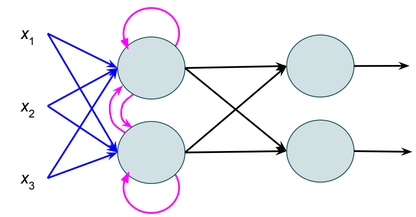
Training

We want to find model parameters, i.e., weights for units in our network, that minimize our cost function $J(\mathbf{W})$ on the training data

Gradient Descent:

- Initialize weights to different random values close to 0
- Iteratively update weights in order to reduce the cost
  - ➔ Use forward propagation to compute the output a of each unit
  - ➔ Use backpropagation to compute the errors δ of each unit
  - ➔ The gradients, i.e., the partial derivatives of the cost function with respect to the weight parameters, are determined from a and δ as $a_L \cdot \delta_{L+1}$

Recurrent Neural Network



- RNNs are a type of neural network designed to recognize patterns in *sequences* of data
- RNNs take as input ***both*** the current data point as well as output of the RNN's previous computation
- RNNs have "memory", i.e., they share weight parameters over time
- For example, if you want to predict the next word in a sentence, it is useful to know which word came before it

Convolutional Neural Network

- CNNs (or ConvNets) are used primarily for image analysis
- In a traditional NN, each input (pixel) is connected to each unit in the first hidden layer, which makes for a lot of parameters to learn
- Traditional NNs do not take spatial structure of data into account
- With CNNs, each unit is connected only to a small local region of the input
- Convolutional layer consists of learnable filters (kernels); each filter is convolved across the input data.

Convolutional Neural Network