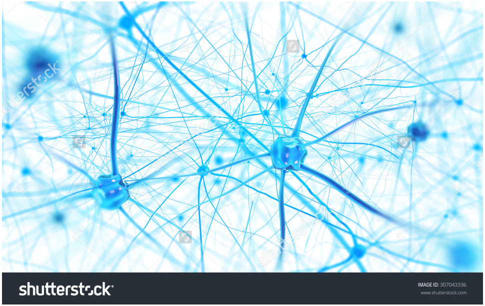
# Feedforward Neural Nets and Backpropagation

Julie Nutini

*University of British Columbia*




MLRG

September $28^{\text{th}}$, 2016

## Supervised Learning Roadmap

- Supervised Learning:
  - Assume that we are given the features $x_i$.
  - Could also use basis functions or kernels.

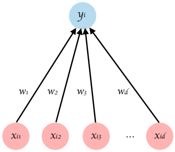
# Supervised Learning Roadmap

- Supervised Learning:
  - Assume that we are given the features $x_i$.
  - Could also use basis functions or kernels.
- Unsupervised Learning:
  - Learn a representation $z_i$ based on features $x_i$.
  - Also used for supervised learning: use $z_i$ as features.

# Supervised Learning Roadmap

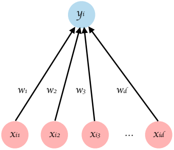
- Supervised Learning:
  - Assume that we are given the features $x_i$.
  - Could also use basis functions or kernels.
- Unsupervised Learning:
  - Learn a representation $z_i$ based on features $x_i$.
  - Also used for supervised learning: use $z_i$ as features.
- Supervised Learning:
  - Learn features $z_i$ that are good for supervised learning.

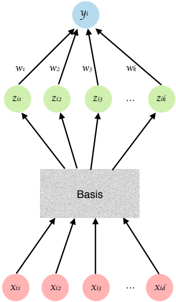
Linear Model



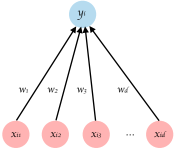
Change of Basis

Linear Model

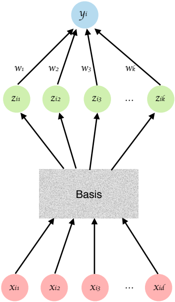


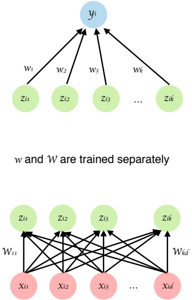
## Linear Model



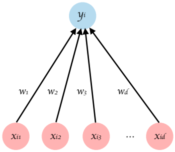

## Change of Basis



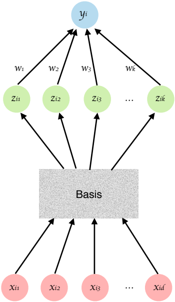

## Basis from Latent-Factor Model



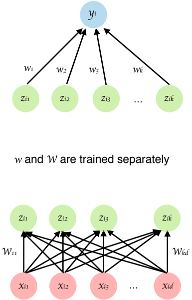
Linear Model



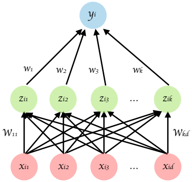

Change of Basis




Basis from Latent-Factor Model

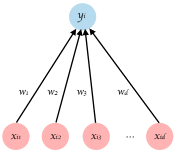



Simultaneously Learn Features for Task and Regression Model

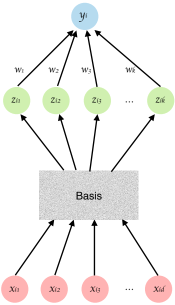


# Supervised Learning Roadmap

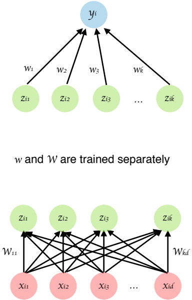
## Linear Model



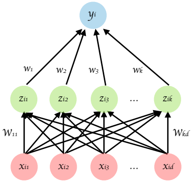

## Change of Basis




## Basis from Latent-Factor Model



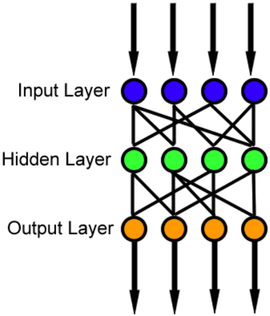

## Simultaneously Learn Features for Task and Regression Model




→ These are all examples of Feedforward Neural Networks.

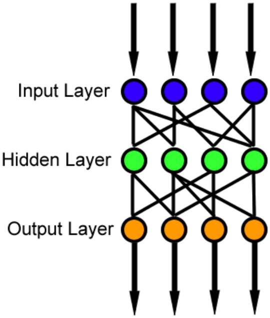
# Feed-forward Neural Network

- Information always moves one direction.
  - No loops.
  - Never goes backwards.
  - Forms a directed acyclic graph.



# Feed-forward Neural Network

- Information always moves one direction.
  - No loops.
  - Never goes backwards.
  - Forms a directed acyclic graph.



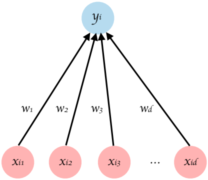

- Each node receives input only from immediately preceding layer.
- Simplest type of artificial neural network.

# Neural Networks - Historical Background

- **1943**: McCulloch and Pitts proposed first computational model of neuron
- **1949**: Hebb proposed the first learning rule
- **1958**: Rosenblatt's work on perceptrons
- **1969**: Minsky and Papert's paper exposed limits of theory
- **1970s**: Decade of dormancy for neural networks
- **1980-90s**: Neural network return (self-organization, back-propagation algorithms, etc.)

# Model of Single Neuron

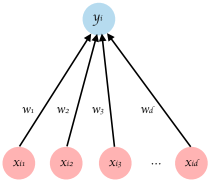
- McCulloch and Pitts (1943): "integrate and fire" model (no hidden layers)




- Denote the $d$ input values for sample $i$ by $x_{i1}, x_{i2}, \ldots, x_{id}$.
- Each of the $d$ inputs has a weight $w_1, w_2, \ldots, w_d$.

# Model of Single Neuron

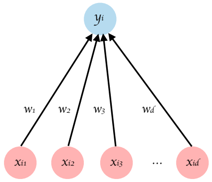
- McCulloch and Pitts (1943): “integrate and fire” model (no hidden layers)




- Denote the $d$ input values for sample $i$ by $x_{i1}, x_{i2}, \ldots, x_{id}$.
- Each of the $d$ inputs has a weight $w_1, w_2, \ldots, w_d$.
- Compute prediction as weighted sum,

$$\hat{y}_i = w_1 x_{i1} + w_2 x_{i2} + \cdots + w_d x_{id} = \sum_{j=1}^{d} w_j x_{ij}$$

- McCulloch and Pitts (1943): “integrate and fire” model (no hidden layers)



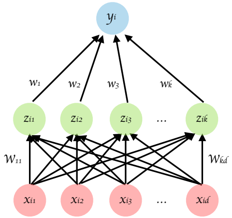

- Denote the $d$ input values for sample $i$ by $x_{i1}, x_{i2}, \ldots, x_{id}$.
- Each of the $d$ inputs has a weight $w_1, w_2, \ldots, w_d$.
- Compute prediction as weighted sum,

$$\hat{y}_i = w_1 x_{i1} + w_2 x_{i2} + \cdots + w_d x_{id} = \sum_{j=1}^{d} w_j x_{ij}$$

- Use $\hat{y}_i$ in some loss function:

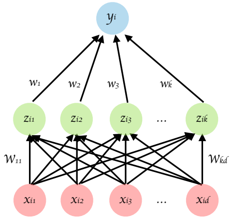
$$\frac{1}{2}(y_i - \hat{y}_i)^2$$

- Consider more than one neuron:



# Incorporating Hidden Layers

- Consider more than one neuron:

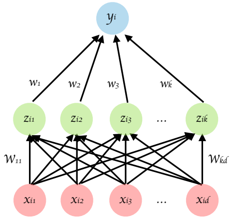



- **Input to hidden layer**: function $h$ of features from latent-factor model:

$$z_i = h(Wx_i).$$

  - Each neuron has directed connection to ALL neurons of a subsequent layer.

- Consider more than one neuron:




- **Input to hidden layer**: function $h$ of features from latent-factor model:

$$z_i = h(Wx_i).$$

  - Each neuron has directed connection to ALL neurons of a subsequent layer.
- This function $h$ is often called the activation function.
  - Each unit/node applies an activation function.

# Linear Activation Function

- A linear activation function has the form

$$h(Wx_i) = a + Wx_i,$$

where $a$ is called bias (intercept).

# Linear Activation Function

- A linear activation function has the form

$$h(Wx_i) = a + Wx_i,$$

  where a is called bias (intercept).
- **Example**: linear regression with linear bias (linear-linear model)
  - Representation: $z_i = h(Wx_i)$ (from latent-factor model)
  - Prediction: $\hat{y}_i = w^T z_i$
  - Loss: $\frac{1}{2}(y_i - \hat{y}_i)^2$

# Linear Activation Function

- A linear activation function has the form

$$h(Wx_i) = a + Wx_i,$$

  where a is called bias (intercept).
- **Example**: linear regression with linear bias (linear-linear model)
  - Representation: $z_i = h(Wx_i)$ (from latent-factor model)
  - Prediction: $\hat{y}_i = w^T z_i$
  - Loss: $\frac{1}{2}(y_i - \hat{y}_i)^2$
- To train this model, we solve:

$$\underset{w \in \mathbb{R}^k, W \in \mathbb{R}^{k \times d}}{\operatorname{argmin}} \quad \underbrace{\frac{1}{2}\sum_{i=1}^{n}(y_i - w^T z_i)^2}_{\text{linear regression with } z_i \text{ as features}} \quad = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T h(Wx_i))^2$$

# Linear Activation Function

- A linear activation function has the form

$$h(Wx_i) = a + Wx_i,$$

where a is called bias (intercept).

- **Example**: linear regression with linear bias (linear-linear model)
  - Representation: $z_i = h(Wx_i)$ (from latent-factor model)
  - Prediction: $\hat{y}_i = w^T z_i$
  - Loss: $\frac{1}{2}(y_i - \hat{y}_i)^2$

- To train this model, we solve:

$$\underset{w \in \mathbb{R}^k, W \in \mathbb{R}^{k \times d}}{\text{argmin}} \quad \underbrace{\frac{1}{2}\sum_{i=1}^{n}(y_i - w^T z_i)^2}_{\text{linear regression with } z_i \text{ as features}} \quad = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T h(Wx_i))^2$$

- But this is just a linear model:

$$w^T(Wx_i) = (W^T w)^T x_i = \tilde{w}^T x_i$$

# Binary Activation Function

- To increase flexibility, something needs to be non-linear.

# Binary Activation Function

- To increase flexibility, something needs to be non-linear.
- A Heaviside step function has the form

$$h(v) = \begin{cases} 1 & \text{if } v \geq a \\ 0 & \text{otherwise} \end{cases}$$

where $a$ is the threshold.

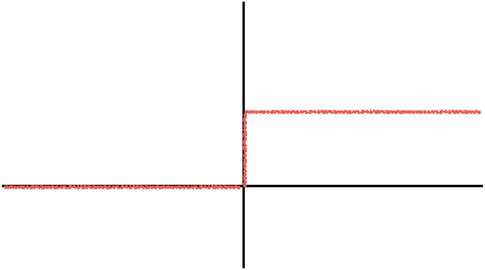
# Binary Activation Function

- To increase flexibility, something needs to be non-linear.
- A Heaviside step function has the form

$$h(v) = \begin{cases} 1 & \text{if } v \geq a \\ 0 & \text{otherwise} \end{cases}$$

  where $a$ is the threshold.

- **Example**: Let $a = 0$,



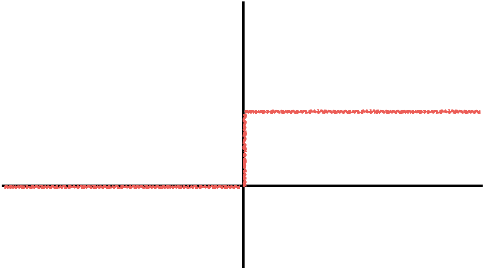
- This yields a binary $z_i = h(Wx_i)$.

# Binary Activation Function

- To increase flexibility, something needs to be non-linear.
- A Heaviside step function has the form

$$h(v) = \begin{cases} 1 & \text{if } v \geq a \\ 0 & \text{otherwise} \end{cases}$$

  where $a$ is the threshold.
- **Example**: Let $a = 0$,



- This yields a binary $z_i = h(Wx_i)$.
- $Wx_i$ has a concept encoded by each of its $2^k$ possible signs.

- Minsky and Papert (late 50s).
- Algorithm for supervised learning of binary classifiers.
  - Decides whether input belongs to a specific class or not.

# Perceptrons

- Minsky and Papert (late 50s).
- Algorithm for supervised learning of binary classifiers.
  - Decides whether input belongs to a specific class or not.
- Uses binary activation function.
  - Can learn “AND”, “OR”, “NOT” functions.

# Perceptrons

- Minsky and Papert (late 50s).
- Algorithm for supervised learning of binary classifiers.
  - Decides whether input belongs to a specific class or not.
- Uses binary activation function.
  - Can learn “AND”, “OR”, “NOT” functions.
- Perceptrons only capable of learning linearly separable patterns.

# Learning Algorithm for Perceptrons

- The perceptron learning rule is given as follows:
  1. For each $x_i$ and desired output $y_i$ in training set,
     1. Calculate output: $\hat{y}_i^{(t)} = h\left(\left(w^{(t)}\right)^T x_i\right)$.

# Learning Algorithm for Perceptrons

- The perceptron learning rule is given as follows:
  1. For each $x_i$ and desired output $y_i$ in training set,
     1. Calculate output: $\hat{y}_i^{(t)} = h\left(\left(w^{(t)}\right)^T x_i\right)$.
     2. Update the weights: $w_j^{(t+1)} = w_j^{(t)} + (y_i - \hat{y}_i^{(t)})x_{ji} \; \forall \; 1 \leq j \leq d$.

# Learning Algorithm for Perceptrons

- The perceptron learning rule is given as follows:
  1. For each $x_i$ and desired output $y_i$ in training set,
     1. Calculate output: $\hat{y}_i^{(t)} = h\left(\left(w^{(t)}\right)^T x_i\right)$.
     2. Update the weights: $w_j^{(t+1)} = w_j^{(t)} + (y_i - \hat{y}_i^{(t)})x_{ji} \; \forall \; 1 \leq j \leq d$.
  2. Continue these steps until $\frac{1}{2}\sum_{i=1}^{n}\left|y_i - \hat{y}_i^{(t)}\right| < \gamma$ (threshold).

# Learning Algorithm for Perceptrons

- The perceptron learning rule is given as follows:
  1. For each $x_i$ and desired output $y_i$ in training set,
     1. Calculate output: $\hat{y}_i^{(t)} = h\left(\left(w^{(t)}\right)^T x_i\right)$.
     2. Update the weights: $w_j^{(t+1)} = w_j^{(t)} + (y_i - \hat{y}_i^{(t)})x_{ji} \; \forall \; 1 \leq j \leq d$.
  2. Continue these steps until $\frac{1}{2}\sum_{i=1}^{n}\left|y_i - \hat{y}_i^{(t)}\right| < \gamma$ (threshold).

- If training set is linearly separable, then guaranteed to converge.

(Rosenblatt, 1962).

# Learning Algorithm for Perceptrons

- The perceptron learning rule is given as follows:
  1. For each $x_i$ and desired output $y_i$ in training set,
     1. Calculate output: $\hat{y}_i^{(t)} = h\left(\left(w^{(t)}\right)^T x_i\right)$.
     2. Update the weights: $w_j^{(t+1)} = w_j^{(t)} + (y_i - \hat{y}_i^{(t)})x_{ji} \;\forall\, 1 \leq j \leq d$.
  2. Continue these steps until $\frac{1}{2}\sum_{i=1}^{n}\left|y_i - \hat{y}_i^{(t)}\right| < \gamma$ (threshold).
- If training set is linearly separable, then guaranteed to converge.
  (Rosenblatt, 1962).
- An upper bound exists on number of times weights adjusted in training.

# Learning Algorithm for Perceptrons

- The perceptron learning rule is given as follows:
  1. For each $x_i$ and desired output $y_i$ in training set,
     1. Calculate output: $\hat{y}_i^{(t)} = h\left(\left(w^{(t)}\right)^T x_i\right)$.
     2. Update the weights: $w_j^{(t+1)} = w_j^{(t)} + (y_i - \hat{y}_i^{(t)})x_{ji} \; \forall \, 1 \leq j \leq d$.
  2. Continue these steps until $\frac{1}{2}\sum_{i=1}^{n} \left| y_i - \hat{y}_i^{(t)} \right| < \gamma$ (threshold).
- If training set is linearly separable, then guaranteed to converge. (Rosenblatt, 1962).
- An upper bound exists on number of times weights adjusted in training.
- Can also use gradient descent if function is differentiable.

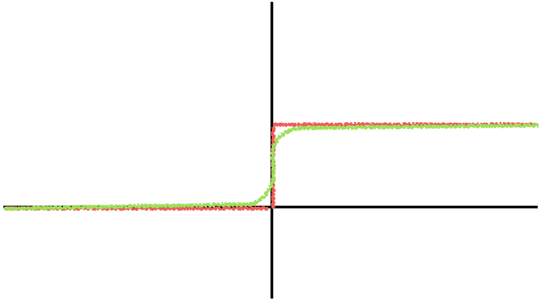
# Non-Linear Continuous Activation Function

- What about a smooth approximation to the step function?

# Non-Linear Continuous Activation Function

- What about a smooth approximation to the step function?
- A sigmoid function has the form

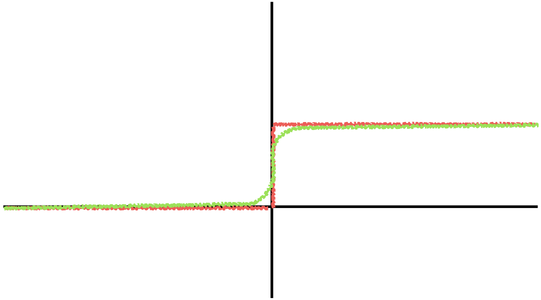
$$h(v) = \frac{1}{1 + e^{-v}}$$



# Non-Linear Continuous Activation Function

- What about a smooth approximation to the step function?
- A sigmoid function has the form

$$h(v) = \frac{1}{1 + e^{-v}}$$

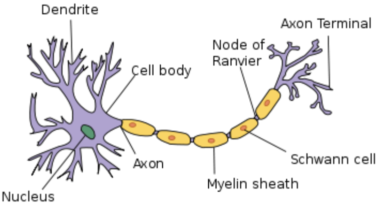


- Applying the sigmoid function element-wise,

$$z_{ic} = \frac{1}{1 + e^{-W_c^T x_i}}.$$

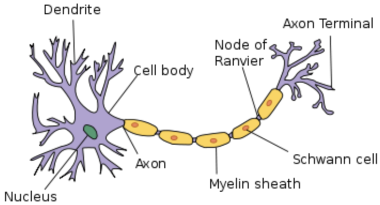
- This is called a multi-layer perceptron or neural network.

- A “typical” neuron.
- Neuron has many dendrites.
  - Each dendrite “takes” input.
- Neuron has a single axon.
  - Axon “sends” output.

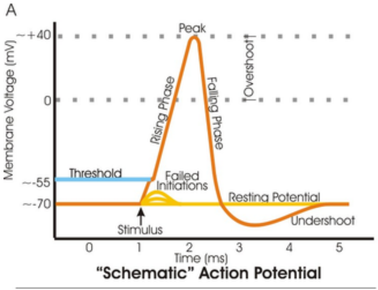


# Why Neural Network?

- A “typical” neuron.
- Neuron has many dendrites.
  - Each dendrite “takes” input.
- Neuron has a single axon.
  - Axon “sends” output.

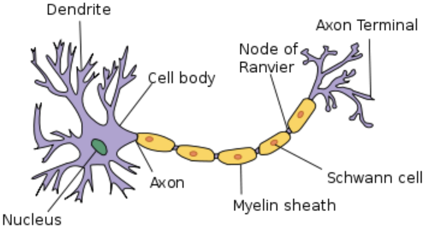



- With the “right” input to dendrites:
  - Action potential along axon.



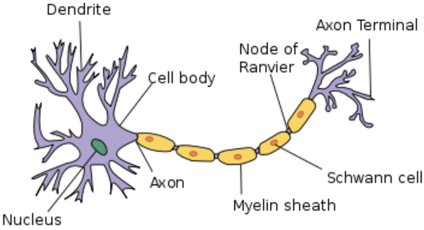
# Why Neural Network?

- First neuron:
  - Each dendrite: $x_{i1}, x_{i1}, \ldots, x_{id}$
  - Nucleus: computes $W_c^T x_i$
  - Axon: sends binary signal $\frac{1}{1+e^{-W_c^T x_i}}$
- Axon terminal: neurotransmitter, synapse
- Second neuron:
  - Each dendrite receives: $z_{i1}, z_{i2}, \ldots, z_{ik}$
  - Nucleus: computes $w^T z_i$
  - Axon: sends signal $\hat{y}_i$



# Why Neural Network?

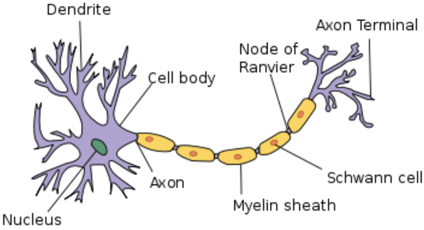
- First neuron:
  - Each dendrite: $x_{i1}, x_{i1}, \ldots, x_{id}$
  - Nucleus: computes $W_c^T x_i$
  - Axon: sends binary signal $\frac{1}{1+e^{-W_c^T x_i}}$
- Axon terminal: neurotransmitter, synapse
- Second neuron:
  - Each dendrite receives: $z_{i1}, z_{i2}, \ldots, z_{ik}$
  - Nucleus: computes $w^T z_i$
  - Axon: sends signal $\hat{y}_i$




- Describes a neural network with one hidden layer (2 neurons).

# Why Neural Network?

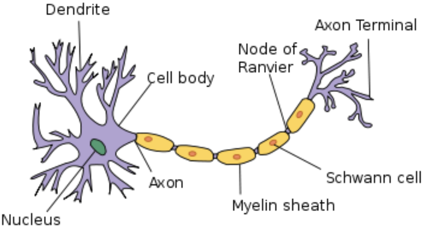
- First neuron:
  - Each dendrite: $x_{i1}, x_{i1}, \ldots, x_{id}$
  - Nucleus: computes $W_c^T x_i$
  - Axon: sends binary signal $\frac{1}{1+e^{-W_c^T x_i}}$
- Axon terminal: neurotransmitter, synapse
- Second neuron:
  - Each dendrite receives: $z_{i1}, z_{i2}, \ldots, z_{ik}$
  - Nucleus: computes $w^T z_i$
  - Axon: sends signal $\hat{y}_i$




- Describes a neural network with one hidden layer (2 neurons).
- Human brain has approximately $10^{11}$ neurons.

# Why Neural Network?

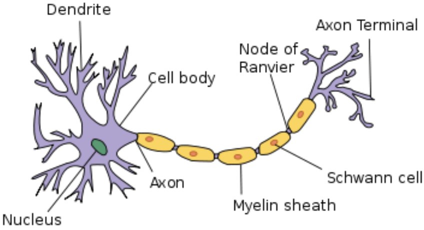
- First neuron:
  - Each dendrite: $x_{i1}, x_{i1}, \ldots, x_{id}$
  - Nucleus: computes $W_c^T x_i$
  - Axon: sends binary signal $\frac{1}{1+e^{-W_c^T x_i}}$
- Axon terminal: neurotransmitter, synapse
- Second neuron:
  - Each dendrite receives: $z_{i1}, z_{i2}, \ldots, z_{ik}$
  - Nucleus: computes $w^T z_i$
  - Axon: sends signal $\hat{y}_i$




- Describes a neural network with one hidden layer (2 neurons).
- Human brain has approximately $10^{11}$ neurons.
- Each neuron connected to approximately $10^4$ neurons.

# Why Neural Network?

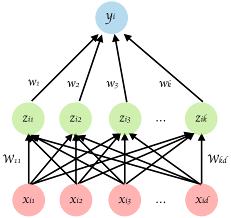
- First neuron:
  - Each dendrite: $x_{i1}, x_{i1}, \ldots, x_{id}$
  - Nucleus: computes $W_c^T x_i$
  - Axon: sends binary signal $\frac{1}{1+e^{-W_c^T x_i}}$
- Axon terminal: neurotransmitter, synapse
- Second neuron:
  - Each dendrite receives: $z_{i1}, z_{i2}, \ldots, z_{ik}$
  - Nucleus: computes $w^T z_i$
  - Axon: sends signal $\hat{y}_i$




- Describes a neural network with one hidden layer (2 neurons).
- Human brain has approximately $10^{11}$ neurons.
- Each neuron connected to approximately $10^4$ neurons.
- ...

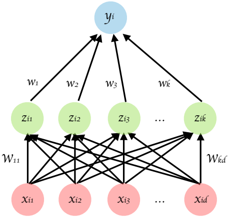
# Artificial Neural Nets vs. Real Neural Nets

- Artificial neural network:
  - $x_i$ is measuring of the world.
  - $z_i$ is internal representation of the world.
  - $\hat{y}_i$ as output of neuron for classification/regression.



# Artificial Neural Nets vs. Real Neural Nets

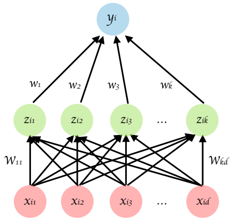
- Artificial neural network:
  - $x_i$ is measuring of the world.
  - $z_i$ is internal representation of the world.
  - $\hat{y}_i$ as output of neuron for classification/regression.




- Real neural networks are more complicated:
  - Timing of action potentials seems to be important.
    - Rate coding: frequency of action potentials simulates continuous output.

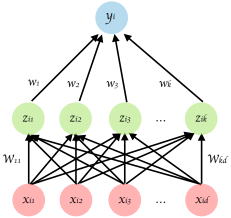
# Artificial Neural Nets vs. Real Neural Nets

- Artificial neural network:
  - $x_i$ is measuring of the world.
  - $z_i$ is internal representation of the world.
  - $\hat{y}_i$ as output of neuron for classification/regression.




- Real neural networks are more complicated:
  - Timing of action potentials seems to be important.
    - Rate coding: frequency of action potentials simulates continuous output.
  - Neural networks don't reflect sparsity of action potentials.

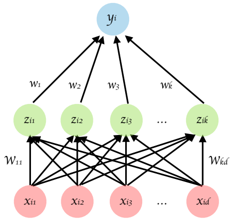
# Artificial Neural Nets vs. Real Neural Nets




- Artificial neural network:
  - $x_i$ is measuring of the world.
  - $z_i$ is internal representation of the world.
  - $\hat{y}_i$ as output of neuron for classification/regression.
- Real neural networks are more complicated:
  - Timing of action potentials seems to be important.
    - Rate coding: frequency of action potentials simulates continuous output.
  - Neural networks don't reflect sparsity of action potentials.
  - How much computation is done inside neuron?

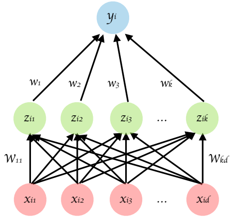
# Artificial Neural Nets vs. Real Neural Nets

- Artificial neural network:
  - $x_i$ is measuring of the world.
  - $z_i$ is internal representation of the world.
  - $\hat{y}_i$ as output of neuron for classification/regression.




- Real neural networks are more complicated:
  - Timing of action potentials seems to be important.
    - Rate coding: frequency of action potentials simulates continuous output.
  - Neural networks don't reflect sparsity of action potentials.
  - How much computation is done inside neuron?
  - Brain is highly organized (e.g., substructures and cortical columns).
  - Connection structure changes.

# Artificial Neural Nets vs. Real Neural Nets




- Artificial neural network:
  - $x_i$ is measuring of the world.
  - $z_i$ is internal representation of the world.
  - $\hat{y}_i$ as output of neuron for classification/regression.

- Real neural networks are more complicated:
  - Timing of action potentials seems to be important.
    - Rate coding: frequency of action potentials simulates continuous output.
  - Neural networks don't reflect sparsity of action potentials.
  - How much computation is done inside neuron?
  - Brain is highly organized (e.g., substructures and cortical columns).
  - Connection structure changes.
  - Different types of neurotransmitters.

- One of the first versions proven by George Cybenko (1989).

# The Universal Approximation Theorem

- One of the first versions proven by George Cybenko (1989).
- For feedforward neural networks, the universal approximation theorem:

  > "... claims that every continuous function defined on a compact set can be arbitrarily well approximated by a neural network with one hidden layer".

# The Universal Approximation Theorem

- One of the first versions proven by George Cybenko (1989).
- For feedforward neural networks, the universal approximation theorem:

  > "... claims that every continuous function defined on a compact set can be arbitrarily well approximated by a neural network with one hidden layer".

- Thus, a simple neural network capable of representing a wide variety of functions when given appropriate parameters.

# The Universal Approximation Theorem

- One of the first versions proven by George Cybenko (1989).
- For feedforward neural networks, the universal approximation theorem:

  > "... claims that every continuous function defined on a compact set can be arbitrarily well approximated by a neural network with one hidden layer".

- Thus, a simple neural network capable of representing a wide variety of functions when given appropriate parameters.
  - "Algorithmic learnability" of those parameters?

- With squared loss, our objective function is:

$$\underset{w \in \mathbb{R}^k, W \in \mathbb{R}^{k \times d}}{\text{argmin}} \frac{1}{2} \sum_{i=1}^{n} (y_i - w^T h(W x_i))^2$$

- With squared loss, our objective function is:

$$\underset{w \in \mathbb{R}^k, W \in \mathbb{R}^{k \times d}}{\text{argmin}} \frac{1}{2} \sum_{i=1}^{n} (y_i - w^T h(W x_i))^2$$

- Usual training procedure: stochastic gradient.
  - Compute gradient of random example $i$, update $w$ and $W$.
- Computing the gradient is known as backpropagation.

- With squared loss, our objective function is:

$$\underset{w \in \mathbb{R}^k, W \in \mathbb{R}^{k \times d}}{\text{argmin}} \frac{1}{2} \sum_{i=1}^{n} (y_i - w^T h(W x_i))^2$$

- Usual training procedure: stochastic gradient.
  - Compute gradient of random example $i$, update $w$ and $W$.
- Computing the gradient is known as backpropagation.
- Adding regularization to $w$ and/or $W$ is known as weight decay.

- Output values $\hat{y}_i$ compared to correct values $y_i$ to compute error.

- Output values $\hat{y}_i$ compared to correct values $y_i$ to compute error.
- Error is fed back through the network (backpropagation of errors).

# Backpropagation

- Output values $\hat{y}_i$ compared to correct values $y_i$ to compute error.
- Error is fed back through the network (backpropagation of errors).
- Weights are adjusted to reduce value of error function by small amount.

- Output values $\hat{y}_i$ compared to correct values $y_i$ to compute error.
- Error is fed back through the network (backpropagation of errors).
- Weights are adjusted to reduce value of error function by small amount.
- After number of cycles, network converges to state where error is small.

- Output values $\hat{y}_i$ compared to correct values $y_i$ to compute error.
- Error is fed back through the network (backpropagation of errors).
- Weights are adjusted to reduce value of error function by small amount.
- After number of cycles, network converges to state where error is small.

- To adjust weights, use non-linear optimization method gradient descent.
  $\rightarrow$ Backpropagation can only be applied to differentiable activation functions.

repeat

**until** some stopping criterion is satisfied

# Backpropagation Algorithm

repeat

**for each** weight $w_{i,j}$ in the network **do**

$w_{i,j} \leftarrow$ a small random number

**for each** example $(x, y)$ **do**

**until** some stopping criterion is satisfied

# Backpropagation Algorithm

repeat

**for each** weight $w_{i,j}$ in the network **do**

$w_{i,j} \leftarrow$ a small random number

**for each** example $(x, y)$ **do**

/*Propagate the inputs forward to compute the outputs*/

/*Propagate the deltas backwards from output layer to input layer*/

/*Update every weight in network using deltas*/

**until** some stopping criterion is satisfied

repeat

**for each** weight $w_{i,j}$ in the network **do**

$w_{i,j} \leftarrow$ a small random number

**for each** example $(x, y)$ **do**

/*Propagate the inputs forward to compute the outputs*/

**for each** node $i$ in the input layer **do**

$a_i \leftarrow x_i$

**for** $\ell = 2$ **to** $L$ **do**

**for each** node $j$ in layer $\ell$ **do**

$v_j \leftarrow \sum_i w_{i,j} a_i$

$a_j \leftarrow h(v_j)$

/*Propagate the deltas backwards from output layer to input layer*/

/*Update every weight in network using deltas*/

**until** some stopping criterion is satisfied

repeat

**for each** weight $w_{i,j}$ in the network **do**

$w_{i,j} \leftarrow$ a small random number

**for each** example $(x, y)$ **do**

/*Propagate the inputs forward to compute the outputs*/

**for each** node $i$ in the input layer **do**

$a_i \leftarrow x_i$

**for** $\ell = 2$ **to** $L$ **do**

**for each** node $j$ in layer $\ell$ **do**

$v_j \leftarrow \sum_i w_{i,j} a_i$

$a_j \leftarrow h(v_j)$

/*Propagate the deltas backwards from output layer to input layer*/

**for each** node $j$ in output layer **do**

$\Delta[j] \leftarrow h'(v_j)(y_j - a_j)$

**for** $\ell = L - 1$ **to** $1$ **do**

**for each** node $i$ in layer $\ell$ **do**

$\Delta[i] \leftarrow h'(v_i) \sum_j w_{i,j} \Delta[j]$

/*Update every weight in network using deltas*/

**until** some stopping criterion is satisfied

# Backpropagation Algorithm

repeat

**for each** weight $w_{i,j}$ in the network **do**

$w_{i,j} \leftarrow$ a small random number

**for each** example $(x, y)$ **do**

/*Propagate the inputs forward to compute the outputs*/

**for each** node $i$ in the input layer **do**

$a_i \leftarrow x_i$

**for** $\ell = 2$ **to** $L$ **do**

**for each** node $j$ in layer $\ell$ **do**

$v_j \leftarrow \sum_i w_{i,j} a_i$

$a_j \leftarrow h(v_j)$

/*Propagate the deltas backwards from output layer to input layer*/

**for each** node $j$ in output layer **do**

$\Delta[j] \leftarrow h'(v_j)(y_j - a_j)$

**for** $\ell = L - 1$ **to** $1$ **do**

**for each** node $i$ in layer $\ell$ **do**

$\Delta[i] \leftarrow h'(v_i) \sum_j w_{i,j} \Delta[j]$

/*Update every weight in network using deltas*/

**for each** weight $w_{i,j}$ in network **do**

$w_{i,j} \leftarrow w_{i,j} + \alpha a_i \Delta[j]$

**until** some stopping criterion is satisfied

# Backpropagation

- Derivations of the back-propagation equations are very similar to the gradient calculation for logistic regression.
  - Uses the multivariable chain rule more than once.

# Backpropagation

- Derivations of the back-propagation equations are very similar to the gradient calculation for logistic regression.
  - Uses the multivariable chain rule more than once.
- Consider the loss for a single node:

$$f(w, W) = \frac{1}{2}(y_i - \sum_{j=1}^{k} w_j h(W_j x_i))^2.$$

- Derivations of the back-propagation equations are very similar to the gradient calculation for logistic regression.
  - Uses the multivariable chain rule more than once.
- Consider the loss for a single node:

$$f(w,W) = \frac{1}{2}(y_i - \sum_{j=1}^{k} w_j h(W_j x_i))^2.$$

- Derivatives with respect to $w_j$ (weights connect hidden to output layer):

$$\frac{\partial f(w,W)}{\partial w_j} = -(y_i - \sum_{j=1}^{k} w_j h(W_j x_i)) \cdot h(W_j x_i)$$

# Backpropagation

- Derivations of the back-propagation equations are very similar to the gradient calculation for logistic regression.
  - Uses the multivariable chain rule more than once.
- Consider the loss for a single node:

$$f(w, W) = \frac{1}{2}(y_i - \sum_{j=1}^{k} w_j h(W_j x_i))^2.$$

- Derivatives with respect to $w_j$ (weights connect hidden to output layer):

$$\frac{\partial f(w, W)}{\partial w_j} = -(y_i - \sum_{j=1}^{k} w_j h(W_j x_i)) \cdot h(W_j x_i)$$

- Derivative with respect to $W_{ij}$ (weights connect input to hidden layer):

$$\frac{\partial f(w, W)}{\partial W_{ij}} = -(y_i - \sum_{j=1}^{k} w_j h(W_j x_i)) \cdot w_j h'(W_j x_i) x_{ij}$$

- Notice repeated calculations in gradients:

$$\frac{\partial f(w,W)}{\partial w_j} = -(y_i - \sum_{j=1}^{k} w_j h(W_j x)) \cdot h(W_j x)$$

$$\frac{\partial f(w,W)}{\partial W_{ij}} = -(y_i - \sum_{j=1}^{k} w_j h(W_j x)) \cdot w_j h'(W_j x) x_i$$

- Notice repeated calculations in gradients:

$$\frac{\partial f(w,W)}{\partial w_j} = -(y_i - \sum_{j=1}^{k} w_j h(W_j x)) \cdot h(W_j x)$$

$$\frac{\partial f(w,W)}{\partial W_{ij}} = -(y_i - \sum_{j=1}^{k} w_j h(W_j x)) \cdot w_j h'(W_j x) x_i$$

- Same value for $\frac{\partial f(w,W)}{\partial w_j}$ with all $j$ and $\frac{\partial f(w,W)}{\partial W_{ij}}$ for all $i$ and $j$.
- Same value for $\frac{\partial f(w,W)}{\partial W_{ij}}$ with all $i$.

## Disadvantages of Neural Networks

- Cannot be used if very little available data.

# Disadvantages of Neural Networks

- Cannot be used if very little available data.
- Many free parameters (e.g., number of hidden nodes, learning rate, minimal error, etc.)

# Disadvantages of Neural Networks

- Cannot be used if very little available data.
- Many free parameters (e.g., number of hidden nodes, learning rate, minimal error, etc.)
- Not great for precision calculations.

# Disadvantages of Neural Networks

- Cannot be used if very little available data.
- Many free parameters (e.g., number of hidden nodes, learning rate, minimal error, etc.)
- Not great for precision calculations.
- Do not provide explanations (unlike slopes in linear models that represent correlations).

# Disadvantages of Neural Networks

- Cannot be used if very little available data.
- Many free parameters (e.g., number of hidden nodes, learning rate, minimal error, etc.)
- Not great for precision calculations.
- Do not provide explanations (unlike slopes in linear models that represent correlations).
- Network overfits training data, fails to capture true statistical process behind the data.
  - Early stopping can be used to avoid this.

## Disadvantages of Neural Networks

- Cannot be used if very little available data.
- Many free parameters (e.g., number of hidden nodes, learning rate, minimal error, etc.)
- Not great for precision calculations.
- Do not provide explanations (unlike slopes in linear models that represent correlations).
- Network overfits training data, fails to capture true statistical process behind the data.
    - Early stopping can be used to avoid this.
- Speed of convergence.
- Local minimia.

# Advantages of Neural Networks

- Easy to maintain.
- Versatile.
- Used on problems for which analytical methods do not yet exist.

# Advantages of Neural Networks

- Easy to maintain.
- Versatile.
- Used on problems for which analytical methods do not yet exist.
- Models non-linear dependencies.
- Quickly work out patterns, even with noisy data.

# Advantages of Neural Networks

- Easy to maintain.
- Versatile.
- Used on problems for which analytical methods do not yet exist.
- Models non-linear dependencies.
- Quickly work out patterns, even with noisy data.
- Always outputs answer, even when input is incomplete.

# Advantages of Neural Networks

- Easy to maintain.
- Versatile.
- Used on problems for which analytical methods do not yet exist.
- Models non-linear dependencies.
- Quickly work out patterns, even with noisy data.
- Always outputs answer, even when input is incomplete.