CS685: Data Mining
Artificial Neural Networks

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs685/

1st semester, 2020-21

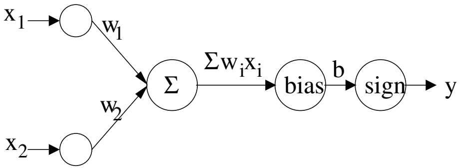
Mon 1030-1200 (online)

Perceptron

- A perceptron is a simple binary *linear* classifier
- Input attributes $x_1, \ldots, x_n$ are weighted and summed
- A bias b is added as well
- Final class ($y = \pm 1$) is *sign* of the output
- The *sign* node is called the activation function

Perceptron

- A perceptron is a simple binary *linear* classifier
- Input attributes $x_1, \ldots, x_n$ are weighted and summed
- A bias b is added as well
- Final class ($y = \pm 1$) is *sign* of the output
- The *sign* node is called the activation function
- It is a simple *linear* classifier
- Decision boundary is $w.x + b = \sum_{i=1}^{n} w_i x_i + b$
- Therefore, sign of $w.x + b$ predicts the class

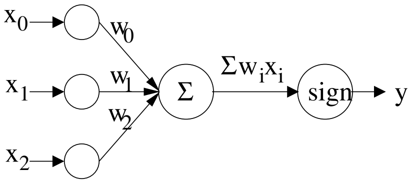


Bias

- Why is bias needed?

Bias

- Why is bias needed?
- Otherwise, hyperplane passes through origin
- Simple trick to model input uniformly: include 1 as x_0 of data
- Decision boundary becomes $w_0 x_0 + w.x = \sum_{i=0}^{n} w_i x_i$
- Weight on $x_0 = 1$ becomes the constant term, i.e., $w_0 = b$



Examples of Perceptrons

- Different boolean functions
- AND (of x_1 and x_2)

Examples of Perceptrons

- Different boolean functions
- AND (of x_1 and x_2)
  - $w_1 = w_2 = 1$, $w_0 = -1.5$
- OR (of x_1 and x_2)

Examples of Perceptrons

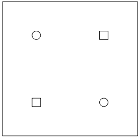
- Different boolean functions
- AND (of x_1 and x_2)
    - $w_1 = w_2 = 1$, $w_0 = -1.5$
- OR (of x_1 and x_2)
    - $w_1 = w_2 = 1$, $w_0 = -0.5$
- NOT (of x_1)

Examples of Perceptrons

- Different boolean functions
- AND (of x_1 and x_2)
  - $w_1 = w_2 = 1$, $w_0 = -1.5$
- OR (of x_1 and x_2)
  - $w_1 = w_2 = 1$, $w_0 = -0.5$
- NOT (of x_1)
  - $w_1 = -1$, $w_0 = 0.5$
- XOR (of x_1 and x_2)

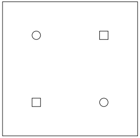
Examples of Perceptrons

- Different boolean functions
- AND (of x_1 and x_2)
  - $w_1 = w_2 = 1$, $w_0 = -1.5$
- OR (of x_1 and x_2)
  - $w_1 = w_2 = 1$, $w_0 = -0.5$
- NOT (of x_1)
  - $w_1 = -1$, $w_0 = 0.5$
- XOR (of x_1 and x_2)
  - Cannot be done as the two classes are not linearly separable



Examples of Perceptrons

- Different boolean functions
- AND (of x_1 and x_2)
  - $w_1 = w_2 = 1,\ w_0 = -1.5$
- OR (of x_1 and x_2)
  - $w_1 = w_2 = 1,\ w_0 = -0.5$
- NOT (of x_1)
  - $w_1 = -1,\ w_0 = 0.5$
- XOR (of x_1 and x_2)
  - Cannot be done as the two classes are not linearly separable



  - Combination of perceptrons can do it, though

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

$$w_i = w_i - \eta(\hat{y}_i - y_i)x_i$$

where $\hat{y}_i$ is the predicted value and η is the *learning rate*

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

$$w_i = w_i - \eta(\hat{y}_i - y_i)x_i$$

  where $\hat{y}_i$ is the predicted value and η is the *learning rate*
- If $y_i = \hat{y}_i$, there is no change in weight

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

$$w_i = w_i - \eta(\hat{y}_i - y_i)x_i$$

  where $\hat{y}_i$ is the predicted value and η is the *learning rate*
- If $y_i = \hat{y}_i$, there is no change in weight
- If $y_i = +1$ and $\hat{y}_i = -1$, i.e., $\hat{y}_i - y_i < 0$
  - Weights of positive x_i are increased and those of negative x_i are decreased thereby pushing $\hat{y}_i$ towards positive

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

$$w_i = w_i - \eta(\hat{y}_i - y_i)x_i$$

where $\hat{y}_i$ is the predicted value and η is the *learning rate*

- If $y_i = \hat{y}_i$, there is no change in weight
- If $y_i = +1$ and $\hat{y}_i = -1$, i.e., $\hat{y}_i - y_i < 0$
  - Weights of positive x_i are increased and those of negative x_i are decreased thereby pushing $\hat{y}_i$ towards positive
- If $y_i = -1$ and $\hat{y}_i = +1$, i.e., $\hat{y}_i - y_i > 0$
  - Weights of positive x_i are decreased and those of negative x_i are increased thereby pushing $\hat{y}_i$ towards negative

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

$$w_i = w_i - \eta(\hat{y}_i - y_i)x_i$$

where $\hat{y}_i$ is the predicted value and η is the *learning rate*

- If $y_i = \hat{y}_i$, there is no change in weight
- If $y_i = +1$ and $\hat{y}_i = -1$, i.e., $\hat{y}_i - y_i < 0$
  - Weights of positive x_i are increased and those of negative x_i are decreased thereby pushing $\hat{y}_i$ towards positive
- If $y_i = -1$ and $\hat{y}_i = +1$, i.e., $\hat{y}_i - y_i > 0$
  - Weights of positive x_i are decreased and those of negative x_i are increased thereby pushing $\hat{y}_i$ towards negative
- If the data *is* linearly separable, a perceptron *will* learn it, i.e., it will converge to the global optimum; otherwise, it may oscillate

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

$$w_i = w_i - \eta(\hat{y}_i - y_i)x_i$$

where $\hat{y}_i$ is the predicted value and η is the *learning rate*

- If $y_i = \hat{y}_i$, there is no change in weight
- If $y_i = +1$ and $\hat{y}_i = -1$, i.e., $\hat{y}_i - y_i < 0$
  - Weights of positive x_i are increased and those of negative x_i are decreased thereby pushing $\hat{y}_i$ towards positive
- If $y_i = -1$ and $\hat{y}_i = +1$, i.e., $\hat{y}_i - y_i > 0$
  - Weights of positive x_i are decreased and those of negative x_i are increased thereby pushing $\hat{y}_i$ towards negative
- If the data *is* linearly separable, a perceptron *will* learn it, i.e., it will converge to the global optimum; otherwise, it may oscillate
- The learning rates η may be modified to give more importance to recent examples

Learning a Perceptron

- What is new in a linear classifier?
- Training a perceptron, i.e., learning the weights w
- Perceptron learning rule or perceptron training rule

$$w_i = w_i - \eta(\hat{y}_i - y_i)x_i$$

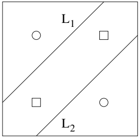
  where $\hat{y}_i$ is the predicted value and η is the *learning rate*
- If $y_i = \hat{y}_i$, there is no change in weight
- If $y_i = +1$ and $\hat{y}_i = -1$, i.e., $\hat{y}_i - y_i < 0$
    - Weights of positive x_i are increased and those of negative x_i are decreased thereby pushing $\hat{y}_i$ towards positive
- If $y_i = -1$ and $\hat{y}_i = +1$, i.e., $\hat{y}_i - y_i > 0$
    - Weights of positive x_i are decreased and those of negative x_i are increased thereby pushing $\hat{y}_i$ towards negative
- If the data *is* linearly separable, a perceptron *will* learn it, i.e., it will converge to the global optimum; otherwise, it may oscillate
- The learning rates η may be modified to give more importance to recent examples
- Weights can be learned through *gradient descent* method as well

Combination of Perceptrons

- A single perceptron can learn only a single hyperplane
- If the data can be separated using two or more hyperplanes, a *combination* of perceptrons can learn it
- Example: XOR

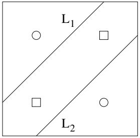
Combination of Perceptrons

- A single perceptron can learn only a single hyperplane
- If the data can be separated using two or more hyperplanes, a *combination* of perceptrons can learn it
- Example: XOR

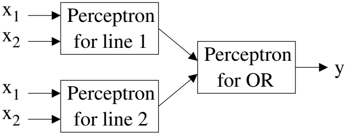


Combination of Perceptrons

- A single perceptron can learn only a single hyperplane
- If the data can be separated using two or more hyperplanes, a *combination* of perceptrons can learn it
- Example: XOR



- Each hyperplane can be modeled by a perceptron and the outputs can be combined using OR

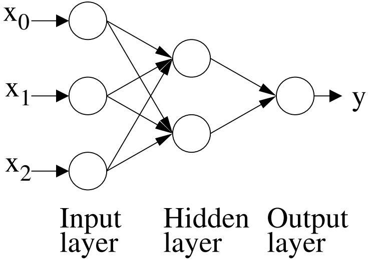


Artificial Neural Networks

- Artificial neural networks (ANNs) are modeled on the human brain
- Nodes have connections ala neurons in human nervous system
- Nodes are also called neurodes

Artificial Neural Networks

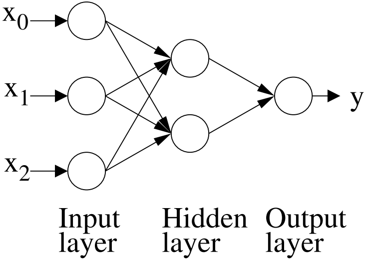
- Artificial neural networks (ANNs) are modeled on the human brain
- Nodes have connections ala neurons in human nervous system
- Nodes are also called neurodes
- Three types of nodes: input layer, *hidden* layer, output layer



- ANN with one hidden layer is considered as *two-layered*
  - Input layer is not counted
- Can have multiple hidden layers – deep neural network (DNN)

Artificial Neural Networks

- Artificial neural networks (ANNs) are modeled on the human brain
- Nodes have connections ala neurons in human nervous system
- Nodes are also called neurodes
- Three types of nodes: input layer, *hidden* layer, output layer



- ANN with one hidden layer is considered as *two-layered*
  - Input layer is not counted
- Can have multiple hidden layers – deep neural network (DNN)
- Learning through ANNs is also called connectionist learning

Connections

- ANNs are of many types depending on the connections
- The most common are multilayer feed-forward networks
  - Connections are directed from one layer *only* to the next
  - There are *no* back edges or same-layer connections

Connections

- ANNs are of many types depending on the connections
- The most common are multilayer feed-forward networks
  - Connections are directed from one layer *only* to the next
  - There are *no* back edges or same-layer connections
- In recurrent networks, there can be back edges or same-layer connections

Connections

- ANNs are of many types depending on the connections
- The most common are multilayer feed-forward networks
  - Connections are directed from one layer *only* to the next
  - There are *no* back edges or same-layer connections
- In recurrent networks, there can be back edges or same-layer connections
- *Fully connected*, i.e., each node in one layer is connected to every node in the next layer

Connections

- ANNs are of many types depending on the connections
- The most common are multilayer feed-forward networks
  - Connections are directed from one layer *only* to the next
  - There are *no* back edges or same-layer connections
- In recurrent networks, there can be back edges or same-layer connections
- *Fully connected*, i.e., each node in one layer is connected to every node in the next layer
- Training is learning the weights on each of these edges
- Akin to layers of perceptrons
- Activation functions in the nodes are *non-linear*

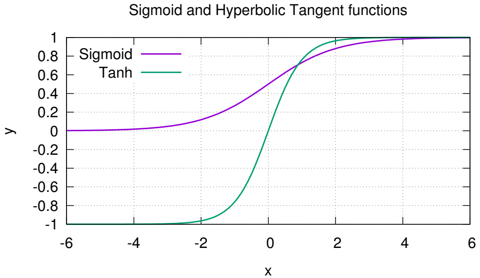
Connections

- ANNs are of many types depending on the connections
- The most common are multilayer feed-forward networks
  - Connections are directed from one layer *only* to the next
  - There are *no* back edges or same-layer connections
- In recurrent networks, there can be back edges or same-layer connections
- *Fully connected*, i.e., each node in one layer is connected to every node in the next layer
- Training is learning the weights on each of these edges
- Akin to layers of perceptrons
- Activation functions in the nodes are *non-linear*
  - Non-linear needed since combination of linear functions can only learn linear separators
- Activation function is sigmoid (logistic) or hyperbolic tangent

Sigmoid and Hyperbolic Tangent Functions

- If input of a node is x, then output y is

$$y = \sigma(x) = \frac{1}{1 + e^{-x}} \qquad y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



- Approximates the step (or sign) function
- Continuous and differentiable
- Output constrained to $(0, 1)$ or $(-1, +1)$
- Scaled versions of each other
- Also called squashing functions

Nodes and Weights

- Output of a node is the sigmoid of the weighted sum of its inputs
- Inputs, in turn, are outputs of previous layers

Nodes and Weights

- Output of a node is the sigmoid of the weighted sum of its inputs
- Inputs, in turn, are outputs of previous layers
- Inputs are normalized to $(0, 1)$
- Outputs are already constrained to $(0, 1)$

Nodes and Weights

- Output of a node is the sigmoid of the weighted sum of its inputs
- Inputs, in turn, are outputs of previous layers
- Inputs are normalized to $(0, 1)$
- Outputs are already constrained to $(0, 1)$
- Weight from a node i to node j is w_{ij}
- Output from node i is O_i
- Input to node j is I_j

$$I_j = \sum_{\forall i} (w_{ij} \cdot O_i)$$

- Output from node j is O_j

$$O_j = \sigma(I_j)$$

Training an ANN

- Training an ANN requires

Training an ANN

- Training an ANN requires
  - Designing the topology: how many layers and many nodes in each layer

Training an ANN

- Training an ANN requires
  - Designing the topology: how many layers and many nodes in each layer
  - Learning the weights of the connections

Training an ANN

- Training an ANN requires
  - Designing the topology: how many layers and many nodes in each layer
  - Learning the weights of the connections
- Designing the topology requires either extensive domain knowledge or simply try-and-test

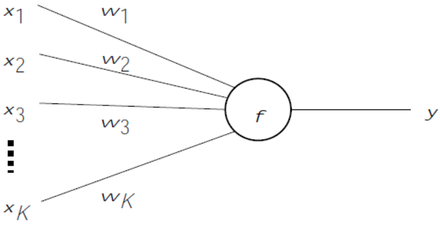
Training an ANN

- Training an ANN requires
  - Designing the topology: how many layers and many nodes in each layer
  - Learning the weights of the connections
- Designing the topology requires either extensive domain knowledge or simply try-and-test
- The final outputs are some non-linear functions of inputs
- Hence, can be trained using gradient descent
- However, it is too complex and slow

Training an ANN

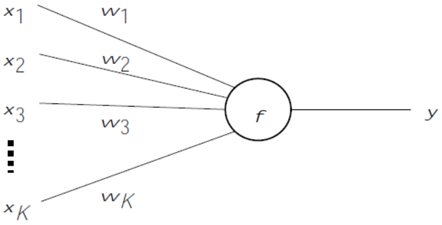
- Training an ANN requires
  - Designing the topology: how many layers and many nodes in each layer
  - Learning the weights of the connections
- Designing the topology requires either extensive domain knowledge or simply try-and-test
- The final outputs are some non-linear functions of inputs
- Hence, can be trained using gradient descent
- However, it is too complex and slow
- Weights are learned through the backpropagation algorithm

Backpropagation



- Main idea
  - Start with arbitrary weights
  - Propagate forward the values
  - Propagate backward the errors
  - Update the weights using gradient descent

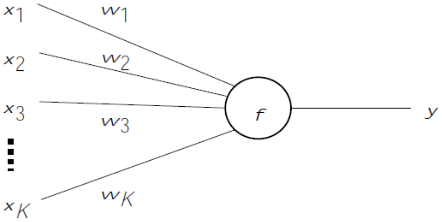
Backpropagation



- Main idea
    - Start with arbitrary weights
    - Propagate forward the values
    - Propagate backward the errors
    - Update the weights using gradient descent
- Output, using sigmoid function $\sigma(u) = \frac{1}{1+e^{-u}}$, is

$$y = f(u) = \sigma\left(\sum_{\forall x_i} w_i.x_i\right)$$

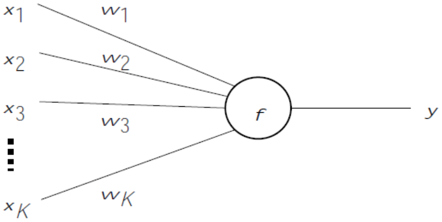
Backpropagation



- Output, using sigmoid function $\sigma(u) = \frac{1}{1+e^{-u}}$, is

$$y = f(u) = \sigma\left(\sum_{\forall x_i} w_i . x_i\right)$$

Backpropagation



- Output, using sigmoid function $\sigma(u) = \frac{1}{1+e^{-u}}$, is

$$y = f(u) = \sigma\left(\sum_{\forall x_i} w_i . x_i\right)$$

- Properties of sigmoid function

$$\frac{d\sigma(u)}{du} = \sigma(u)\sigma(-u) = \sigma(u)(1 - \sigma(u))$$

Updating Weights

- If true output is t, error is squared error

$$E = \frac{1}{2}(t - y)^2$$

Updating Weights

- If true output is t, error is squared error

$$E = \frac{1}{2}(t - y)^2$$

- *Stochastic gradient descent*
- Error function with respect to a single weight w_i

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial w_i} = (y - t).y(1 - y).x_i$$

Updating Weights

- If true output is t, error is squared error

$$E = \frac{1}{2}(t - y)^2$$

- *Stochastic gradient descent*
- Error function with respect to a single weight w_i

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y}.\frac{\partial y}{\partial u}.\frac{\partial u}{\partial w_i} = (y - t).y(1 - y).x_i$$

- Therefore, update equation is

$$w_i^{(new)} = w_i^{(old)} - \eta.(y - t).y(1 - y).x_i$$

Updating Weights

- If true output is t, error is squared error

$$E = \frac{1}{2}(t - y)^2$$

- *Stochastic gradient descent*
- Error function with respect to a single weight w_i

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} . \frac{\partial y}{\partial u} . \frac{\partial u}{\partial w_i} = (y - t).y(1 - y).x_i$$

- Therefore, update equation is

$$w_i^{(new)} = w_i^{(old)} - \eta.(y - t).y(1 - y).x_i$$

- Learning rate or *momentum* η controls the speed of training
- Can be continued till error is below a threshold

Representational Power of an ANN

- Boolean functions

Representational Power of an ANN

- Boolean functions
  - Can approximate *any* finite Boolean function with one layer of hidden nodes
  - Number of hidden nodes may be equal to exponential factor of number of Boolean variables
  - Output layer wired through AND or OR

Representational Power of an ANN

- Boolean functions
  - Can approximate *any* finite Boolean function with one layer of hidden nodes
  - Number of hidden nodes may be equal to exponential factor of number of Boolean variables
  - Output layer wired through AND or OR
- Continuous functions

Representational Power of an ANN

- Boolean functions
  - Can approximate *any* finite Boolean function with one layer of hidden nodes
  - Number of hidden nodes may be equal to exponential factor of number of Boolean variables
  - Output layer wired through AND or OR
- Continuous functions
  - Can approximate *any* continuous function with one layer of hidden nodes up to any arbitrary error factor
  - Due to the properties of sigmoid functions
  - Number of hidden nodes may be large

Representational Power of an ANN

- Boolean functions
  - Can approximate *any* finite Boolean function with one layer of hidden nodes
  - Number of hidden nodes may be equal to exponential factor of number of Boolean variables
  - Output layer wired through AND or OR
- Continuous functions
  - Can approximate *any* continuous function with one layer of hidden nodes up to any arbitrary error factor
  - Due to the properties of sigmoid functions
  - Number of hidden nodes may be large
- Arbitrary functions

Representational Power of an ANN

- Boolean functions
  - Can approximate *any* finite Boolean function with one layer of hidden nodes
  - Number of hidden nodes may be equal to exponential factor of number of Boolean variables
  - Output layer wired through AND or OR
- Continuous functions
  - Can approximate *any* continuous function with one layer of hidden nodes up to any arbitrary error factor
  - Due to the properties of sigmoid functions
  - Number of hidden nodes may be large
- Arbitrary functions
  - Can approximate *any* arbitrary function with *two* layers of hidden nodes up to any arbitrary error factor
  - Due to the properties of sigmoid functions
  - Number of hidden nodes may be large

Discussion

Discussion

- Determining number of nodes and layers is problematic

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise
- Overfitting is a problem

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
    - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise
- Overfitting is a problem
- Gradient descent converges to local minimum only

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise
- Overfitting is a problem
- Gradient descent converges to local minimum only
- Susceptible to class imbalance problem

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise
- Overfitting is a problem
- Gradient descent converges to local minimum only
- Susceptible to class imbalance problem
- Generally, requires large training data

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise
- Overfitting is a problem
- Gradient descent converges to local minimum only
- Susceptible to class imbalance problem
- Generally, requires large training data
- Very slow to train

Discussion

- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise
- Overfitting is a problem
- Gradient descent converges to local minimum only
- Susceptible to class imbalance problem
- Generally, requires large training data
- Very slow to train
- Can be easily parallelized

Discussion

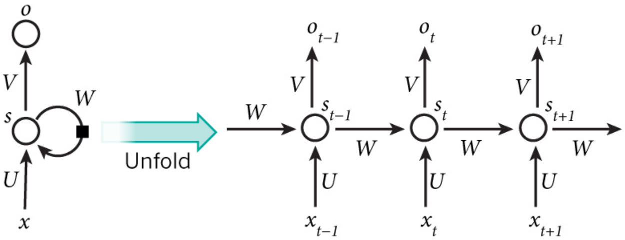
- Determining number of nodes and layers is problematic
- Universal approximators
  - Sigmoid and hyperbolic tangent functions
- Redundant or irrelevant features will have less weight
- Robust to noise
- Overfitting is a problem
- Gradient descent converges to local minimum only
- Susceptible to class imbalance problem
- Generally, requires large training data
- Very slow to train
- Can be easily parallelized
- Notoriously non-explainable

Recurrent Neural Networks

- Recurrent Neural Networks (RNNs) model recurring patterns
  - Same task is repeated for every element of a sequence

Recurrent Neural Networks

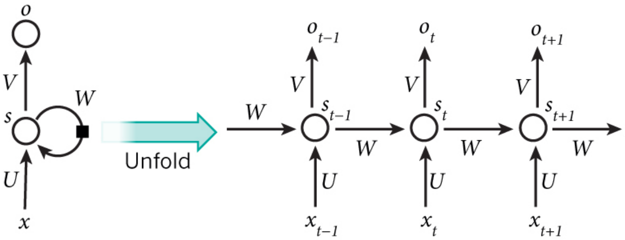
- Recurrent Neural Networks (RNNs) model recurring patterns
  - Same task is repeated for every element of a sequence
- Hidden nodes are *not* independent of each other



- Output depends on previous steps, i.e., it uses "memory"

Recurrent Neural Networks

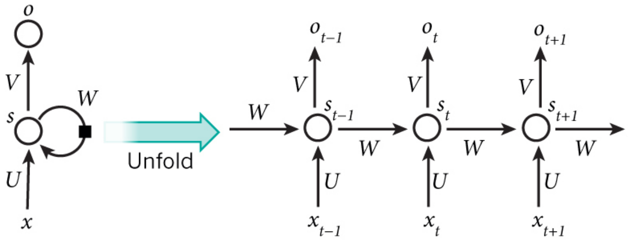
- Recurrent Neural Networks (RNNs) model recurring patterns
  - Same task is repeated for every element of a sequence
- Hidden nodes are *not* independent of each other



- Output depends on previous steps, i.e., it uses "memory"
- "Unrolling" or "unfolding" produces the layers
- If a 5-length *context* is needed, RNN is unfolded to 5 layers

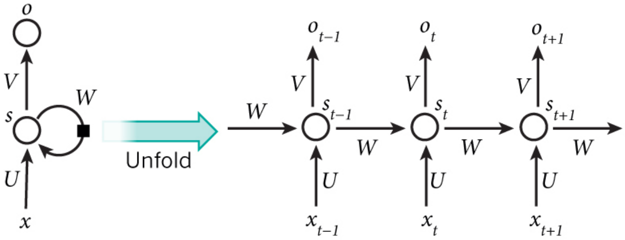
Recurrent Neural Networks

- Recurrent Neural Networks (RNNs) model recurring patterns
  - Same task is repeated for every element of a sequence
- Hidden nodes are *not* independent of each other



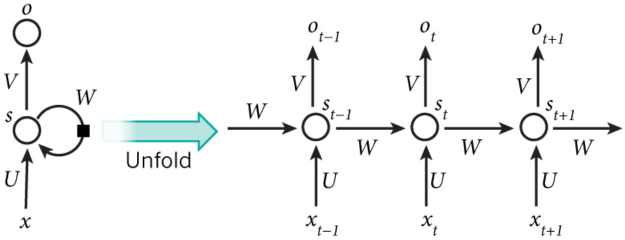
- Output depends on previous steps, i.e., it uses "memory"
- "Unrolling" or "unfolding" produces the layers
- If a 5-length *context* is needed, RNN is unfolded to 5 layers
- Same parameters U, V, W are shared across the layers
  - General deep networks are not constrained by this

Components of an RNN



- x at each step is the "one-hot" vector (i.e., only 1 element is on)

Components of an RNN



- x at each step is the "one-hot" vector (i.e., only 1 element is on)
- s_t is memory as it captures everything previous

$$s_t = f(U.x_t + W.s_{t-1} + b_s)$$

  - f is a non-linear function such as sigmoid or hyperbolic tangent
- o_t is the output at step t

$$o_t = g(V.s_t + b_o)$$

  - Generally, g is the *softmax* function to produce distributions

Training RNNs

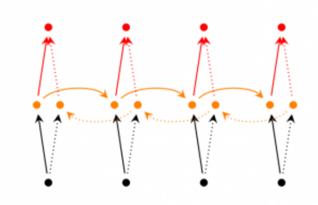
- Standard *backpropagation* does not work since there are loops
- Unfolding removes loops
- Backpropagation is then adopted as backpropagation through time (BPTT)

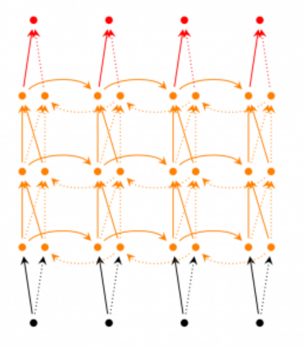
Training RNNs

- Standard *backpropagation* does not work since there are loops
- Unfolding removes loops
- Backpropagation is then adopted as backpropagation through time (BPTT)
- Suffers from vanishing/exploding gradients problem for long chains

Types of RNNs

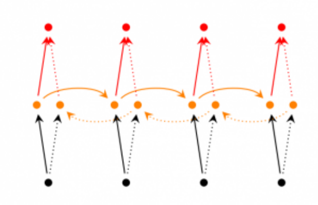
- Bi-directional RNNs use future as well as past to model present
- Deep bi-directional RNNs use multiple layers per time step

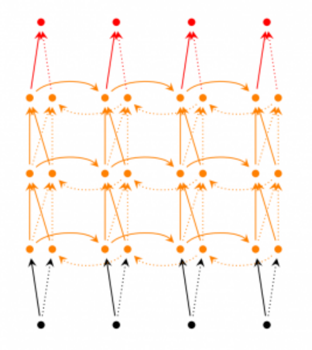




Types of RNNs

- Bi-directional RNNs use future as well as past to model present
- Deep bi-directional RNNs use multiple layers per time step





- Most famous is LSTM (long short-term memory) RNNs
  - Can use long-term memory or can ignore it
  - Instead of a simple non-linear function f at s_t, LSTM uses a complicated neural network structure

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) approximate fully connected feed-forward networks

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) approximate fully connected feed-forward networks
- Especially useful for images
- Full connections are too many in number and require impractical training size

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) approximate fully connected feed-forward networks
- Especially useful for images
- Full connections are too many in number and require impractical training size
- Convolutional layer combines *spatially local* neurons into one output neuron in the next layer
  - Tiling of 4×4 pixels into 1 value requires learning only 16 weights

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) approximate fully connected feed-forward networks
- Especially useful for images
- Full connections are too many in number and require impractical training size
- Convolutional layer combines *spatially local* neurons into one output neuron in the next layer
  - Tiling of 4×4 pixels into 1 value requires learning only 16 weights
- Pooling layer combines several input neurons into one output neuron in the next layer
  - Common is *max pooling*

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) approximate fully connected feed-forward networks
- Especially useful for images
- Full connections are too many in number and require impractical training size
- Convolutional layer combines *spatially local* neurons into one output neuron in the next layer
  - Tiling of 4×4 pixels into 1 value requires learning only 16 weights
- Pooling layer combines several input neurons into one output neuron in the next layer
  - Common is *max pooling*
- Activation function is often simplified to ReLU (rectified linear units): $f(x) = \max\{0, x\}$

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) approximate fully connected feed-forward networks
- Especially useful for images
- Full connections are too many in number and require impractical training size
- Convolutional layer combines *spatially local* neurons into one output neuron in the next layer
  - Tiling of 4×4 pixels into 1 value requires learning only 16 weights
- Pooling layer combines several input neurons into one output neuron in the next layer
  - Common is *max pooling*
- Activation function is often simplified to ReLU (rectified linear units): $f(x) = \max\{0, x\}$
  - Relaxed version is Leaky ReLU

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) approximate fully connected feed-forward networks
- Especially useful for images
- Full connections are too many in number and require impractical training size
- Convolutional layer combines *spatially local* neurons into one output neuron in the next layer
  - Tiling of 4×4 pixels into 1 value requires learning only 16 weights
- Pooling layer combines several input neurons into one output neuron in the next layer
  - Common is *max pooling*
- Activation function is often simplified to ReLU (rectified linear units): $f(x) = \max\{0, x\}$
  - Relaxed version is Leaky ReLU
- Includes *fully connected layers* (*dense layers*) as well