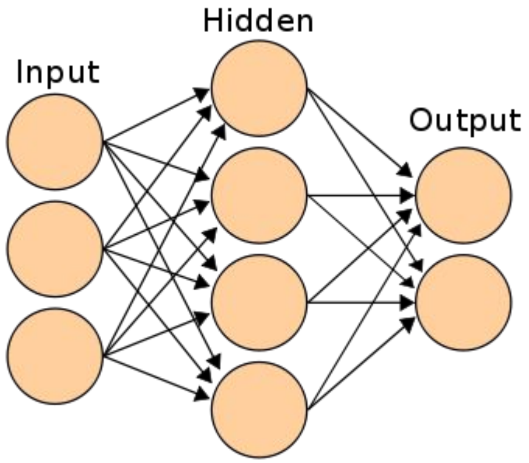
CS380
Artificial Intelligence for Games

Neural Network

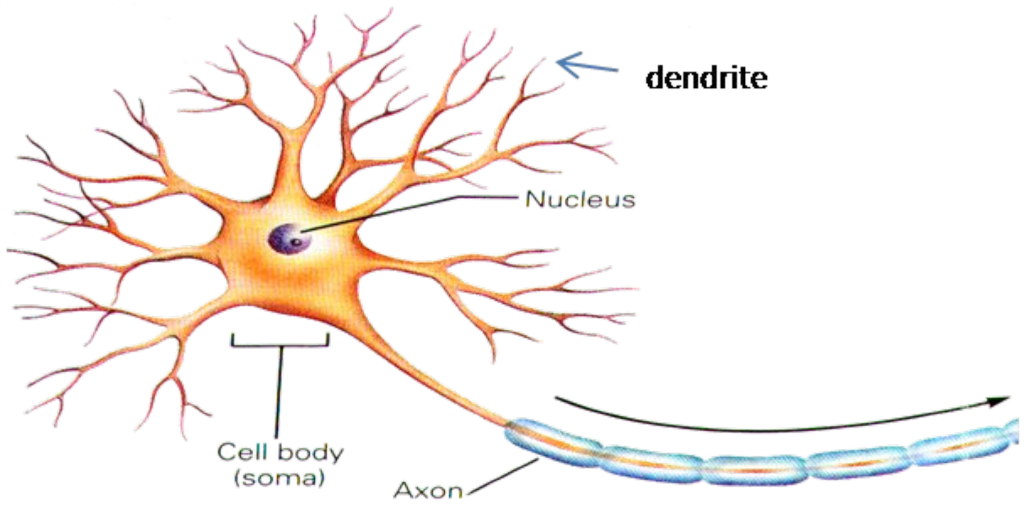
Biological Motivation

- The study of artificial neural networks has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons.
- Artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

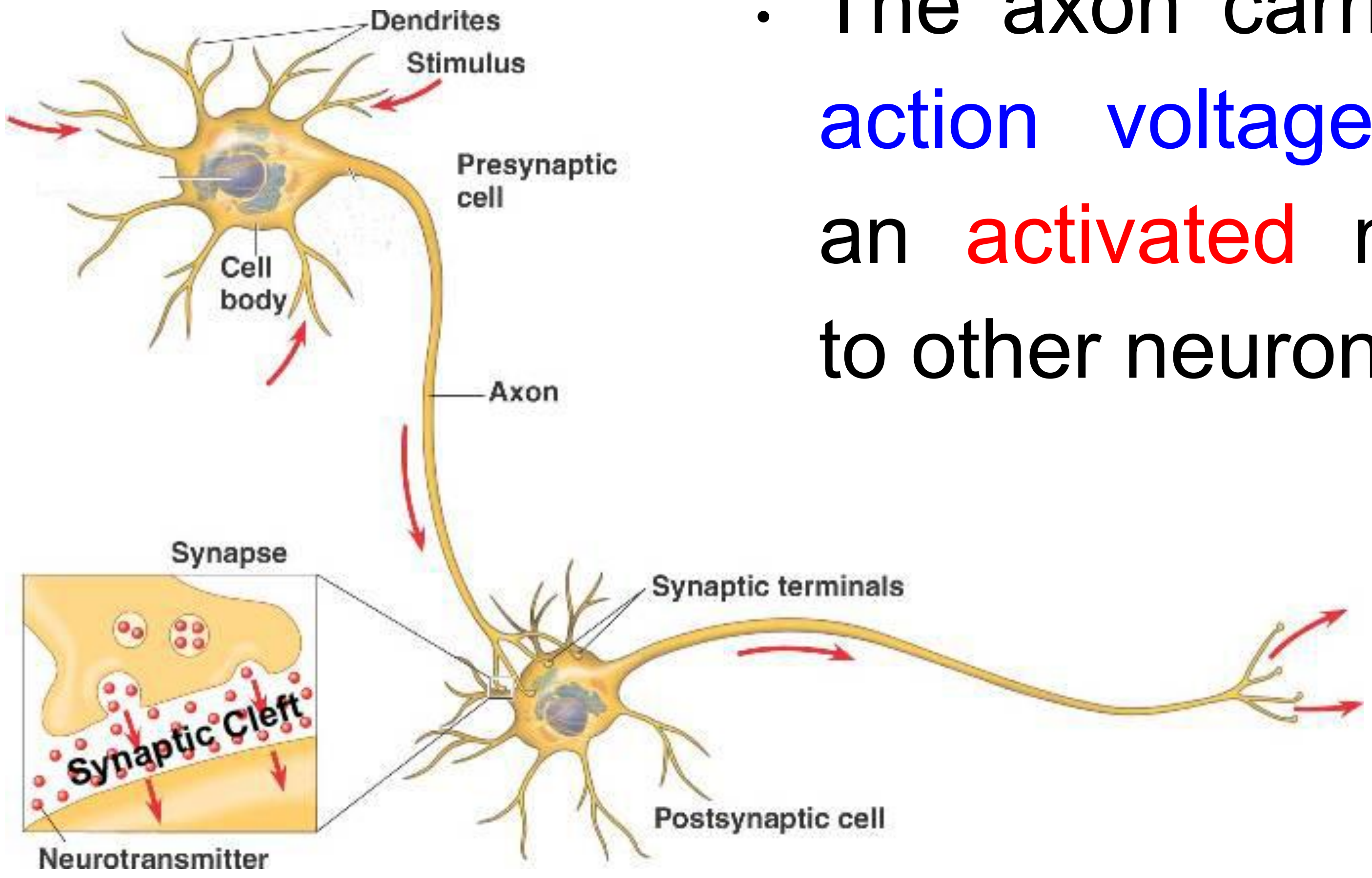


The Biological Neural Network

- The human brain is composed of billions of neurons.
- The brain of an adult human contains around 10^{11} neurons.
- Each neuron is connected to thousands (around 10^4) of other neurons.
- Information is transmitted from one neuron to another via the axon and the dendrites.

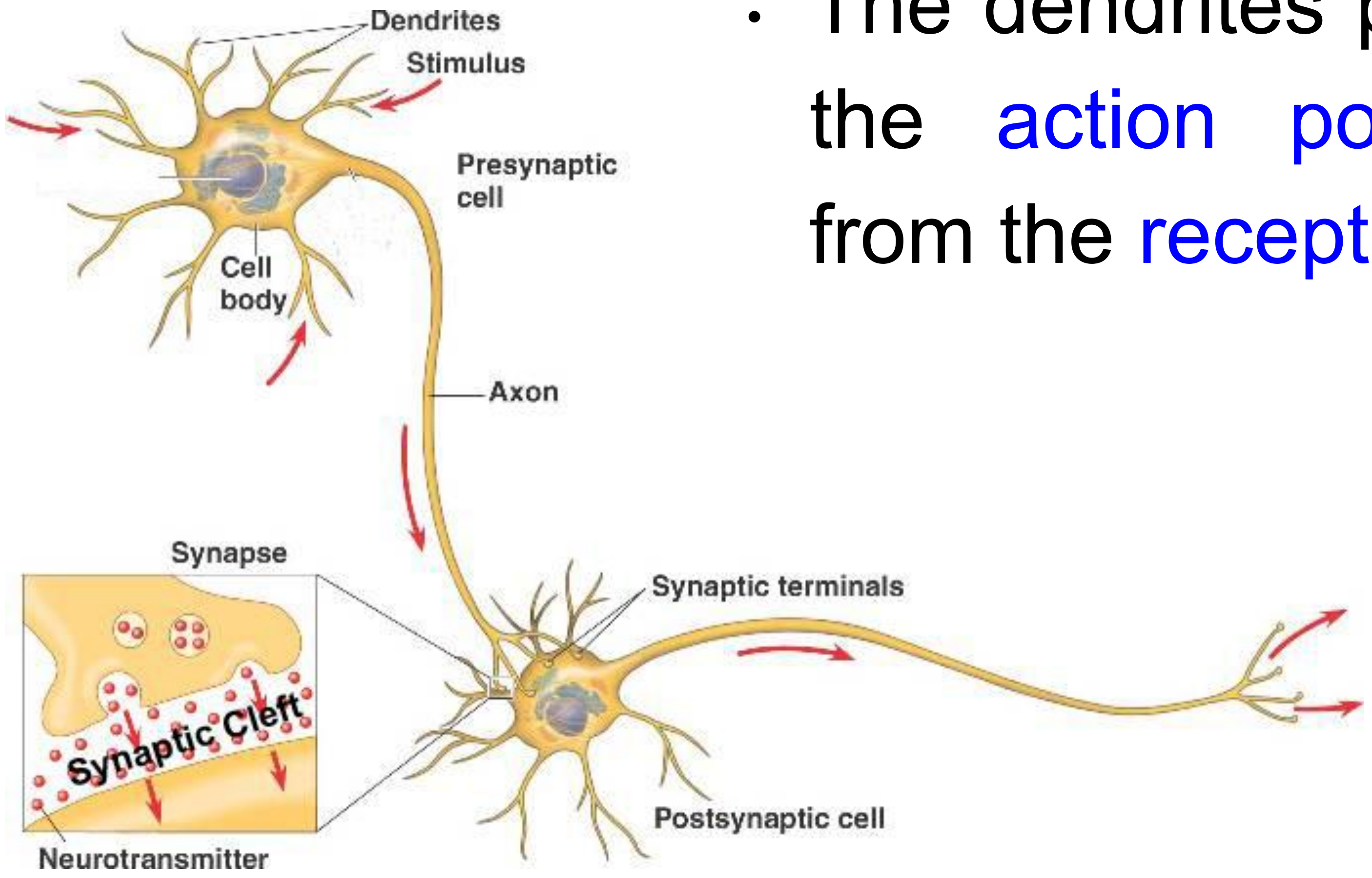


The Biological Neural Network



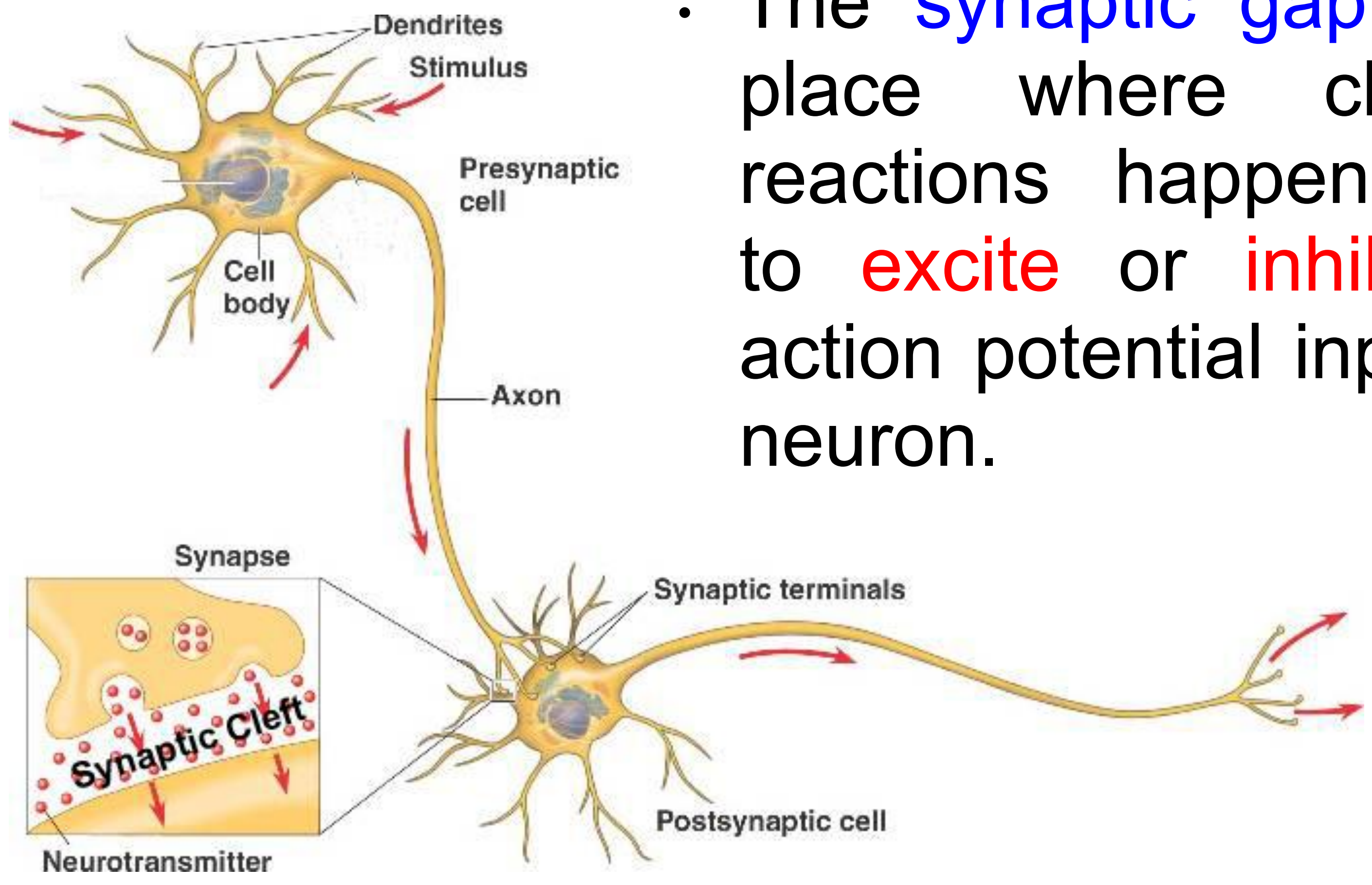
- The axon carries the action voltage from an activated neuron to other neurons.

The Biological Neural Network



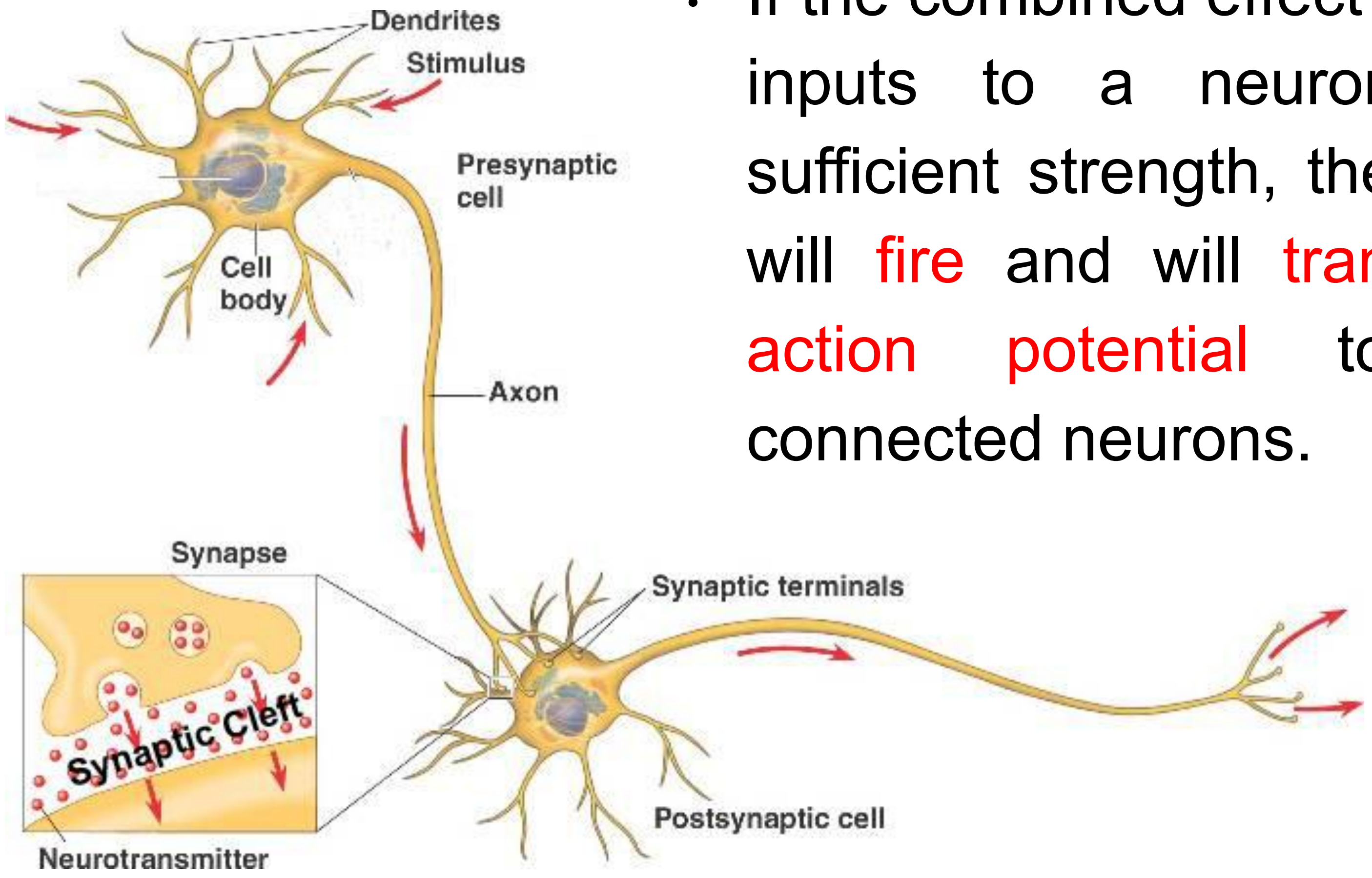
- The dendrites pick up the action potential from the receptors.

The Biological Neural Network



- The synaptic gap is the place where chemical reactions happen either to excite or inhibit the action potential input to a neuron.

The Biological Neural Network



- If the combined effect of all the inputs to a neuron is of sufficient strength, the neuron will fire and will transmit its action potential to other connected neurons.

The Biological Neural Network

- The fastest neuron switching times are in the order 10^{-3} seconds.
- It requires approximately 10^{-1} seconds to visually recognize your mother.
- There cannot be more than a few hundred steps in the sequence of neuron firings in 0.1 second interval.
  - Speculation: information processing abilities of biological neural systems must follow highly parallel processes operating on representations that are distributed over many neurons.

The Artificial Neural Network

- An artificial neural network is a mathematical function approximation.
  - Input to the network represents independent variables, while output represents dependent variables.
  - It gives a unique set of output for a given input. The function is usually difficult to write in equation form and is highly non-linear.
- The artificial neural networks used in most applications are much simpler than our brain, composed of only a dozen or so of neurons.
  - But even these simple nets can be very useful.

Advantages: Neural Networks in Games

- They allow game programmers to simplify coding of complex state machines or rule based systems.
  - This is done by transferring key decision making processes to trained neural networks.

- They offer the ability for the game's AI to adapt as the game is played.

- They are able to solve a specific game AI task as part of an integrated AI system that uses traditional AI techniques.

Disadvantages: Neural Networks in Games

- Neural network is very hard to debug and trace, and the solution offered is always like a **black box** for the programmer.

- It is difficult to predict the overall output of a neural network especially the one that is programmed to learn within the game.

Neural Network Usability

- Input can be
  - high-dimensional discrete
  - real-valued (e.g. raw sensor input).
- Output can be
  - discrete or real values
  - a vector of values
- Form of target function is unknown
- Humans do not need to interpret the results (black box model)

Neural Network Usability

- Neural network can be used in many domains and there are many forms of them.
- What we will see is one of the very useful forms, the Multi-layer feed-forward network.
- Neural networks algorithms can be applied to solve problems and learning in finance, engineering, virtual simulation and many other domains.

Input (x_i)

- What input should we choose?
  - problem specific.

- Of what form should they be?
  - Boolean (if an enemy is engaged in a combat/not?)
  - Enumerated (the kind of weapon an enemy is holding)
  - Continuous (speed or health level of an enemy)
    - Need normalization

- How many inputs do we need?
  - As few as possible
  - Combine a set of input into a more compact form.

Weights (w)

- Weights in a neural network are similar to synaptic connections in a biological neural network.

- Weights affect the strength of the input. A weight can be either inhibitory (<1) or excitatory (>1).

- The net input to a given neuron with weights w and inputs x is evaluated using the following function:

$$\text{net} = \sum w_i x_i$$

Activation Functions

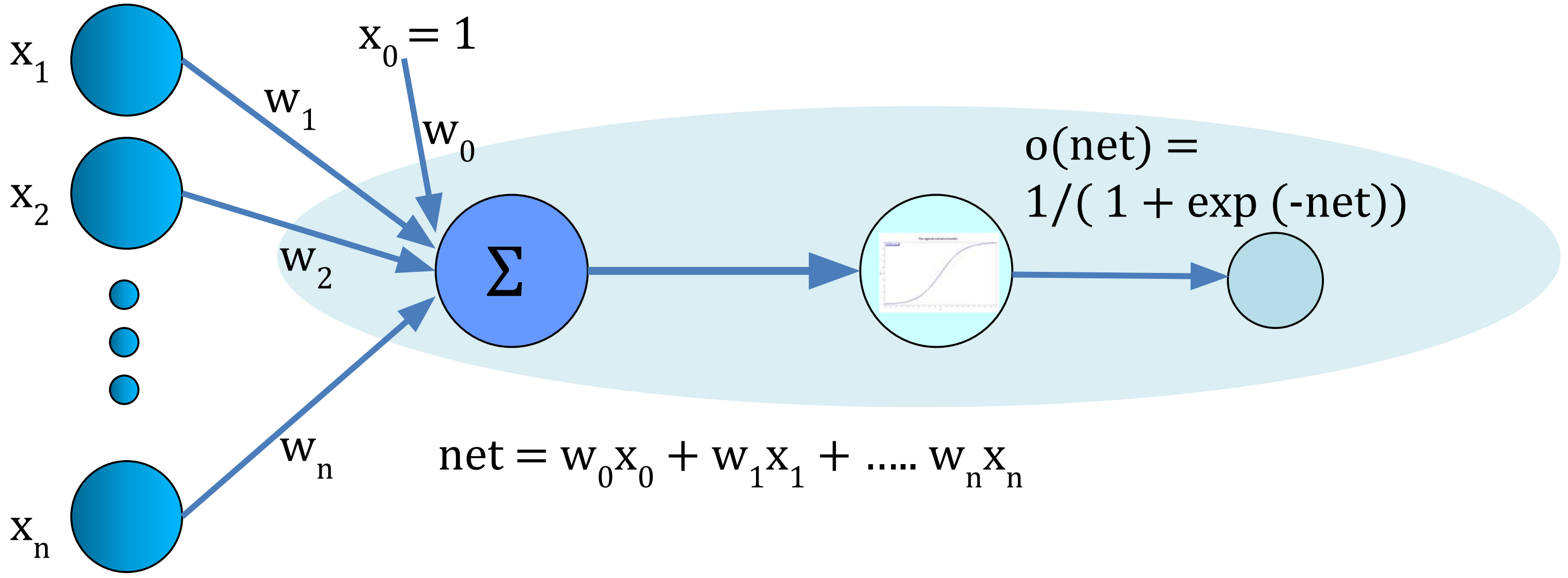
- An activation function takes the net input to a neuron and produces an output for the neuron.
- In general, activation functions are non linear. A common choice is the so-called logistic function or sigmoid:

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

Also can be a step function output (with threshold) for output layer.

- If the activation function is linear, the neural net becomes a linear combination of linear function, and thus becomes incapable of approximating non linear functions and relationships.

Activation Functions



Bias

- Each neuron has an associated bias. The bias is represented by a bias value (x_0) and a bias weight (w_0).
- The bias term shifts the net input along the horizontal axis of the activation function, resulting in a change in the threshold at which a neuron is activated.
- The bias value is set to 1 or -1, while the weight term is adjusted through training. It does not matter which value to choose (-1 or 1) because in either case the bias weight will adjust the sign of the result.

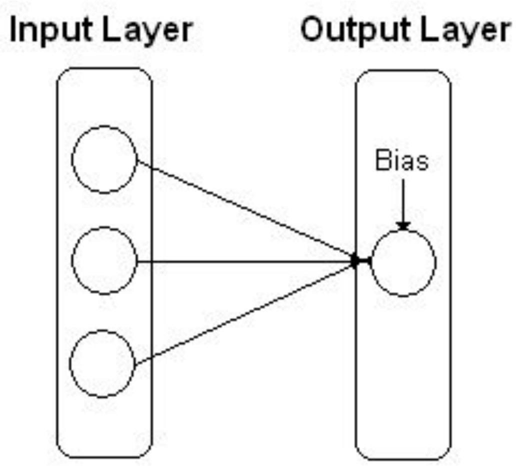
Output

- The choice of the output neurons and their number is problem dependent, but it is better to keep the number of output neurons to a minimum in order to reduce computation time and complexity.

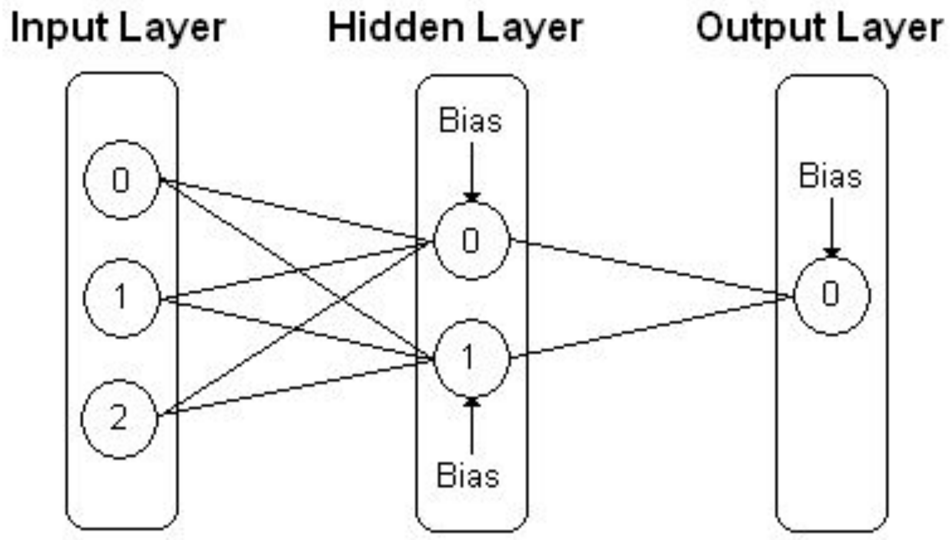
- If the desired output is a classification of the input, then you can use only one output neuron.

Neural Networks Types

- Single-layer perceptrons

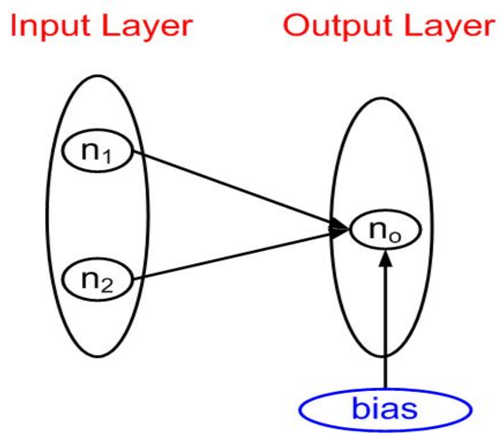


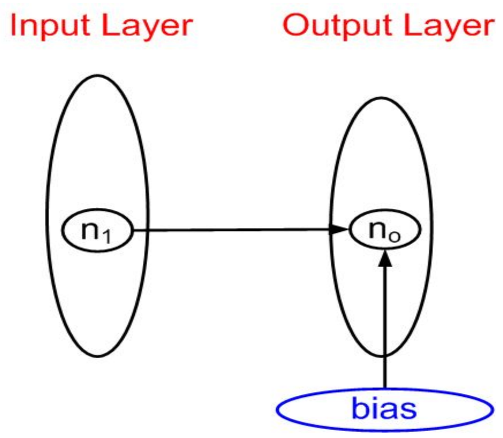
- Multi-layer perceptrons



Logic Gates Example

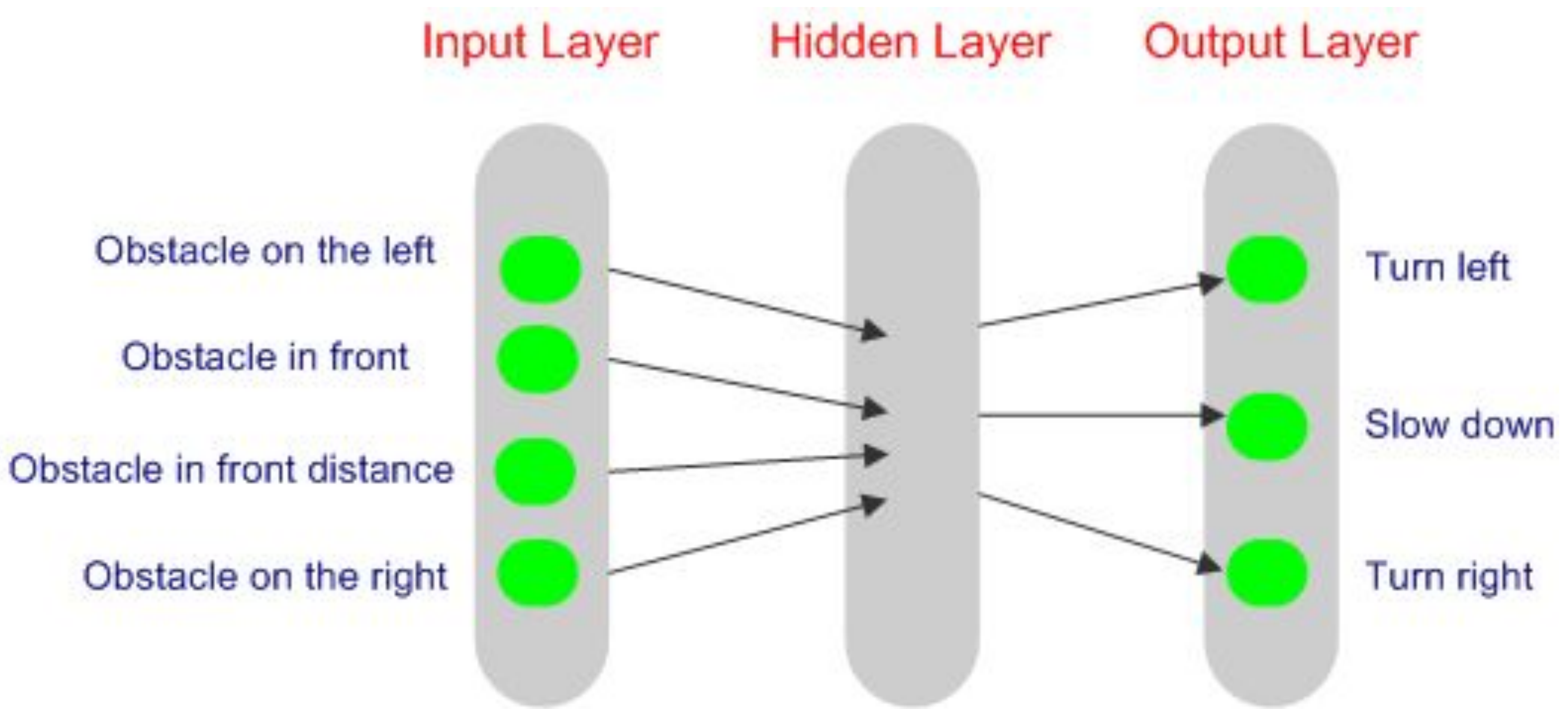
- AND: bias=-1, threshold function is g(x),
- OR: bias=0 or (bias=-1 and weights=2),
- NOT: bias=1 and weight=-1.





$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Self-driving Example



Hidden Layer

- The hidden layer gives the neural net the ability to process features in the input data.
- The more hidden neurons are put, the more features the neural net can process.

Hidden Layer

- If the neural network's function is to approximate a noisy function (meaningless input and output), using few hidden neurons will yield in a function that captures the trend of the input but misses the local noisy features.
  - If this is what you want, it's fine.
  - Otherwise you need to increase the number of hidden neurons in order to pick up the noisy features as well as the general trend of the function.

Hidden Layer

- It is hard to tell the exact number of neurons in the hidden layer.
- A general rule for two layer networks is that the number of neurons in the hidden layer is approximately equal to the square root of the product of the number of neurons in the input layer and the number of neurons in the output layer.
- Although this is just an approximation, it is a good place to start.

Single- vs Multi-layer Perceptrons

- A single-layer perceptron can be used to represent many boolean functions
  - AND, OR, NAND, NOR are representable by a perceptron
- But not all of them
  - XOR cannot be representable by any single perceptron,
  - Nonetheless, it was known that multi-layer perceptrons are capable of producing any possible boolean function.

Training/Learning

- The purpose of training a neural network is to find the values for the weights that connect all the neurons in a way that the input values generate the desired output values.

- Training a neural network is somehow an optimization process where we are trying to find the best weight values that make the network produce the desired results.

Representational Power of Perceptrons

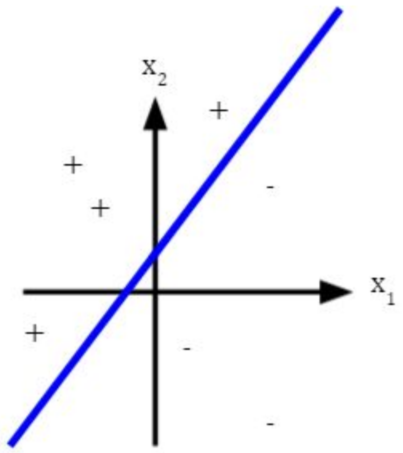
- A perceptron represents a hyperplane decision surface in the n-dimensional space of instances, where n is number of inputs.
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 or 0 for instances lying on the other side.
- The equation for this decision hyperplane is

$$w \cdot x = 0$$

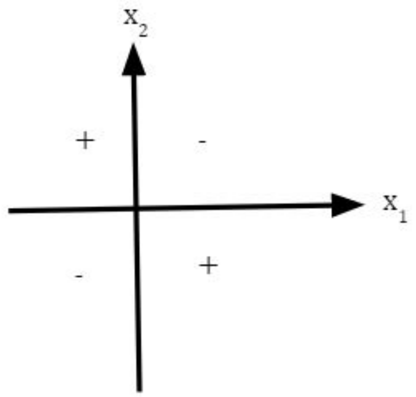
(dot product of vectors w and x).

Representational Power of Perceptrons

- Some sets of positive and negative outputs cannot be separated by any hyperplane.
- Those that can be separated are called **linearly separable** sets of examples.



Linearly Separable



Not linearly separable

Perceptron Learning

- To learn an acceptable weight vector
  - begin with random weights, then
  - iteratively apply the perceptron to each training example,
    - modifying the perceptron weights whenever it misclassifies an example.

Perceptron Learning

- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.

- Each pass through all of the training examples is called one epoch.

- Weights are modified at each step according to perceptron learning rule.

Perceptron Learning Rule

$$w_i = w_i + \Delta w_i,$$

where

$$\Delta w_i = \eta\ (t - o)\ x_i$$

- t - the target output of the current training example
- o - the output generated by the perceptron
- η - a positive learning constant usually set to a small value, e.g. 0.1.

Perceptron Learning Rule

- If the output is correct $(t = o)$, then the weights w_i are not changed.
- If the output is incorrect $(t \neq o)$, then the weights w_i are changed such that the output of the perceptron for the new weights is *closer* to t.
- The algorithm converges to the correct classification if
  - the training data is linearly separable
  - and η is sufficiently small

Perceptron Learning Rule – OR

x_1	x_2	t	o	Δw_1	w_1	Δw_2	w_2	Δw_0	w_0
-	-	-	-	-	.1	-	.5	-	-.8
0	0	0	0	0	.1	0	.5	0	-.8
0	1	**1**	**0**	0	.1	.2	.7	.2	-.6
1	0	**1**	**0**	.2	.3	0	.7	.2	-.4
1	1	1	1	0	.3	0	.7	0	-.4
0	0	0	0	0	.3	0	.7	0	-.4
0	1	1	1	0	.3	0	.7	0	-.4
1	0	**1**	**0**	.2	.5	0	.7	.2	-.2
1	1	1	1	0	.5	0	.7	0	-.2
0	0	0	0	0	.5	0	.7	0	-.2
0	1	1	1	0	.5	0	.7	0	-.2
1	0	1	1	0	.5	0	.7	0	-.2
1	1	1	1	0	.5	0	.7	0	-.2

$$w_i = w_i + \Delta w_i$$
$$\Delta w_i = \eta\,(t - o)\,x_i$$
$$\eta = 0.2$$

The result of executing the learning algorithm for 3 epochs.

Perceptron Learning Rule – AND

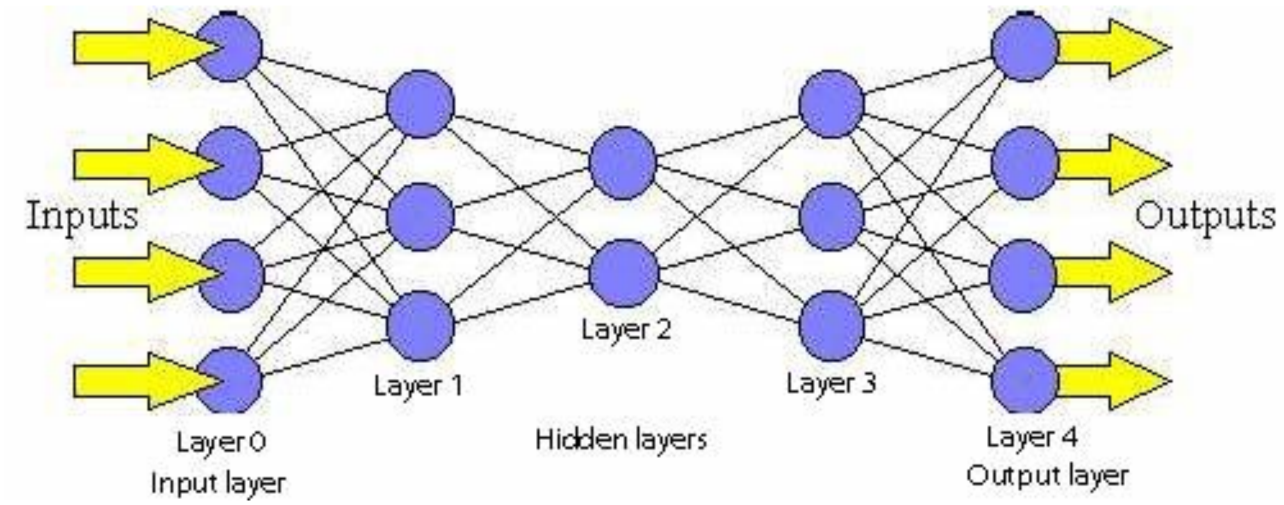
x_1	x_2	t	o	Δw_1	w_1	Δw_2	w_2	Δw_0	w_0
-	-	-	-	-	.1	-	.5	-	-.8
0	0	0	0	0	.1	0	.5	0	-.8
0	1	0	0	0	.1	0	.5	0	-.8
1	0	0	0	0	.1	0	.5	0	-.8
1	**1**	**1**	**0**	**.2**	**.3**	**.2**	**.7**	**.2**	**-.6**
0	0	0	0	0	.3	0	.7	0	-.6
0	**1**	**0**	**1**	**0**	**.3**	**-.2**	**.5**	**-.2**	**-.8**
1	0	0	0	0	.3	0	.5	0	-.8
1	**1**	**1**	**0**	**.2**	**.5**	**.2**	**.7**	**.2**	**-.6**
0	0	0	0	0	.5	0	.7	0	-.6
0	**1**	**0**	**1**	**0**	**.5**	**-.2**	**.5**	**-.2**	**-.8**
1	0	0	0	0	.5	0	.5	0	-.8
1	1	1	1	0	.5	0	.5	0	-.8

$$w_i = w_i + \Delta w_i$$
$$\Delta w_i = \eta\,(t - o)\,x_i$$
$$\eta = 0.2$$

The result of executing the learning algorithm for 4 epochs.
1 epoch not shown in the table.

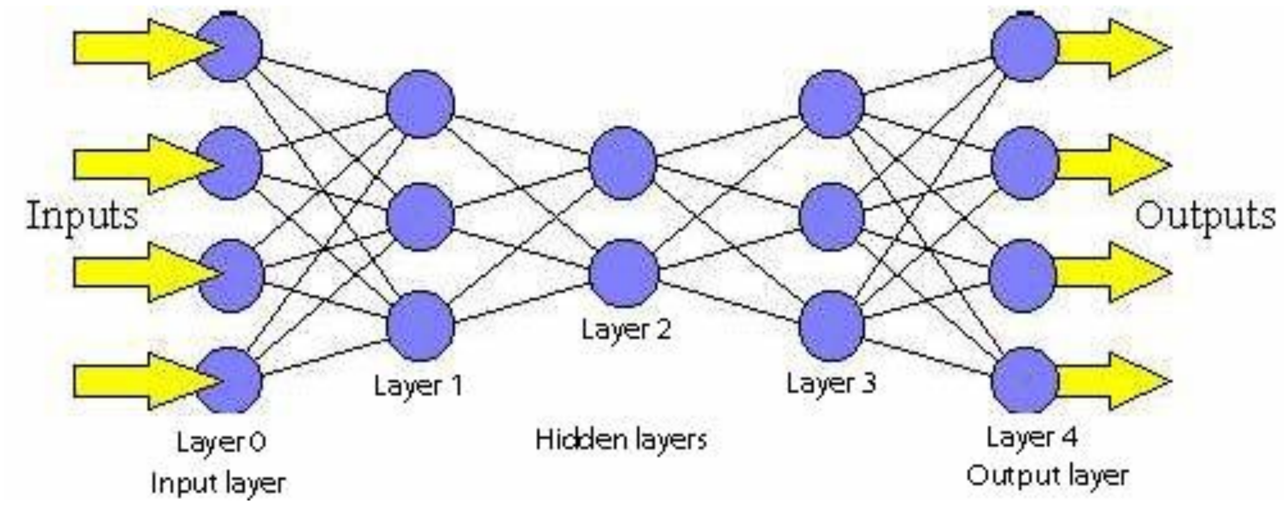
Feed-Forward Network

- Perceptrons are arranged in layers, with the first layer taking in inputs and the last layer producing outputs.
- The hidden layers have no connection with the external world



Feed-Forward Network

- Each perceptron in one layer is connected to every perceptron on the next layer.
- Hence information is constantly "fed forward" from one layer to the next.
- There is no connection among perceptrons in the same layer.

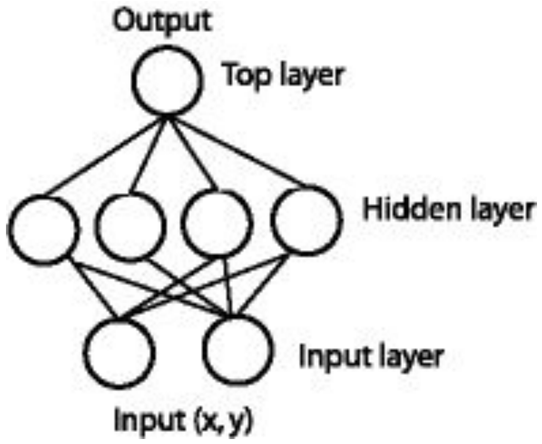


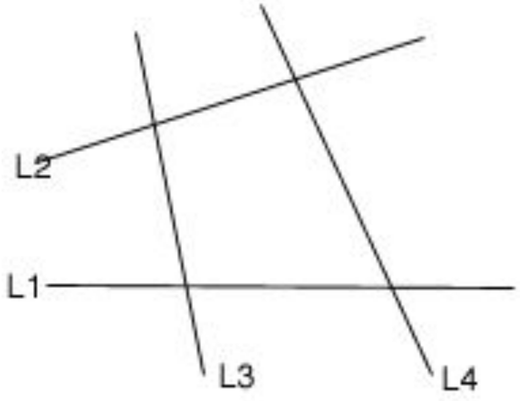
Feed-Forward Network

- Recall that a single perceptron can classify points into two regions that are linearly separable.
- Now let us extend the discussion into the separation of points into two regions that are not linearly separable.

Feed-Forward Network

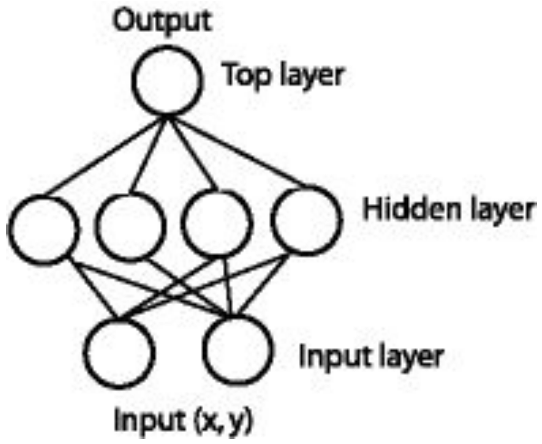
- Consider the following network.
- The same (x, y) is fed into the network through the perceptrons in the input layer.
- With four perceptrons that are independent of each other in the hidden layer, the point is classified into 4 pairs of linearly separable regions, each of which has a unique line separating the region.

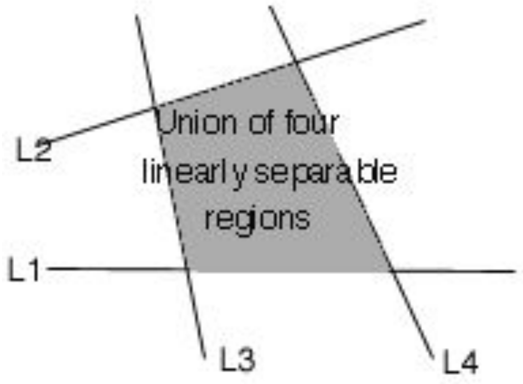




Feed-Forward Network

- The top perceptron performs logical operations on the outputs of the hidden layers so that the whole network classifies input points in 2 regions that might not be linearly separable.
- For instance, using the AND operator on these four outputs, one gets the intersection of the 4 regions that forms the center region.





Feed-Forward Network

- By varying the number of nodes in the hidden layer, the number of layers, and the number of input and output nodes, one can classification of points in arbitrary dimension into an arbitrary number of groups.
- Hence feed-forward networks are commonly used for classification.

Backpropagation

- Backpropagation - learning in feed-forward networks by using Delta Rule with gradient descent

Delta Rule

- If samples are not linearly separable, perceptron rule may fail to converge.
- How about to converge toward a best-fit approximation to the target concept?
- Use the delta rule as an update rule for single layer perceptrons to get such approximation.
- Key idea: Gradient descent to search the hypothesis space to find the best weights.

Delta Rule

- Consider the case of a linear unit with no threshold, the output for input x_i is computed as

$$o_i = \mathbf{w} \cdot \mathbf{x_i} = w_1 x_{i1} + w_2 x_{i2} + ... + w_n x_{in}$$

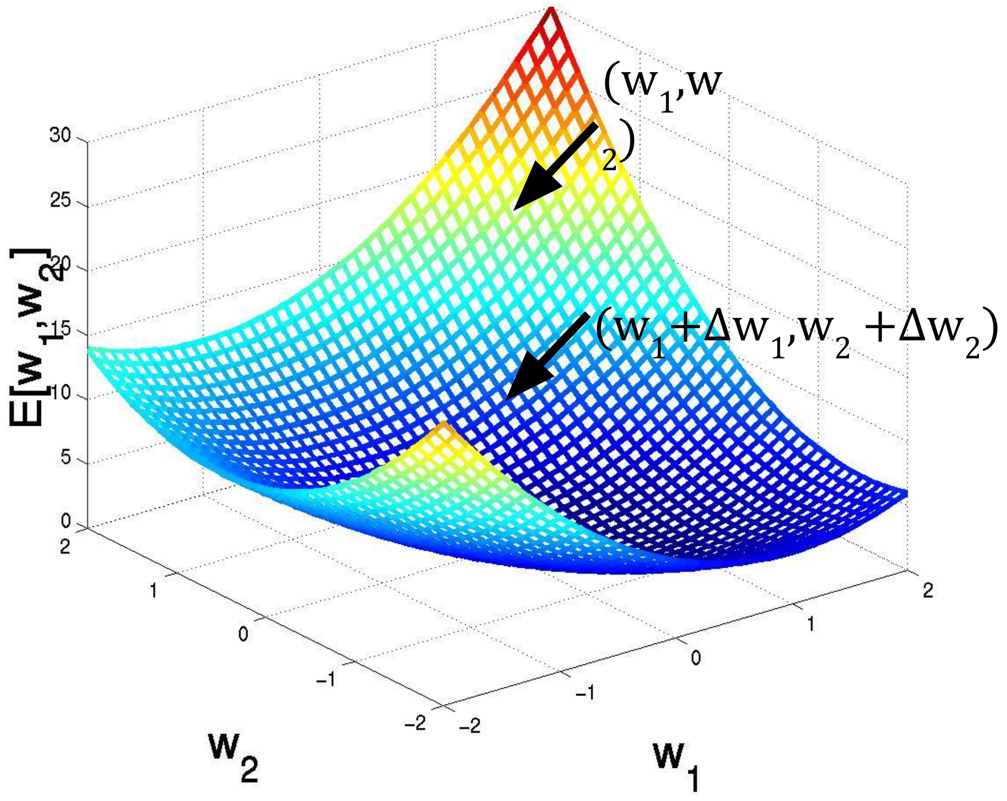
- The training error for the training data set is defined as

$$E(\mathbf{w}) = ½ \sum_i (t_i - o_i)^2$$

where t_i is the target values and o_i is the unit's output for input x_i.

Gradient Descent and Delta Rule

- The axes w_1 and w_2 represent possible values of the two weights of a simple linear unit.
- The vertical axis indicates the error E relative to the 4 training examples.
- Gradient descent search determines a weight vector that minimizes E by moving along the steepest descent direction along the error surface.



Visualizing the hypothesis space

Derivation of the Gradient Descent Rule

- The direction of steepest descent along the hypothesis space can be obtained by computing the derivative of the error function $\mathrm{E}(\mathbf{w})$.

- The derivative (**gradient**) is vector

$$\nabla \mathrm{E}(\mathbf{w}) = (\partial \mathrm{E}/\partial \mathrm{w}_0, \partial \mathrm{E}/\partial \mathrm{w}_1, \ldots, \partial \mathrm{E}/\partial \mathrm{w}_n)$$

- The negative of this gradient gives the direction of steepest descent

- The training rule for gradient descent is $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$, where $\Delta\mathbf{w} = -\eta\, \nabla \mathrm{E}(\mathbf{w})$

Derivation of the Gradient Descent Rule

- At each iteration we need to compute $\partial E/\partial w_j$

- This can be done as follows:

$$\begin{aligned}\partial E/\partial w_j &= \partial/\partial w_j \ \tfrac{1}{2} \textstyle\sum_i (t_i - o_i)^2 \\ &= \tfrac{1}{2} \textstyle\sum_i \partial/\partial w_j \ (t_i - o_i)^2 \\ &= \tfrac{1}{2} \textstyle\sum_i 2(t_i - o_i) \ \partial/\partial w_j (t_i - o_i) \\ &= \tfrac{1}{2} \textstyle\sum_i 2 (t_i - o_i) \ \partial/\partial w_j (t_i - w \cdot x_i) \\ &= \textstyle\sum_i (t_i - o_i) (- x_{ij})\end{aligned}$$

- The weight update rule for gradient descent is then:

$$\Delta w_j = \eta \textstyle\sum_i (t_i - o_i) \, x_{ij}$$

Gradient Descent Algorithm

Input:

- (x_i,t_i),
  - $i = 1,2, \ldots. N$,
  - x_i is the vector of K input values,
  - t_i is its corresponding target value.
- Learning parameter: η.

1. Initialize each w_j to a small random value for $j=1,2,..,K$

Gradient Descent Algorithm

2. Until termination condition is met, do
   - Initialize each Δw_j to zero, for j=1,2,..,K
   - For each (x_i, t_i) in the data set, do
     - Input the data x_i to the unit and compute the output o_i
     - For each linear unit weight w_j, do
       - $\Delta w_j = \Delta w_j + \eta\,(t_i - o_i)\,x_{ij}$
   - For each w_j, update the weight w_j:
     $w_j = w_j + \Delta w_j$

Gradient Descent

- Gradient descent can be applied for learning where
  - The hypothesis space contains continuously parameterized hypothesis.
  - The error can be differentiated with respect to the hypothesis parameters.
- Some practical difficulties in gradient descent learning:
  - Slow convergence to a local minimum
  - No guaranteed convergence to a global minimum.

Incremental Gradient Descent

- Idea of optimization:
  - Instead of updating the weights with the sum over all training examples, approximate the gradient descent by updating the weights incrementally, following the calculation of the error for each individual sample.

Incremental Gradient Descent

2. Until termination condition is met, do
   - For each (x_i, t_i) in the data set, do
     - Input the data x_i to the unit and compute the output o_i
     - For each linear unit weight w_i, do
       $w_j = w_j + \eta\ (t_i – o_i)\ x_{ij}$

Neural Network In Games

Role Playing Game Example

- A group of NPC / AI enemies.
- Possible actions: attack the player, flock with other AI soldiers, evade the player
- AI enemies will only need to decision making when they engage with the player.
- Adaptive neural network learning.

Role Playing Game Example

- The AI soldier will receive some amount of hit points each game loop (while in radius).
- The player will receive some amount of hit points each game loop proportional to the number of AI soldier he’s engaged with.
- When any of the characters on the scene reaches the maximum hit points he dies and will be re-spawned automatically.

```
class NeuralNetwork
{
   NeuralNetworkLayer inputLayer;
   NeuralNetworkLayer hiddenLayer;
   NeuralNetworkLayer outputLayer;

   void Initialize(int neuronInput, int neuronHidden,
                    int neuronOutput);
  void SetInput(int i, double value);
                         // i is the neuron index
  double GetOutput(int i);
   void DesiredOutput(int i, double value);

   void FeedForward();
   void BackPropagate();

   int GetTheBestOutputIndex();
   void CalculateError();
   SetLearningRate(double rate);
};
```

```
void NeuralNetwork::FeedForward() {
    hiddenLayer.CalculateNeuronValues();
    outputLayer.CalculateNeuronValues();
}
void NeuralNetwork::BackPropagate(){
    outputLayer.CalculateErrors();
    hiddenLayer.CalculateErrors();
    hiddenLayerAdjustWeights();
    inputLayerAdjustWeights();
}
double NeuralNetwork::CalculateError(){
    double error = 0.0f;
    for (int i = 0; i < outputLayer.numberOfNeurons; ++i)
        error += pow(outputLayer.neuronValues[i] -
                     outputLayer.desiredValues[i], 2);
    error = error / outputLayer.numberOfNeurons;
    return error;
}
```

```
class NeuralNetworkLayer {
    int numberOfNeurons, numberOfChildNeurons,
         numberOfParentNeurons;
    double ** weights;
    double * neuronValues, * desiredValues, * errors,
            * biasWeight, * biasValues;
    double learningRate;
     bool linearOuput;
    NeuralNetworkLayer * parentLayer, * childLayer;

    void Initialize(int numberOfNeurons,
       NeuralNetworkLayer * parent,
       NeuralNetworkLayer * child);
    void RandomizeWeights();
    void CalculateErrors();
    void AdjustWeights();
    void CalculateNeuronValues();
};
```

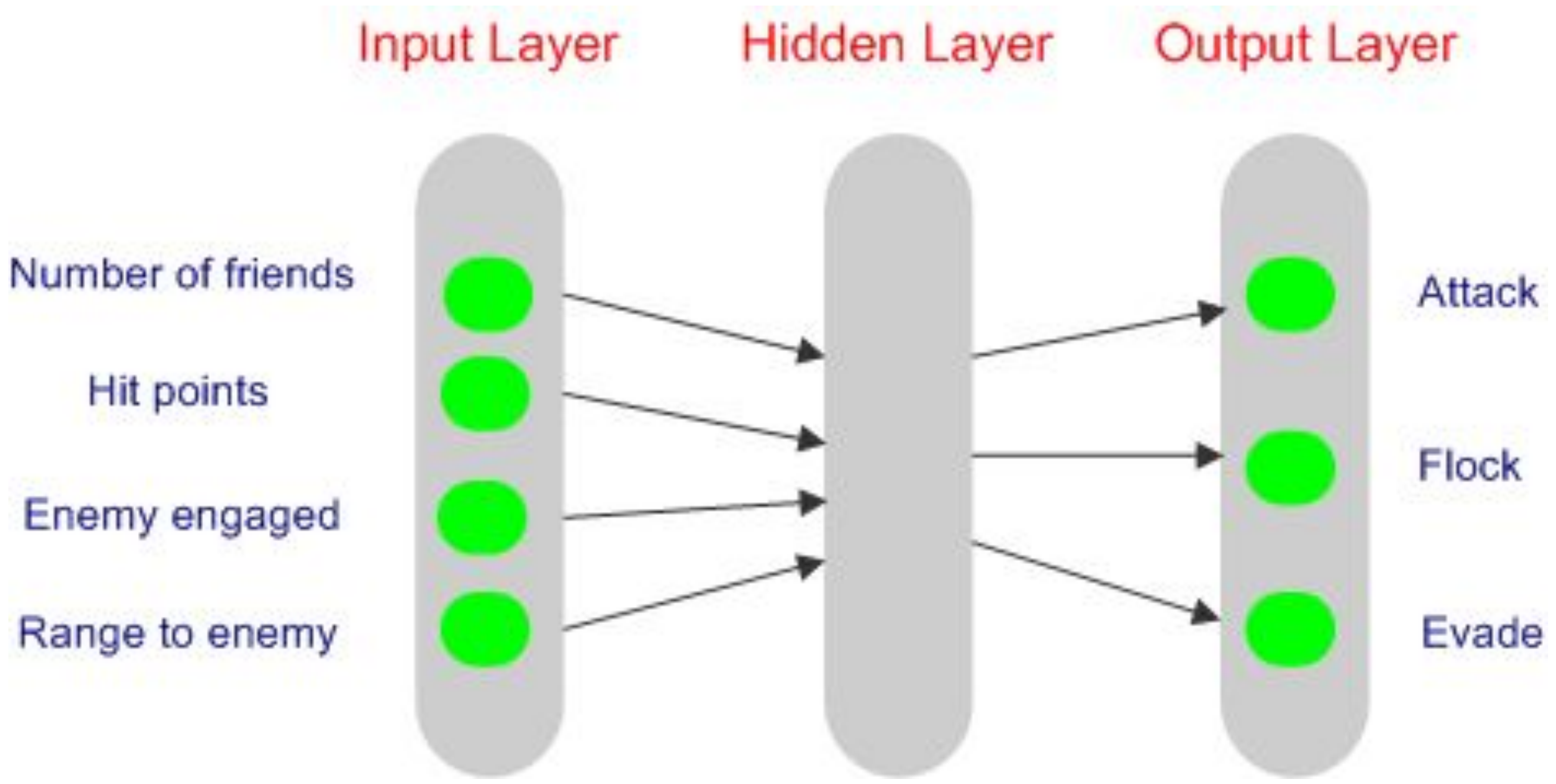
```
void NeuralNetworkLayer::CalculateNeuronValues() {
    for (int j = 0; j < numberOfNeurons; ++j) {
        x = 0;
        for (int i = 0; i < numberOfParentNeurons; ++i){
            x += parentLayer->neuronValues[i] *
                         parentLayer->weights[i][j];
        }
        x += parentLayer->biasValues[j] *
                 parentLayer->biasWeights[j];

        if (linearOutput)
            neuronValues[j] = x;
              // applying linear activation on output
        else
            neuronValues[j] = 1.0f / (1 + exp(-x));
              // applying sigmoid on hidden
      }
}
```

```
void NeuralNetworkLayer::CalculateErrors(){
    if (childLayer == 0){//output layer
       for (int i = 0; i < numberOfNeurons; ++i){
         if (linearOutput)
           errors[i] = (desiredValues[i] - neuronValues[i]);
         else
           errors[i] = (desiredValues[i] - neuronValues[i]) *
                neuronValues[i] * (1.0f - neuronValues[i]);
    }  }
    else { //hidden layer
        for (int i = 0; i < numberOfNeurons; ++i){
            sum = 0;
          for(int j = 0; j < numberOfChildNeurons; ++j)
              sum += childLayer->errors[j] * weights[i][j];
          if(linearOutput)
             errors[i] = sum;
          else
             errors[i] = sum * neuronValues[i] * (1.0f -
                  neuronValues[i]);

}  }   }
```

```
void NeuralNetworkLayer::AdjustWeights(){
  //weights are adjusted only if the layer has a child layer
  if(childLayer == 0) // the output layer
      return;

  for(int i = 0; i < numberOfNeurons; ++i){
      for(int j = 0; j < numberOfChildNeurons; ++j){
          dw = learningRate * childLayer->errors[j] *
            neuronValues[i];
          weights[i][j] += dw;
      }
   }

   for(int j = 0; j < numberOfChildNeurons; ++j)
      biasWeights[j] += learningRate * childLayer->errors[j]
        * biasValues[j];
  }
```

```
NeuralNetwork ai_network;
Soldiers ai_characters[MAX_UNITS];

void InitializeGame(){
    …
   for(int i = 0; i < MAX_UNITS; ++i) {
        …
        ai_characters[i].hitPoints = 0;
            //will lose when reaching MAX_HITPOINTS
        ai_characters[i].chase = false;
        ai_characters[i].flock = false;
        ai_characters[i].evade = false;
    }
    …
    player.hitPoints = 0;
    ai_network.initialize(4,3,3);
      // 4 input neurons, 3 hidden neurons, 3 output neurons
    ai_network.SetLearningRate(0.2f);
    TrainTheNetwork(); //func. to do the offline initial training
}
```

Role Playing Game Example



```
void TrainTheNetwork(){
    double error;
    int counter;
    // here we can dump the initial data of the network
    // before we start training
    while((error > 0.05f) && (counter < 50000)){
        error  = 0.0f;
        ++counter;
        for(int i = 0; i < 10; ++i){
            ai_network.SetInput(0, trainingSet[i][0]);
            ai_network.SetInput(1, trainingSet[i][1]);
            ai_network.SetInput(2, trainingSet[i][2]);
            ai_network.SetInput(3, trainingSet[i][3]);

            ai_network.SetDesiredOutput(0, trainingSet[i][4]);
            ai_network.SetDesiredOutput(1, trainingSet[i][5]);
            ai_network.SetDesiredOutput(2, trainingSet[i][6]);

            ai_network.FeedForward();
            error += ai_network.CalculateError();
            ai_network.BackPropagate();
        }
        error = error / 10.0f;
    }
    //here we can dump the data after the training
}
```

Training Set

```
double trainingSet[10][7] =
{
    //Friends, Hit pts, Enemy engaged,  Range,  Chase, Flock, Evade

    0,         1,           0,          0.2,    0.9,   0.1,   0.1,
    0,         1,           1,          0.2,    0.9,   0.1,   0.1,
    0,         1,           0,          0.8,    0.1,   0.1,   0.1,
    0.1,       0.5,         0,          0.2,    0.9,   0.1,   0.1,
    0,         0.25,        1,          0.5,    0.1,   0.9,   0.1,
    0,         0.2,         1,          0.2,    0.1,   0.1,   0.9,
    0.3,       0.2,         0,          0.2,    0.9,   0.1,   0.1,
    0,         0.2,         0,          0.3,    0.1,   0.9,   0.1,
    0,         0.6,         0,          0.2,    0.1,   0.1,   0.9,
    0.1,       0.2,         0,          0.2,    0.1,   0.1,   0.9
};
```

Runtime Training

- Running the example, we should see that the AI soldiers characters behavior does indeed adapt while playing.
- Assume that we can control how much or how often the player affects and hit the AI characters; if we let the player die without affecting damage on the AI characters, we should see the AI characters attacking the player more often.
- On the other hand, if we let the player make high damage on the AI characters, we should see the AI characters adapting to avoid engagement with the player.