Perceptron training rule, linear units, gradient descent, stochastic gradient descent, delta rule

Relevant Readings: Section 4.4 in Mitchell

CS495 - Machine Learning, Fall 2009
Before we build up Artificial Neural Networks, we will first consider how to train single units.
Training ANN units

► Before we build up Artificial Neural Networks, we will first consider how to train single units
► Recall that perceptrons take a weighted sum of their inputs and produce a 1 or -1 output
Training ANN units

- Before we build up Artificial Neural Networks, we will first consider how to train single units.
- Recall that perceptrons take a weighted sum of their inputs and produce a 1 or -1 output.
  - In this case, we will apply the aptly named *perceptron training rule*.
Before we build up Artificial Neural Networks, we will first consider how to train single units.

Recall that perceptrons take a weighted sum of their inputs and produce a 1 or -1 output.

In this case, we will apply the aptly named *perceptron training rule*.

*Linear units* are like perceptrons, but the output is used directly (not thresholded to 1 or -1).
Training ANN units

- Before we build up Artificial Neural Networks, we will first consider how to train single units.
- Recall that perceptrons take a weighted sum of their inputs and produce a 1 or -1 output.
  - In this case, we will apply the aptly named *perceptron training rule*.
- *Linear units* are like perceptrons, but the output is used directly (not thresholded to 1 or -1).
  - In that case, we will use either *gradient descent* or the *delta rule*.
Training ANN units

- Before we build up Artificial Neural Networks, we will first consider how to train single units.
- Recall that perceptrons take a weighted sum of their inputs and produce a 1 or -1 output.
  - In this case, we will apply the aptly named *perceptron training rule*.
- *Linear units* are like perceptrons, but the output is used directly (not thresholded to 1 or -1).
  - In that case, we will use either *gradient descent* or the *delta rule*. 
Perceptron training rule

- The *perceptron training rule* updates perceptron weights according to training examples as follows:

  \[ w_i \rightarrow w_i + \Delta w_i \]

  where:

  \[ \Delta w_i = \eta (t - o) x_i \]

  - \( w_i \) is the weight associated with the \( i \)th input
  - \( x_i \) is the \( i \)th input
  - \( t \) is the current training example's output value
  - \( o \) is the output of the perceptron under the current training example
  - the learning rate \( \eta \) is a small constant (like .01)
Perceptron training rule

- The *perceptron training rule* updates perceptron weights according to training examples as follows:
  - If the perceptron correctly classifies a training example, don’t do anything
Perceptron training rule

The *perceptron training rule* updates perceptron weights according to training examples as follows:

- If the perceptron correctly classifies a training example, don’t do anything
- If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

More precisely:

- \( w_i \) becomes \( w_i + \Delta w_i \)
- \( \Delta w_i = \eta (t - o) x_i \)
- \( x_i \) is the \( i \)th input
- \( w_i \) is the weight associated with the \( i \)th input
- \( \eta \) is the learning rate (like 0.01)
- \( t \) is the current training example’s output value
- \( o \) is the output of the perceptron under the current training example

Try an example or two
Perceptron training rule

- The *perceptron training rule* updates perceptron weights according to training examples as follows:
  - If the perceptron correctly classifies a training example, don’t do anything
  - If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

- More precisely:
Perceptron training rule

- The *perceptron training rule* updates perceptron weights according to training examples as follows:
  - If the perceptron correctly classifies a training example, don’t do anything
  - If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

- More precisely:
  - $w_i$ becomes $w_i + \Delta w_i$
Perceptron training rule

- The *perceptron training rule* updates perceptron weights according to training examples as follows:
  - If the perceptron correctly classifies a training example, don’t do anything
  - If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

- More precisely:
  - \( w_i \) becomes \( w_i + \Delta w_i \)
  - where:
Perceptron training rule

- The *perceptron training rule* updates perceptron weights according to training examples as follows:
  - If the perceptron correctly classifies a training example, don’t do anything
  - If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

- More precisely:
  - $w_i$ becomes $w_i + \Delta w_i$
  - where:
    - $\Delta w_i = \eta(t - o)x_i$
The perceptron training rule updates perceptron weights according to training examples as follows:

- If the perceptron correctly classifies a training example, don’t do anything
- If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

More precisely:

- $w_i$ becomes $w_i + \Delta w_i$
- where:
  - $\Delta w_i = \eta(t - o)x_i$
  - $x_i$ is the $i^{th}$ input
The *perceptron training rule* updates perceptron weights according to training examples as follows:

- If the perceptron correctly classifies a training example, don’t do anything.
- If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example.

More precisely:

- $w_i$ becomes $w_i + \Delta w_i$.
- where:
  - $\Delta w_i = \eta (t - o) x_i$.
  - $x_i$ is the $i^{th}$ input.
  - $w_i$ is the weight associated with the $i^{th}$ input.
The *perceptron training rule* updates perceptron weights according to training examples as follows:

- If the perceptron correctly classifies a training example, don’t do anything.
- If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example.

More precisely:

- $w_i$ becomes $w_i + \Delta w_i$
- $\Delta w_i = \eta(t - o)x_i$
- $x_i$ is the $i^{th}$ input.
- $w_i$ is the weight associated with the $i^{th}$ input.
- The learning rate $\eta$ is a small constant (like .01).
Perceptron training rule

- The *perceptron training rule* updates perceptron weights according to training examples as follows:
  - If the perceptron correctly classifies a training example, don’t do anything
  - If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

- More precisely:
  - $w_i$ becomes $w_i + \Delta w_i$
  - where:
    - $\Delta w_i = \eta(t - o)x_i$
    - $x_i$ is the $i^{th}$ input
    - $w_i$ is the weight associated with the $i^{th}$ input
    - the *learning rate* $\eta$ is a small constant (like .01)
    - $t$ is the current training example’s output value

- Try an example or two
The perceptron training rule updates perceptron weights according to training examples as follows:

- If the perceptron correctly classifies a training example, don’t do anything
- If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

More precisely:

- $w_i$ becomes $w_i + \Delta w_i$
- where:
  - $\Delta w_i = \eta(t - o)x_i$
  - $x_i$ is the $i^{\text{th}}$ input
  - $w_i$ is the weight associated with the $i^{\text{th}}$ input
  - the learning rate $\eta$ is a small constant (like .01)
  - $t$ is the current training example’s output value
  - $o$ is the output of the perceptron under the current training example

Try an example or two.
The perceptron training rule updates perceptron weights according to training examples as follows:

- If the perceptron correctly classifies a training example, don’t do anything
- If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

More precisely:

- $w_i$ becomes $w_i + \Delta w_i$
- where:
  - $\Delta w_i = \eta(t - o)x_i$
  - $x_i$ is the $i^{th}$ input
  - $w_i$ is the weight associated with the $i^{th}$ input
  - the learning rate $\eta$ is a small constant (like .01)
  - $t$ is the current training example’s output value
  - $o$ is the output of the perceptron under the current training example

Try an example or two
Perceptron training rule

The *perceptron training rule* updates perceptron weights according to training examples as follows:

- If the perceptron correctly classifies a training example, don’t do anything
- If the perceptron incorrectly classifies a training example, each of the input weights is nudged a little bit in the “right direction” for that training example

More precisely:

- $w_i$ becomes $w_i + \Delta w_i$
- where:
  - $\Delta w_i = \eta (t - o) x_i$
  - $x_i$ is the $i^{th}$ input
  - $w_i$ is the weight associated with the $i^{th}$ input
  - the *learning rate* $\eta$ is a small constant (like .01)
  - $t$ is the current training example’s output value
  - $o$ is the output of the perceptron under the current training example

Try an example or two
Perceptron training rule

- **Strength:**
  - If the data is linearly separable and $\eta$ is set to a sufficiently small value, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations.
  - Weakness:
    - If the data is not linearly separable, it will not converge.
Perceptron training rule

- **Strength:**
  - If the data is *linearly separable* and $\eta$ is set to a sufficiently small value, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations.
Perceptron training rule

- **Strength:**
  - If the data is *linearly separable* and $\eta$ is set to a sufficiently small value, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations.

- **Weakness:**
Perceptron training rule

- **Strength:**
  - If the data is *linearly separable* and $\eta$ is set to a sufficiently small value, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations

- **Weakness:**
  - If the data is not linearly separable, it will not converge
Perceptron training rule

Strength:
- If the data is *linearly separable* and $\eta$ is set to a sufficiently small value, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations.

Weakness:
- If the data is not linearly separable, it will not converge.
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron

\[
\text{The output of an } k\text{-input linear unit is } \sum_{i=0}^{k-1} w_i x_i
\]

It isn't reasonable to use a boolean notion of error for linear units, so we need to use something else. We will use a sum-of-squares measure of error \( E \), under hypothesis (weights) \((w_0, \ldots, w_{k-1})\) and training set \( D \):

\[
E(w_0, \ldots, w_{k-1}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2,
\]

where:

- \( t_d \) is training example \( d \)'s output value
- \( o_d \) is the output of the linear unit under \( d \)'s inputs

This \( E \) is a parabola, and has a global minimum.

Gradient descent aims to find the minimum by repeatedly taking a small step in the direction of the gradient:

\[
\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_d i
\]

Pseudocode is given in Table 4.1 in Mitchell.
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an \( k \)-input linear unit is \( \sum_{i=0}^{k-1} w_i x_i \)

It isn't reasonable to use a boolean notion of error for linear units, so we need to use something else. We will use a sum-of-squares measure of error \( E \), under hypothesis (weights) \( (w_0, \ldots, w_{k-1}) \) and training set \( D \):

\[
E(w_0, \ldots, w_{k-1}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2,
\]
where:
- \( t_d \) is training example \( d \)'s output value
- \( o_d \) is the output of the linear unit under \( d \)'s inputs

This \( E \) is a parabola, and has a global minimum. Gradient descent aims to find the minimum by repeatedly taking a small step in the direction of the gradient:

\[
\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_d i
\]
Pseudocode is given in Table 4.1 in Mitchell.
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_i x_i$
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_i x_i$
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error $E$, under hypothesis (weights) $(w_0, \ldots, w_{k-1})$ and training set $D$:

$$E(w_0, \ldots, w_{k-1}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2,$$

where:

- $t_d$ is training example $d$’s output value
- $o_d$ is the output of the linear unit under $d$’s inputs

This $E$ is a parabola, and has a global minimum

Gradient descent aims to find the minimum by repeatedly taking a small step in the direction of the gradient

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_d$$

Pseudocode is given in Table 4.1 in Mitchell
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_i x_i$
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error $E$, under hypothesis (weights) $(w_0, \ldots, w_{k-1})$ and training set $D$:
  - $E(w_0, \ldots, w_{k-1}) = (1/2) \sum_{d \in D} (t_d - o_d)^2$, where:
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an \( k \)-input linear unit is \( \sum_{i=0}^{k-1} w_i x_i \)
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error \( E \), under hypothesis (weights) \( (w_0, \ldots, w_{k-1}) \) and training set \( D \):
  - \( E(w_0, \ldots, w_{k-1}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \), where:
    - \( t_d \) is training example \( d \)’s output value
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_i x_i$
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error $E$, under hypothesis (weights) $(w_0, \ldots, w_{k-1})$ and training set $D$:
  - $E(w_0, \ldots, w_{k-1}) = (1/2) \sum_{d \in D} (t_d - o_d)^2$, where:
    - $t_d$ is training example $d$’s output value
    - $o_d$ is the output of the linear unit under $d$’s inputs
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_i x_i$
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error $E$, under hypothesis (weights) $(w_0, \ldots, w_{k-1})$ and training set $D$:
  - $E(w_0, \ldots, w_{k-1}) = (1/2) \sum_{d \in D} (t_d - o_d)^2$, where:
    - $t_d$ is training example $d$’s output value
    - $o_d$ is the output of the linear unit under $d$’s inputs
- This $E$ is a parabola, and has a global minimum
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_i x_i$
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error $E$, under hypothesis (weights) $(w_0, \ldots, w_{k-1})$ and training set $D$:
  - $E(w_0, \ldots, w_{k-1}) = (1/2) \sum_{d \in D} (t_d - o_d)^2$, where:
    - $t_d$ is training example $d$’s output value
    - $o_d$ is the output of the linear unit under $d$’s inputs
- This $E$ is a parabola, and has a global minimum
- Gradient descent aims to find the minimum by repeatedly taking a small step in the direction of the gradient
A linear unit can be thought of as an unthresholded perceptron

The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_ix_i$

It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else

We will use a sum-of-squares measure of error $E$, under hypothesis (weights) $(w_0, \ldots, w_{k-1})$ and training set $D$:

$E(w_0, \ldots, w_{k-1}) = (1/2) \sum_{d \in D} (t_d - o_d)^2$, where:

- $t_d$ is training example $d$’s output value
- $o_d$ is the output of the linear unit under $d$’s inputs

This $E$ is a parabola, and has a global minimum

Gradient descent aims to find the minimum by repeatedly taking a small step in the direction of the gradient

$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)x_{di}$
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an \( k \)-input linear unit is \( \sum_{i=0}^{k-1} w_i x_i \)
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error \( E \), under hypothesis (weights) \((w_0, \ldots, w_{k-1})\) and training set \( D \):
  - \( E(w_0, \ldots, w_{k-1}) = (1/2) \sum_{d \in D} (t_d - o_d)^2 \), where:
    - \( t_d \) is training example \( d \)’s output value
    - \( o_d \) is the output of the linear unit under \( d \)’s inputs
- This \( E \) is a parabola, and has a global minimum
- **Gradient descent** aims to find the minimum by repeatedly taking a small step in the direction of the gradient
  - \( \Delta w_i = \eta \sum_{d \in D} (t_d - o_d)x_i \)
- Pseudocode is given in Table 4.1 in Mitchell
Linear units and gradient descent

- A linear unit can be thought of as an unthresholded perceptron
- The output of an $k$-input linear unit is $\sum_{i=0}^{k-1} w_i x_i$
- It isn’t reasonable to use a boolean notion of error for linear units, so we need to use something else
- We will use a sum-of-squares measure of error $E$, under hypothesis (weights) $(w_0, \ldots, w_{k-1})$ and training set $D$:
  - $E(w_0, \ldots, w_{k-1}) = (1/2) \sum_{d \in D} (t_d - o_d)^2$, where:
    - $t_d$ is training example $d$’s output value
    - $o_d$ is the output of the linear unit under $d$’s inputs
- This $E$ is a parabola, and has a global minimum
- *Gradient descent* aims to find the minimum by repeatedly taking a small step in the direction of the gradient
  - $\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_d$
- Pseudocode is given in Table 4.1 in Mitchell
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface.

The idea: instead of using the actual error surface's gradient, we use the gradient with respect to one training example at a time.

This leads to the following definition of error with respect to instance $d$:

$$E_d(w_0, \ldots, w_{k-1}) = \frac{1}{2} (t_d - o_d)^2$$

Then gradient descent becomes the delta rule:

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

This is the LMS rule we used in checkers.

Note that the delta rule is almost the same as the perceptron rule.

The difference is that the output value $o$ is continuous, rather than $\pm 1$. 
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface
- *Stochastic gradient descent* can help deal with these issues somewhat

\[
E_d(w_0, \ldots, w_k - 1) = \frac{1}{2} (t_d - o_d)^2
\]

Then gradient descent becomes the delta rule:

\[
\Delta w_i = \eta (t - o) x_i
\]

This is the LMS rule we used in checkers.

Note that the delta rule is almost the same as the perceptron rule.

The difference is that the output value $o$ is continuous, rather than $\pm 1$. 
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface.

- *Stochastic gradient descent* can help deal with these issues somewhat.

- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time.

\[
E_d(w_0, \ldots, w_{k-1}) = \frac{1}{2} (t_d - o_d)^2
\]

Then gradient descent becomes the delta rule:

\[
\Delta w_i = \eta (t - o) x_i
\]

This is the LMS rule we used in checkers.

Note that the delta rule is almost the same as the perceptron rule. The difference is that the output value \( o \) is continuous, rather than \( \pm 1 \).
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface.
- *Stochastic gradient descent* can help deal with these issues somewhat.
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time.
- This leads to the following definition of error with respect to instance $d$:
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface
- *Stochastic gradient descent* can help deal with these issues somewhat
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time
- This leads to the following definition of error with respect to instance $d$: 
  
  $$E_d(w_0, \ldots, w_{k-1}) = (1/2)(t_d - o_d)^2$$
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface
- *Stochastic gradient descent* can help deal with these issues somewhat
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time
- This leads to the following definition of error with respect to instance $d$:
  - $E_d(w_0, \ldots, w_{k-1}) = (1/2)(t_d - o_d)^2$
- Then gradient descent becomes the *delta rule*:
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface
- *Stochastic gradient descent* can help deal with these issues somewhat
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time
- This leads to the following definition of error with respect to instance $d$:
  - $E_d(w_0, \ldots, w_{k-1}) = (1/2)(t_d - o_d)^2$
- Then gradient descent becomes the *delta rule*:
  - $\Delta w_i = \eta(t - o)x_i$
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface.
- *Stochastic gradient descent* can help deal with these issues somewhat.
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time.
- This leads to the following definition of error with respect to instance $d$:
  - $E_d(w_0, \ldots, w_{k-1}) = (1/2)(t_d - o_d)^2$
- Then gradient descent becomes the *delta rule*:
  - $\Delta w_i = \eta(t - o)x_i$
- This is the LMS rule we used in checkers.
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface
- *Stochastic gradient descent* can help deal with these issues somewhat
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time
- This leads to the following definition of error with respect to instance $d$:
  
  $$E_d(w_0, \ldots, w_{k-1}) = \frac{1}{2}(t_d - o_d)^2$$

- Then gradient descent becomes the *delta rule*:
  
  $$\Delta w_i = \eta(t - o)x_i$$

- This is the LMS rule we used in checkers.
- Note that the delta rule is *almost* the same as the perceptron rule.
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface
- **Stochastic gradient descent** can help deal with these issues somewhat
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time
- This leads to the following definition of error with respect to instance \( d \):
  \[
  E_d(w_0, \ldots, w_{k-1}) = \frac{1}{2}(t_d - o_d)^2
  \]
- Then gradient descent becomes the **delta rule**:
  \[
  \Delta w_i = \eta(t - o)x_i
  \]
- This is the LMS rule we used in checkers.
- Note that the delta rule is *almost* the same as the perceptron rule.
  - The difference is that the output value \( o \) is continuous, rather than \( \pm 1 \).
Stochastic gradient descent

- Gradient descent can be slow, and there are no guarantees if there are multiple local minima in the error surface.
- *Stochastic gradient descent* can help deal with these issues somewhat.
- The idea: instead of using the actual error surface’s gradient, we use the gradient with respect to one training example at a time.
- This leads to the following definition of error with respect to instance $d$:
  \[ E_d(w_0, \ldots, w_{k-1}) = \frac{1}{2}(t_d - o_d)^2 \]
- Then gradient descent becomes the *delta rule*:
  \[ \Delta w_i = \eta(t - o)x_i \]
- This is the LMS rule we used in checkers.
- Note that the delta rule is *almost* the same as the perceptron rule.
  \[ \text{The difference is that the output value } o \text{ is continuous, rather than } \pm 1. \]
Delta rule

▶ Strengths:

- Converges to least squares error for the training data
- The data doesn't need to be linearly separable
- Can be used with multi-layer ANNs

▶ Weakness:

- Doesn't necessarily converge to a "perfect" hypothesis on linearly separable data
Delta rule

- **Strengths:**
  - Converges to least squares error for the training data

- **Weakness:**
  - Doesn't necessarily converge to a "perfect" hypothesis on linearly separable data
Delta rule

- **Strengths:**
  - Converges to least squares error for the training data
  - The data doesn’t need to be linearly separable
Delta rule

- **Strengths:**
  - Converges to least squares error for the training data
  - The data doesn’t need to be linearly separable
  - Can be used with multi-layer ANNs

- **Weakness:**
  - Doesn’t necessarily converge to a “perfect” hypothesis on linearly separable data
Delta rule

- **Strengths:**
  - Converges to least squares error for the training data
  - The data doesn’t need to be linearly separable
  - Can be used with multi-layer ANNs

- **Weakness:**
  

Delta rule

Strengths:
- Converges to least squares error for the training data
- The data doesn’t need to be linearly separable
- Can be used with multi-layer ANNs

Weakness:
- Doesn’t necessarily converge to a “perfect” hypothesis on linearly separable data
Delta rule

- **Strengths:**
  - Converges to least squares error for the training data
  - The data doesn’t need to be linearly separable
  - Can be used with multi-layer ANNs

- **Weakness:**
  - Doesn’t necessarily converge to a “perfect” hypothesis on linearly separable data