Neural Networks

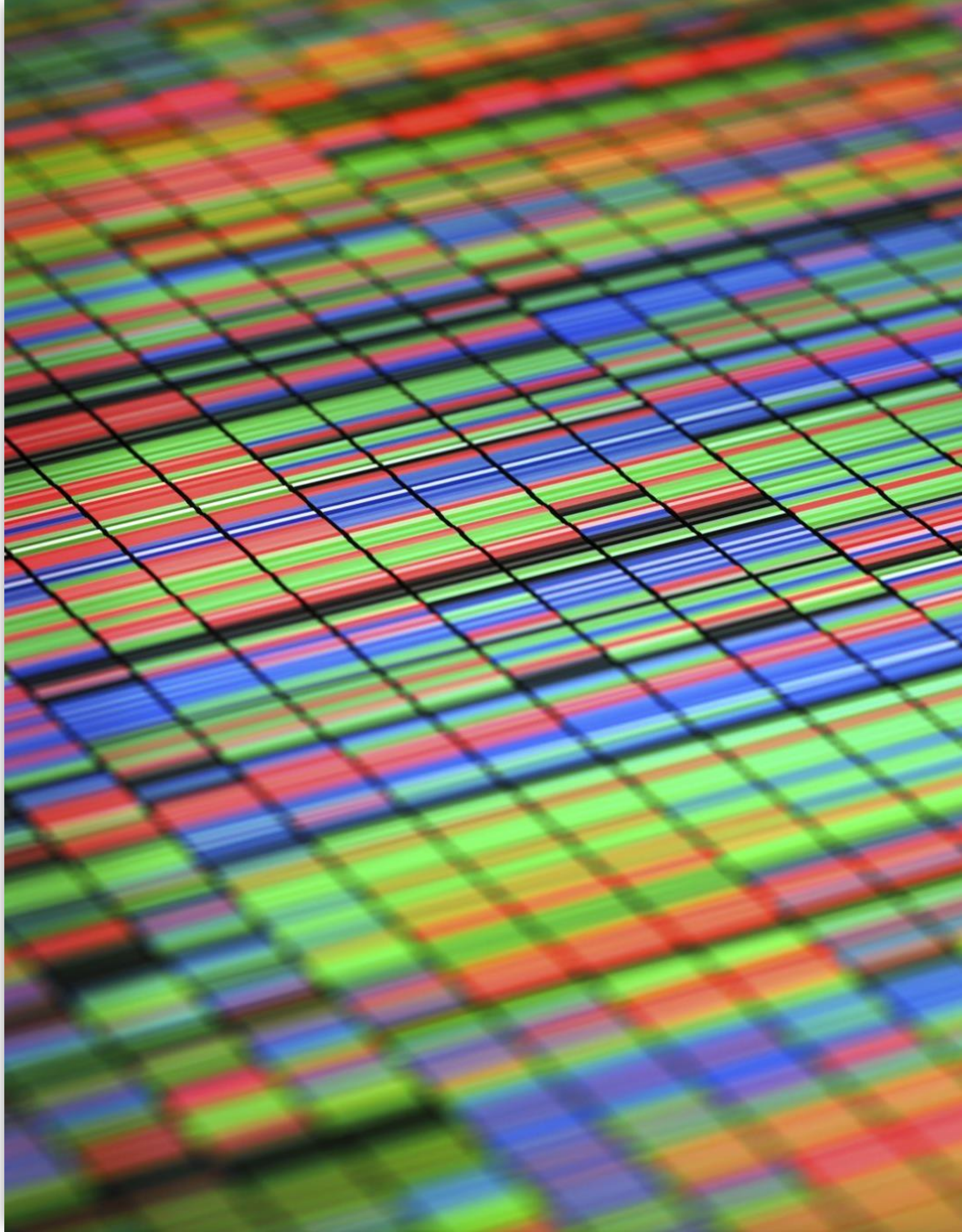
Perceptrons and Single-Layer Networks

CSCI 4850/5850

Pattern Recognition

- The ***classification problem*** involves assigning one of a discrete set of **category labels** to an **object**, based on a collection of measurements (i.e **features**) of that object:

- The **statistical pattern recognition problem** involves finding a function which does a good job of classifying objects, based on a combination of prior knowledge and a **data set** of labeled objects (i.e. **feature vectors**).
  - Goodness is formalized in terms of a performance measure, typically an **error function**.
  - The main goal is **generalization**: good classification performance on objects ***not*** used to find the classification function (i.e. “held-out” feature vectors).

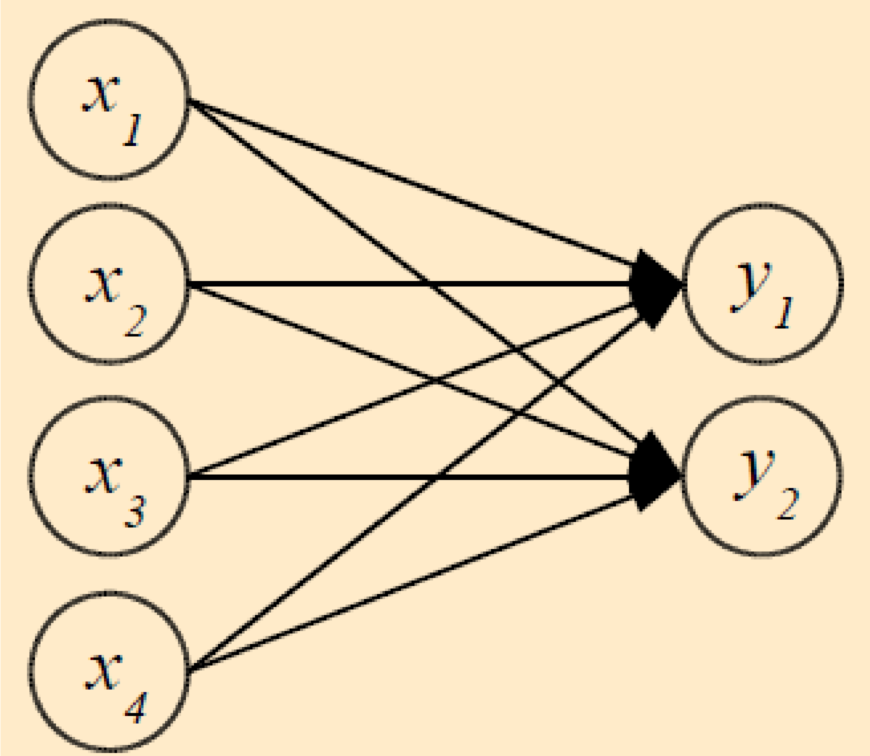


Feature Extraction

- Often, features provided may not be related to the class membership in a simple way, but other features calculated *based on* the input features may be. Thus, transforming input features into a more useful set of features before finding class regions is common.
  – One key advantage of *deep-learning* networks that we will see later is that such transformations are not as necessary as in other machine learning methods, but studying these transformations can help us understand why.
- Such methods are known as feature **pre-processing** or **feature extraction** which often require **prior knowledge** concerning the problem to domain to be successful.
- Feature extraction often seeks to leverage **invariances** (translation, rotation, scaling, skew, etc.)

Example: Single-layer Network

- Consider the case where we have a network comprised of only two layers: **input** and **output**
- The neural units in the input layer are **fully-connected** to the units in the output layer: every input unit, x_i, connects to every output unit, y_j.
- All weights initially set to **zero**.
- Output units use the **linear** activation function (without a bias weight, w_0).



[Note that the **input layer** has no neural units upstream**. By convention, we set the activations of these units directly to the input vectors we are processing with the network**. Therefore, since they perform no computation, we don't count them in the number of layers in the network (hence, single-layer network).]

A Training Procedure

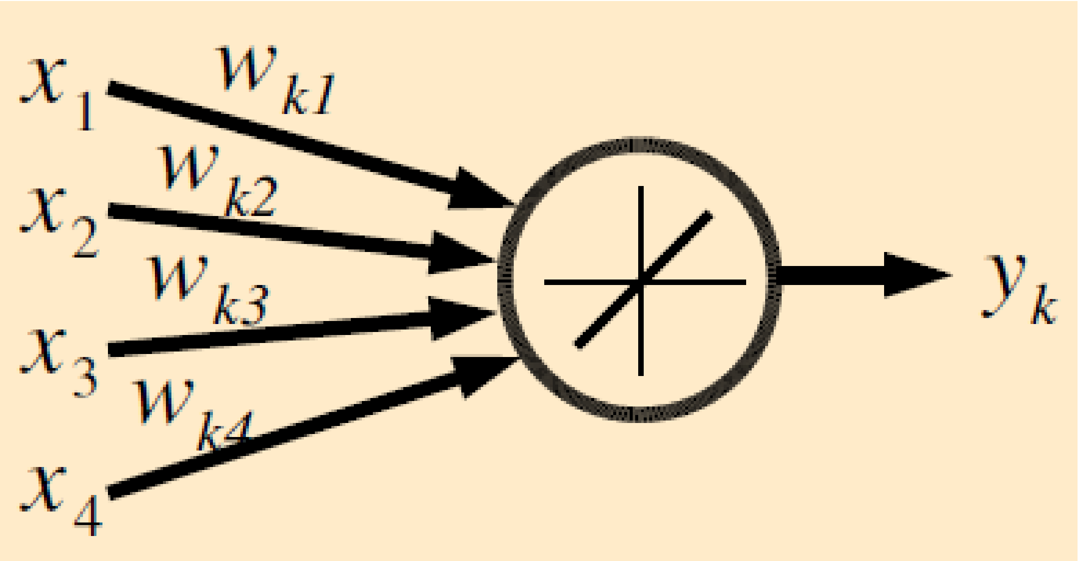
- Consider that we are calculating c=2 linear discriminant regions given d=5 dimensional input vectors:

$$\begin{array}{cccccccc} +1 & -1 & +1 & -1 & \Rightarrow & +1 & +1 \\ +1 & +1 & +1 & +1 & \Rightarrow & +1 & -1 \\ +1 & +1 & +1 & -1 & \Rightarrow & -1 & -1 \\ +1 & -1 & -1 & +1 & \Rightarrow & -1 & +1 \end{array}$$

- We hope to achieve **good performance** on these training patterns, i.e. we want the network to produce the c-dimensional output pattern on the right from the output units when providing the d-dimensional pattern on the left to the input layer
- We will present these patterns to the network one at a time to our network during learning.
- Each pattern presentation is called a **training trial**
- Exposure to all patterns in the set counts as a **training epoch**

A Training Procedure

How can we adapt the weights to produce a “better” output?

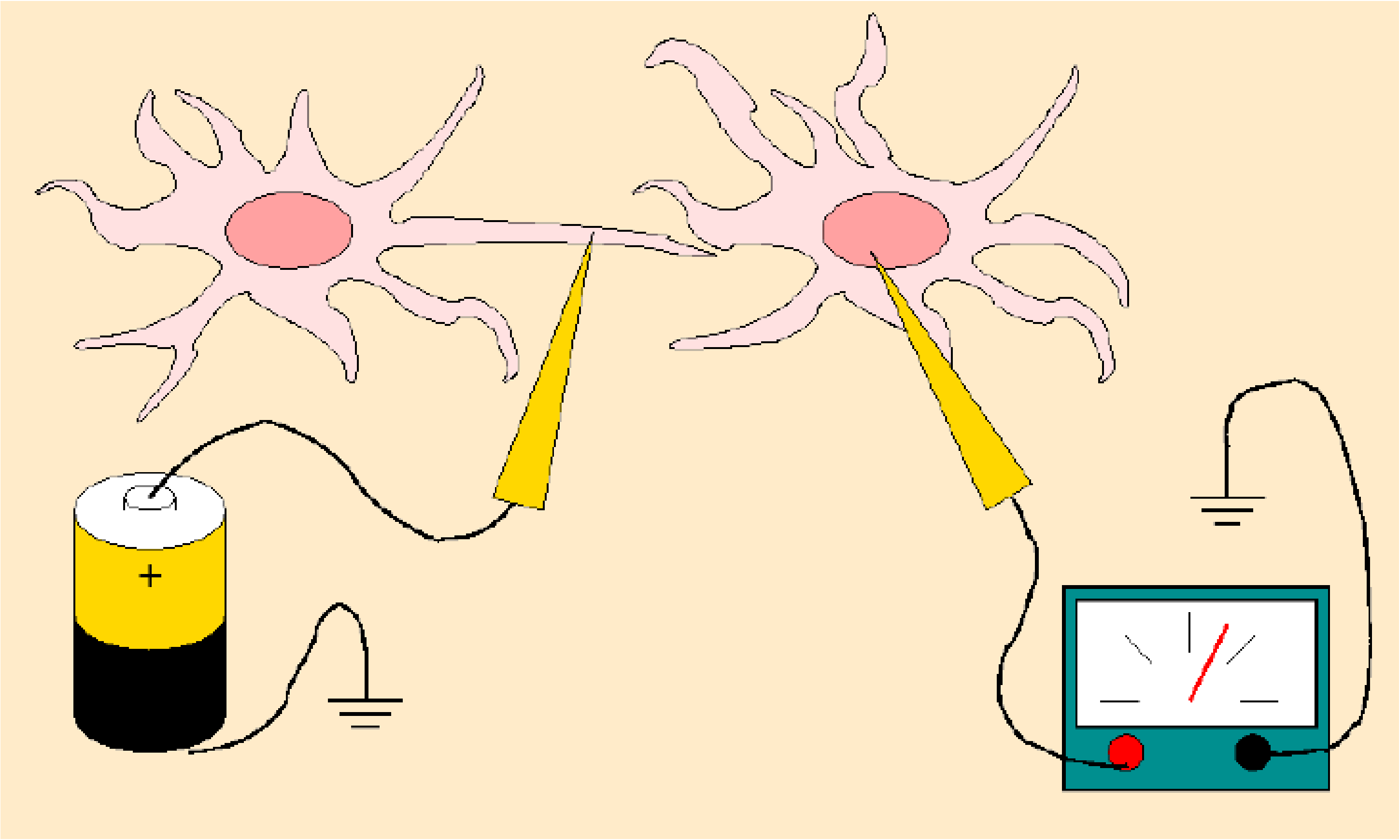


$$\vec{y} = W\,\vec{x}$$

$$y_k = \sum_i x_i\, w_{ki}$$

Biological Inspiration

- Long-term potentiation (LTP)
- Long-term depression (LTD)



Hebbian Learning (1949 theory, 1982 practice)

- Modify each connection weight in proportion to the product of the activation levels on both sides of the connection.

$$\Delta w_{ki} = \eta x_i t_k \qquad\qquad w_{ki}^{new} = w_{ki}^{old} + \Delta w_{ki}$$

- The target value, t_k, is the desired output for output unit, y_k, given the corresponding input vector. The parameter, η, is called the learning rate, and specifies the step size of the weight update.
- If starting from W=0, then after N epochs W will be:

$$w_{ki} = \eta N_e \sum_{n=1}^{N} x_i^n t_k^n$$

Hebbian/Correlation Learning

- The correlation coefficient (normalized covariance) for an input and target value, given both have zero mean:

$$r_{ki} = \frac{cov[x_i, t_k]}{\sqrt{var[x_i]\, var[t_k]}} = \frac{\frac{1}{N}\sum_{n=1}^{N} x_i^n\, t_k^n}{\sqrt{\left(\frac{1}{N}\sum_{n=1}^{N}(x_i^n)^2\right)\left(\frac{1}{N}\sum_{n=1}^{N}(t_k^n)^2\right)}} = k_{train}\sum_{n=1}^{N} x_i^n\, t_k^n$$

Hebbian = Correlation

- Weight values will be proportional to the correlation between the input and the output

$$r_{ki} = k_{train} \sum_{n=1}^{N} x_i^n t_k^n \quad + \quad w_{ki} = \eta N_e \sum_{n=1}^{N} x_i^n t_k^n$$

$$\Downarrow$$

$$w_{ki} = \frac{\eta N_e}{k_{train}} r_{ki}$$

Successful Hebbian Learning

- Proven: if the input patterns are **mutually orthogonal**, then Hebbian learning will result in outputs that are **proportional** to the desired targets
- But it's not enough:

$$
\begin{array}{cccccc}
+1 & -1 & +1 & -1 & \Rightarrow & +1 \\
+1 & +1 & +1 & +1 & \Rightarrow & +1 \\
+1 & +1 & +1 & -1 & \Rightarrow & -1 \\
+1 & -1 & -1 & +1 & \Rightarrow & -1
\end{array}
$$

- There is a weight vector which can solve this problem, but Hebbian learning cannot find it… $(-1,-1,+2,+1)^T$

- **Why?** Weights updates are not sensitive to **other connections** (even those coming into the same output unit) are doing (too local).

Error-Correction Learning

Hebbian learning involves learning correlations, but *this is not the objective of classification or regression problems*.

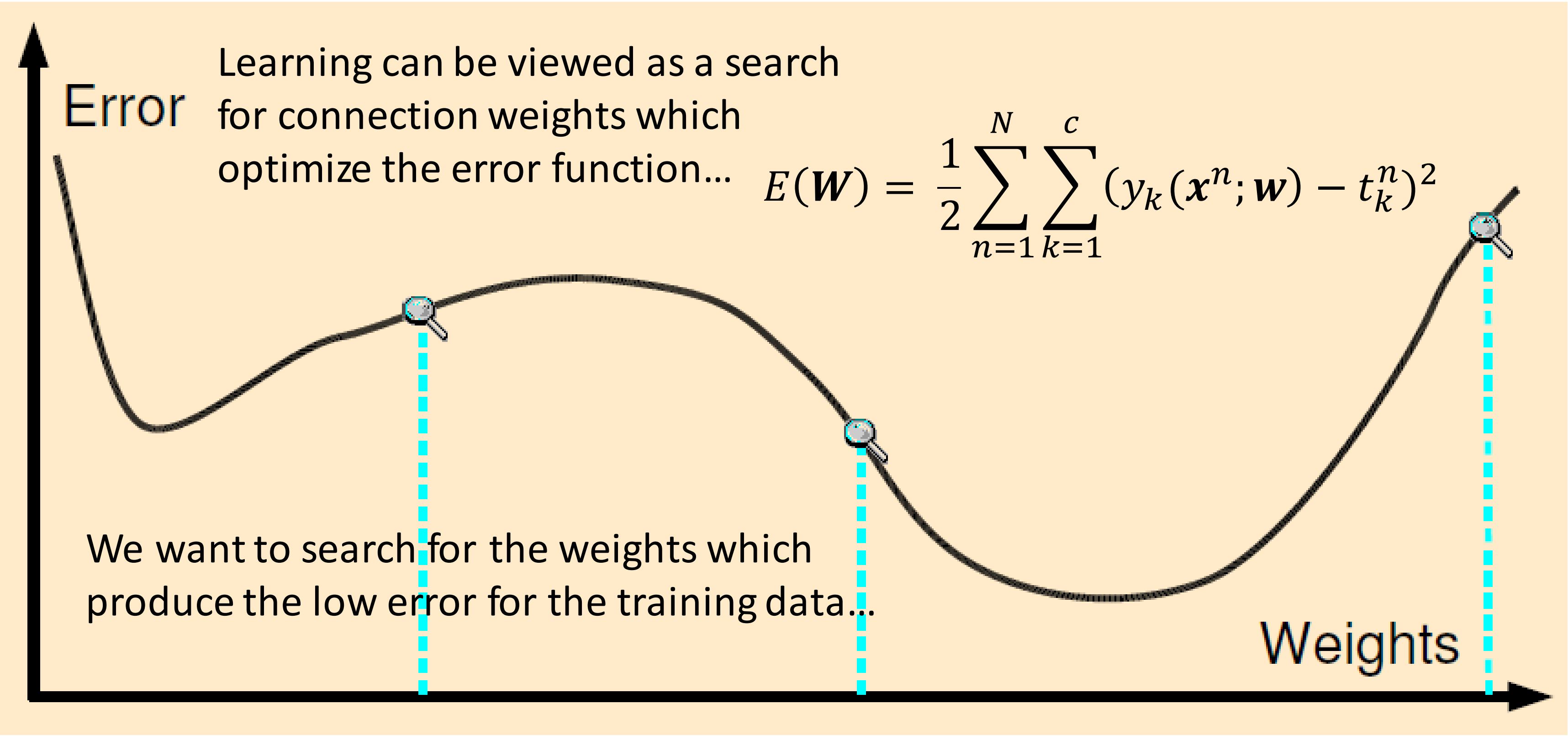
Error-correction learning involves using an explicit measure of **loss/error** which should be **minimized** to produce good performance.

One example of such a function is the sum-squared error (SSE):

$$E(\boldsymbol{W}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}(y_k(\boldsymbol{x}^n;\boldsymbol{w}) - t_k^n)^2$$

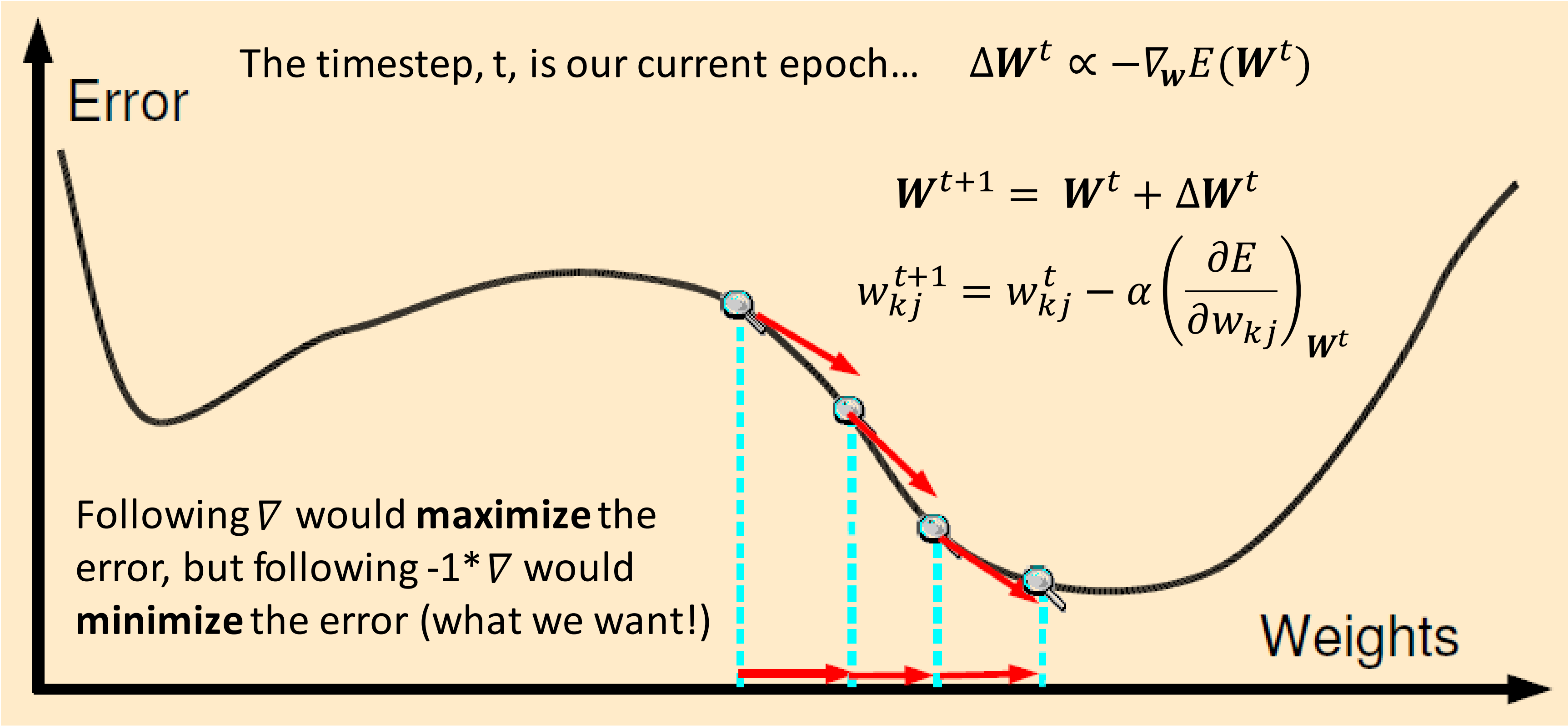
The Loss/Error Function

- Notice that by **summing over** all of the individual training vectors, $\mathbf{x}^n$, the error function becomes **dependent** *only on the **weights*** in the network…



Gradient Descent

- This search isn't uninformed since we can calculate the **partial derivative** of the error with respect to **each weight**



Stochastic Gradient Descent

In practice, some **data sets** are very **large**, so a full pass through the data is computationally expensive (long time to wait between weight updates = **slow learning**)

In neuroscience, we see evidence of learning **before** a completing a full sequence of experiences

A variant of gradient descent called **stochastic gradient descent** is therefore often used, where we compute the error for each training pattern (or a subset of patterns)

This variant doesn't follow the gradient in precisely the same manner since the particular patterns experienced and the order in which they are experienced differs between epochs…

$$E^Q(\boldsymbol{W}) = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{c}\left(y_k\left(\boldsymbol{x}^{q^p};\boldsymbol{w}\right) - t_k^{q^p}\right)^2$$

$$where\ Q \in \{1 \dots N\}\ of\ length\ P$$

One **batch** is now a complete pass through the **subset** Q:

P=N: **gradient descent**

P < N: **stochastic gradient descent**

Non-linear Activation Function

- Result: the Delta Rule
  - Least Mean Squares
  - Widrow-Hoff
  - Adaline Rule

$$y_k(\boldsymbol{x}^n;\boldsymbol{W}) = g(net_k^n)$$

$$net_k^n = \sum_{i=1}^{M} w_{ki}x_i^n$$

$$\frac{\partial y_k(\boldsymbol{x}^n;\boldsymbol{W})}{\partial w_{kj}} = \frac{\partial g(net_k^n)}{\partial net_k^n}\frac{\partial net_k^n}{\partial w_{kj}}$$

$$= g'(net_k^n)\frac{\partial}{\partial w_{kj}}\left(\sum_{i=1}^{M} w_{ki}x_i^n\right)$$

$$\frac{\partial y_k(\boldsymbol{x}^n;\boldsymbol{W})}{\partial w_{kj}} = g'(net_k^n)x_j^n$$

$$\frac{\partial E^n(\boldsymbol{W})}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}}\left(\frac{1}{2}(y_k(\boldsymbol{x}^n;\boldsymbol{W}) - t_k^n)^2\right)$$

$$= (y_k(\boldsymbol{x}^n;\boldsymbol{W}) - t_k^n)\frac{\partial}{\partial w_{kj}}(y_k(\boldsymbol{x}^n;\boldsymbol{W}) - t_k^n)$$

$$\delta_k^n = (y_k(\boldsymbol{x}^n;\boldsymbol{W}) - t_k^n)$$

$$\frac{\partial E(\boldsymbol{W})}{\partial w_{kj}} = \delta_k^n\frac{\partial y_k(\boldsymbol{x}^n;\boldsymbol{W})}{\partial w_{kj}} = \delta_k^n g'(net_k^n)x_j^n$$

$$\Delta w_{kj}^n = -\alpha\delta_k^n g'(net_k^n)\, x_j^n$$

The Perceptron (1957, Rosenblatt)

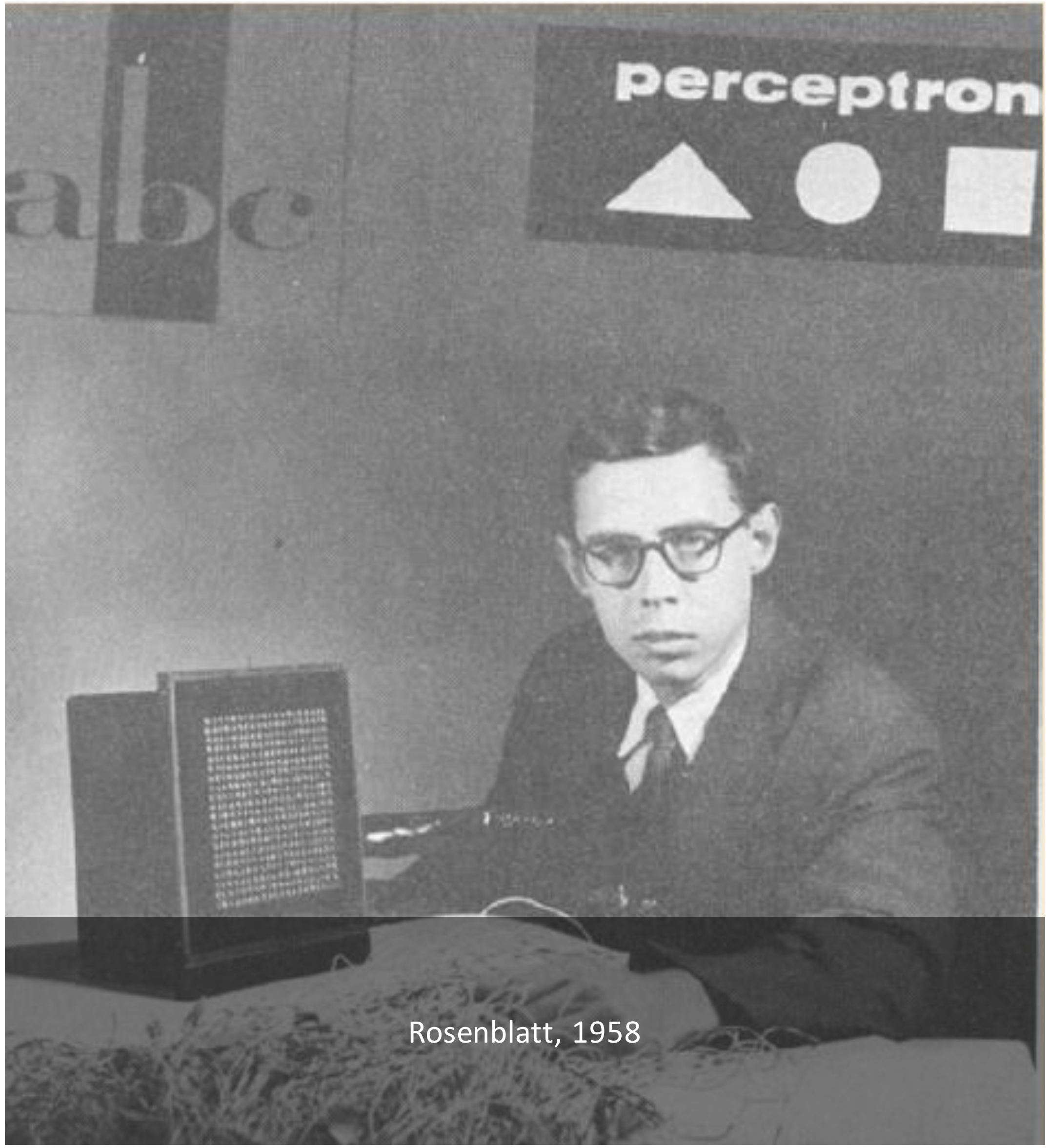
- Delta Rule requires functions to be differentiable
- However, a slight modification makes error correction possible with a discontinuous activation function (eg. step function)
  - Let g be the step function -1 to 1 centered at 0

$$if\ y_k^n = t_k^n\ then\ let\ \Delta w_{kj} = 0$$
$$if\ y_k^n > t_k^n\ then\ let\ \Delta w_{kj} = -\alpha x_j^n$$
$$if\ y_k^n < t_k^n\ then\ let\ \Delta w_{kj} = +\alpha x_j^n$$

$$\Delta w_{kj} = \alpha x_j^n t_k^n$$

Similar to the delta rule, but with an error term that is always 0, -1 or +1

The Perceptron Convergence Theorem



Rosenblatt, 1958

Learning: Some Observations

- In classification, we start with a rich observation of features and end with a discrete class assignment
- In regression, we start with a rich observation of features and often end with only a continuous value of interest
- Classification and regression involve throwing away information
  - Learning involves throwing away information
  - Memorization is **not** learning
  - Need to learn a *general* concept/function

Key Concept: Generalization

- Consider a stock option problem…
  - You want to know if a stock will increase or decrease in value in the next month (classification)
  - You want to know what the value of a stock will be at the end of the next month (regression)
  - Classification and Regression are closely related concepts…
- We must utilize data (eg. past stock option properties and performance information) to **train** a network to make accurate **predictions**
- Performance improves over time for predictions on this *training data*, but we really **want to do better on data not experienced while training**
- That is, we want the network to **generalize** to new experiences
- For this, we often separate **data sets** into three distinct sets
  - **Training**: used to learn the **parameters** (i.e. weights)
  - **Validation**: used to check generalization and tune **hyper-parameters** (eg. α, number of layers, number of units, etc.)
  - **Testing**: used to check generalization performance –after- training/tuning (no more parameter/hyperparameter changes allowed)

Multiclass Activation Function = Softmax

Softmax Activation Function

$$y_k = \frac{e^{net_k}}{\sum_{c=1}^{C} e^{net_c}}$$

$$\frac{\partial y_j}{\partial net_k} = -y_j y_k \qquad \frac{\partial y_k}{\partial net_k} = y_k - y_k^2$$

Categorical CrossEntropy Loss Function

$$E = -\sum_{n=1}^{N}\sum_{c=1}^{C} t_c^n \ln y_c^n$$

$$\delta_k = y_k - t_k$$

The softmax activation function is a generalization of the sigmoid activation function for multiple classes

It considers the relative activity of the other units and scales activity accordingly

Each activation will be between 0 and 1, just like the sigmoid

The sum of the activations across all output units will be 1, which is appropriate for representing discrete probability distributions