Neural Networks and Deep Learning

Learning II

Nicholas Dronen

Department of Computer Science
dronen@colorado.edu

January 28, 2019

University of Colorado **Boulder**

Review

The Perceptron

Multilayer Neural Networks and Backprop

Autoencoders and Interpolation

(c) Sequences of points interpolated at different depths

Better Mixing via Deep Representations, Bengio et al, 2012

Where We're Going

Different models, capabilities, tasks, depths.

Where We're Going

Different models, capabilities, tasks, depths.

- Hebbian learning - linear - regression - shallow 😕

Where We're Going

Different models, capabilities, tasks, depths.

- Hebbian learning - linear - regression - shallow 😕
- LMS (delta rule) - linear - regression - shallow 😐

Where We're Going

Different models, capabilities, tasks, depths.

- Hebbian learning - linear - regression - shallow 😕
- LMS (delta rule) - linear - regression - shallow 😐
- Perceptrons - non-linear - classification - shallow 🙂

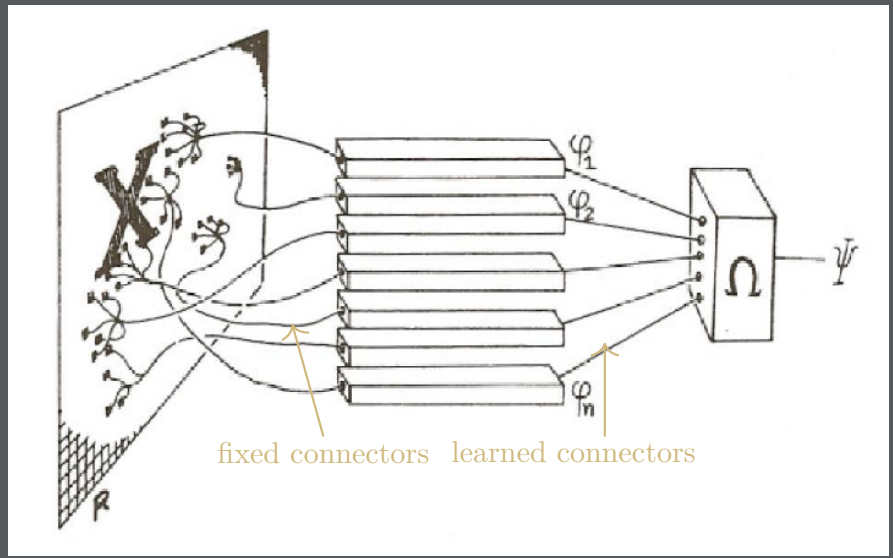
Where We're Going

Different models, capabilities, tasks, depths.

- Hebbian learning - linear - regression - shallow 😕
- LMS (delta rule) - linear - regression - shallow 😐
- Perceptrons - non-linear - classification - shallow 🙂

Perceptron

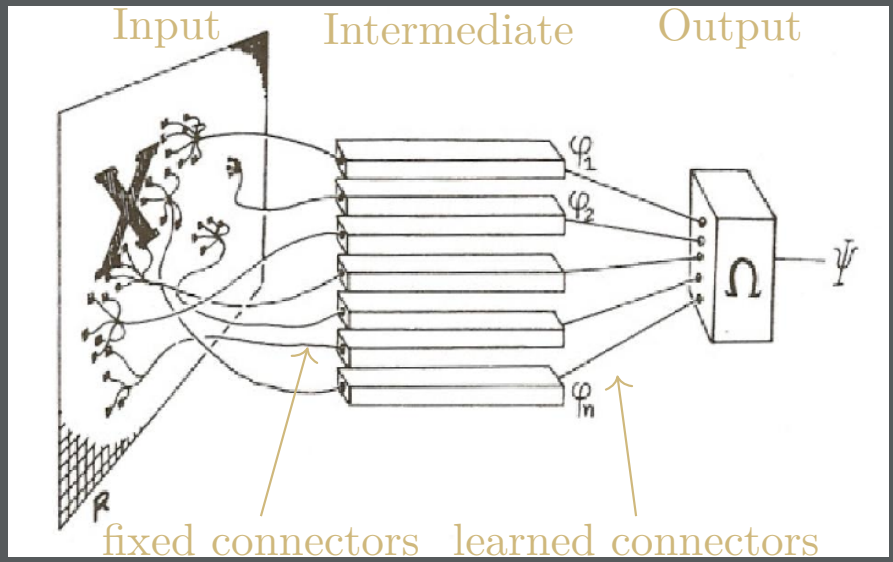
The Perceptron is a device capable of computing all predicates Ψ that are linear in some set $\{\varphi_1, \varphi_2, \varphi_3\}$ of partial predicates.



Some confusion about definition. Perceptron is used to refer to -

- This architecture
- A binary threshold unit
- A learning rule

Architecture



Intermediate units compute some binary function, φ_i, of the inputs. Output units compute some binary function, Ψ_1 of the intermediate units.

$$\Psi = \begin{cases} 1 & \sum w_i \varphi_i > \theta \\ 0 & otherwise \end{cases}$$

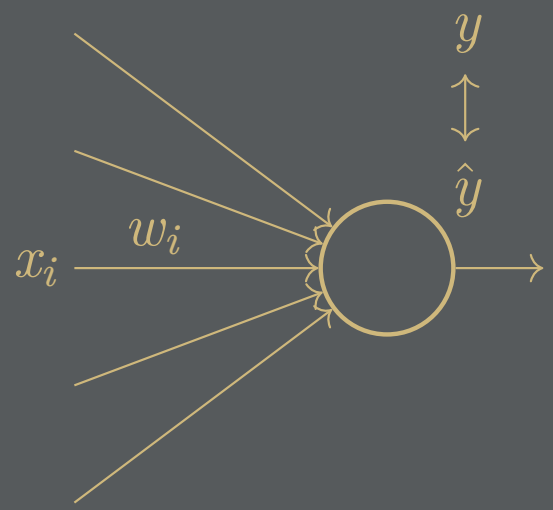
Perceptron Learning Rule (Rosenblatt, 1962)

1. Initialize $\mathbf{w} = 0$
2. Process example i, (x_i, y_i).
3. Update as follows

$$\Delta w_i = \begin{cases} 0 & \text{if } \hat{y} = y \\ x_i & \text{if } \hat{y} = 0 \text{ and } y = 1 \\ -x_i & \text{if } \hat{y} = 1 \text{ and } y = 0 \end{cases}$$

$\Delta w_i = (y - \hat{y}) x_i \leftarrow$ Delta rule, same as LMS!

No learning rate. Why?



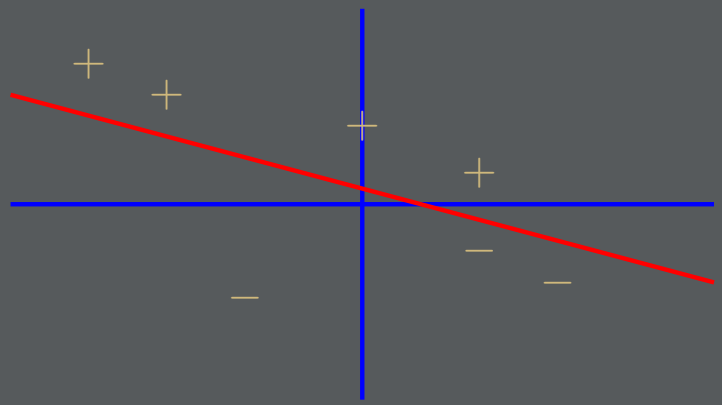
$$\hat{y} = \begin{cases} 1 & \sum_i w_i x_i > 0 \\ 0 & otherwise \end{cases}$$

Precondition of (Guaranteed) Success

The Perceptron learning rule is guaranteed to succeed if the data are linearly separable.

- A hyperplane must exist that can separate positive and negative examples
- The weights define this hyperplane

E.g. A network with 2 inputs, 1 output



$$\hat{y} = \begin{cases} 1 & w_1 x_1 + w_x x_2 + b > 0 \\ 0 & otherwise \end{cases}$$

$+ : y = 1$

$- : y = 0$

$- : w_1 x_1 + w_x x_2 + b = 0$

Capabilities

- Low-order problems are solved by combining a few input features
- High-order problems → require an exponential number of features.

Perceptrons *can* solve low-order problems.

Limitations of Perceptrons

What's needed to perform translation- and rotation-invariant object recognition?

What would a Perceptron's weights need to be tuned to detect?

- Position
- Orientation
- Scale

This is exponential in the number of input features (e.g. $h \times w$, where h (w) is the height (width) of an input image).

Informal Proof of Perceptron Convergence (Rosenblatt, 1962)

Consider the case $y = 0$, $\hat{y} = 1$

- $\hat{y} = 1$ implies $wx^T > 0$
- $w' = w - x$

Performance only improves if

- $w'x^T < wx^T$
- $(w - x)x^T < wx^T$
- $xx^T > 0$

Always true because

- $xx^T = \|x\|^2$

$$\hat{y} = \begin{cases} 1 & \sum_i w_i x_i > 0 \\ 0 & otherwise \end{cases}$$

$$\Delta w_i = \begin{cases} 0 & \text{if } \hat{y} = y \\ x_i & \text{if } \hat{y} = 0 \text{ and } y = 1 \\ -x_i & \text{if } \hat{y} = 1 \text{ and } y = 0 \end{cases}$$

Another Proof of Perceptron Convergence (Novikoff, 1962)

- On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622, 1962.
- Perceptron Mistake Bounds, Mohri and Rostamizadeh
- Convergence Proof for the Perceptron Algorithm, Michael Collins

We will cover this in the lecture on optimization.

Perceptrons and Feature Engineering

Major issue with perceptron architecture: we must specify the representation (a.k.a *features*, input vectors $\mathbf{x}$)

- Features must be designed and implemented $\rightarrow$ feature *engineering*
- Exponential number of hidden can always solve problem
- But leads to large network + poor generalization

With domain knowledge, we *can* engineer appropriate features, but is that what we want?

Feature Learning, Representation Learning

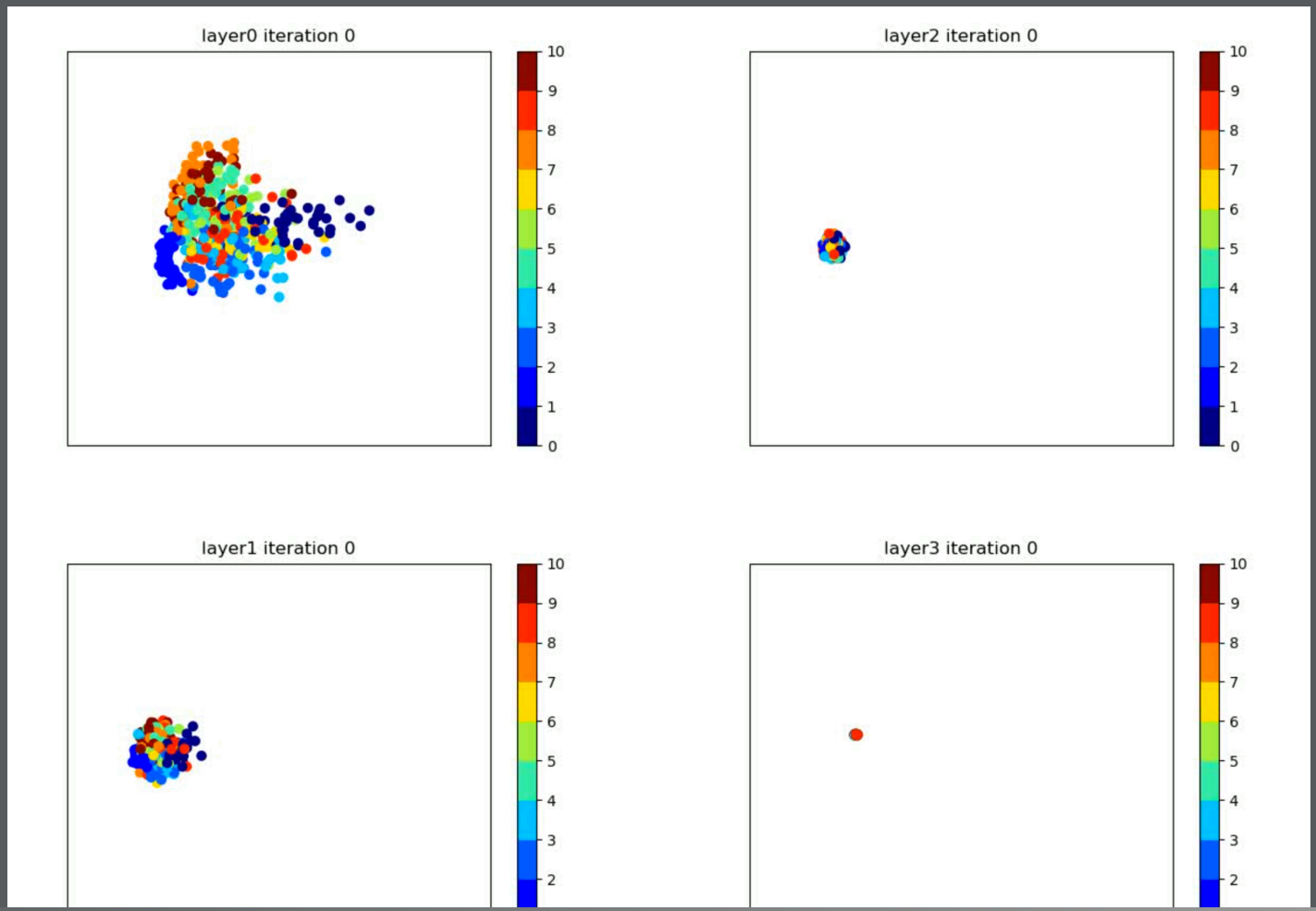
A better approach: feature learning, representation learning.

- Still supervised. Training data encodes:
  - "Raw" inputs
  - Desired outputs/targets
- The state of data as they pass through the network's hidden units are *learned features* or *representations*.
- The network must learn weights so the hidden representations/features/states maximize performance on the supervised task *at the output layer*.
- To learn good features, the supervised error must be pushed back through the network. How?

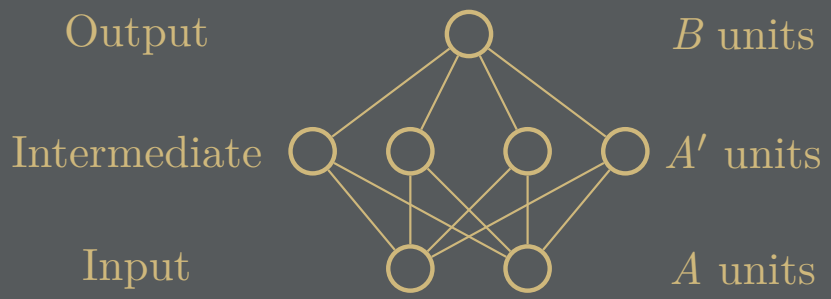
What problems does this approach solve?

Feature/Representation Learning Video

This video shows how each layer of a 4-layer feed-forward network represents (a sample from) the MNIST test set during training.

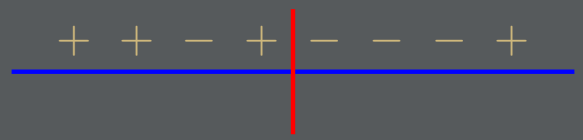


The Power of Intermediate Layers (=Representations)

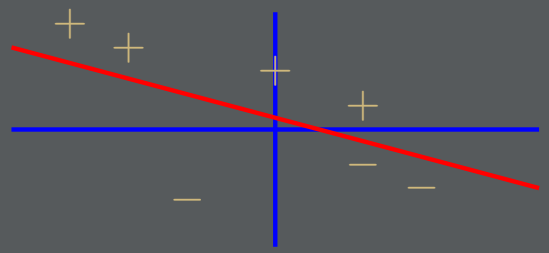


Projecting input vector from an A-dimensional space to A'-dimensional space.

E.g.



classification error



no error

Extending LMS to handle squashing non-linearities and hidden units

First, let's derive the LMS update rule (delta rule) again.

$$\Delta w_{ji} \approx -\frac{\partial E}{\partial w_{ji}} = -\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial w_{ji}}$$

$$E = \sum_k \frac{1}{2}(y_k - o_k)^2$$

$$o_j = \sum_k w_{jk} o_k$$

$$\frac{\partial E}{\partial o_j} = -(y_j - o_j) \qquad (1)$$

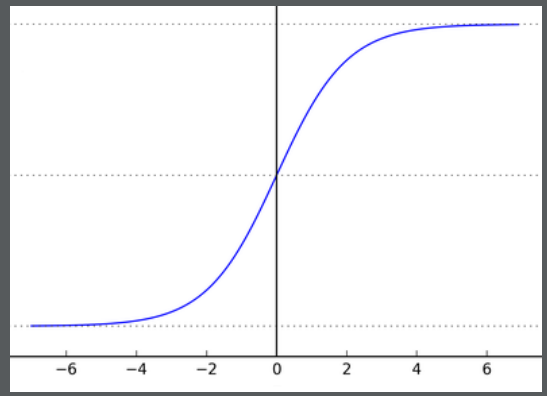
$$\frac{\partial o_j}{\partial w_{ji}} = o_i \qquad (2)$$

$$\approx (y_j - o_j) o_i$$

Extending LMS to handle squashing non-linearities and hidden units

Suppose output unit has sigmoid squashing function

$$o_j = \frac{1}{1 + e^{-net_j}}$$
$$o_j = (1 + e^{-net_j})^{-1}$$



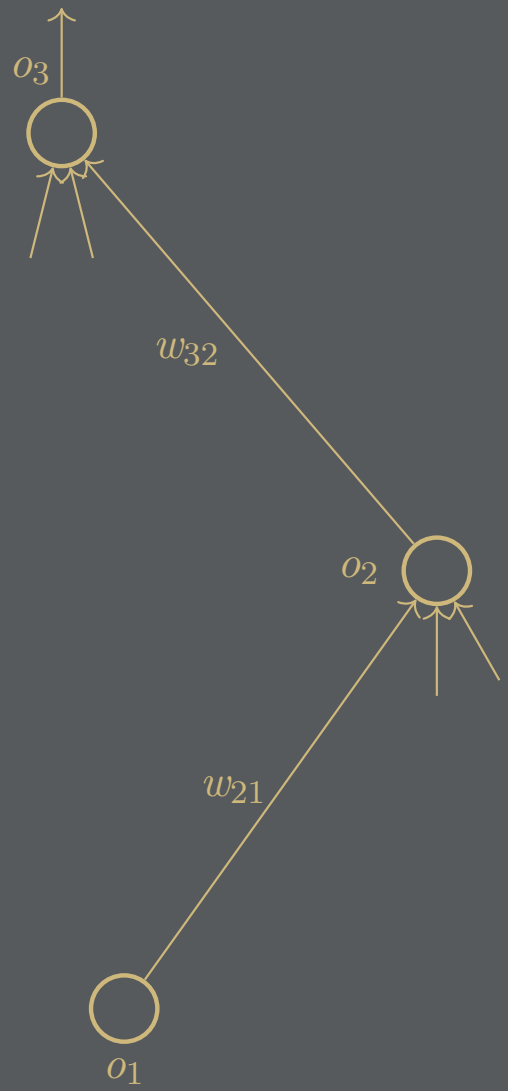
$$\frac{\partial o_j}{\partial net_j} = -1 \cdot (1 + e^{-net_j})^{-2} . - e^{-net_j} \tag{3}$$
$$= \frac{1}{1 + e^{-net_j}} \frac{e^{-net_j}}{1 + e^{-net_j}} \tag{4}$$
$$= \frac{1}{1 + e^{-net_j}} \left(1 - \frac{1}{1 + e^{-net_j}}\right) \tag{5}$$
$$= o_j(1 - o_j) \tag{6}$$

Extending LMS to handle squashing non-linearities and hidden units

From (1), (2) and (6) -

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \approx -(y_j - o_j)o_j(1 - o_j)o_i$$

Back Propagation (Reinhard, Hinton, Willliams, LeCun, Parker, Werbos)



$$E = \frac{1}{2}(y_3 - o_3)^2 \qquad \frac{\partial E}{\partial o_3} = -(y_3 - o_3)$$

$$o_3 = \frac{1}{1 + e^{-net_3}} \qquad \frac{\partial o_3}{\partial net_3} = o_3(1 - o_3)$$

$$net_3 = \sum_i w_{3i} o_i \qquad \frac{\partial net_3}{\partial w_{32}} = o_2$$

$$\frac{\partial net_3}{\partial o_2} = w_{32}$$

$$o_2 = \frac{1}{1 + e^{-net_2}} \qquad \frac{\partial o_2}{\partial net_2} = o_2(1 - o_2)$$

$$net_2 = \sum_i w_{2i} o_i \qquad \frac{\partial net_2}{\partial w_{21}} = o_1$$

Back Propagation (Rumelhart, Hinton, Willliams; LeCun; Parker; Werbos)

$$\frac{\partial E}{\partial w_{32}} = \frac{\partial E}{\partial o_3}\frac{\partial o_3}{\partial net_3}\frac{\partial net_3}{\partial w_{32}} \approx -(y_3 - o_3)o_3(1 - o_3)o_2$$

$$\frac{\partial E}{\partial w_{21}} = \frac{\partial E}{\partial o_2}\frac{\partial o_2}{\partial net_2}\frac{\partial net_2}{\partial w_{21}} \approx -(y_3 - o_3)o_3(1 - o_3)w_{32}o_2(1 - o_2)o_1$$

$$\frac{\partial E}{\partial o_2} = \frac{\partial E}{\partial o_3}\frac{\partial o_3}{\partial net_3}\frac{\partial net_3}{\partial o_2} \approx -(y_3 - o_3)o_3(1 - o_3)w_{32}$$

What it boils down to

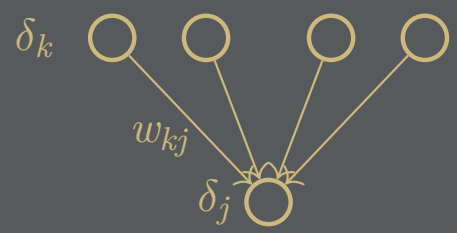
$$\Delta w_{ji} = \epsilon \delta_j o_i \tag{7}$$

For output unit,

$$\delta_j = (y_j - o_j)\, o_j(1 - o_j) \tag{8}$$

For hidden unit,

$$\delta_j = \left[\sum_k \delta_k w_{kj} \right] o_j(1 - o_j) \tag{9}$$



Δw_{ji} for output unit is the same as LMS with a nonlinear output.
(LMS $\equiv$ delta rule back prop $\equiv$ generalized delta rule)

- Two phase process : Forward (activation propagation) phase and Backward (error propagation phase)

- As with the LMS rule, back prop performs gradient descent in error space, i.e. finding best set of weights that minimize error. weights = all weights (biases) in network.
- Back propagation works for arbitrarily deep feedforward networks.