Neural networks

Logistic regression

Suppose $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$.

Logistic regression: statistical model for $Y \mid \boldsymbol{X} = \boldsymbol{x}$ for each $\boldsymbol{x} \in \mathbb{R}^d$:

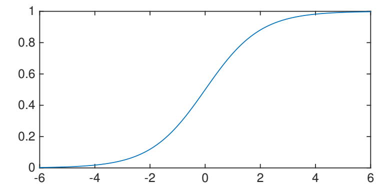
$$\mathcal{P} \;=\; \left\{ P_{\boldsymbol{w}} : \boldsymbol{w} \in \mathbb{R}^d \right\},$$

where

$$f_{\boldsymbol{w}}(\boldsymbol{x}) \;:=\; P_{\boldsymbol{w}}(Y = 1 \mid \boldsymbol{X} = \boldsymbol{x}) \;=\; g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle)$$

and

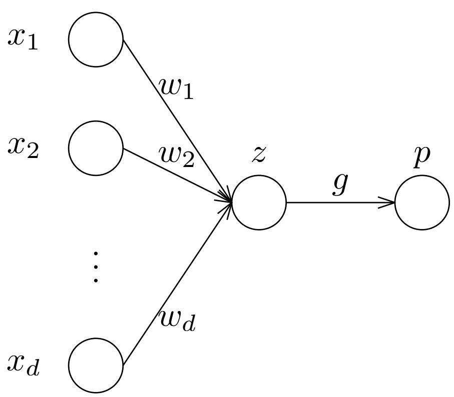
$$g(z) \;:=\; \operatorname{logistic}(z) \;=\; \frac{1}{1 + e^{-z}}\,.$$



(Bias term omitted for simplicity.)

Logistic regression

Network diagram for $f_{\boldsymbol{w}}$:



$$z \;:=\; \sum_{j=1}^{d} w_j x_j\,, \qquad p \;:=\; g(z) \quad (g = \operatorname{logistic})\,.$$

Learning $\boldsymbol{w}$ from data

Recall $f_{\boldsymbol{w}}(\boldsymbol{x}) = g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle)$ where $g = \operatorname{logistic}$ (called the *link function*).

Using log-loss function

$$\ell_{\log}(y, p) \;:=\; y \log \frac{1}{p} + (1 - y) \log \frac{1}{1 - p}$$

in ERM framework with training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \quad \sum_{i=1}^{n} \ell_{\log}(y_i, g(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle))\,.$$

Stochastic gradient method

Given: training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$.

- Start with some initial $\boldsymbol{w}^{(1)} \in \mathbb{R}^d$.
- For $t = 1, 2, \dots$ until some stopping condition is satisfied.
  - Pick $(\boldsymbol{x}, y) \in \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ uniformly at random.
  - Compute
    $$\boldsymbol{\lambda}^{(t)} := \nabla_{\boldsymbol{w}}\, \ell_{\log}(y, g(\langle \boldsymbol{w}, \boldsymbol{x}\rangle))\Big|_{\boldsymbol{w}=\boldsymbol{w}^{(t)}}.$$
  - Update:
    $$\boldsymbol{w}^{(t+1)} := \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{\lambda}^{(t)}.$$

How to compute $\boldsymbol{\lambda}^{(t)}$?

Using the chain rule

Usual *chain rule* from calculus gives generic way to compute gradient of log-loss (w.r.t. $\boldsymbol{w}$) on example $(\boldsymbol{x}, y)$:

$$\nabla_{\boldsymbol{w}}\{\ell_{\log}(y, f_{\boldsymbol{w}}(\boldsymbol{x}))\} = \tfrac{\partial}{\partial p}\{\ell_{\log}(y,p)\}\Big|_{p=f_{\boldsymbol{w}}(\boldsymbol{x})} \cdot \nabla_{\boldsymbol{w}}\{f_{\boldsymbol{w}}(\boldsymbol{x})\}.$$

- First term:
  $$\frac{\partial}{\partial p}\{\ell_{\log}(y,p)\} = -\frac{y}{p} + \frac{1-y}{1-p},$$
  and plug-in $f_{\boldsymbol{w}}(\boldsymbol{x})$ for p.
- Second term: since $f_{\boldsymbol{w}}(\boldsymbol{x}) = \operatorname{logistic}(\langle \boldsymbol{w}, \boldsymbol{x}\rangle)$, applying chain rule again gives

$$\begin{aligned}\nabla_{\boldsymbol{w}}\{f_{\boldsymbol{w}}(\boldsymbol{x})\} &= \tfrac{\partial}{\partial z}\operatorname{logistic}(z)\Big|_{z=\langle \boldsymbol{w}, \boldsymbol{x}\rangle} \cdot \boldsymbol{x} \\ &= \operatorname{logistic}(\langle \boldsymbol{w}, \boldsymbol{x}\rangle) \cdot \big(1 - \operatorname{logistic}(\langle \boldsymbol{w}, \boldsymbol{x}\rangle)\big) \cdot \boldsymbol{x}.\end{aligned}$$

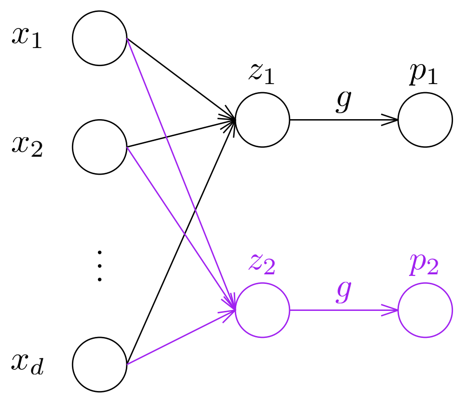
Extensions

- Using log-loss and logistic link function, objective
  $$\min_{\boldsymbol{w}\in\mathbb{R}^d} \quad \sum_{i=1}^n \ell_{\log}(y_i, \operatorname{logistic}(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle))$$
  is same as negative log-likelihood for logistic regression.
- Can also use other loss functions $\ell(y,p)$ and other link functions $g(z)$.
  E.g., $\ell(y,p) = (y-p)^2$, $g(z) = \tanh(z)$.

  Note: objective not necessarily convex, even if $p \to \ell(y,p)$ is convex.

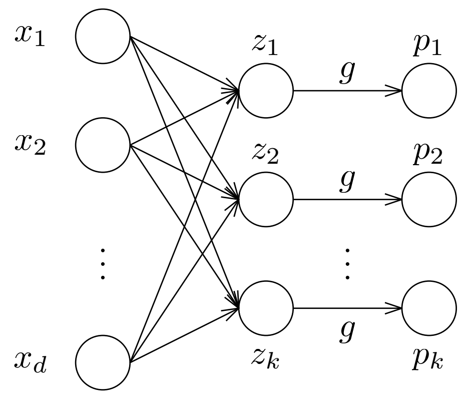
  Nevertheless, stochastic gradient method may still be effective.

Two-output network



$$z_i := \sum_{j=1}^d W_{i,j} x_j, \qquad y_i := g(z_i), \qquad i \in \{1,2\}.$$

k-output network



$$z_i \;:=\; \sum_{j=1}^{d} W_{i,j} x_j \,, \qquad y_i \;:=\; g(z_i) \,, \qquad i \in \{1, 2, \dots, k\} \,.$$

Compositional structure

- Suppose we have two functions

$$f_{\boldsymbol{W}^{(1)}} : \mathbb{R}^d \to \mathbb{R}^k \,, \qquad (\boldsymbol{W}^{(1)} \in \mathbb{R}^{k \times d}) \,,$$
$$f_{\boldsymbol{W}^{(2)}} : \mathbb{R}^k \to \mathbb{R}^\ell \,, \qquad (\boldsymbol{W}^{(2)} \in \mathbb{R}^{\ell \times k}) \,,$$

  where

$$f_{\boldsymbol{W}}(\boldsymbol{x}) \;:=\; G(\boldsymbol{W}\boldsymbol{x}) \,,$$

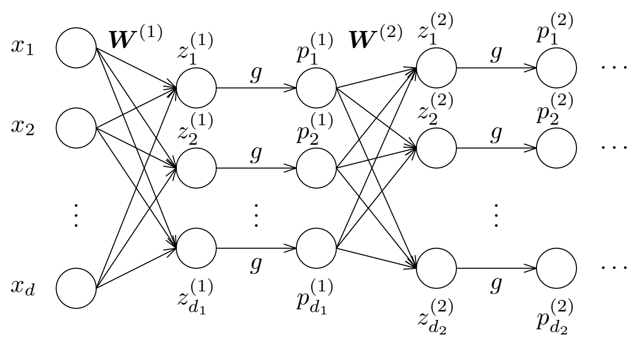
  and G applies the link function g coordinate-wise to a vector.

- **Composition**: $f_{\boldsymbol{W}^{(1)},\boldsymbol{W}^{(2)}} := f_{\boldsymbol{W}^{(2)}} \circ f_{\boldsymbol{W}^{(1)}}$ is defined by

$$f_{\boldsymbol{W}^{(1)},\boldsymbol{W}^{(2)}}(\boldsymbol{x}) \;:=\; f_{\boldsymbol{W}^{(2)}}\big(f_{\boldsymbol{W}^{(1)}}(\boldsymbol{x})\big) \,.$$

  This is a *two-layer neural network*.

Multilayer neural network



- $(z_i^{(l)}, p_i^{(l)})$ regarded as a single "**unit**".
  (Typically represented by a single node.)
- All except *input* ($\boldsymbol{x}$) and *output* units are considered "**hidden**".

Why use multilayer neural networks?

- A one-layer neural network with a monotonic link function is (essentially) a *linear classifier*.

  Cannot represent XOR function (Minsky and Papert, 1969).

- *Any continuous function* f can be approximated arbitrarily well by a two-layer neural network

$$f \;\approx\; \underbrace{f_{\boldsymbol{W}^{(1)}}}_{\mathbb{R}^k \to \mathbb{R}} \circ \underbrace{f_{\boldsymbol{W}^{(0)}}}_{\mathbb{R}^d \to \mathbb{R}^k}$$

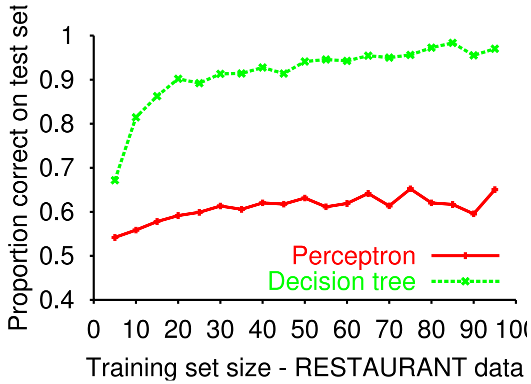
  (Cybenko, 1989; Barron, 1993).

  Caveat: may need a very large number of nodes.

- Some functions can be approximated with *exponentially fewer nodes* by using more than two layers (Eldan and Shamir, 2015; Telgarsky, 2015).

Limits of single-layer network

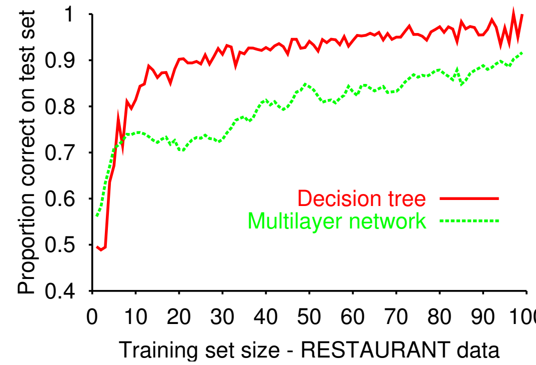
Linear classifier unable to represent complex functions.



(Figure from Stuart Russell `aima.eecs.berkeley.edu/slides-pdf/chapter20b.pdf`)

Benefits of depth

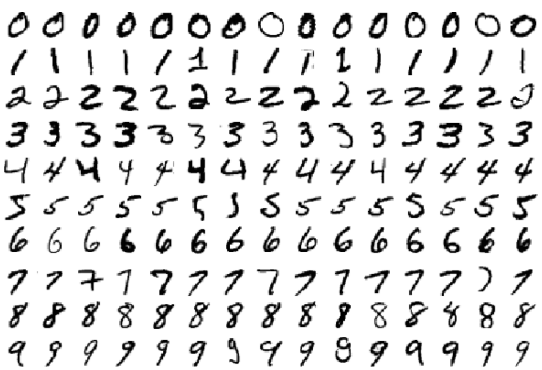
Neural network with four hidden units.



(Figure from Stuart Russell `aima.eecs.berkeley.edu/slides-pdf/chapter20b.pdf`)

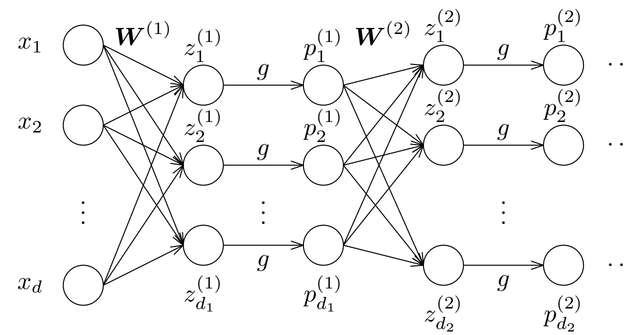
MNIST

OCR digits classification task



- 3-nearest-neighbor: 3.0% error rate
- Neural network with 700 hidden units: 1.6% error rate
- Current best convolutional network: 0.23% error rate

H-layer neural network



- Index layers by $l = 0, 1, 2, \ldots, H$; d_l "units" in layer l.
- $p_j^{(l)}$ is output of unit j in layer l. ($\boldsymbol{p}^{(0)} = \boldsymbol{x} \in \mathbb{R}^{d_0}$ is the *input layer*.)
- $w_{i,j}^{(l)}$ is weight of edge from unit j in layer $l-1$ to unit i in layer l:

$$z_i^{(l)} = \langle \boldsymbol{w}_i^{(l)}, \boldsymbol{p}^{(l-1)} \rangle = \sum_{j=1}^{d_{l-1}} w_{i,j}^{(l)} \cdot p_j^{(l-1)} .$$

- (Transpose of) $\boldsymbol{w}_i^{(l)}$ is i-th row of $\boldsymbol{W}^{(l)}$.

Evaluating a neural network

H-layer neural network f with weights $\{\boldsymbol{w}_i^{(l)} : 0 \le l \le H-1, 1 \le i \le d_l\}$ where $\boldsymbol{w}_i^{(l)} \in \mathbb{R}^{d_{l-1}}$.

Forward propagation to compute $f(\boldsymbol{x})$.

- Start with $\boldsymbol{p}^{(0)} := \boldsymbol{x}$.
- For $l = 1, 2, \ldots, H$:
  - $z_i^{(l)} := \langle \boldsymbol{w}_i^{(l)}, \boldsymbol{p}^{(l-1)} \rangle$ for each $i = 1, 2, \ldots, d_l$.
  - $p_i^{(l)} := g(z_i^{(l)})$ for each $i = 1, 2, \ldots, d_l$.
- Return $\boldsymbol{p}^{(H)}$.

Training a neural network

How to compute gradient of loss $\ell(y, f(\boldsymbol{x}))$ w.r.t weights?

Direct calculation of gradients w.r.t. $\boldsymbol{w}_i^{(l)}$ is complicated.

Mid-to-late 20th century researchers discovered how to use *partial derivatives* to manage gradient computation: **backpropagation algorithm**.

Another view of backpropagation: introduce auxiliary variables $\boldsymbol{p}^{(l)}$ and constraints

$$\boldsymbol{p}^{(l)} \;=\; f_{\boldsymbol{W}^{(l)}}(\boldsymbol{p}^{(l-1)})$$

for $1 \le l \le H$, and apply "method of adjoints".

Recipe for method of adjoints

- Introduce auxiliary variables $\boldsymbol{p}^{(l)}$ for $1 \le l \le H$, and constraints

  $$\boldsymbol{p}^{(l)} \;=\; f_{\boldsymbol{W}^{(l)}}(\boldsymbol{p}^{(l-1)}), \qquad \text{for } 1 \le l \le H.$$

  (Also define $\boldsymbol{p}^{(0)} := \boldsymbol{x}$; not variable.)
- Introduce Lagrange variables $\boldsymbol{\alpha}^{(l)} \in \mathbb{R}^{d_l}$ for $1 \le l \le H$, and Lagrangian

  $$\mathcal{L} \;:=\; \ell(y, f(\boldsymbol{x})) - \sum_{l=1}^{H} \Big(\boldsymbol{p}^{(l)} - f_{\boldsymbol{W}^{(l)}}(\boldsymbol{p}^{(l-1)})\Big)^{\top} \boldsymbol{\alpha}^{(l)}.$$

- Compute values of auxiliary variables and Lagrange variables that satisfy

  $$\nabla_{\boldsymbol{\alpha}^{(l)}} \mathcal{L} \;=\; \boldsymbol{0}, \qquad \nabla_{\boldsymbol{p}^{(l)}} \mathcal{L} \;=\; \boldsymbol{0}, \qquad \text{for } 1 \le l \le H.$$

- Compute gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{w}_i^{(l)}$ in terms of auxiliary and Lagrange variables; much simpler!

  (This method is justified by some very beautiful optimization theory.)

Setting the auxiliary and Lagrange variables

Lagrangian: $\mathcal{L} \;:=\; \ell(y, f(\boldsymbol{x})) - \sum_{l=1}^{H} \Big(\boldsymbol{p}^{(l)} - f_{\boldsymbol{W}^{(l)}}(\boldsymbol{p}^{(l-1)})\Big)^{\top} \boldsymbol{\alpha}^{(l)}$.

- For $1 \le l \le H$: $\quad \nabla_{\boldsymbol{\alpha}^{(l)}} \mathcal{L} \;=\; \boldsymbol{p}^{(l)} - f_{\boldsymbol{W}^{(l)}}(\boldsymbol{p}^{(l-1)})$.

  This is zero if we set $\boldsymbol{p}^{(l)} := f_{\boldsymbol{W}^{(l)}}(\boldsymbol{p}^{(l-1)})$ (i.e., forward propagation).
- For $l = H$: $\quad \nabla_{\boldsymbol{p}^{(H)}} \mathcal{L} \;=\; \nabla_{\boldsymbol{p}^{(H)}} \Big\{\ell(y, \boldsymbol{p}^{(H)})\Big\} - \boldsymbol{\alpha}^{(H)}$.

  This is zero if we set $\boldsymbol{\alpha}^{(H)} := \nabla_{\boldsymbol{p}^{(H)}} \Big\{\ell(y, \boldsymbol{p}^{(H)})\Big\}$.
- For $1 \le l < H$: $\quad \nabla_{\boldsymbol{p}^{(l)}} \mathcal{L} \;=\; \nabla_{\boldsymbol{p}^{(l)}} \Big\{f_{\boldsymbol{W}^{(l+1)}}(\boldsymbol{p}^{(l)})^{\top} \boldsymbol{\alpha}^{(l+1)}\Big\} - \boldsymbol{\alpha}^{(l)}$.

  This is zero if we set

  $$\boldsymbol{\alpha}^{(l)} \;:=\; \nabla_{\boldsymbol{p}^{(l)}} \Big\{f_{\boldsymbol{W}^{(l+1)}}(\boldsymbol{p}^{(l)})^{\top} \boldsymbol{\alpha}^{(l+1)}\Big\} \;=\; \sum_{i=1}^{d_{l+1}} \alpha_i^{(l+1)} \cdot g'(\langle \boldsymbol{w}_i^{(l+1)}, \boldsymbol{p}^{(l)} \rangle) \cdot \boldsymbol{w}_i^{(l+1)}.$$

- For $1 \le l \le H$: $\quad \nabla_{\boldsymbol{w}_i^{(l)}} \mathcal{L} \;=\; \alpha_i^{(l)} \cdot g'(\langle \boldsymbol{w}_i^{(l)}, \boldsymbol{p}^{(l-1)} \rangle) \cdot \boldsymbol{p}^{(l-1)}$ (simple!).

Backward propagation

Backward propagation ("backprop") to compute gradient of $\ell(y, f(\boldsymbol{x}))$ with respect to weights.

Input: labeled example $(\boldsymbol{x}, y)$, current weights $\boldsymbol{W}^{(l)}$ for $1 \le l \le H$.

- Run forward propagation, saving intermediate computations: $\boldsymbol{p}^{(l)} = f_{\boldsymbol{W}^{(l)}}(\boldsymbol{p}^{(l-1)})$ and $z_i^{(l)} = \langle \boldsymbol{w}_i^{(l)}, \boldsymbol{p}^{(l-1)} \rangle$ for $1 \le l \le H$.
- Set $\boldsymbol{\alpha}^{(H)} := \nabla_{\boldsymbol{p}^{(H)}} \ell(y, \boldsymbol{p}^{(H)})$.
- For $l = H, H-1, \ldots, 1$:
  - Set $\nabla_{\boldsymbol{w}_i^{(l)}} \ell(y, f(\boldsymbol{x})) := \alpha_i^{(l)} \cdot g'(z_i^{(l)}) \cdot \boldsymbol{p}^{(l-1)}$ for $1 \le i \le d_l$.
  - Set $\boldsymbol{\alpha}^{(l-1)} := \sum_{i=1}^{d_l} \alpha_i^{(l)} \cdot g'(z_i^{(l)}) \cdot \boldsymbol{w}_i^{(l)}$ (unless $l = 1$).
- Return gradients $\nabla_{\boldsymbol{w}_i^{(l)}} \ell(y, f(\boldsymbol{x}))$ for $1 \le l \le H$, $1 \le i \le d_l$.

Practical tips

- Apply stochastic gradient method to examples in random order.
- Standardize inputs (i.e., center and divide by standard deviation).
- (More expensive): completely de-correlate inputs (i.e., PCA).
- Can also use other loss functions, e.g., $\ell(y, p) = (y - p)^2$ (square loss).
- Link function: prefer "centered" range, e.g., $\tanh$ (as opposed to $\mathrm{logistic}$).
- Initialization: random? But take care so that weights are not too large or too small.

  E.g., for node with d inputs, draw weights iid from $\mathrm{N}(0, 1/d)$.

Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What's new?
  - Designing complex "architectures" that work well for specific problems.
  - Fast algorithms / hardware for optimizing neural network weights on large data sets (e.g., using stream-based processing, distributed systems).
  - Applications: visual detection and recognition, speech recognition, general function fitting (e.g., learning "reward" functions of different actions of video games), etc.
  - ...

Key takeaways

1. Structure of simple multilayer neural networks; concept of link functions.
2. Optimization problems with neural networks; use of chain rule to compute gradients.
3. Arguments for depth.
4. High-level idea of backprop.