Getting started on machine learning

with one little artificial neural net

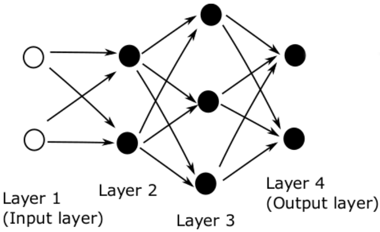
Ed Bueler

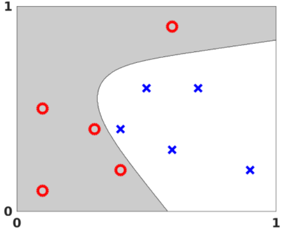
MATH 692 Mathematics for Machine Learning
UAF

13 January
20 January

- sign-up sheet!
- in-person or hybrid?
  - is this classroom adequate?
- what will be the topics?
  - are there out-of-bounds topics?
  - who is volunteering to talk, and when?
- my existing webpages . . . improvements?
  - bueler.github.io/M692S22
  - github.com/bueler/ml-seminar

- **my topic:** how this ↓ neural network does this ↓ classification task





- an **example** from this ↓ paper:

HH19 $=$ C. F. Higham & D. J. Higham (2019). *Deep learning: An introduction for applied mathematicians.* SIAM Review, 61(4), 860-891

goal for today

- know the meanings of some machine learning (ML) language:

  artificial neuron   *activation function*
  weight matrix   *bias vector*
  training   *stochastic gradient descent*
  back-propagation

- which standard mathematical concept(s) match these buzzwords?

- I am no expert on what I am talking about here
  - many in the room know more than me
- I volunteered to give one intro talk, that's all!

Outline

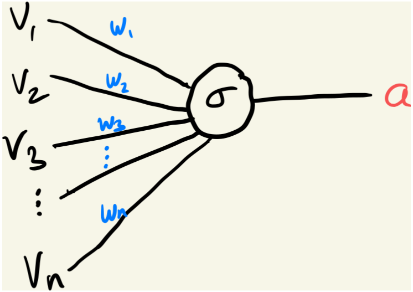
artificial neuron = nonlinear-ized inner product

- given (column) vectors $v, w \in \mathbb{R}^n$
- recall *inner product*:

$$\langle w, v \rangle = w^\top v = \sum_{j=1}^{n} w_j v_j$$

- apply a nonlinear function $\sigma : \mathbb{R}^1 \to \mathbb{R}^1$:

$$a = \sigma\left(\sum_{j=1}^{n} w_j v_j\right) \in \mathbb{R}^1$$



  - detail: add a bias $b \in \mathbb{R}^1$
  - that's it! an **artificial neuron**

artificial neuron = nonlinear-ized inner product

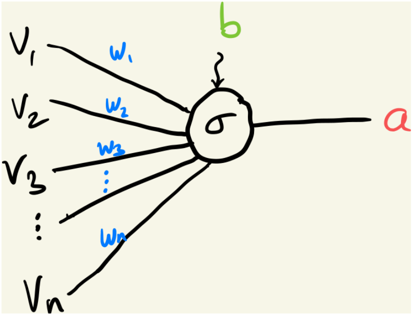
- given (column) vectors $v, w \in \mathbb{R}^n$
- recall *inner product*:

$$\langle w, v \rangle = w^\top v = \sum_{j=1}^{n} w_j v_j$$

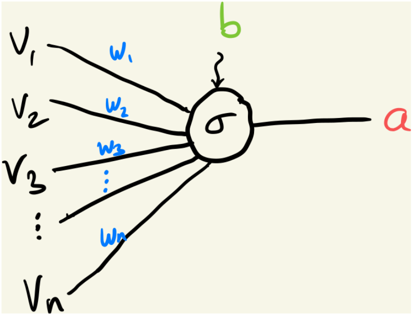
- apply a nonlinear function $\sigma : \mathbb{R}^1 \to \mathbb{R}^1$:

$$a = \sigma\left(\sum_{j=1}^{n} w_j v_j + b\right) \in \mathbb{R}^1$$

  - detail: add a bias $b \in \mathbb{R}^1$
  - that's it! an **artificial neuron**

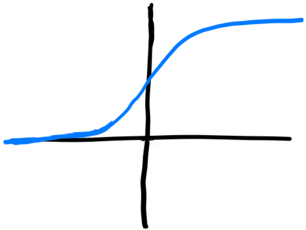


- v is input
- **weights** w and **biases** b are parameters
  - they need **training**
- the **activation function** σ is fixed
- the output a is the **activation** of the neuron

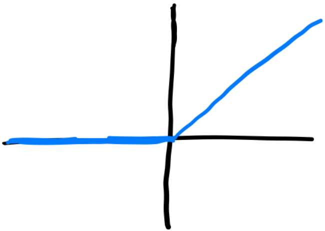


$$a = \sigma\left(\sum_{j=1}^{n} w_j v_j + b\right)$$

nonlinear activation function



sigmoid



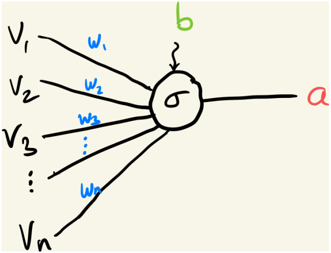
ReLU

- σ is the **activation function**
  - an increasing scalar function with bounded derivative
- some possibilities:
  - **sigmoid**, e.g. $$\sigma(z) = \frac{1}{1 + e^{-z}}$$
  - **rectified linear unit (ReLU)**, $$\sigma(z) = \begin{cases} z, & z > 0 \\ 0, & z \le 0 \end{cases}$$

- a **trained neuron** has known parameters w, b
- then $a : \mathbb{R}^n \to \mathbb{R}^1$ is a known function:

$$a = a(v)$$

  - similar cost to inner product
  - backward stable
  - one might write $a(v; w, b)$ to make dependence on parameters clear



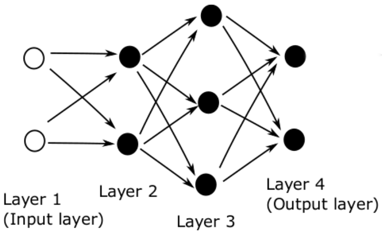
- Rosenblatt (1958): from biological motivation, proposes a **perceptron**, a single artificial neuron with binary output:

$$\sigma(z) = \begin{cases} 1, & z > 0 \\ 0, & z \le 0 \end{cases}$$

  - with learning algorithm
- Minsky & Papert (1969): a single layer of perceptrons cannot even learn the XOR function!
  - single layer perceptrons are linear separators
  - **support vector machines** are perceptrons of optimal stability
- feedforward artificial neural networks (ANN), the next topic, are sometimes called **multilayer perceptrons**
  - ...which ignores activation function details

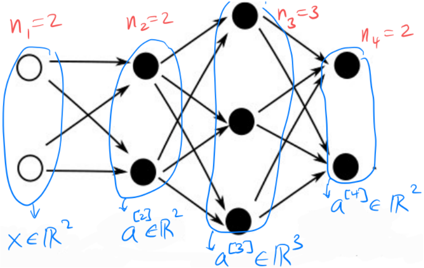
Outline

feed-forward networks

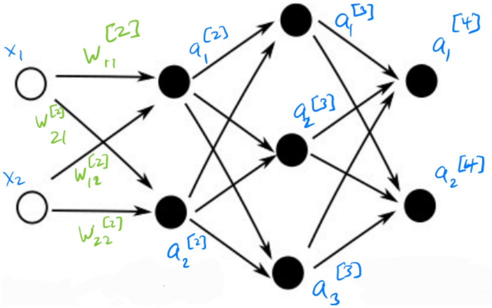


- considering only **feed-forward** networks in this talk
  - edges connect consecutive layers, in order
  - in ML language: feed-forward versus **recurrent**
  - in graph language: "feed-forward network" $=$ connected, directed acyclic graph which is equal to its own transitive reduction . . . ?

network notation



- notation from HH19
- n_ℓ is number of neurons in layer $\ell = 1, \dots, L$
  - $\ell = 1$ is input layer
  - input values are first-layer activations: $x = a^{[1]} \in \mathbb{R}^{n_1}$
  - $\ell = L$ is output layer
  - output values are final-layer activations: $y = a^{[L]} \in \mathbb{R}^{n_L}$
- activations in layer ℓ form a vector $a^{[\ell]} \in \mathbb{R}^{n_\ell}$
  - $a_j^{[\ell]}$ is activation of neuron j in layer ℓ

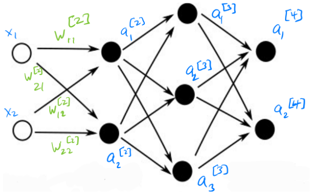


- weight $w_{jk}^{[\ell]}$ on the edge from neuron $a_k^{[\ell-1]}$ to neuron $a_j^{[\ell]}$
- thus

$$a_j^{[\ell]} = \sigma\left(\sum_{k=1}^{n_{\ell-1}} w_{jk}^{[\ell]} a_k^{[\ell-1]} + b_j\right)$$

- which suggests matrix-vector multiplication!

weight notation using vectors and matrices



- one (row) vector of weights for each neuron
- the inputs to the neurons in a layer are weighted by a matrix:

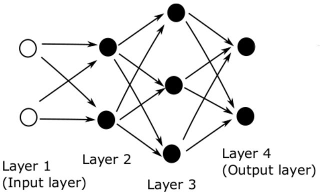
$$W^{[\ell]} = (\text{weights } \textit{into} \text{ layer } \ell) = \left[w_{jk}^{[\ell]} \right]$$

  - $W^{[\ell]}$ is $n_\ell \times n_{\ell-1}$
- assuming σ applied entrywise:

$$a^{[\ell]} = \sigma \left(W^{[\ell]} a^{[\ell-1]} + b^{[\ell]} \right)$$

  - bias vector $b^{[\ell]} \in \mathbb{R}^{n_\ell}$

forward pass = nonlinear-ized matrix multiplication



- for this small $L = 4$ network:

$$y = a^{[4]} = \sigma\left(W^{[4]} a^{[3]} + b^{[4]}\right) = \dots$$
$$= \sigma\left(W^{[4]} \sigma\left(W^{[3]} \sigma\left(W^{[2]} x + b^{[2]}\right) + b^{[3]}\right) + b^{[4]}\right)$$

- *over-simplified:* $\sigma(z) = z$ and $b^{[\ell]} = 0$ implies $y = W^{[4]} W^{[3]} W^{[2]} x$
- feed-forward network = nonlinear- & affine-ized matrix product

- compute from input $x = a^{[1]}$ to output $y = a^{[L]}$ by layers:

$$a^{[\ell]} = \sigma\left(W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}\right) \qquad \text{for } \ell = 2, 3, \dots, L$$

  - equation (3.2) in HH19
- thus a forward pass is an obvious loop:

```
FORWARD(x):
    a^[1] = x
    for ℓ = 2, 3, ..., L:
        z^[ℓ] = W^[ℓ] a^[ℓ-1] + b^[ℓ]
        a^[ℓ] = σ(z^[ℓ])
    return y = a^[L]
```

  - $\{W^{[\ell]}\}$ and $\{b^{[\ell]}\}$ are stored in some data structure
  - $\sigma(z)$ is implemented entrywise

forward-pass computation work model (*minor point*)

- **work** $=$ number of floating-point operations
- work at layer ℓ:

$$2n^{[\ell-1]}n^{[\ell]} + O(n^{[\ell]}) = O(n^{[\ell-1]}n^{[\ell]})$$

  - using big-O in the "$n \to \infty$" limit of big layers
  - evaluating activation functions is cheap
  - the work at one layer is basically just a matrix-vector product
- the total forward-pass work in an L-layer network is asymptotically the same as L matrix-vector products:

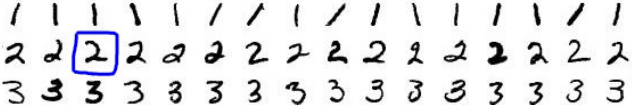
$$\sum_{\ell=2}^{L} O(n^{[\ell-1]}n^{[\ell]}) = O(Ln_{\text{max}}^2)$$

- *note:* a forward pass is easily computed by GPU hardware
  - versus solving linear systems ...

Outline

training is a biologically-motivated procedure

- imagine teaching your dog to read numbers 1,2,3:



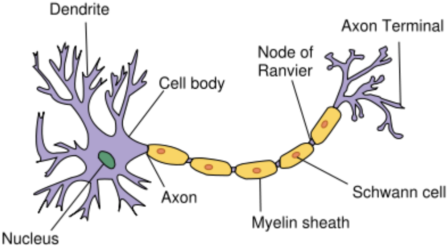
- example training procedure:
  1. randomly present one image of a digit from above

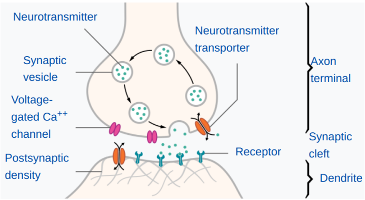
  2. if dog barks correct number of times then gets treat

bark! bark! $\Longrightarrow$ 

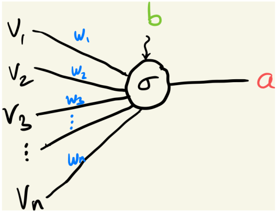
  3. otherwise move on to next image

- biological learning can last from hours to decades
- something permanent/persistent changes within the brain
  - but neuron excitation is electrical (activation is temporary),
  - and neuron count is relatively fixed
- training yields chemical or morpological changes in connections between neurons, especially the **synapses** where the axon connects to another neuron's dendrite





- but biological realism is *not* needed for machine learning!
- we use a simplified artificial neuron:



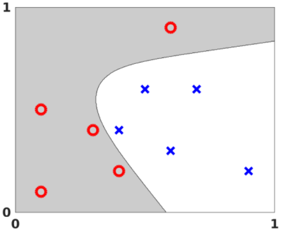
  - this model is merely a formula: $$a = \sigma\left(\sum_{k=1}^{n} w_k v_k + b\right)$$
- however, training needs to make "permanent" changes in the weights w_k and biases b
- after training, forward passes are the network's "learned behavior"

- so, how does training work in artificial neural networks?
- only considering **supervised training** here
- given: N pieces of **labeled data** (pairs)

$$(x^{\{i\}}, y^{\{i\}}) \qquad \text{for } i = 1, \dots, N$$

  - $x^{\{i\}} \in \mathbb{R}^{n_1}$ are the **data**
  - $y^{\{i\}} \in \mathbb{R}^{n_L}$ are the **labels**
  - in a **classification task**, the $y^{\{i\}}$ only take on finitely-many values
- observation: the labeled data determine the number of neurons in the first and last layers

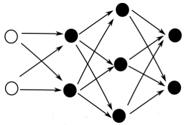
- example **classification task** data $(x^{\{i\}}, y^{\{i\}})$ in one figure



  - $N = 10$
  - marker coordinates give $x^{\{i\}} \in [0,1]^2$
  - marker type gives $y^{\{i\}}$:

$$\circ : y^{\{i\}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad \times : y^{\{i\}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

  - a neural net for this data must have $n_1 = n_L = 2$:
  - ignore shading for now . . .



- **supervised training** means choosing the weights $W^{[\ell]}$ and biases $b^{[\ell]}$, in *all* the layers, so as to approximately minimize the **average misfit** between the the network output from input data vector $x^{\{i\}}$ and the corresponding label vector $y^{\{i\}}$
- using the squared 2-norm for the misfit, this is a formula:

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|y^{\{i\}} - a^{[L]}(x^{\{i\}})\|_2^2$$

  - $a^{[L]}(x^{\{i\}})$ = output-layer activation from forward pass with input $x^{\{i\}}$
  - the Cost is a scalar-valued function (**functional**) of the weights and biases

better training notation

- let's give all the parameters a single-letter name:

$$p = \{W^{[2]}, W^{[3]}, \dots, W^{[L]}, b^{[2]}, b^{[3]}, \dots, b^{[L]}\} \in \mathbb{R}^s$$

  - p collects all weight matrices and biases into one big column vector
- define the cost (misfit) of the network for one data pair:

$$C^{\{i\}}(p) = \frac{1}{2} \left\| y^{\{i\}} - a^{[L]}(x^{\{i\}}; p) \right\|_2^2$$

- key idea:

  The output from a forward pass through the network, namely $a^{[L]}(x^{\{i\}}; p)$, depends both on the input data $x^{\{i\}}$ and all the weights and biases p. We emphasize that the cost of one data pair is a function of p: $C^{\{i\}}(p)$.

- now define a total **cost functional**, or **objective**, $C(p)$
- $C(p)$ is the **average** misfit over all the labeled data:

$$\begin{aligned} C(p) &= \frac{1}{N} \sum_{i=1}^{N} C^{\{i\}}(p) \\ &= \frac{1}{2N} \sum_{i=1}^{N} \left\| y^{\{i\}} - a^{[L]}(x^{\{i\}}; p) \right\|_2^2 \end{aligned}$$

training = nonlinear least-squares optimization

- compare

$$C(p) = \frac{1}{2N} \sum_{i=1}^{N} \left\| y^{\{i\}} - a^{[L]}(x^{\{i\}}; p) \right\|_2^2$$

to "nonlinear least-squares" in a standard optimization textbook (Nocedal & Wright, 2006; Chapter 10):

In least-squares problems, the objective function f has the following special form:

$$f(x) = \frac{1}{2} \sum_{j=1}^{m} r_j^2(x),$$

where each r_j is a smooth function from $\mathbb{R}^n$ to $\mathbb{R}$. We refer to each r_j as a *residual*,

- training a neural net is **nonlinear least-squares optimization**
  - $\left\| y^{\{i\}} - a^{[L]}(x^{\{i\}}; p) \right\|_2$ is the **residual norm** for *i*th data
  - it becomes zero when the network fully-learns the *i*th data

fundamental idea: cost is a function of weights and biases

- recall $p = \{W^{[2]}, W^{[3]}, \dots, W^{[L]}, b^{[2]}, b^{[3]}, \dots, b^{[L]}\}$
- the **cost** for one data pair **is a function of** p:

$$C^{\{i\}}(p) = \frac{1}{2} \left\| y^{\{i\}} - a^{[L]}(x^{\{i\}}; p) \right\|_2^2$$

- that is, given parameters p, one forward pass through the network, using input $x^{\{i\}}$, is needed to evaluate $C^{\{i\}}(p)$
- from now on we simplify notation: $a^{[L]} = a^{[L]}(x^{\{i\}}; p)$

Q. why is $C(p) = \frac{1}{2N} \sum_{i=1}^{N} \left\| y^{\{i\}} - a^{[L]} \right\|_2^2$ a function of p?

a. because the activations of the final layer, namely $a^{[L]}$, are determined by the weights and biases in the network

fundamental idea: cost is a function of weights and biases

- recall $p = \{W^{[2]}, W^{[3]}, \dots, W^{[L]}, b^{[2]}, b^{[3]}, \dots, b^{[L]}\}$
- the **cost** for one data pair **is a function of** p:

$$C^{\{i\}}(p) = \frac{1}{2} \left\| y^{\{i\}} - a^{[L]}(x^{\{i\}}; p) \right\|_2^2$$

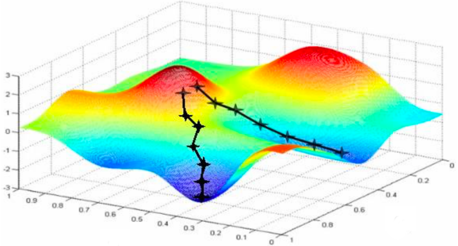
- that is, given parameters p, one forward pass through the network, using input $x^{\{i\}}$, is needed to evaluate $C^{\{i\}}(p)$
- from now on we simplify notation: $a^{[L]} = a^{[L]}(x^{\{i\}}; p)$

Q. why is $C(p) = \frac{1}{2N} \sum_{i=1}^{N} \left\| y^{\{i\}} - a^{[L]} \right\|_2^2$ a function of p?

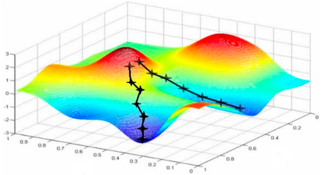
a. because the activations of the final layer, namely $a^{[L]}$, are determined by the weights and biases in the network

- our goal is to minimize $C(p) = \frac{1}{2N} \sum_{i=1}^{N} \left\| y^{\{i\}} - a^{[L]} \right\|_2^2$
- the function $C(p)$ is differentiable
  - *why? what does this assume about the network?*
- ...thus we can compute the **gradient** $\nabla C(p)$
- the gradient points *up hill* on the surface $C : \mathbb{R}^s \to \mathbb{R}$,
- natural idea: do **gradient descent**

GD(p)**:**
  for $s = 1, 2, \dots$**:**
    $p \leftarrow p - \eta \nabla C(p)$
  return p



GD(p):
  for $s = 1, 2, \dots$ **:**
    $p \leftarrow p - \eta \nabla C(p)$
  return p



- GD is simple to program
  - …but it will always let you down
- known issues with naive GD:
  - it is not clear how far to step (how to set η?)
    - $C(p)$, $\nabla C(p)$ provide no information
    - provable convergence requires a *line search* or *trust region* approach, otherwise $G(p)$ may not even decrease
    - η is called the **learning rate** in machine learning
  - if GD converges, it may be to a *local* minimum only

- GD is widely used for training in **machine learning** (ML)
  - a seminar priority?: GD limitations, modifications, alternatives
- ML applies 2 “insights” (*habits*?) about how GD should work:

1. **stochastic gradient descent**: since N is big, and because overfitting should be avoided, do *not* compute the whole gradient $\nabla C(p)$, but instead a randomly chosen $\nabla C^{\{i\}}(p)$
   - i.e. choose data $(x^{\{i\}}, y^{\{i\}})$ and do

$$p \leftarrow p - \eta \nabla C^{\{i\}}(p)$$

   - or a choose a **batch**: $p \leftarrow p - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C^{\{k_i\}}(p)$

2. **back-propagation**: when computing $\nabla C^{\{i\}}(p)$, regard the chain rule as information which can be fed backward through the network
   - back-propagation uses info found in computing forward for $C^{\{i\}}(p)$

SGD(p):
  for $s = 1, 2, \dots$**:**
    $i =$ **(**random uniform from $\{1, \dots, N\}$**)**
    $p \leftarrow p - \eta \nabla C^{\{i\}}(p)$
  return p

- above is **vanilla** SGD
  - note i is chosen *with* replacement
- variations:
  - choose i without replacement
  - batching: $p \leftarrow p - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C^{\{k_i\}}(p)$
  - **online**: N unknown; data pairs $(x^{\{i\}}, y^{\{i\}})$ are provided by a stream

observations about the cost gradient

$$C(p) = \frac{1}{N} \sum_{i=1}^{N} C^{\{i\}}(p), \qquad C^{\{i\}}(p) = \frac{1}{2} \left\| y^{\{i\}} - a^{[L]} \right\|_2^2$$

- N = amount of training data $\therefore$ N is (should be) large
- gradient for one data pair:

$$\begin{aligned} \nabla C^{\{i\}}(p) &= \nabla \left[\frac{1}{2} (y^{\{i\}} - a^{[L]})^\top (y^{\{i\}} - a^{[L]}) \right] \\ &= - \sum_{j=1}^{n_L} (y_j^{\{i\}} - a_j^{[L]}) \, \nabla a_j^{[L]} \end{aligned}$$

- chain rule will be needed to expand further
  - network output $a_j^{[L]}$ is a *composition* of matrix-vector products and (nonlinear) σ applications
- how to compute $\nabla a_j^{[L]}$ efficiently? ... time for the chain rule!

interlude: the buzzword list

- **artificial neuron**
  - **activation**
  - **activation function**
    - sigmoid, ReLU
  - **weight** matrix
  - **bias** vector
- **artificial neural network** = ANN
  - feed-forward network
- **training**
  - **supervised learning**
  - labeled data
  - nonlinear least-squares optimization
- **stochastic gradient descent** = SGD
  - learning rate
- **back-propagation** = BP

- **machine learning** = ML
  - **deep learning** if $L > 2$

Outline

cost gradient with respect to weights and biases

- recall:
  - $p = \{W^{[2]}, W^{[3]}, \dots, W^{[L]}, b^{[2]}, b^{[3]}, \dots, b^{[L]}\} \in \mathbb{R}^s$ is a grab-bag of parameters
  - cost for one pair $(x^{\{i\}}, y^{\{i\}})$: $\qquad C^{\{i\}}(p) = \frac{1}{2}\left\|y^{\{i\}} - a^{[L]}\right\|_2^2$
- want: $\qquad \nabla C^{\{i\}}(p) = \left[\frac{\partial C^{\{i\}}}{\partial p_1}, \dots, \frac{\partial C^{\{i\}}}{\partial p_s}\right]$
- the components of this gradient come in two types:

$$\frac{\partial C^{\{i\}}}{\partial w_{jk}^{[\ell]}}, \qquad \frac{\partial C^{\{i\}}}{\partial b_j^{[\ell]}}$$

chain rule on the cost (for the last layer)

- recall: $z_j^{[L]} = \sum_{k=1}^{n_{L-1}} w_{jk}^{[L]} a_k^{[L-1]} + b_j^{[L]}$
- expand the 2-norm and the activation in the one-pair cost formula:

$$C^{\{i\}}(p) = \frac{1}{2} \sum_{j=1}^{n_L} (y_j^{\{i\}} - \underbrace{\sigma(z_j^{[L]})}_{=a_j^{[L]}})^2$$

- thus by the chain rule:

$$\frac{\partial C^{\{i\}}}{\partial w_{jk}^{[L]}} = \boxed{\frac{\partial C^{\{i\}}}{\partial z_j^{[L]}}} \frac{\partial z_j^{[L]}}{\partial w_{jk}^{[L]}} = \boxed{\frac{\partial C^{\{i\}}}{\partial z_j^{[L]}}} a_k^{[L-1]}$$

$$\frac{\partial C^{\{i\}}}{\partial b_j^{[L]}} = \boxed{\frac{\partial C^{\{i\}}}{\partial z_j^{[L]}}} \frac{\partial z_j^{[L]}}{\partial b_j^{[L]}} = \boxed{\frac{\partial C^{\{i\}}}{\partial z_j^{[L]}}}$$

  - the boxed quantity shows up a lot, so it gets a name . . .

chain rule on the cost

- define, following HH19:

$$\delta_j^{[\ell]} = \frac{\partial C^{\{i\}}}{\partial z_j^{[\ell]}}$$

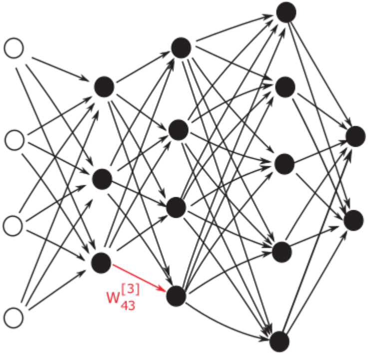
  - this definition is for *any layer* ℓ
  - remember that this is for one data pair $(x^{\{i\}}, y^{\{i\}})$
- for the final layer $\ell = L$ we already have:

$$\begin{aligned}
\delta_j^{[L]} &= -(y_j^{\{i\}} - a_j^{[L]})\,\sigma'(z_j^{[L]}) \\
\frac{\partial C^{\{i\}}}{\partial w_{jk}^{[L]}} &= \delta_j^{[L]} a_k^{[L-1]} \\
\frac{\partial C^{\{i\}}}{\partial b_j^{[L]}} &= \delta_j^{[L]}
\end{aligned}$$

- to go further back into the network, follow multiple routes:

  $a_j^{[L]}$ depends on $a_k^{[L-1]}$ for different k, then $a_k^{[L-1]}$ depends on $a_s^{[L-2]}$ for different s, ...

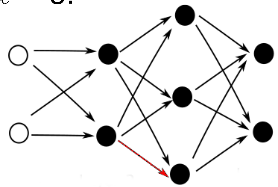
- example: what is derivative of cost $C^{\{i\}}$ with respect to $w_{43}^{[3]}$?



find inside the chain rule: multiplication by the transpose

- *example.* $L = 4$; consider a weight into layer $\ell = 3$:

$$\frac{\partial C^{\{i\}}}{\partial w_{jk}^{[3]}} = \underbrace{\frac{\partial C^{\{i\}}}{\partial z_j^{[3]}}}_{multi-route} \underbrace{\frac{\partial z_j^{[3]}}{\partial w_{jk}^{[3]}}}_{easy} = \delta_j^{[3]} a_k^{[2]}$$



- but $\delta_j^{[3]}$ relates to the *next*-layer deltas $\delta^{[4]}$ by matrix multiplication:

$$\begin{aligned}\delta_j^{[3]} = \frac{\partial C^{\{i\}}}{\partial z_j^{[3]}} &= \sum_{s=1}^{n_4} \frac{\partial C^{\{i\}}}{\partial z_s^{[4]}} \frac{\partial z_s^{[4]}}{\partial z_j^{[3]}} \\ &= \sum_{s=1}^{n_4} \delta_s^{[4]} \frac{\partial z_s^{[4]}}{\partial a_j^{[3]}} \frac{\partial a_j^{[3]}}{\partial z_j^{[3]}} \\ &= \sum_{s=1}^{n_4} \delta_s^{[4]} w_{sj}^{[4]} \sigma'(z_j^{[3]}) = \left((W^{[4]})^\top \delta^{[4]} \right)_j \sigma'(z_j^{[3]})\end{aligned}$$

- define the entrywise (*Hadamard*?) product of vectors $x, y \in \mathbb{R}^m$:

$$(x \circ y)_j = x_j y_j$$

Lemma

the vector $\delta^{[\ell]} = \left[\frac{\partial C^{\{i\}}}{\partial z_j^{[\ell]}}\right] \in \mathbb{R}^{n_\ell}$ *can be computed by (back-) multiplying the weight matrix transposes (adjoints):*

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^{[L]} - y^{\{i\}}),$$
$$\delta^{[\ell]} = \sigma'(z^{[\ell]}) \circ \underbrace{(W^{[\ell+1]})^\top \delta^{[\ell+1]}}_{\text{matrix-vector product}}, \qquad \ell = L-1, L-2, \dots, 2$$

- *key point:* start with last layer $\ell = L$ and count *down* to $\ell = 2$

back-propagation provides the cost gradient

Corollary

once the $\delta^{[\ell]}$ are calculated, all components of the gradient are easy:

$$\frac{\partial C^{\{i\}}}{\partial w_{jk}^{[\ell]}} = \delta_j^{[\ell]} a_k^{[\ell-1]}, \qquad \frac{\partial C^{\{i\}}}{\partial b_j^{[\ell]}} = \delta_j^{[\ell]}$$

for $\ell = 2, \dots, L$

- *observation:* as a matrix,

$$\frac{\partial C^{\{i\}}}{\partial W^{[\ell]}} = \delta^{[\ell]} \left(a^{[\ell-1]}\right)^\top$$

is a rank-one outer product

from x, y and $p = \{W^{[\ell]}, b^{[\ell]}\}$ compute $C(p)$ and $\nabla C(p)$:

FORBACK$(x, y, W^{[\ell]}, b^{[\ell]})$**:**

$a^{[1]} = x$

for $\ell = 2, \dots, L$**:**

$z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}$

$a^{[\ell]} = \sigma(z^{[\ell]})$, $\quad r^{[\ell]} = \sigma'(z^{[\ell]})$

$C = \frac{1}{2}\|y - a^{[L]}\|_2^2$

$\delta^{[L]} = r^{[L]} \circ (a^{[L]} - y)$

for $\ell = L, \dots, 2$**:**

if $\ell < L$**:**

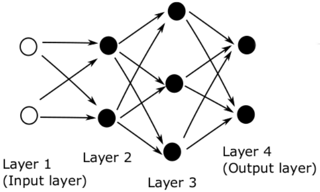
$\delta^{[\ell]} = r^{[\ell]} \circ (W^{[\ell+1]})^\top \delta^{[\ell+1]}$

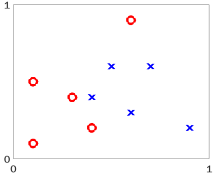
$\frac{\partial C}{\partial W^{[\ell]}} = \delta^{[\ell]} (a^{[\ell-1]})^\top$

$\frac{\partial C}{\partial b^{[\ell]}} = \delta^{[\ell]}$

return $C, \left\{\frac{\partial C}{\partial W^{[\ell]}}\right\}, \left\{\frac{\partial C}{\partial b^{[\ell]}}\right\}$

- we are ready to train a network, e.g. on a classification task:





- one could call FORBACK() in a SGD training loop:

$$
\begin{aligned}
&\textbf{for } s = 1, 2, \dots \\
&\qquad i = \textbf{(}\text{random uniform from } \{1, \dots, N\}\textbf{)} \\
&\qquad C^{\{i\}}, \frac{\partial C^{\{i\}}}{\partial W^{[\ell]}}, \frac{\partial C^{\{i\}}}{\partial b^{[\ell]}} = \text{FORBACK}(x^{\{i\}}, y^{\{i\}}, \dots) \\
&\qquad W^{[\ell]} \leftarrow W^{[\ell]} - \eta \, \frac{\partial C^{\{i\}}}{\partial W^{[\ell]}} \\
&\qquad b^{[\ell]} \leftarrow b^{[\ell]} - \eta \, \frac{\partial C^{\{i\}}}{\partial b^{[\ell]}}
\end{aligned}
$$

- also natural to combine in one loop: (forward pass) $+$ BP $+$ SGD

pseudocode: training using SGD and BP

integrate SGD into the FORBACK() loop; see `netbp.m` in HH19:

TRAINING$(\{x^{\{i\}}\},\{y^{\{i\}}\},\{W^{[\ell]}\},\{b^{[\ell]}\})$:

for $s = 1, 2, \dots$

- $i =$ (random uniform from $\{1, \dots, N\}$)
- $a^{[1]} = x^{\{i\}}$
- **for** $\ell = 2, \dots, L$:
  - $z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}$
  - $a^{[\ell]} = \sigma(z^{[\ell]}), \quad r^{[\ell]} = \sigma'(z^{[\ell]})$
- $\delta^{[L]} = r^{[L]} \circ (a^{[L]} - y^{\{i\}})$
- **for** $\ell = L, \dots, 2$:
  - **if** $\ell < L$:
    - $\delta^{[\ell]} = r^{[\ell]} \circ (W^{[\ell+1]})^\top \delta^{[\ell+1]}$
  - $W^{[\ell]} \leftarrow W^{[\ell]} - \eta\, \delta^{[\ell]} (a^{[\ell-1]})^\top$
  - $b^{[\ell]} \leftarrow b^{[\ell]} - \eta\, \delta^{[\ell]}$

return $\{W^{[\ell]}\}, \{b^{[\ell]}\}$

Outline

Matlab implementations

- HH19 includes a Matlab implementation of TRAINING() for the small ($L = 4$) ANN and classification task I have been showing
- see `netbp.m` and `netbpfull.m` at

  www.maths.ed.ac.uk/~dhigham/algfiles.html
- I rewrote this code for my own amusement; see `example1.m` at

  github.com/bueler/ml-seminar/tree/main/talk1/code
  - my codes are tested in both Matlab and Octave

- as a UAF person you have access to Matlab online if you want it

  matlab.mathworks.com
- from now on, I'll assume you can run these things

HH19 code `netbp.m` fits on one page

```
function netbp
%NETBP  Uses backpropagation to train a network

%%%%%%% DATA %%%%%%%%%%%
x1 = [0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7];
x2 = [0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6];
y = [ones(1,5) zeros(1,5); zeros(1,5) ones(1,5)];

% Initialize weights and biases
rng(5000);
W2 = 0.5*randn(2,2); W3 = 0.5*randn(3,2); W4 = 0.5*randn(2,3);
b2 = 0.5*randn(2,1); b3 = 0.5*randn(3,1); b4 = 0.5*randn(2,1);

% Forward and Back propagate
eta = 0.05;                 % learning rate
Niter = 1e6;                % number of SG iterations
savecost = zeros(Niter,1); % value of cost function at each iteration
for counter = 1:Niter
    k = randi(10);          % choose a training point at random
    x = [x1(k); x2(k)];
    % Forward pass
```

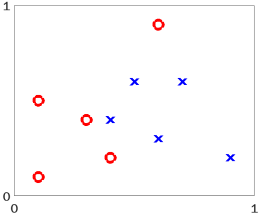
⋮

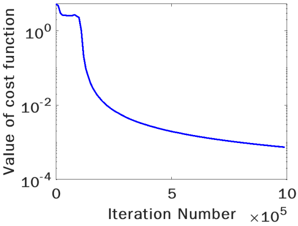
running `netbp.m`

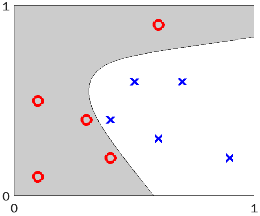
- run graphics version of `netbp.m` in Matlab online:

```
>> tic, netbpfull, toc
   ... spews cost values
Elapsed time is 148.599924 seconds.
```

- does 10^6 SGD iterations ... that's not great for a small network
- *result:* right figure below shows the contour where $a_1^{[4]} > a_2^{[4]}$
- my octave version produces similar result in similar time





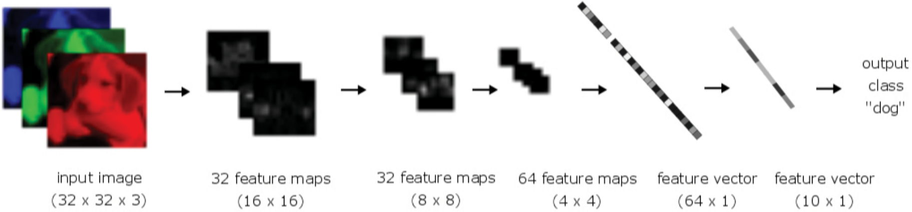


Outline

there is so much more to say

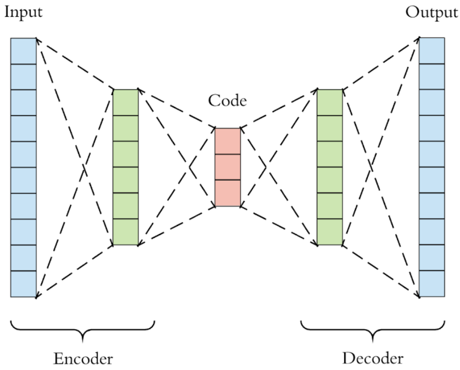
- please actually read HH19?
- next are 4 topics which might catch your interest as a talk
  - these topics have math inside!

- based on discrete convolution of time series and images
  - a talk to explain convolution, *sans* neural nets?
- CNNs win at image classification
- example given in HH19



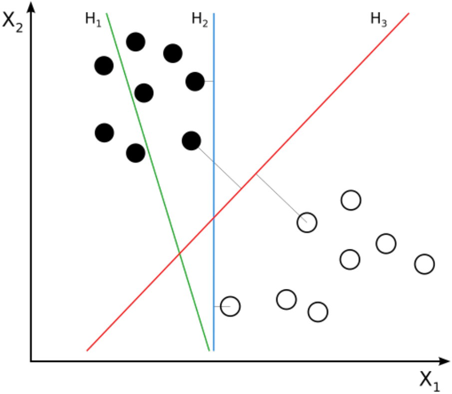
autoencoders

- one form of *unsupervised training*
- fit the identity map with a fancy nonlinear function (!)



support vector machines

- recall: “single layer perceptrons can’t compute XOR”
  - linearly-separable classification problems
- support vector machines (SVM)
  - add stable optimality to linearly-separable classification



- stochastic optimization
  - the objective function is random; the goal is to minimize the expected objective value
  - SGD is well-suited for this goal?
- online machine learning model
- improvements on SGD
  - momentum
  - dropout
  - Adam ($\to$), and etc.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

additional topics?

- recurrent neural networks
- graph neural networks
- algorithmic/automatic differentiation
- classical nonlinear least-squares methods
  - Gauss-Newton
  - Levenberg-Marquardt
- classical optimization: Newton-type methods
  - quasi-Newton methods, especially L-BFGS
- and on and on through the buzzwords ...