Lecture 25

Neural networks

COURSE MATH 637: Mathematical Techniques in Data Science

WEBSITE https://jacobcalvert.com/teaching/math637fall2026/

INSTRUCTOR Jacob Calvert (calvert@udel.edu)

OFFICE HOURS MW 2:45pm–3:45pm, Ewing Hall 509

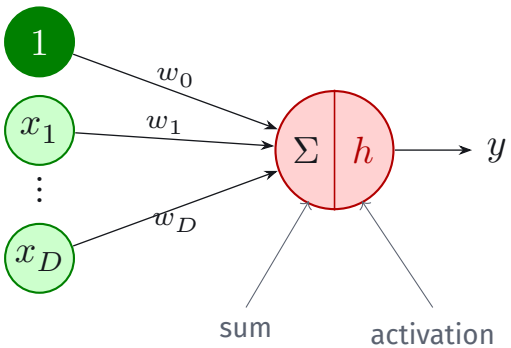
Announcements & Reminders

- HW5 (1-page project proposal) is due by class on Wed. Nov. 11

Plan for today

• Neurons, networks	15 min
• Activation functions	8 min
• Large networks and their parameters	7 min
• Error functions	5 min
• Optimization	15 min

Definition: neuron



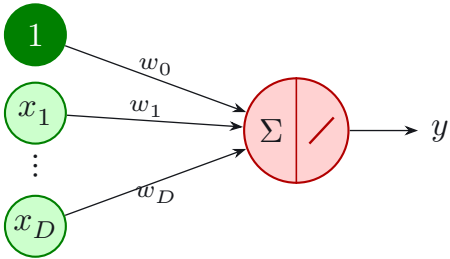
- $x_1, \ldots, x_D$ are the inputs
- $w_1, \ldots, w_D$ are the weights
- $x_0 = 1$, w_0 is the bias
- a is the pre-activation

$$a = \sum_{i=0}^{D} w_i x_i = w_0 + \sum_{i=1}^{D} w_i x_i$$

- h is the activation function
- $y = h(a)$ is the output

Example: linear and logistic regression

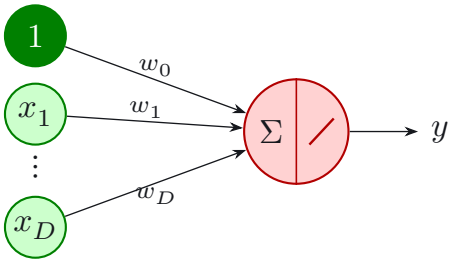
$h(a) = a$ gives linear regression



$$y = w_0 + w_1 x_1 + \cdots + w_D x_D$$

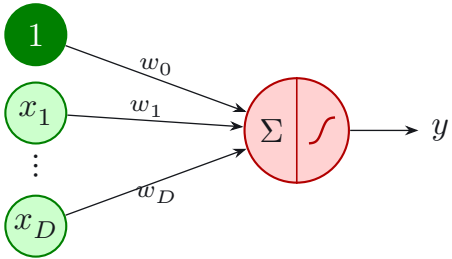
Example: linear and logistic regression

$h(a) = a$ gives linear regression



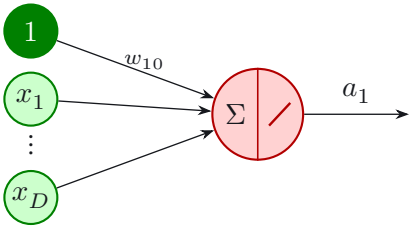
$$y = w_0 + w_1 x_1 + \cdots + w_D x_D$$

$h = \sigma$ gives logistic regression



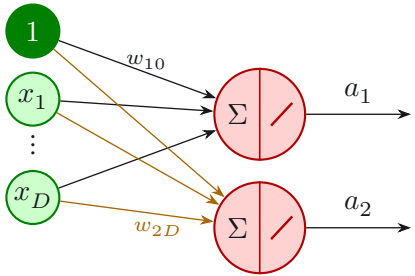
$$y = \sigma(\mathbf{w}^\mathsf{T}\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\mathsf{T}\mathbf{x}}}$$

Example: a network for K-class classification



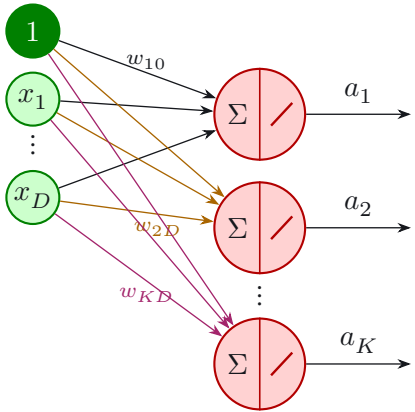
- the neuron from before:
  $h(a_1) = a_1 = \mathbf{w}_1^\mathsf{T}\mathbf{x}$

Example: a network for K-class classification



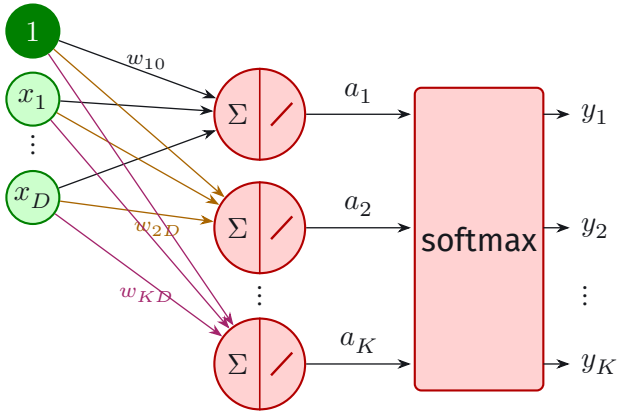
- the neuron from before: $h(a_1) = a_1 = \mathbf{w}_1^\mathsf{T}\mathbf{x}$
- a second neuron: $h(a_2) = a_2 = \mathbf{w}_2^\mathsf{T}\mathbf{x}$

Example: a network for K-class classification



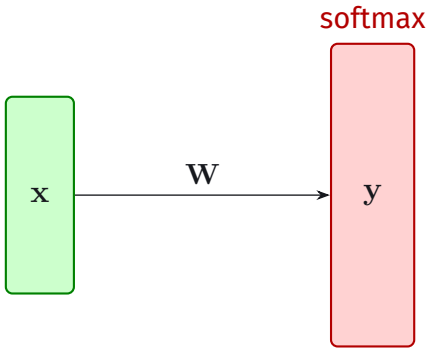
- the neuron from before: $h(a_1) = a_1 = \mathbf{w}_1^\mathsf{T}\mathbf{x}$
- a second neuron: $h(a_2) = a_2 = \mathbf{w}_2^\mathsf{T}\mathbf{x}$
- K neurons form a layer, $\mathbf{w}_k^\mathsf{T}$ is kth row of $\mathbf{W}$

Example: a network for K-class classification



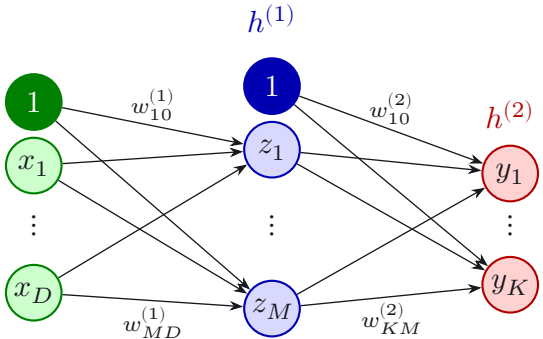
- the neuron from before: $h(a_1) = a_1 = \mathbf{w}_1^\mathsf{T}\mathbf{x}$
- a second neuron: $h(a_2) = a_2 = \mathbf{w}_2^\mathsf{T}\mathbf{x}$
- K neurons form a layer, $\mathbf{w}_k^\mathsf{T}$ is kth row of $\mathbf{W}$
- softmax uses all K sums at once: $y_k = e^{a_k} / \sum_l e^{a_l}$

Example: a network for K-class classification



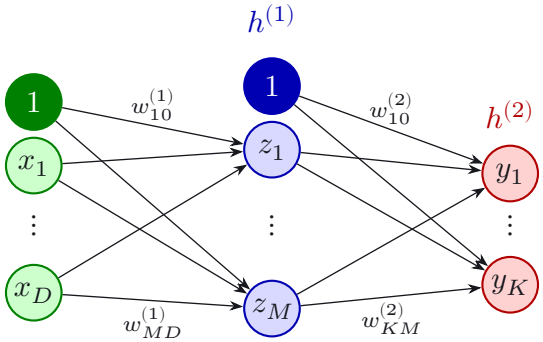
- the neuron from before: $h(a_1) = a_1 = \mathbf{w}_1^\mathsf{T}\mathbf{x}$
- a second neuron: $h(a_2) = a_2 = \mathbf{w}_2^\mathsf{T}\mathbf{x}$
- K neurons form a layer, $\mathbf{w}_k^\mathsf{T}$ is kth row of $\mathbf{W}$
- softmax uses all K sums at once: $y_k = e^{a_k} / \sum_l e^{a_l}$
- as blocks: $\mathbf{y} = \mathrm{softmax}(\mathbf{W}\mathbf{x})$

A network with a hidden layer



- $h^{(l)} =$ activation function for layer l
- $w_{ji}^{(l)} =$ weight in layer l from unit i to unit j
- "hidden units" $z_1, \ldots, z_M$

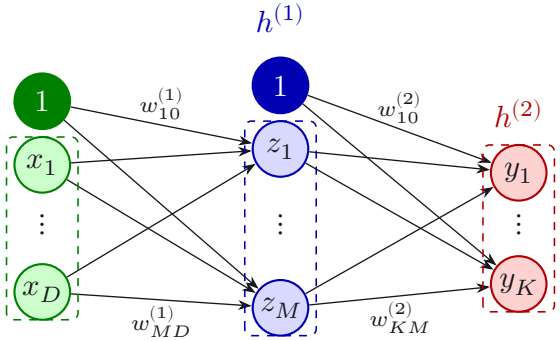
A network with a hidden layer



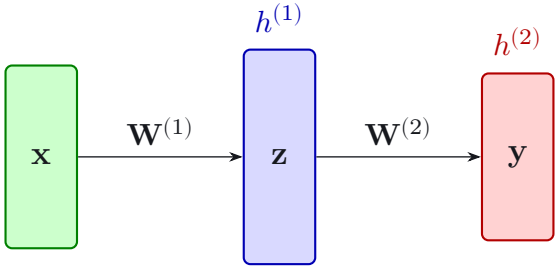
- $h^{(l)} =$ activation function for layer l
- $w_{ji}^{(l)} =$ weight in layer l from unit i to unit j
- "hidden units" $z_1, \dots, z_M$

$$z_j = h^{(1)}\Big(\sum_{i=0}^{D} w_{ji}^{(1)}\, x_i\Big), \qquad y_k = h^{(2)}\Big(\sum_{j=0}^{M} w_{kj}^{(2)}\, z_j\Big)$$

From units to blocks

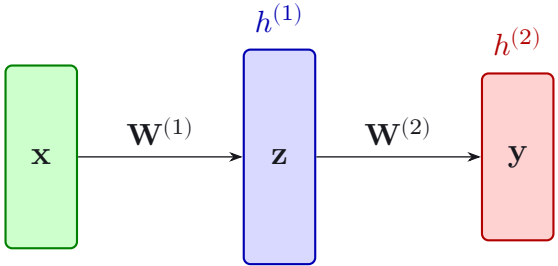


From units to blocks



- name of layer's units inside, activation above
- arrow $=$ apply matrix $\mathbf{W}^{(l)}$ of weights between consecutive layers
- not drawn: the constants 1 and biases, i.e., column "0" of $\mathbf{W}^{(l)}$

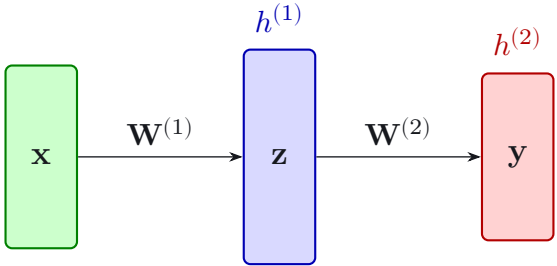
From units to blocks



- name of layer's units inside, activation above
- arrow $=$ apply matrix $\mathbf{W}^{(l)}$ of weights between consecutive layers
- not drawn: the constants 1 and biases, i.e., column "0" of $\mathbf{W}^{(l)}$

$$\mathbf{z} = h^{(1)}(\mathbf{W}^{(1)}\mathbf{x}), \quad \mathbf{y} = h^{(2)}(\mathbf{W}^{(2)}\mathbf{z})$$

From units to blocks

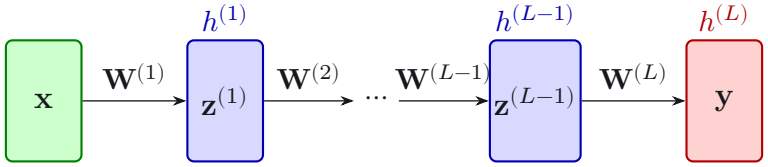


- name of layer's units inside, activation above
- arrow $=$ apply matrix $\mathbf{W}^{(l)}$ of weights between consecutive layers
- not drawn: the constants 1 and biases, i.e., column "0" of $\mathbf{W}^{(l)}$

$$\mathbf{z} = h^{(1)}(\mathbf{W}^{(1)}\mathbf{x}), \quad \mathbf{y} = h^{(2)}(\mathbf{W}^{(2)}\mathbf{z})$$

Note: The dimensions are $\mathbf{W}^{(1)} \in \mathbb{R}^{M\times(D+1)}$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{K\times(M+1)}$.

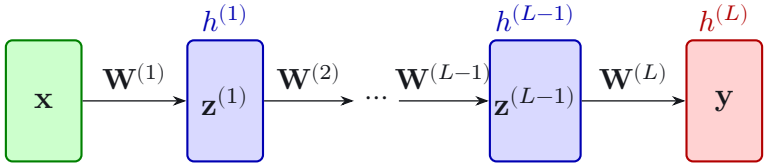
A network is a composition



Definition (Neural network with L layers)

$\mathbf{z}^{(l)} = h^{(l)}\big(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)}\big)$ for $l = 1, \dots, L$, with $\mathbf{z}^{(0)} = \mathbf{x}$ and $\mathbf{y} = \mathbf{z}^{(L)}$.

A network is a composition



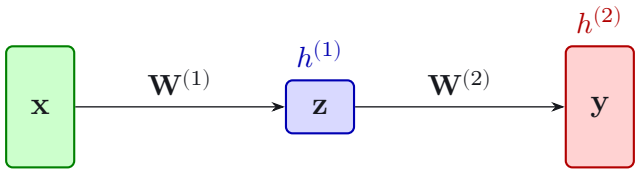
Definition (Neural network with L layers)

$\mathbf{z}^{(l)} = h^{(l)}(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)})$ for $l = 1, \dots, L$, with $\mathbf{z}^{(0)} = \mathbf{x}$ and $\mathbf{y} = \mathbf{z}^{(L)}$.

- each layer is an affine map, then an activation applied entrywise
- $\mathbf{y} = h^{(L)}(\mathbf{W}^{(L)} \cdots h^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))$
- $\mathbf{W}^{(l)}$ acts on $\mathbf{z}^{(l-1)}$ with the entry $z_0^{(l-1)} = 1$, which is not shown

Example: a network with a bottleneck

The middle layer $\mathbf{z} \in \mathbb{R}^M$ has fewer units than the outer ones $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$.

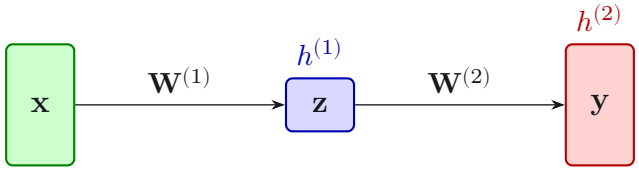


Question

What does this network remind you of?

Example: a network with a bottleneck

The middle layer $\mathbf{z} \in \mathbb{R}^M$ has fewer units than the outer ones $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$.



Question

What does this network remind you of?

Question

How does it differ from the encoder–decoder model of the dimension reduction module?

Summary of notation

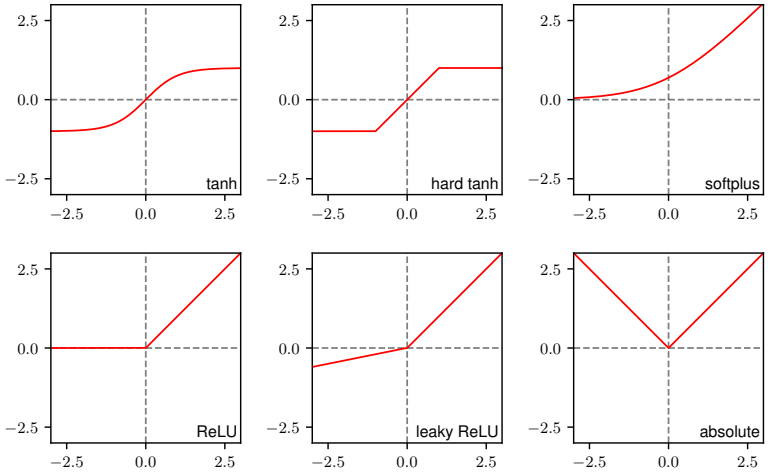
symbol	meaning
$\mathbf{x} = (x_0, x_1, \ldots, x_D)$	an input, with $x_0 = 1$
$\mathbf{t}$, $\mathbf{y}(\mathbf{x}, \mathbf{w})$	the target, and the network's output (K entries)
$\mathbf{W}^{(l)}$, $w_{ji}^{(l)}$	layer l's weights; entry (j, i) joins unit i to unit j; column 0: biases
a_j, $z_j = h(a_j)$	a unit's pre-activation and its value; $z_0 = 1$
$h^{(l)}$, σ	layer l's activation function; the logistic sigmoid
$\mathbf{w}$, P	all parameters, stacked; their number
N, n	the number of examples; the index of an example

- vectors like $\mathbf{x}$, $\mathbf{w}$ and matrices $\mathbf{W}^{(l)}$ are bold, while numbers, including their entries x_i, $w_{ji}^{(l)}$, are italic.

Plan for today

• Neurons, networks	15 min
• Activation functions	8 min
• Large networks and their parameters	7 min
• Error functions	5 min
• Optimization	15 min

Activation functions are usually nonlinear

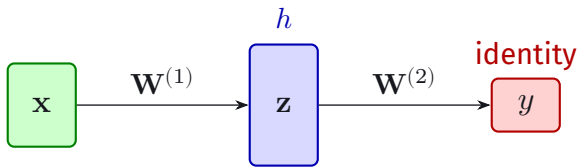


Bishop and Bishop (2024), Fig. 6.12

Why nonlinear? Approximation w/ one hidden layer

Theorem (Universal approximation (Leshno et al., 1993))

Let h be continuous and not a polynomial, f continuous on a closed, bounded set $K \subset \mathbb{R}^D$, and $\epsilon > 0$. For some M and some weights, the network below has $|y(\mathbf{x}) - f(\mathbf{x})| < \epsilon$ for all $\mathbf{x}$ in K.

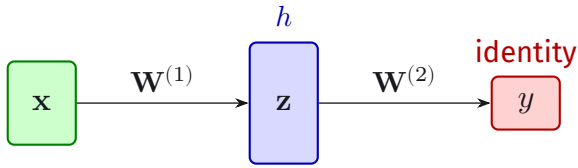


- M hidden units, one output: $y(\mathbf{x}) = w_0^{(2)} + \sum_{j=1}^{M} w_j^{(2)} h(a_j)$

Why nonlinear? Approximation w/ one hidden layer

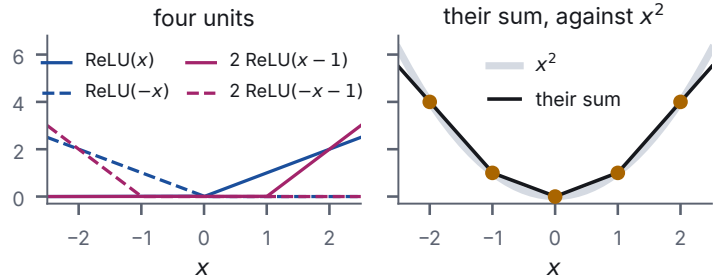
Theorem (Universal approximation (Leshno et al., 1993))

Let h be continuous and not a polynomial, f continuous on a closed, bounded set $K \subset \mathbb{R}^D$, and $\epsilon > 0$. For some M and some weights, the network below has $|y(\mathbf{x}) - f(\mathbf{x})| < \epsilon$ for all $\mathbf{x}$ in K.



- M hidden units, one output: $y(\mathbf{x}) = w_0^{(2)} + \sum_{j=1}^{M} w_j^{(2)} h(a_j)$
- existence only: no bound on M, and no method for finding the weights

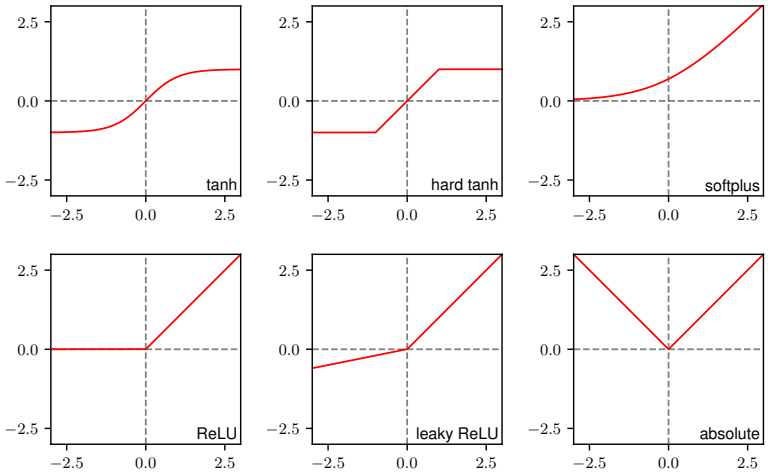
Example: Building x^2 from ReLU units



$$\mathrm{ReLU}(x) + \mathrm{ReLU}(-x) + 2\,\mathrm{ReLU}(x-1) + 2\,\mathrm{ReLU}(-x-1)$$

- equals x^2 at $x = -2, -1, 0, 1, 2$, and is linear in between
- a linear combination of ReLU units is piecewise linear

ReLU is a common choice of activation function



Bishop and Bishop (2024), Fig. 6.12

Why?: vanishing gradients problem

- we'll train neural nets using (variants of) gradient descent to minimize an error function $E(\mathbf{w})$ over weight vectors $\mathbf{w}$

Why?: vanishing gradients problem

- we'll train neural nets using (variants of) gradient descent to minimize an error function $E(\mathbf{w})$ over weight vectors $\mathbf{w}$
- a gradient descent step changes each weight w by $-\eta\,\partial_w E(\mathbf{w})$

Why?: vanishing gradients problem

- we'll train neural nets using (variants of) gradient descent to minimize an error function $E(\mathbf{w})$ over weight vectors $\mathbf{w}$
- a gradient descent step changes each weight w by $-\eta\,\partial_w E(\mathbf{w})$
- for a weight in an early layer, each term of $\partial_w E(\mathbf{w})$ contains one factor $h'(a_j)$ for each later layer due to the chain rule

Why?: vanishing gradients problem

- we'll train neural nets using (variants of) gradient descent to minimize an error function $E(\mathbf{w})$ over weight vectors $\mathbf{w}$
- a gradient descent step changes each weight w by $-\eta\,\partial_w E(\mathbf{w})$
- for a weight in an early layer, each term of $\partial_w E(\mathbf{w})$ contains one factor $h'(a_j)$ for each later layer due to the chain rule
- Example: for 10 sigmoid layers: these factors multiply to at most $4^{-10} \approx 10^{-6}$

Why?: vanishing gradients problem

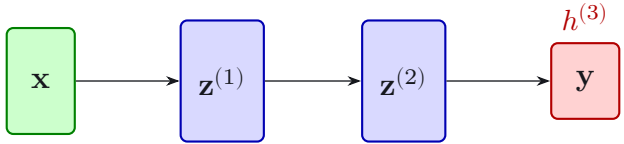
- we'll train neural nets using (variants of) gradient descent to minimize an error function $E(\mathbf{w})$ over weight vectors $\mathbf{w}$
- a gradient descent step changes each weight w by $-\eta\,\partial_w E(\mathbf{w})$
- for a weight in an early layer, each term of $\partial_w E(\mathbf{w})$ contains one factor $h'(a_j)$ for each later layer due to the chain rule
- Example: for 10 sigmoid layers: these factors multiply to at most $4^{-10} \approx 10^{-6}$
- so, with a step size η that suits the last layers, the weights of the first layers barely change: the vanishing gradients problem

Why?: vanishing gradients problem

- we'll train neural nets using (variants of) gradient descent to minimize an error function $E(\mathbf{w})$ over weight vectors $\mathbf{w}$
- a gradient descent step changes each weight w by $-\eta\,\partial_w E(\mathbf{w})$
- for a weight in an early layer, each term of $\partial_w E(\mathbf{w})$ contains one factor $h'(a_j)$ for each later layer due to the chain rule
- Example: for 10 sigmoid layers: these factors multiply to at most $4^{-10} \approx 10^{-6}$
- so, with a step size η that suits the last layers, the weights of the first layers barely change: the vanishing gradients problem

Key idea: The derivative of ReLU is exactly 1 wherever a unit is on, so these factors do not shrink with layer, and early layers keep learning!

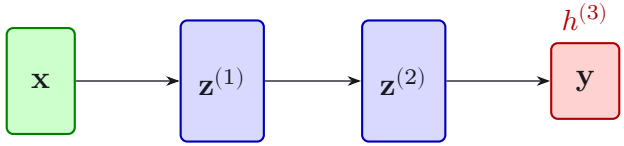
An exception: the output activation



- hidden layers: ReLU activation chosen to make training work
- output layer: activation chosen so that $\mathbf{y}$ has the form of the target

target	output activation	output
real numbers	identity	$\mathbf{y} \in \mathbb{R}^K$
one of two classes	σ	$y \in (0, 1)$, a probability
one of K classes	softmax	$y_k > 0$, $\sum_k y_k = 1$

An exception: the output activation



- hidden layers: ReLU activation chosen to make training work
- output layer: activation chosen so that $\mathbf{y}$ has the form of the target

target	output activation	output
real numbers	identity	$\mathbf{y} \in \mathbb{R}^K$
one of two classes	σ	$y \in (0,1)$, a probability
one of K classes	softmax	$y_k > 0$, $\sum_k y_k = 1$

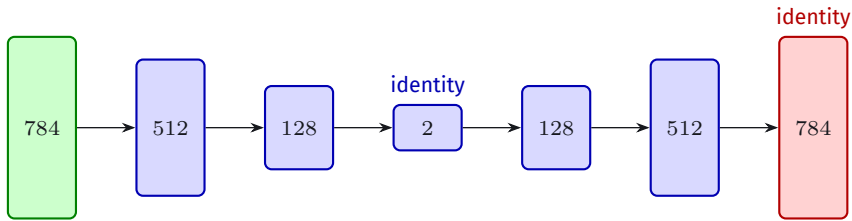
Note: From here on, a hidden block with no activation written above it uses ReLU, the output activation is always written.

Plan for today

- Neurons, networks — 15 min
- Activation functions — 8 min
- Large networks and their parameters — 7 min
- Error functions — 5 min
- Optimization — 15 min

Example: a larger network

A 28×28 MNIST image $=$ vector w/ $D = 784$ entries. A network for encoding such images:



- blocks $=$ layers labeled by width; arrows $=$ weight matrices $\mathbf{W}^{(l)}$
- activation functions $=$ ReLU except for those labeled identity

Counting parameters: a unit, a layer

One unit with q inputs

- weights $w_{j1}, \ldots, w_{jq}$ and bias w_{j0}: $q + 1$
- the activation has no parameters

One layer: q inputs, m units

- m units, each with $q + 1$: $m(q + 1)$
- the size of $\mathbf{W}^{(l)} \in \mathbb{R}^{m \times (q+1)}$: the $+1$ is the column for the constant 1

Counting parameters: a unit, a layer

One unit with q inputs

- weights $w_{j1}, \ldots, w_{jq}$ and bias w_{j0}: $q+1$
- the activation has no parameters

One layer: q inputs, m units

- m units, each with $q+1$: $m(q+1)$
- the size of $\mathbf{W}^{(l)} \in \mathbb{R}^{m \times (q+1)}$: the $+1$ is the column for the constant 1

A network with widths $d_0, d_1, \ldots, d_L$

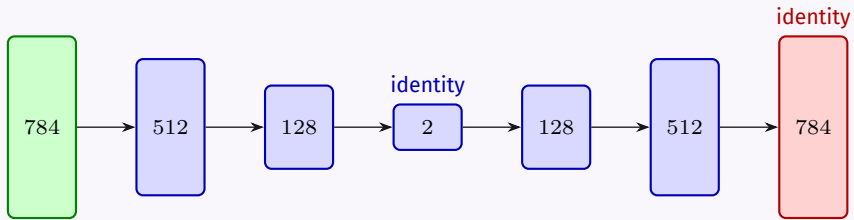
$$P = \sum_{l=1}^{L} d_l \left(d_{l-1} + 1\right)$$

- $P =$ the number of parameters

Exercise: counting parameters

Question

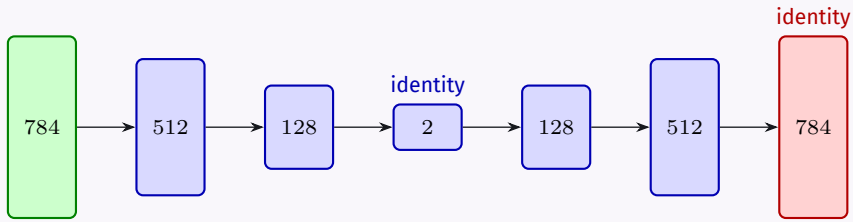
How many parameters does this network have?



Exercise: counting parameters

Question

How many parameters does this network have?



Answer

$P = 512 \cdot 785 + 128 \cdot 513 + 2 \cdot 129 + 128 \cdot 3 + 512 \cdot 129 + 784 \cdot 513 = 936{,}466$

Plan for today

• Neurons, networks	15 min
• Activation functions	8 min
• Large networks and their parameters	7 min
• Error functions	5 min
• Optimization	15 min

Error functions

Note: Given example $\mathbf{x}_n$ with label of "target" t_n or $\mathbf{t}_n$, output of network with weights $\mathbf{w}$ is denoted y_n or $\mathbf{y}_n$.

task	output activation	$E_n(\mathbf{w})$
regression	identity	$\frac{1}{2}(y_n - t_n)^2$
two classes, $t_n \in \{0, 1\}$	σ	$-t_n \ln y_n - (1 - t_n) \ln(1 - y_n)$
K classes, $\mathbf{t}_n$ one-hot	softmax	$-\sum_k t_{nk} \ln y_{nk}$
dimension reduction	identity	$\frac{1}{2} \left\| \mathbf{y}_n - \mathbf{x}_n \right\|^2$

Error functions

Note: Given example $\mathbf{x}_n$ with label of "target" t_n or $\mathbf{t}_n$, output of network with weights $\mathbf{w}$ is denoted y_n or $\mathbf{y}_n$.

task	output activation	$E_n(\mathbf{w})$
regression	identity	$\frac{1}{2}(y_n - t_n)^2$
two classes, $t_n \in \{0, 1\}$	σ	$-t_n \ln y_n - (1 - t_n) \ln(1 - y_n)$
K classes, $\mathbf{t}_n$ one-hot	softmax	$-\sum_k t_{nk} \ln y_{nk}$
dimension reduction	identity	$\frac{1}{2} \left\| \mathbf{y}_n - \mathbf{x}_n \right\|^2$

Key idea: The task tells you what output activation and error to use.

The training problem

Stack all weights into one vector $\mathbf{w}$, with P entries.

Problem (Training)

$$\text{minimize} \quad E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \quad \text{over } \mathbf{w} \in \mathbb{R}^P$$

- Example: $P = 936{,}466$ for the encoder–decoder network

The training problem

Stack all weights into one vector $\mathbf{w}$, with P entries.

Problem (Training)

$$\text{minimize} \quad E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \quad \text{over } \mathbf{w} \in \mathbb{R}^P$$

- Example: $P = 936{,}466$ for the encoder–decoder network
- Issue 1: unlike linear regression, no closed form

The training problem

Stack all weights into one vector $\mathbf{w}$, with P entries.

Problem (Training)

$$\text{minimize} \quad E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \quad \text{over } \mathbf{w} \in \mathbb{R}^P$$

- Example: $P = 936{,}466$ for the encoder–decoder network
- Issue 1: unlike linear regression, no closed form
- Issue 2: Newton's method needs a $P \times P$ Hessian, which has $936{,}466^2 \approx 9 \times 10^{11}$ entries in our example

The training problem

Stack all weights into one vector $\mathbf{w}$, with P entries.

Problem (Training)

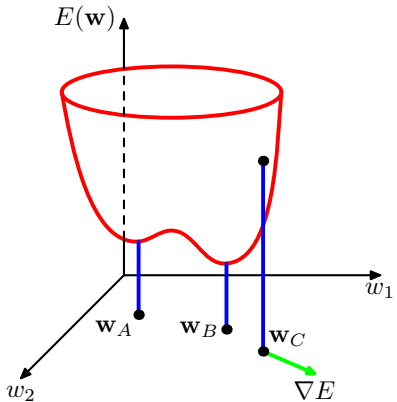
$$\text{minimize} \quad E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \quad \text{over } \mathbf{w} \in \mathbb{R}^P$$

- Example: $P = 936{,}466$ for the encoder–decoder network
- Issue 1: unlike linear regression, no closed form
- Issue 2: Newton's method needs a $P \times P$ Hessian, which has $936{,}466^2 \approx 9 \times 10^{11}$ entries in our example
- Issue 3: E likely not convex (consider exchanging two hidden units, with their weights)

Plan for today

• Neurons, networks	15 min
• Activation functions	8 min
• Large networks and their parameters	7 min
• Error functions	5 min
• Optimization	15 min

The error surface



Bishop and Bishop, Fig. 7.1

- $E(\mathbf{w})$ is a surface over weight space $\mathbb{R}^P$
- $E(\mathbf{w} + \delta\mathbf{w}) \approx E(\mathbf{w}) + \delta\mathbf{w}^\mathsf{T}\nabla E(\mathbf{w})$, so $-\nabla E$ is direction of steepest descent
- $\nabla E(\mathbf{w}) = \mathbf{0}$ can give minima, maxima, saddle points
- $\mathbf{w}_A =$ a local minimum; $\mathbf{w}_B =$ global min

Batch gradient descent

At step τ, with learning rate $\eta > 0$, do:

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \nabla E(\mathbf{w}^{(\tau-1)}), \qquad \nabla E = \sum_{n=1}^{N} \nabla E_n.$$

Key idea: Every step of batch gradient descent uses all N training examples, requiring N gradient evaluations. When N is large this is slow.

- Example: MNIST has $N = 10{,}000$, so $10{,}000$ gradients must be calculated for one step of batch gradient descent.

The other extreme: stochastic gradient descent

Algorithm 7.1: Stochastic gradient descent

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Error function per data point $E_n(\mathbf{w})$
Learning rate parameter η
Initial weight vector $\mathbf{w}$

Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
repeat
  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_n(\mathbf{w})$ `// update weight vector`
  $n \leftarrow n + 1 (\mathrm{mod}\ N)$ `// iterate over data`
until convergence
return $\mathbf{w}$

Bishop and Bishop, Algorithm 7.1

Key idea: Use only example per step, hence one gradient evaluation per step. This gives "noisy" steps, since $\nabla E_n \approx \nabla E$.

Happy medium: Mini-batch SGD

Algorithm 7.2: Mini-batch stochastic gradient descent

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Initial weight vector $\mathbf{w}$

Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
repeat
  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_{n:n+B-1}(\mathbf{w})$ `// weight vector update`
  $n \leftarrow n + B$
  if $n > N$ **then**
    shuffle data
    $n \leftarrow 1$
  end if
until convergence
return $\mathbf{w}$

Bishop and Bishop, Algorithm 7.2

Key idea: Use B examples per step. Note that $B = 1$ is SGD, $B = N$ is batch gradient descent.

- the gradient's standard deviation falls like $1/\sqrt{B}$, but cost grows like B
- MNIST with $B = 64$ requires 157 updates per epoch

Now add momentum

Algorithm 7.3: Stochastic gradient descent with momentum

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Momentum parameter μ
Initial weight vector $\mathbf{w}$

Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
$\Delta\mathbf{w} \leftarrow \mathbf{0}$
repeat
  $\Delta\mathbf{w} \leftarrow -\eta\nabla E_{n:n+B-1}(\mathbf{w}) + \mu\Delta\mathbf{w}$ // calculate update term
  $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ // weight vector update
  $n \leftarrow n + B$
  if $n > N$ **then**
    shuffle data
    $n \leftarrow 1$
  end if
until convergence
return $\mathbf{w}$

Bishop and Bishop, Algorithm 7.3

Key idea: $\Delta\mathbf{w}$ accumulates past steps. A constant gradient gives steps tending to $-\eta\nabla E/(1-\mu)$, ten times larger at $\mu = 0.9$, while oscillating steps cancel.

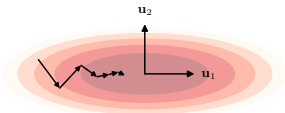


Fig. 7.6

Adam

Algorithm 7.4: Adam optimization

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Decay parameters β_1 and β_2
Stabilization parameter δ
Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
$\mathbf{s} \leftarrow \mathbf{0}$
$\mathbf{r} \leftarrow \mathbf{0}$
repeat
  Choose a mini-batch at random from $\mathcal{D}$
  $\mathbf{g} = -\nabla E_{n:n+B-1}(\mathbf{w})$ // evaluate gradient vector
  $\mathbf{s} \leftarrow \beta_1 \mathbf{s} + (1 - \beta_1)\mathbf{g}$
  $\mathbf{r} \leftarrow \beta_2 \mathbf{r} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$ // element-wise multiply
  $\widehat{\mathbf{s}} \leftarrow \mathbf{s}/(1 - \beta_1^\tau)$ // bias correction
  $\widehat{\mathbf{r}} \leftarrow \mathbf{r}/(1 - \beta_2^\tau)$ // bias correction
  $\Delta\mathbf{w} \leftarrow -\eta \frac{\widehat{\mathbf{s}}}{\sqrt{\widehat{\mathbf{r}}} + \delta}$ // element-wise operations
  $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ // weight vector update
  $n \leftarrow n + B$
  if $n + B > N$ **then**
    shuffle data
    $n \leftarrow 1$
  end if
until convergence
return $\mathbf{w}$

Bishop and Bishop, Algorithm 7.4

Key idea: A step size for each weight based on s, which averages the gradient, and its square r its square. Each step size is about η in every coordinate, whatever the scale of the gradient.

Adam

Algorithm 7.4: Adam optimization

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Decay parameters β_1 and β_2
Stabilization parameter δ
Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
$\mathbf{s} \leftarrow \mathbf{0}$
$\mathbf{r} \leftarrow \mathbf{0}$
repeat
  Choose a mini-batch at random from $\mathcal{D}$
  $\mathbf{g} = -\nabla E_{n:n+B-1}(\mathbf{w})$ // evaluate gradient vector
  $\mathbf{s} \leftarrow \beta_1 \mathbf{s} + (1 - \beta_1)\mathbf{g}$
  $\mathbf{r} \leftarrow \beta_2 \mathbf{r} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$ // element-wise multiply
  $\hat{\mathbf{s}} \leftarrow \mathbf{s}/(1 - \beta_1^\tau)$ // bias correction
  $\hat{\mathbf{r}} \leftarrow \mathbf{r}/(1 - \beta_2^\tau)$ // bias correction
  $\Delta\mathbf{w} \leftarrow -\eta \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ // element-wise operations
  $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ // weight vector update
  $n \leftarrow n + B$
  if $n + B > N$ **then**
    shuffle data
    $n \leftarrow 1$
  end if
until convergence
return $\mathbf{w}$

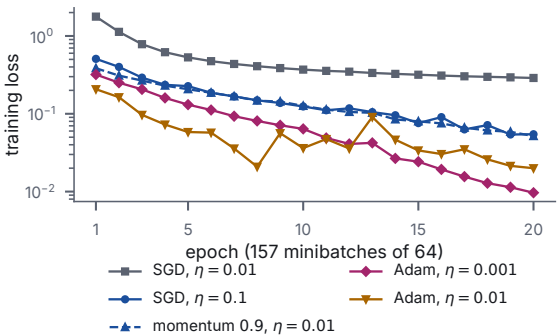
Bishop and Bishop, Algorithm 7.4

Key idea: A step size for each weight based on s, which averages the gradient, and its square r its square. Each step size is about η in every coordinate, whatever the scale of the gradient.

Watch out: Some typos in Bishop–Bishop: use $\mathbf{g} = \nabla E$ instead and if-statement should use $n > N$ instead of $n + B > N$.

Example: comparison of five optimizers

MNIST, 784–128–10, $B = 64$, 20 epochs, one run each:



optimizer	test accuracy
SGD, $\eta = 0.01$	92.6%
SGD, $\eta = 0.1$	96.55%
momentum, $\eta = 0.01$	96.35%
Adam, $\eta = 0.001$	96.6%
Adam, $\eta = 0.01$	95.45%

- SGD at 0.01 is far from fitting
- momentum at 0.01 tracks SGD at $0.1 = 0.01/(1 - 0.9)$

How to calculate the gradients?

Every update needs the P derivatives $\partial_{w_i} E_n(\mathbf{w})$ on a mini-batch.

Finite differences

- for each i calculate $(E_n(\mathbf{w} + \epsilon \mathbf{e}_i) - E_n(\mathbf{w}))/\epsilon$ for small ϵ
- requires $P + 1$ evaluations of the network

Backpropagation

- all P derivatives, by the chain rule
- cost is a small multiple of one evaluation

How to calculate the gradients?

Every update needs the P derivatives $\partial_{w_i} E_n(\mathbf{w})$ on a mini-batch.

Finite differences

- for each i calculate $(E_n(\mathbf{w} + \epsilon \mathbf{e}_i) - E_n(\mathbf{w}))/\epsilon$ for small ϵ
- requires $P + 1$ evaluations of the network

Backpropagation

- all P derivatives, by the chain rule
- cost is a small multiple of one evaluation

- SGD, mini-batches, momentum, and Adam need only the gradient, however it is computed

Python: The training loop

```
model = nn.Sequential(
    nn.Linear(784, 128), nn.ReLU(),
    nn.Linear(128, 10))       # no softmax layer
lossf = nn.CrossEntropyLoss()  # softmax + log loss
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
for epoch in range(20):
    for xb, yb in loader:     # shuffled minibatches
        loss = lossf(model(xb), yb)
        opt.zero_grad()
        loss.backward()       # all 101,770 derivatives
        opt.step()            # one Adam update
```

- `Linear(784, 128)`: $\mathbf{W}^{(1)}$, stored as a 128×784 `weight` and a `bias` (column 0)

Summary

- A network is a composition of layers that apply weight matrices, followed by an activation
- With one layer and one output, we recover linear and logistic regression
- A bottleneck network trained to reproduce its input recalls the dimension reduction problem
- Training minimizes $E(\mathbf{w})$ by mini-batch SGD or Adam, given the gradient