

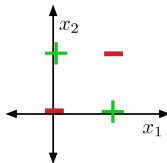
CSC 2515 Lecture 5: Neural Networks I

Roger Grosse

University of Toronto

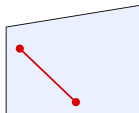
Limits of Linear Classification

- Visually, it's obvious that **XOR** is not linearly separable. But how to show this?



Limits of Linear Classification

Convex Sets



- A set \mathcal{S} is **convex** if any line segment connecting points in \mathcal{S} lies entirely within \mathcal{S} . Mathematically,

$$\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{S} \implies \lambda \mathbf{x}_1 + (1 - \lambda) \mathbf{x}_2 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1.$$

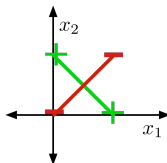
- A simple inductive argument shows that for $\mathbf{x}_1, \dots, \mathbf{x}_N \in \mathcal{S}$, **weighted averages**, or **convex combinations**, lie within the set:

$$\lambda_1 \mathbf{x}_1 + \dots + \lambda_N \mathbf{x}_N \in \mathcal{S} \quad \text{for } \lambda_i > 0, \lambda_1 + \dots + \lambda_N = 1.$$

Limits of Linear Classification

Showing that XOR is not linearly separable

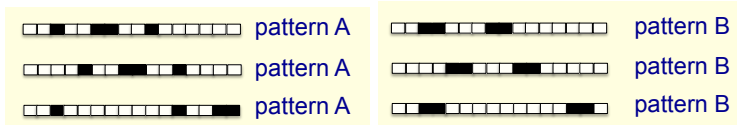
- Half-spaces are obviously convex.
- Suppose there were some feasible hypothesis. If the positive examples are in the positive half-space, then the green line segment must be as well.
- Similarly, the red line segment must lie within the negative half-space.



- But the intersection can't lie in both half-spaces. Contradiction!

Limits of Linear Classification

A more troubling example



- These images represent 16-dimensional vectors. White = 0, black = 1.
- Want to distinguish patterns A and B in all possible translations (with wrap-around)
- Translation invariance is commonly desired in vision!
- Suppose there's a feasible solution. The average of all translations of A is the vector $(0.25, 0.25, \dots, 0.25)$. Therefore, this point must be classified as A.
- Similarly, the average of all translations of B is also $(0.25, 0.25, \dots, 0.25)$. Therefore, it must be classified as B. Contradiction!

Limits of Linear Classification

- Sometimes we can overcome this limitation using feature maps, just like for linear regression. E.g., for **XOR**:

$$\psi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}$$

x_1	x_2	$\psi_1(\mathbf{x})$	$\psi_2(\mathbf{x})$	$\psi_3(\mathbf{x})$	t
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0

- This is linearly separable. (Try it!)
- Not a general solution: it can be hard to pick good **basis functions**. Instead, we'll use neural nets to learn nonlinear hypotheses directly.

Neural Networks

Inspiration: The Brain

- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons

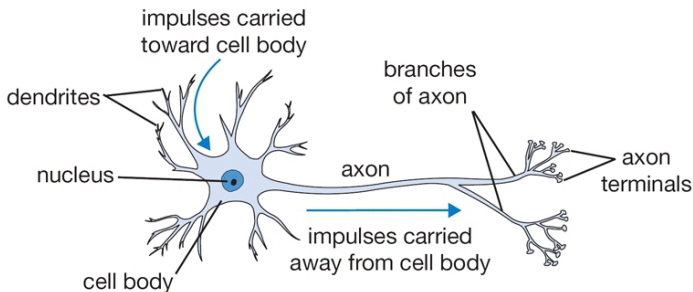
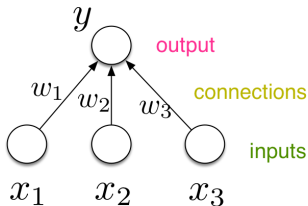


Figure: The basic computational unit of the brain: Neuron

[Pic credit: <http://cs231n.github.io/neural-networks-1/>]

Inspiration: The Brain

- For neural nets, we use a much simpler model neuron, or **unit**:



$$y = \phi(\mathbf{w}^T \mathbf{x} + b)$$

Diagram illustrating the mathematical representation of a neuron unit. The equation is $y = \phi(\mathbf{w}^T \mathbf{x} + b)$. Colored arrows point to the components: a pink arrow points to y (output), a blue arrow points to \mathbf{w} (weights), a blue arrow points to b (bias), a red arrow points to ϕ (activation function), and a green arrow points to \mathbf{x} (inputs).

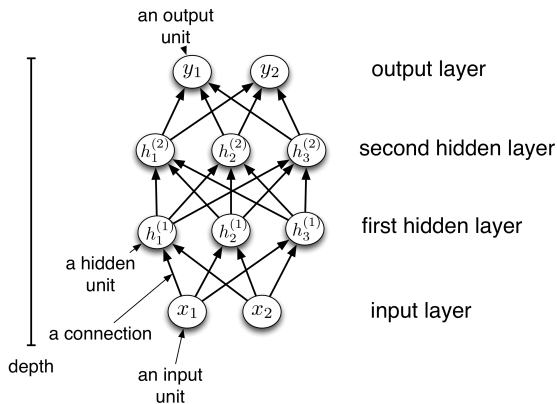
- Compare with logistic regression:

$$y = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

- By throwing together lots of these incredibly simplistic neuron-like processing units, we can do some powerful computations!

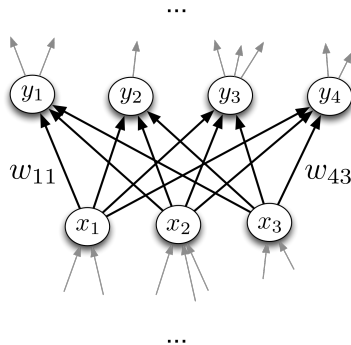
Multilayer Perceptrons

- We can connect lots of units together into a **directed acyclic graph**.
- This gives a **feed-forward neural network**. That's in contrast to **recurrent neural networks**, which can have cycles.
- Typically, units are grouped together into **layers**.



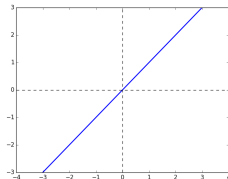
Multilayer Perceptrons

- Each layer connects N input units to M output units.
 - In the simplest case, all input units are connected to all output units. We call this a **fully connected layer**. We'll consider other layer types later.
 - Note: the inputs and outputs for a layer are distinct from the inputs and outputs to the network.
-
- Recall from softmax regression: this means we need an $M \times N$ weight matrix.
 - The output units are a function of the input units:
$$\mathbf{y} = f(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$
 - A multilayer network consisting of fully connected layers is called a **multilayer perceptron**. Despite the name, it has nothing to do with perceptrons!



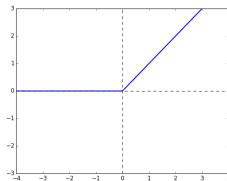
Multilayer Perceptrons

Some activation functions:



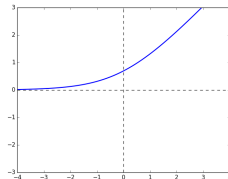
Linear

$$y = z$$



**Rectified Linear Unit
(ReLU)**

$$y = \max(0, z)$$

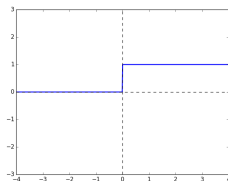


Soft ReLU

$$y = \log 1 + e^z$$

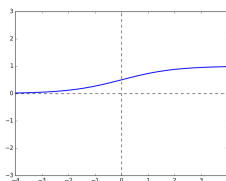
Multilayer Perceptrons

Some activation functions:



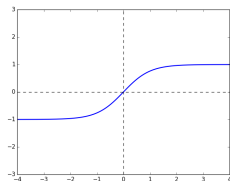
Hard Threshold

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$



Logistic

$$y = \frac{1}{1 + e^{-z}}$$



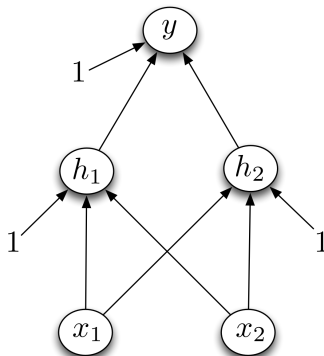
**Hyperbolic Tangent
(tanh)**

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Multilayer Perceptrons

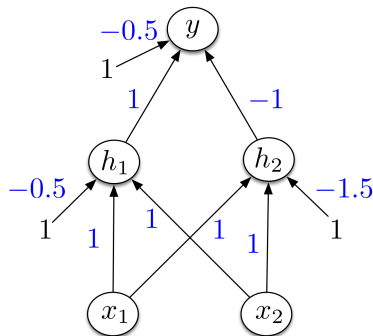
Designing a network to compute XOR:

Assume hard threshold activation function



Multilayer Perceptrons

- h_1 computes x_1 OR x_2
- h_2 computes x_1 AND x_2
- y computes h_1 AND NOT x_2



Multilayer Perceptrons

- Each layer computes a function, so the network computes a composition of functions:

$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x})$$

$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)})$$

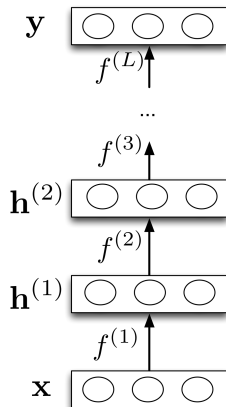
$$\vdots$$

$$\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)})$$

- Or more simply:

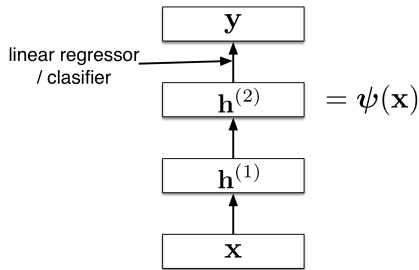
$$\mathbf{y} = f^{(L)} \circ \dots \circ f^{(1)}(\mathbf{x}).$$

- Neural nets provide modularity: we can implement each layer's computations as a black box.

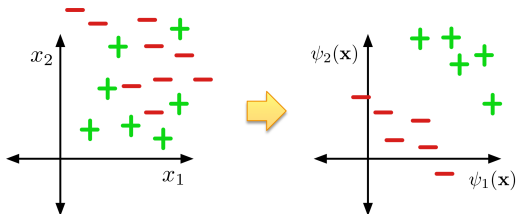


Feature Learning

- Neural nets can be viewed as a way of learning features:



- The goal:



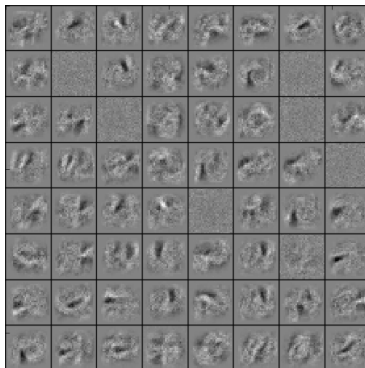
Feature Learning

- Suppose we're trying to classify images of handwritten digits. Each image is represented as a vector of $28 \times 28 = 784$ pixel values.
- Each first-layer hidden unit computes $\sigma(\mathbf{w}_i^T \mathbf{x})$. It acts as a **feature detector**.
- We can visualize \mathbf{w} by reshaping it into an image. Here's an example that responds to a diagonal stroke.



Feature Learning

Here are some of the features learned by the first hidden layer of a handwritten digit classifier:



- We've seen that there are some functions that linear classifiers can't represent. Are deep networks any better?
- Any sequence of *linear* layers can be equivalently represented with a single linear layer.

$$\mathbf{y} = \underbrace{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}}_{\triangleq \mathbf{W}'} \mathbf{x}$$

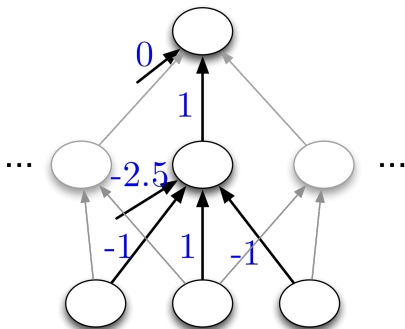
- Deep linear networks are no more expressive than linear regression!
- Linear layers do have their uses — stay tuned!

- Multilayer feed-forward neural nets with *nonlinear* activation functions are **universal function approximators**: they can approximate any function arbitrarily well.
- This has been shown for various activation functions (thresholds, logistic, ReLU, etc.)
 - Even though ReLU is “almost” linear, it’s nonlinear enough!

Universality for binary inputs and targets:

- Hard threshold hidden units, linear output
- Strategy: 2^D hidden units, each of which responds to one particular input configuration

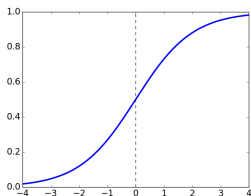
x_1	x_2	x_3	t
	\vdots		\vdots
-1	-1	1	-1
-1	1	-1	1
-1	1	1	1
	\vdots		\vdots



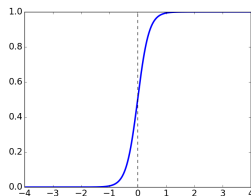
- Only requires one hidden layer, though it needs to be extremely wide!

Expressive Power

- What about the logistic activation function?
- You can approximate a hard threshold by scaling up the weights and biases:



$$y = \sigma(x)$$



$$y = \sigma(5x)$$

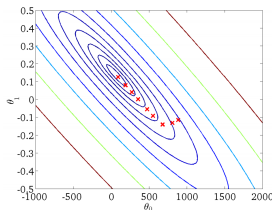
- This is good: logistic units are differentiable, so we can train them with gradient descent. (Stay tuned!)

- Limits of universality
 - You may need to represent an exponentially large network.
 - If you can learn any function, you'll just overfit.
 - Really, we desire a *compact* representation!
- We've derived units which compute the functions AND, OR, and NOT. Therefore, any Boolean circuit can be translated into a feed-forward neural net.
 - This suggests you might be able to learn *compact* representations of some complicated functions

Training neural networks with backpropagation

Recap: Gradient Descent

- **Recall:** gradient descent moves opposite the gradient (the direction of steepest descent)



- Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in *all* the layers
- Conceptually, not any different from what we've seen so far — just higher dimensional and harder to visualize!
- We want to compute the cost gradient $d\mathcal{J}/d\mathbf{w}$, which is the vector of partial derivatives.
 - This is the average of $d\mathcal{L}/d\mathbf{w}$ over all the training examples, so in this lecture we focus on computing $d\mathcal{L}/d\mathbf{w}$.

Univariate Chain Rule

- We've already been using the univariate Chain Rule.
- Recall: if $f(x)$ and $x(t)$ are univariate functions, then

$$\frac{d}{dt}f(x(t)) = \frac{df}{dx} \frac{dx}{dt}.$$

Recall: Univariate logistic least squares model

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Let's compute the loss derivatives.

How you would have done it in calculus class

$$\begin{aligned}\mathcal{L} &= \frac{1}{2}(\sigma(wx + b) - t)^2 \\ \frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w} \left[\frac{1}{2}(\sigma(wx + b) - t)^2 \right] \\ &= \frac{1}{2} \frac{\partial}{\partial w} (\sigma(wx + b) - t)^2 \\ &= (\sigma(wx + b) - t) \frac{\partial}{\partial w} (\sigma(wx + b) - t) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial w} (wx + b) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) x\end{aligned}$$
$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b} \left[\frac{1}{2}(\sigma(wx + b) - t)^2 \right] \\ &= \frac{1}{2} \frac{\partial}{\partial b} (\sigma(wx + b) - t)^2 \\ &= (\sigma(wx + b) - t) \frac{\partial}{\partial b} (\sigma(wx + b) - t) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial b} (wx + b) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b)\end{aligned}$$

What are the disadvantages of this approach?

Univariate Chain Rule

A more structured way to do it

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\frac{d\mathcal{L}}{dy} = y - t$$

$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{dy} \sigma'(z)$$

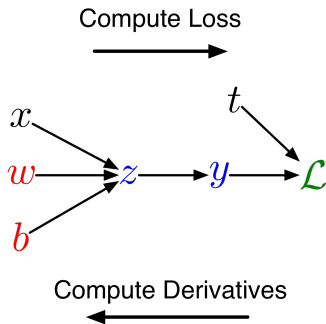
$$\frac{\partial \mathcal{L}}{\partial w} = \frac{d\mathcal{L}}{dz} x$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{d\mathcal{L}}{dz}$$

Remember, the goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Univariate Chain Rule

- We can diagram out the computations using a **computation graph**.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.



Univariate Chain Rule

A slightly more convenient notation:

- Use \bar{y} to denote the derivative $d\mathcal{L}/dy$, sometimes called the [error signal](#).
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).
- This is not a standard notation, but I couldn't find another one that I liked.

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\bar{y} = y - t$$

$$\bar{z} = \bar{y} \sigma'(z)$$

$$\bar{w} = \bar{z} x$$

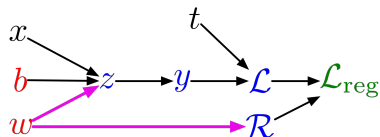
$$\bar{b} = \bar{z}$$

Multivariate Chain Rule

Problem: what if the computation graph has **fan-out** > 1 ?

This requires the **multivariate Chain Rule**!

L_2 -Regularized regression



$$z = wx + b$$

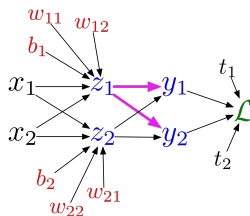
$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \mathcal{R}$$

Softmax regression



$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$

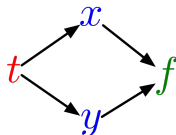
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$

$$\mathcal{L} = - \sum_k t_k \log y_k$$

Multivariate Chain Rule

- Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$. (All the variables here are scalar-valued.) Then

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$



- Example:

$$f(x, y) = y + e^{xy}$$

$$x(t) = \cos t$$

$$y(t) = t^2$$

- Plug in to Chain Rule:

$$\begin{aligned} \frac{df}{dt} &= \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt} \\ &= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t \end{aligned}$$

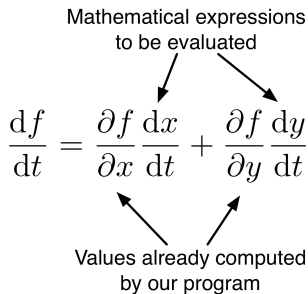
Multivariable Chain Rule

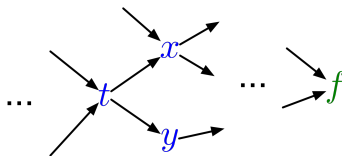
- In the context of backpropagation:

Mathematical expressions
to be evaluated

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

Values already computed
by our program





- In our notation:

$$\bar{t} = \bar{x} \frac{dx}{dt} + \bar{y} \frac{dy}{dt}$$

Backpropagation

Full backpropagation algorithm:

Let v_1, \dots, v_N be a **topological ordering** of the computation graph (i.e. parents come before children.)

v_N denotes the variable we're trying to compute derivatives of (e.g. loss).

forward pass

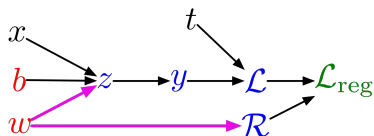
For $i = 1, \dots, N$
Compute v_i as a function of $\text{Pa}(v_i)$

backward pass

$\overline{v_N} = 1$
For $i = N - 1, \dots, 1$
$$\overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$$

Backpropagation

Example: univariate logistic least squares regression



Forward pass:

$$\begin{aligned}z &= wx + b \\y &= \sigma(z) \\\mathcal{L} &= \frac{1}{2}(y - t)^2 \\\mathcal{R} &= \frac{1}{2}w^2 \\\mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda\mathcal{R}\end{aligned}$$

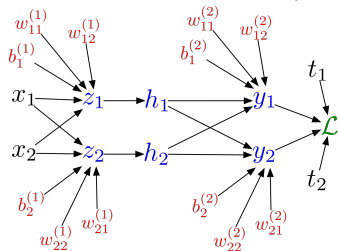
Backward pass:

$$\begin{aligned}\overline{\mathcal{L}_{\text{reg}}} &= 1 \\\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} \\&= \overline{\mathcal{L}_{\text{reg}}} \lambda \\\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} \\&= \overline{\mathcal{L}_{\text{reg}}} \\\overline{y} &= \overline{\mathcal{L}} \frac{d\mathcal{L}}{dy} \\&= \overline{\mathcal{L}}(y - t)\end{aligned}$$

$$\begin{aligned}\overline{z} &= \overline{y} \frac{dy}{dz} \\&= \overline{y} \sigma'(z) \\\overline{w} &= \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{d\mathcal{R}}{dw} \\&= \overline{z} x + \overline{\mathcal{R}} w \\\overline{b} &= \overline{z} \frac{\partial z}{\partial b} \\&= \overline{z}\end{aligned}$$

Backpropagation

Multilayer Perceptron (multiple outputs):



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Backward pass:

$$\bar{\mathcal{L}} = 1$$

$$\bar{y}_k = \bar{\mathcal{L}} (y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \bar{y}_k h_i$$

$$\bar{b}_k^{(2)} = \bar{y}_k$$

$$\bar{h}_i = \sum_k \bar{y}_k w_{ki}^{(2)}$$

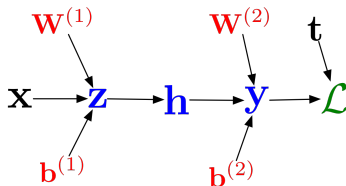
$$\bar{z}_i = \bar{h}_i \sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \bar{z}_i x_j$$

$$\bar{b}_i^{(1)} = \bar{z}_i$$

Vector Form

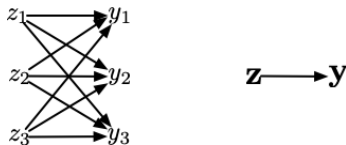
- Computation graphs showing individual units are cumbersome.
- As you might have guessed, we typically draw graphs over the vectorized variables.



- We pass messages back analogous to the ones for scalar-valued nodes.

Vector Form

- Consider this computation graph:



- Backprop rules:

$$\bar{z}_j = \sum_k \bar{y}_k \frac{\partial y_k}{\partial z_j} \qquad \bar{\mathbf{z}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}}^\top \bar{\mathbf{y}},$$

where $\partial \mathbf{y} / \partial \mathbf{z}$ is the **Jacobian matrix**:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n} \end{pmatrix}$$

Examples

- Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \quad \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W} \quad \bar{\mathbf{x}} = \mathbf{W}^\top \bar{\mathbf{z}}$$

- Elementwise operations

$$\mathbf{y} = \exp(\mathbf{z}) \quad \frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix} \quad \bar{\mathbf{z}} = \exp(\mathbf{z}) \circ \bar{\mathbf{y}}$$

- Note: we never explicitly construct the Jacobian. It's usually simpler and more efficient to compute the vector-Jacobian product directly.

Full backpropagation algorithm (vector form):

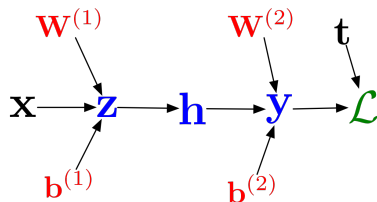
Let $\mathbf{v}_1, \dots, \mathbf{v}_N$ be a [topological ordering](#) of the computation graph (i.e. parents come before children.)

\mathbf{v}_N denotes the variable we're trying to compute derivatives of (e.g. loss).
It's a scalar, which we can treat as a 1-D vector.

forward pass $\left[\begin{array}{l} \text{For } i = 1, \dots, N \\ \text{Compute } \mathbf{v}_i \text{ as a function of } \text{Pa}(\mathbf{v}_i) \end{array} \right.$

backward pass $\left[\begin{array}{l} \overline{\mathbf{v}}_N = 1 \\ \text{For } i = N - 1, \dots, 1 \\ \overline{\mathbf{v}}_i = \sum_{j \in \text{Ch}(\mathbf{v}_i)} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i}^\top \overline{\mathbf{v}}_j \end{array} \right.$

MLP example in vectorized form:



Forward pass:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$\mathbf{h} = \sigma(\mathbf{z})$$

$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \|\mathbf{t} - \mathbf{y}\|^2$$

Backward pass:

$$\bar{\mathcal{L}} = 1$$

$$\bar{\mathbf{y}} = \bar{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$

$$\overline{\mathbf{W}^{(2)}} = \bar{\mathbf{y}}\mathbf{h}^\top$$

$$\overline{\mathbf{b}^{(2)}} = \bar{\mathbf{y}}$$

$$\bar{\mathbf{h}} = \mathbf{W}^{(2)\top} \bar{\mathbf{y}}$$

$$\bar{\mathbf{z}} = \bar{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{\mathbf{W}^{(1)}} = \bar{\mathbf{z}}\mathbf{x}^\top$$

$$\overline{\mathbf{b}^{(1)}} = \bar{\mathbf{z}}$$

Computational Cost

- Computational cost of forward pass: one **add-multiply operation** per weight

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

- Computational cost of backward pass: two add-multiply operations per weight

$$\begin{aligned}\overline{w_{ki}^{(2)}} &= \overline{y_k} h_i \\ \overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)}\end{aligned}$$

- Rule of thumb: the backward pass is about as expensive as two forward passes.
- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.

Backpropagation

- Backprop is used to train the overwhelming majority of neural nets today.
 - Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.
- Despite its practical success, backprop is believed to be neurally implausible.
 - No evidence for biological signals analogous to error derivatives.
 - All the biologically plausible alternatives we know about learn much more slowly (on computers).
 - So how on earth does the brain learn?

Gradient Checking

- We've derived a lot of gradients so far. How do we know if they're correct?
- Recall the definition of the partial derivative:

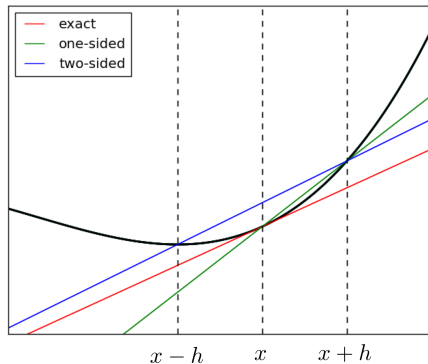
$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i, \dots, x_N)}{h}$$

- Check your derivatives numerically by plugging in a small value of h , e.g. 10^{-10} . This is known as **finite differences**.

Gradient Checking

- Even better: the two-sided definition

$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i - h, \dots, x_N)}{2h}$$



Gradient Checking

- Run gradient checks on small, randomly chosen inputs
- Use double precision floats (not the default for TensorFlow, PyTorch, etc.!)
- Compute the **relative error**:

$$\frac{|a - b|}{|a| + |b|}$$

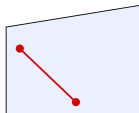
- The relative error should be very small, e.g. 10^{-6}

Gradient Checking

- Gradient checking is really important!
- Learning algorithms often appear to work even if the math is wrong.
- **But:**
 - They might work much better if the derivatives are correct.
 - Wrong derivatives might lead you on a wild goose chase.
- If you implement derivatives by hand, gradient checking is the single most important thing you need to do to get your algorithm to work well.

Convexity

Convex Sets



- A set \mathcal{S} is **convex** if any line segment connecting points in \mathcal{S} lies entirely within \mathcal{S} . Mathematically,

$$\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{S} \implies \lambda \mathbf{x}_1 + (1 - \lambda) \mathbf{x}_2 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1.$$

- A simple inductive argument shows that for $\mathbf{x}_1, \dots, \mathbf{x}_N \in \mathcal{S}$, **weighted averages**, or **convex combinations**, lie within the set:

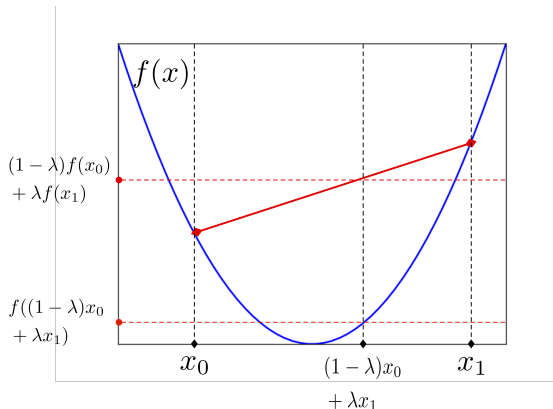
$$\lambda_1 \mathbf{x}_1 + \dots + \lambda_N \mathbf{x}_N \in \mathcal{S} \quad \text{for } \lambda_i > 0, \lambda_1 + \dots + \lambda_N = 1.$$

Convex Functions

- A function f is **convex** if for any $\mathbf{x}_0, \mathbf{x}_1$ in the domain of f ,

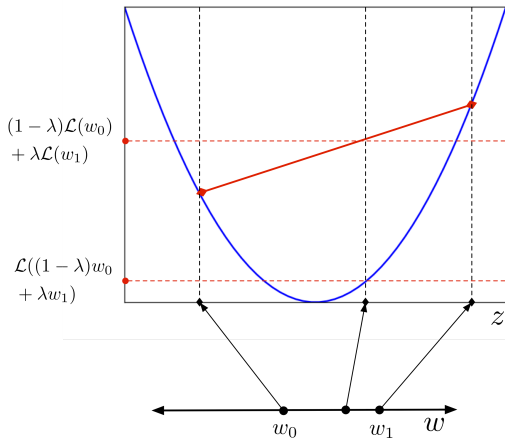
$$f((1 - \lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1) \leq (1 - \lambda)f(\mathbf{x}_0) + \lambda f(\mathbf{x}_1)$$

- Equivalently, the set of points lying above the graph of f is convex.
- Intuitively: the function is bowl-shaped.



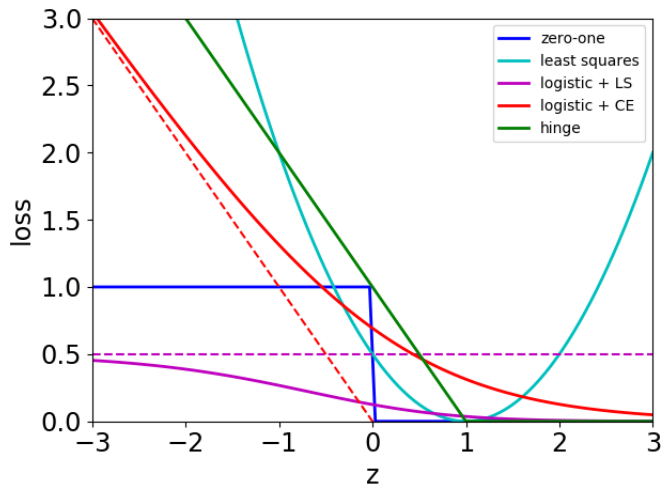
Convex Functions

- We just saw that the least-squares loss function $\frac{1}{2}(y - t)^2$ is convex as a function of y
- For a linear model, $z = \mathbf{w}^\top \mathbf{x} + b$ is a linear function of \mathbf{w} and b . If the loss function is convex as a function of z , then it is convex as a function of \mathbf{w} and b .



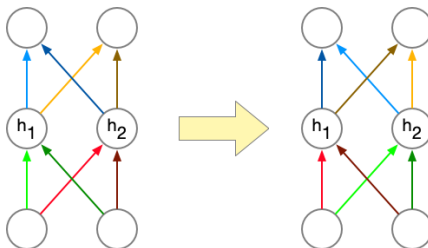
Convex Functions

Which loss functions are convex?



Local Minima

- If a function is convex, it has no **spurious local minima**, i.e. any local minimum is also a global minimum.
- This is very convenient for optimization since if we keep going downhill, we'll eventually reach a global minimum.
- Unfortunately, **training a network with hidden units cannot be convex because of permutation symmetries**.
 - I.e., we can re-order the hidden units in a way that preserves the function computed by the network.



- By definition, if a function \mathcal{J} is convex, then for any set of points $\theta_1, \dots, \theta_N$ in its domain,

$$\mathcal{J}(\lambda_1\theta_1 + \dots + \lambda_N\theta_N) \leq \lambda_1\mathcal{J}(\theta_1) + \dots + \lambda_N\mathcal{J}(\theta_N) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1.$$

- Because of permutation symmetry, there are $K!$ permutations of the hidden units in a given layer which all compute the same function.
- Suppose we average the parameters for all $K!$ permutations. Then we get a degenerate network where all the hidden units are identical.
- If the cost function were convex, this solution would have to be better than the original one, which is ridiculous!
- Hence, training multilayer neural nets is non-convex.

Local Minima (optional, informal)

- Generally, local minima aren't something we worry much about when we train most neural nets.
- It's possible to construct arbitrarily bad local minima even for ordinary classification MLPs. It's poorly understood why these don't arise in practice.
- Intuition pump: if you have enough randomly sampled hidden units, you can approximate any function just by adjusting the output layer.
 - Then it's essentially a regression problem, which is convex.
 - Hence, local optima can probably be fixed by adding more hidden units.
 - Note: this argument hasn't been made rigorous.
- Over the past 5 years or so, CS theorists have made lots of progress proving gradient descent converges to global minima for some non-convex problems, including some specific neural net architectures.