CSC 311: Introduction to Machine Learning

Lecture 4 - Neural Networks

Roger Grosse Chris Maddison Juhan Bae Silviu Pitis

University of Toronto, Fall 2020

Announcements

- Homework 2 is posted! Deadline Oct 14, 23:59.

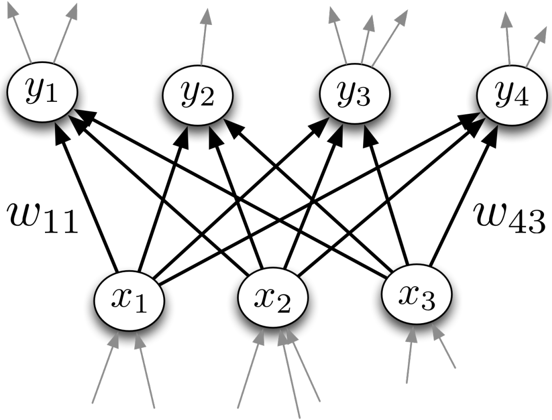
Overview

Design choices so far

- **task:** regression, binary classification, multi-way classification
- **model**: linear, logistic, hard coded feature maps, feed-forward neural network
- **loss**: squared error, 0-1 loss, cross-entropy
- **regularization** L^2, L^p, early stopping
- **optimization**: direct solutions, linear programming, gradient descent (backpropagation)

Neural Networks

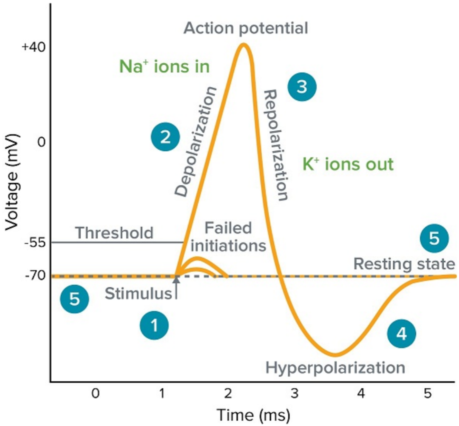
...



...

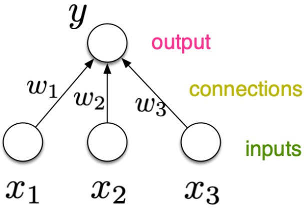
Inspiration: The Brain

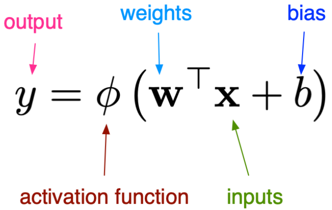
- Neurons receive input signals and accumulate voltage. After some threshold they will fire spiking responses.



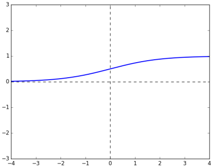
[Pic credit: www.moleculardevices.com]

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





- Compare with logistic regression: $y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

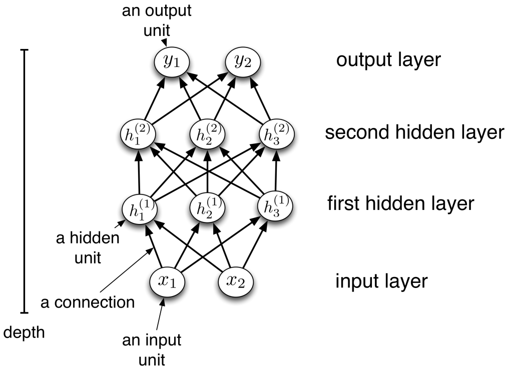


- By throwing together lots of these incredibly simplistic neuron-like processing units, we can do some powerful computations!

Multilayer Perceptrons

Multilayer Perceptrons

- We can connect lots of units together into a **directed acyclic graph**.
- Typically, units are grouped into **layers**.
- This gives a **feed-forward neural network**.



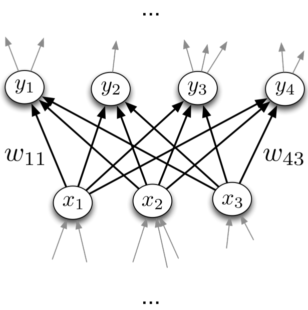
Multilayer Perceptrons

- Each hidden layer i connects N_{i-1} input units to N_i output units.
- In a fully connected layer, all input units are connected to all output units.
- Note: the inputs and outputs for a layer are distinct from the inputs and outputs to the network.

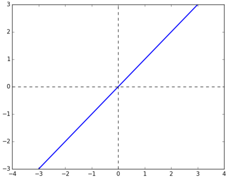
- If we need to compute M outputs from N inputs, we can do so using matrix multiplication. This means we'll be using a $M \times N$ matrix
- The outputs are a function of the input units:

$$\mathbf{y} = f(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

  ϕ is typically applied component-wise.
- A multilayer network consisting of fully connected layers is called a multilayer perceptron.

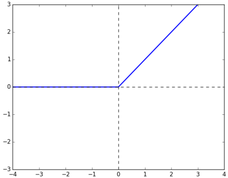


Some activation functions:



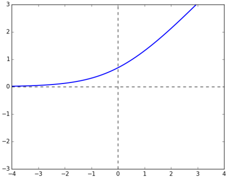
Identity

$$y = z$$



Rectified Linear Unit (ReLU)

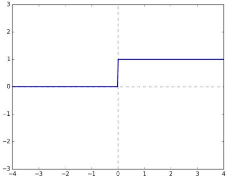
$$y = \max(0, z)$$



Soft ReLU

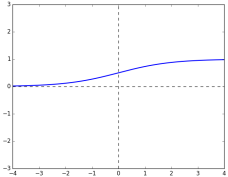
$$y = \log 1 + e^z$$

Some activation functions:



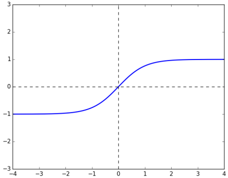
Hard Threshold

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



Logistic

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



Hyperbolic Tangent (tanh)

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

Multilayer Perceptrons

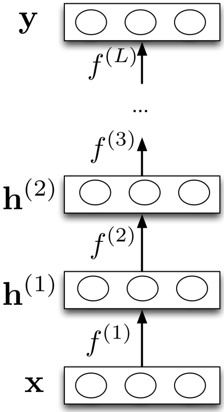
- Each layer computes a function, so the network computes a composition of functions:

$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x}) = \phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)}) = \phi(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$
$$\vdots$$
$$\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)})$$

- Or more simply:

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

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

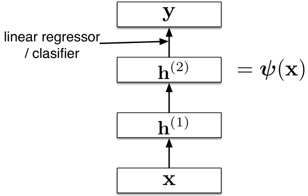


Feature Learning

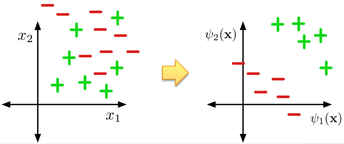
Last layer:

- If task is regression: choose
  $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = (\mathbf{w}^{(L)})^\top \mathbf{h}^{(L-1)} + b^{(L)}$
- If task is binary classification: choose
  $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = \sigma((\mathbf{w}^{(L)})^\top \mathbf{h}^{(L-1)} + b^{(L)})$

So neural nets can be viewed as a way of learning features:

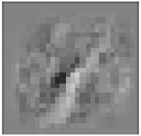


- The goal:



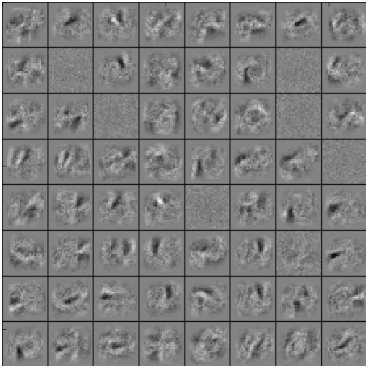
Feature Learning

- 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 $\phi(\mathbf{w}_i^\top \mathbf{x})$. 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:

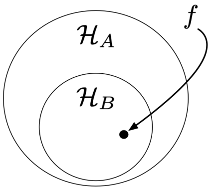


- Unlike hard-coded feature maps (e.g., in polynomial regression), features learned by neural networks adapt to patterns in the data.

Expressivity

- In Lecture 3, we introduced the idea of a hypothesis space $\mathcal{H}$, which is the set of input-output mappings that can be represented by some model. Suppose we are deciding between two models A, B with hypothesis spaces $\mathcal{H}_A, \mathcal{H}_B$.
- If $\mathcal{H}_B \subseteq \mathcal{H}_A$, then A is more expressive than B.

A can represent any function f in $\mathcal{H}_B$.



- Some functions (XOR) can't be represented by linear classifiers. Are deep networks more expressive?

Expressivity—Linear Networks

- Suppose a layer's activation function was the identity, so the layer just computes a affine transformation of the input
  - We call this a linear layer
- Any sequence of *linear* layers can be equivalently represented with a single linear layer.

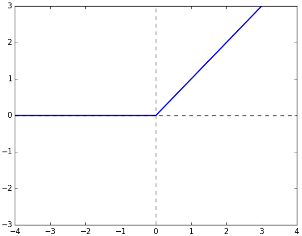
$$\mathbf{y} = \underbrace{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}}_{\triangleq \mathbf{W}'}\mathbf{x}$$

  - Deep linear networks can only represent linear functions.
  - Deep linear networks are no more expressive than linear regression.

Expressive Power—Non-linear Networks

- Multilayer feed-forward neural nets with *nonlinear* activation functions are **universal function approximators**: they can approximate any function arbitrarily well, i.e., for any $f : \mathcal{X} \to \mathcal{T}$ there is a sequence $f_i \in \mathcal{H}$ with $f_i \to f$.

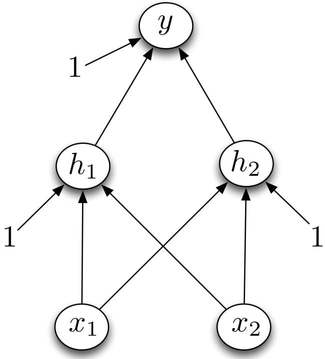
- This has been shown for various activation functions (thresholds, logistic, ReLU, etc.)
  - Even though ReLU is "almost" linear, it's nonlinear enough.



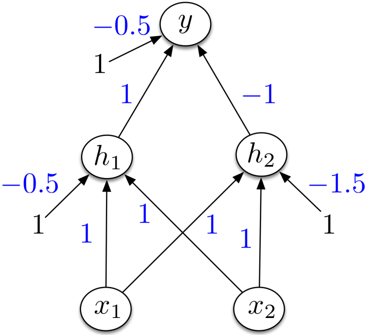
Multilayer Perceptrons

Designing a network to classify XOR:

Assume hard threshold activation function



Multilayer Perceptrons



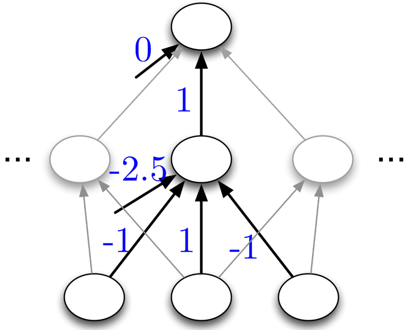
- h_1 computes $\mathbb{I}[x_1 + x_2 - 0.5 > 0]$
  - i.e. x_1 OR x_2
- h_2 computes $\mathbb{I}[x_1 + x_2 - 1.5 > 0]$
  - i.e. x_1 AND x_2
- y computes $\mathbb{I}[h_1 - h_2 - 0.5 > 0] \equiv \mathbb{I}[h_1 + (1 - h_2) - 1.5 > 0]$
  - i.e. h_1 AND (NOT h_2) $= x_1$ XOR x_2

Expressivity

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

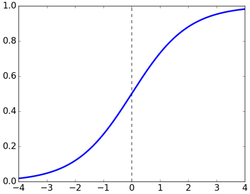
x_1	x_2	x_3	t
	$\vdots$		$\vdots$
-1	-1	1	-1
-1	1	-1	1
-1	1	1	1
	$\vdots$		$\vdots$



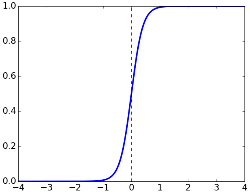
- Only requires one hidden layer, though it needs to be extremely wide.

Expressivity

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



$$y = \sigma(x)$$

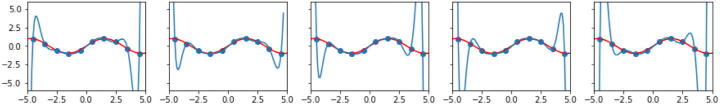


$$y = \sigma(5x)$$

- This is good: logistic units are differentiable, so we can train them with gradient descent.

Expressivity—What is it good for?

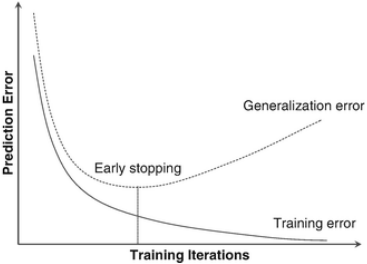
- Universality is not necessarily a golden ticket.
  - You may need a very large network to represent a given function.
  - How can you find the weights that represent a given function?
- Expressivity can be bad: if you can learn any function, overfitting is potentially a serious concern!
  - Recall the polynomial feature mappings from Lecture 2. Expressivity increases with the degree M, eventually allowing multiple perfect fits to the training data.



    This motivated L^2 regularization.
- Do neural networks overfit and how can we regularize them?

Regularization and Overfitting for Neural Networks

- The topic of overfitting (when & how it happens, how to regularize, etc.) for neural networks is not well-understood, even by researchers!
  - In principle, you can always apply L^2 regularization.
  - You will learn more in CSC413.
- A common approach is early stopping, or stopping training early, because overfitting typically increases as training progresses.

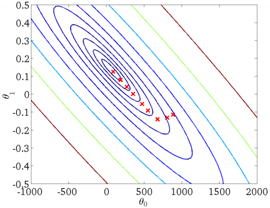


- Unlike L^2 regularization, we don't add an explicit $\mathcal{R}(\boldsymbol{\theta})$ term to our cost.

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 define a loss $\mathcal{L}$ and compute the gradient of the cost $\mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{w}$, which is the vector of partial derivatives.
  - This is the average of $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$ over all the training examples, so in this lecture we focus on computing $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$.

Univariate Chain Rule

- Let's now look at how we compute gradients in neural networks.

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

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t)) = \frac{\mathrm{d}f}{\mathrm{d}x} \frac{\mathrm{d}x}{\mathrm{d}t}.$$

Univariate Chain Rule

Recall: Univariate logistic least squares model

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

Let's compute the loss derivatives $\frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b}$

Univariate Chain Rule

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:

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

Computing the derivatives:

$$
\begin{aligned}
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} &= y - t \\
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z} &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\,\sigma'(z) \\
\frac{\partial\mathcal{L}}{\partial w} &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,x \\
\frac{\partial\mathcal{L}}{\partial b} &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}
\end{aligned}
$$

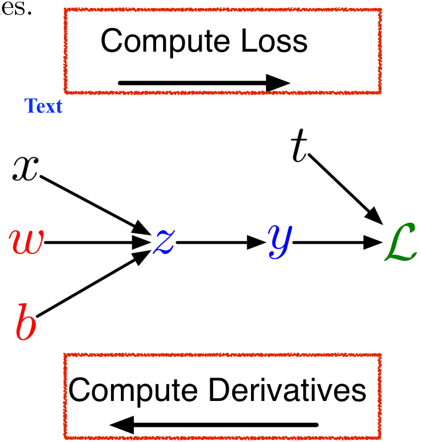
Remember, the goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

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.

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Univariate Chain Rule

A slightly more convenient notation:

- Use $\overline{y}$ to denote the derivative $\mathrm{d}\mathcal{L}/\mathrm{d}y$, sometimes called the **error signal**.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).

Computing the loss:

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

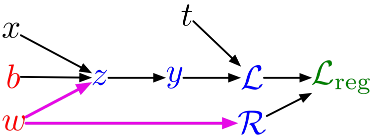
Computing the derivatives:

$$\begin{aligned} \overline{y} &= y - t \\ \overline{z} &= \overline{y}\,\sigma'(z) \\ \overline{w} &= \overline{z}\,x \\ \overline{b} &= \overline{z} \end{aligned}$$

Multivariate Chain Rule

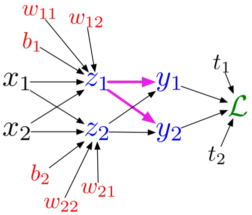
Problem: what if the computation graph has **fan-out** > 1?
This requires the **Multivariate Chain Rule**!

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

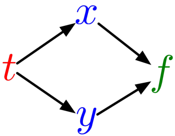


$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$
$$\mathcal{L} = -\sum_k t_k \log y_k$$

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

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



- Example:

$$\begin{aligned} f(x, y) &= y + e^{xy} \\ x(t) &= \cos t \\ y(t) &= t^2 \end{aligned}$$

- Plug in to Chain Rule:

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

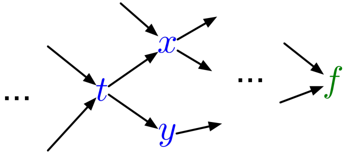
Multivariable Chain Rule

- In the context of backpropagation:

Mathematical expressions
to be evaluated

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed
by our program



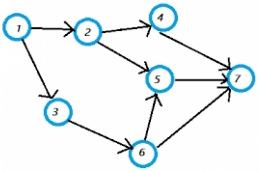
- In our notation:

$$\bar{t} = \overline{x}\,\frac{\mathrm{d}x}{\mathrm{d}t} + \overline{y}\,\frac{\mathrm{d}y}{\mathrm{d}t}$$

Backpropagation

Full backpropagation algorithm:

Let $v_1, \ldots, v_N$ 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 $i = 1, \ldots, N$

Compute v_i as a function of $\mathrm{Pa}(v_i)$

backward pass

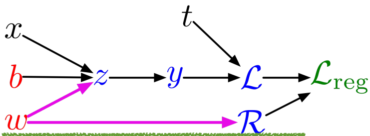
$\overline{v_N} = 1$

For $i = N-1, \ldots, 1$

$\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$

Backpropagation

Example: univariate logistic least squares regression



Forward pass:

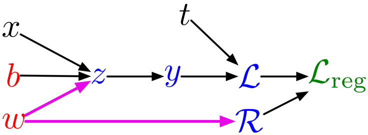
$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2 \\
\mathcal{R} &= \frac{1}{2}w^2 \\
\mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda \mathcal{R}
\end{aligned}
$$

Backward pass:

Text

Backpropagation

Example: univariate logistic least squares regression



Forward pass:

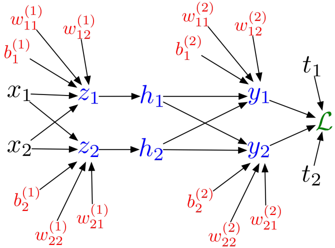
$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2 \\
\mathcal{R} &= \frac{1}{2}w^2 \\
\mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda\mathcal{R}
\end{aligned}
$$

Backward pass:

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

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

Multilayer Perceptron (multiple outputs):



Forward pass:

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

$$h_i = \sigma(z_i)$$

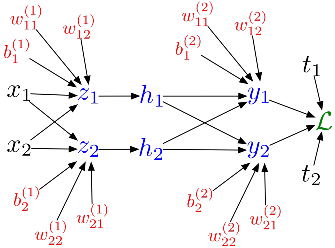
$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

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

Backward pass:

Backpropagation

Multilayer Perceptron (multiple outputs):



Forward pass:

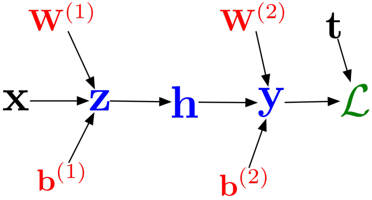
$$
\begin{aligned}
z_i &= \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \\
h_i &= \sigma(z_i) \\
y_k &= \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \\
\mathcal{L} &= \frac{1}{2} \sum_k (y_k - t_k)^2
\end{aligned}
$$

Backward pass:

$$
\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{y_k} &= \overline{\mathcal{L}}\,(y_k - t_k) \\
\overline{w_{ki}^{(2)}} &= \overline{y_k}\,h_i \\
\overline{b_k^{(2)}} &= \overline{y_k} \\
\overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)} \\
\overline{z_i} &= \overline{h_i}\,\sigma'(z_i) \\
\overline{w_{ij}^{(1)}} &= \overline{z_i}\,x_j \\
\overline{b_i^{(1)}} &= \overline{z_i}
\end{aligned}
$$

Backpropagation

In vectorized form:

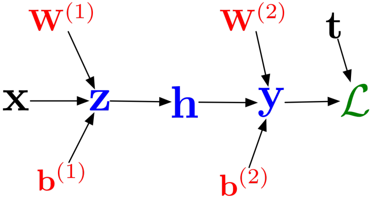


Forward pass:

$$\begin{aligned} \mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{h} &= \sigma(\mathbf{z}) \\ \mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\ \mathcal{L} &= \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2 \end{aligned}$$

Backward pass:

In vectorized form:



Forward pass:

$$
\begin{aligned}
\mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{h} &= \sigma(\mathbf{z}) \\
\mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\
\mathcal{L} &= \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2
\end{aligned}
$$

Backward pass:

$$
\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{\mathbf{y}} &= \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t}) \\
\overline{\mathbf{W}^{(2)}} &= \overline{\mathbf{y}}\mathbf{h}^\top \\
\overline{\mathbf{b}^{(2)}} &= \overline{\mathbf{y}} \\
\overline{\mathbf{h}} &= \mathbf{W}^{(2)\top}\overline{\mathbf{y}} \\
\overline{\mathbf{z}} &= \overline{\mathbf{h}} \circ \sigma'(\mathbf{z}) \\
\overline{\mathbf{W}^{(1)}} &= \overline{\mathbf{z}}\mathbf{x}^\top \\
\overline{\mathbf{b}^{(1)}} &= \overline{\mathbf{z}}
\end{aligned}
$$

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 the algorithm for efficiently computing gradients in neural nets.
- Gradient descent with gradients computed via 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.

Gradient Checking

Gradient Checking

- One way to compute $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$ is numerical. This is useful for checking algorithmically computed gradients, or gradient checking.
- Recall the definition of the partial derivative:

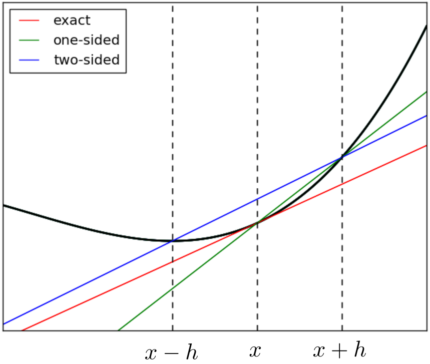
$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) = \lim_{h \to 0} \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i, \dots, x_N)}{h}$$

- We can estimate the gradient numerically by fixing h to a small value, e.g. 10^{-10}, on the right-hand side. This is known as finite differences.

Gradient Checking

- Even better: the two-sided definition

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



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.

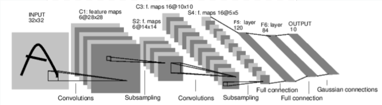
Pytorch, Tensorflow, et al. (Optional)

- If we construct our networks out of a series of “primitive” operations (e.g., add, multiply) with specified routines for computing derivatives, backprop can be done in a completely mechanical, and automatic, way.
- This is called autodifferentiation or just autodiff.
- There are many autodiff libraries (e.g., PyTorch, Tensorflow, Jax, etc.)
- Practically speaking, autodiff automates the backward pass for you — but it’s still important to know how things work under the hood.
- In CSC413, you’ll learn more about how autodiff works and use an autodiff framework to build complex neural networks.

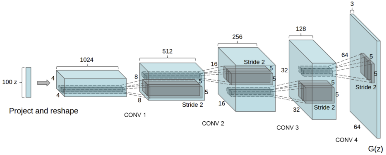
Beyond Feed-forward Neural Networks (Optional)

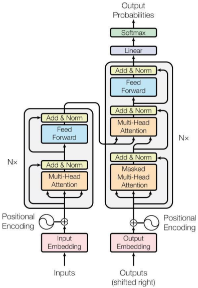
For modern applications (vision, language, games) we use more complicated architectures.

CNN



GAN





Transformer

RNN