CSC 2515: Introduction to Machine Learning

Lecture 5: Neural Networks

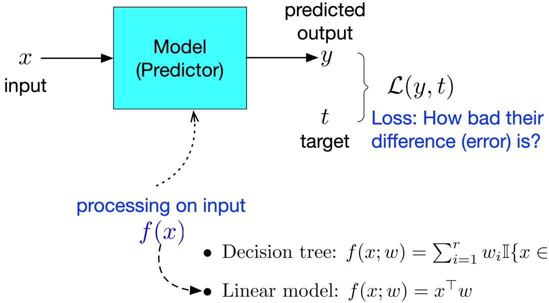
Rahul G. Krishnan[1]

University of Toronto and Vector Institute

[1] Credit for slides goes to many members of the ML Group at the U of T, and beyond, including (recent past): Amir-Massoud Farahmand, Roger Grosse, Murat Erdogdu, Richard Zemel, Juan Felipe Carrasquilla, Emad Andrews, and myself.

Table of Contents

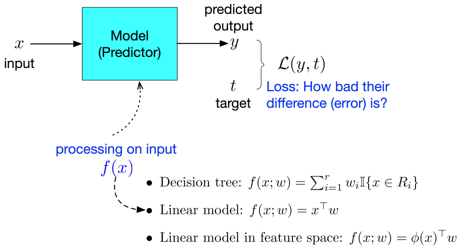
Today



- Decision tree: $f(x; w) = \sum_{i=1}^{r} w_i \mathbb{I}\{x \in R_i\}$
- Linear model: $f(x; w) = x^\top w$
- Linear model in feature space: $f(x; w) = \phi(x)^\top w$

- We have considered a modular framework to ML.
- We considered several loss functions for regression and classifications
- We have "mostly" focused on linear models.

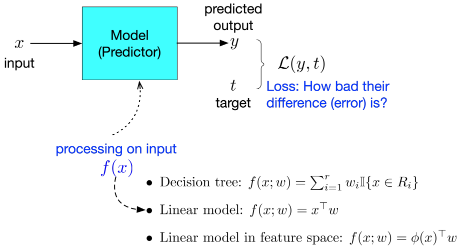
Today



- Decision tree: $f(x;w) = \sum_{i=1}^{r} w_i \mathbb{I}\{x \in R_i\}$
- Linear model: $f(x;w) = x^\top w$
- Linear model in feature space: $f(x;w) = \phi(x)^\top w$

- Feature mapping can make linear models much more powerful.
- Coming up with feature mapping can be challenging.
- Kernel-based approach is a way to partially address it.
- (Artificial) Neural Networks (NN) is a general approach to represent complex models.

Today



x input

Model (Predictor)

predicted output y

t target

$\mathcal{L}(y, t)$

Loss: How bad their difference (error) is?

processing on input $f(x)$

- Decision tree: $f(x; w) = \sum_{i=1}^{r} w_i \mathbb{I}\{x \in R_i\}$
- Linear model: $f(x; w) = x^\top w$
- Linear model in feature space: $f(x; w) = \phi(x)^\top w$

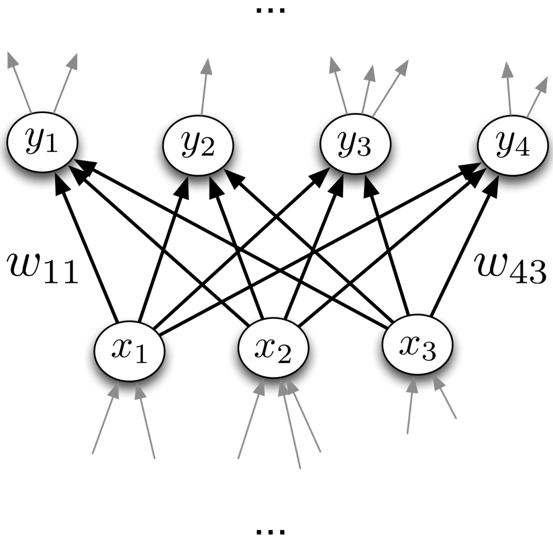
- The predictor can be seen as a computer program that processes the input in order to generate the output. Some programs are simpler, some are more complex.
- Neural networks are one general and flexible way to specify a computer program.
- Different NN architectures correspond to different ways of specifying the overal architecture of the program.

Today

Skills to Learn

- Multi-layer feedforward neural networks
- Backpropagation for training NN

Neural Networks



Inspiration: The Brain

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

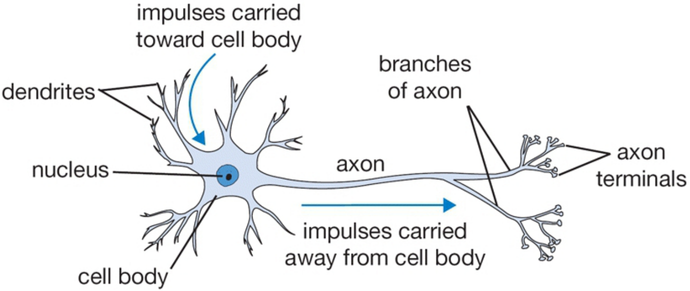


Figure: The basic computational unit of the brain: Neuron

[Image credit: http://cs231n.github.io/neural-networks-1/]

Inspiration: The Brain

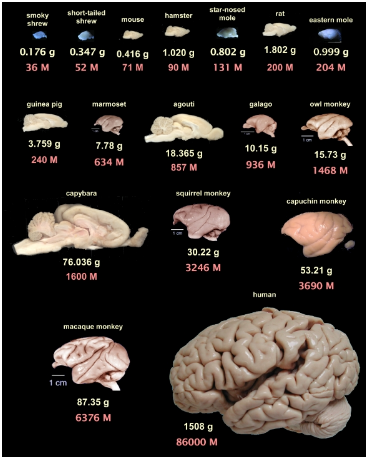
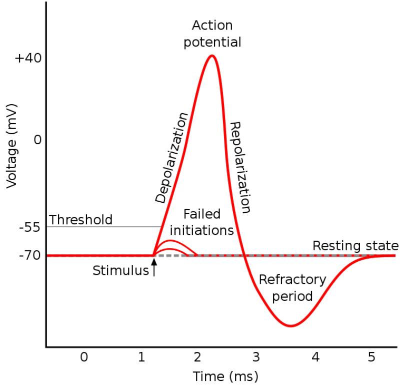


Figure: Brain mass and total number of neurons for the mammalian species.

[Image credit: Suzana Herculano-Houzel, The Human Brain in Numbers: A Linearly Scaled-up Primate Brain, 2009.]

Inspiration: The Brain

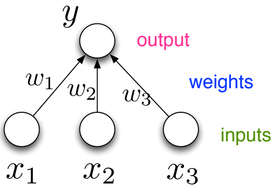
A neuron receives input signals from other neurons and accumulate voltage. If the accumulated voltage passes a threshold, it fires spiking responses.

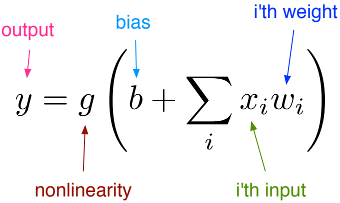


[Image credit: https://en.wikipedia.org/wiki/Action_potential]

Inspiration: The Brain

- For (artificial) neural nets, we use a much simpler model neuron, or unit:





$$y = g\left(b + \sum_i x_i w_i\right)$$

output

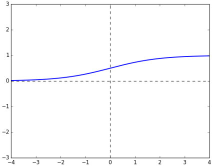
bias

i'th weight

nonlinearity

i'th input

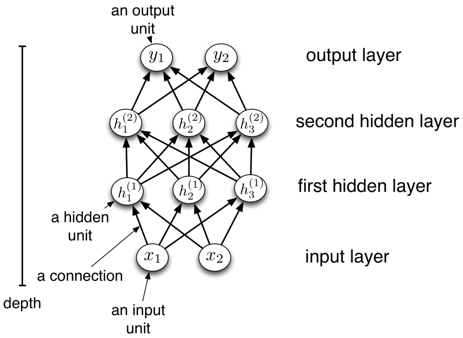
- Compare with logistic activation function used in LR: $y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$



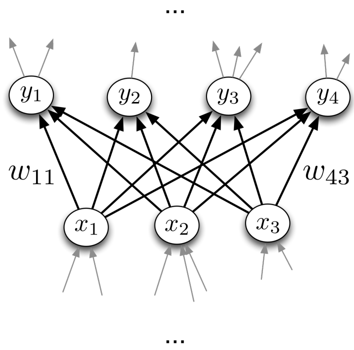
- By throwing together lots of these incredibly simplistic neuron-like

Multilayer Perceptrons (Feedforward Neural Networks)

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

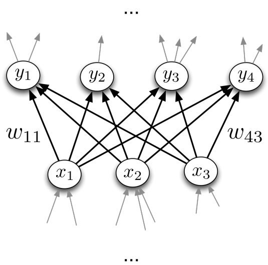


Multilayer Perceptrons (Feedforward Neural Networks)



- Each hidden layer i connects N_{i-1} input units to N_i output units.
- In the simplest case, all input units are connected to all output units. We call this a fully connected layer. We will consider other layer types later.
  - The inputs and outputs for a layer are distinct from the inputs and outputs to the network.

Multilayer Perceptrons (Feedforward Neural Networks)



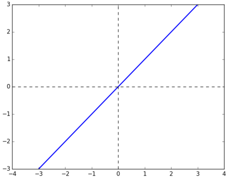
- If we need to compute $M[= N_i]$ outputs from $N = [N_{i-1}]$ inputs, we can do so in parallel using matrix multiplication. This means we will be using a $M \times N$ weight matrix.
- The output units are a function of the input units:

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

- A multilayer network consisting of fully connected layers is called a multilayer perceptron. Despite the name, it has nothing to do with the Perceptron algorithm.

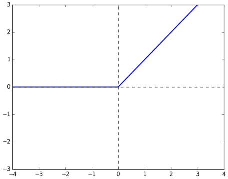
Activation Functions

Some activation functions:



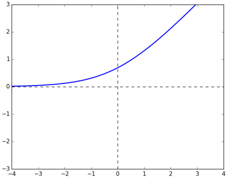
Identity

$$y = z$$



Rectified Linear Unit (ReLU)

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

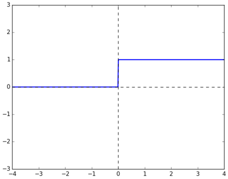


Soft ReLU

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

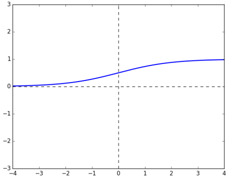
Activation Functions

Some activation functions:



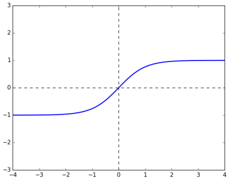
Hard Threshold

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



Logistic

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



Hyperbolic Tangent (tanh)

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

Multilayer Perceptrons (Feedforward Neural Networks)

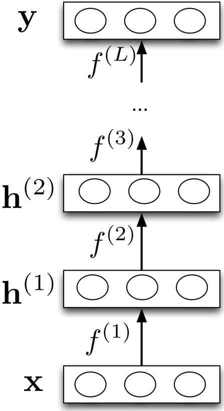
- Each layer computes a function, so the network computes a composition of functions:

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

- Or more compactly:

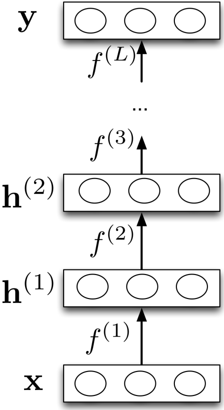
$$\mathbf{y} = f^{(L)} \circ \cdots \circ f^{(1)}(\mathbf{x}).$$

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



Multilayer Perceptrons (Feedforward Neural Networks)

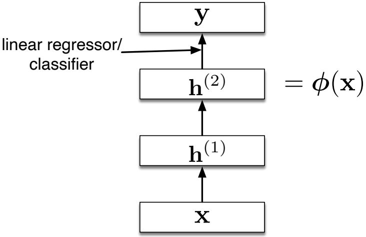
- Q: Write down the equations of a two layer NN (one hidden, one output), two hidden units, ϕ as the activation function of the hidden layer, and a linear one dimensional output layer.



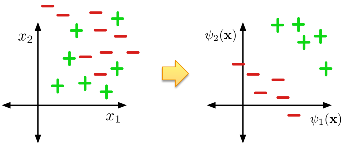
Feature Learning

Last layer:

- If task is regression: choose $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = (\mathbf{w}^{(L)})^T \mathbf{h}^{(L-1)} + b^{(L)}$
- If task is binary classification: choose $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = \sigma((\mathbf{w}^{(L)})^T \mathbf{h}^{(L-1)} + b^{(L)})$
- Neural nets can be viewed as a way of learning features:

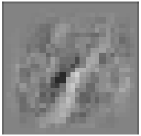


- The goal:



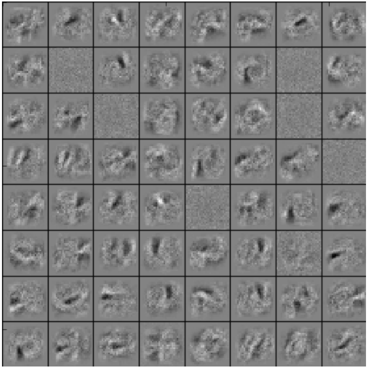
Feature Learning

- Suppose that we are 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 $\phi(\mathbf{w}_i^T \mathbf{x})$. It acts as a feature detector.
- We can visualize $\mathbf{w}$ by reshaping it into an image. Here is 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:



Expressive Power

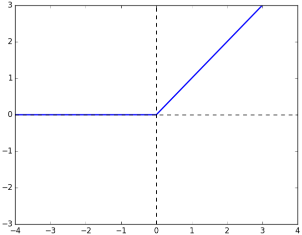
- We have seen that there are some functions that linear classifiers cannot represent. Are deep networks any better?
- Suppose a layer's activation function is the identity function, so the layer just computes an affine transformation of the input
  - We call this a linear layer
- Any sequence of *linear* 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 are no more expressive than linear models.
  - But the dynamics of training can be different than a single layer linear model.
- We need to have nonlinearities to increase expressivity of NN.

Expressive Power

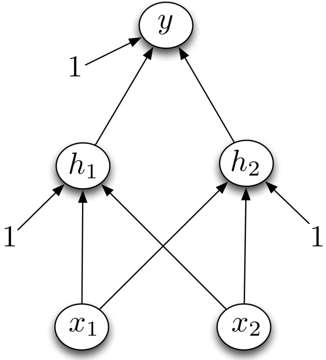
- Multilayer feed-forward neural nets with *nonlinear* activation functions are universal function approximators: they can approximate any function arbitrarily well.
- This has been shown for various activation functions (thresholds, logistic, ReLU, etc.)
  - Even though ReLU is "almost" linear, it is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



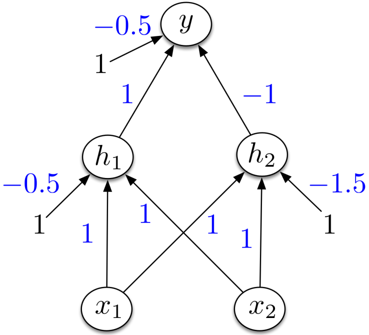
Multilayer Perceptrons

Designing a network to classify XOR:

Assume hard threshold activation function



Multilayer Perceptrons



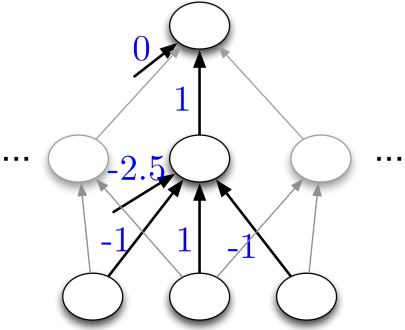
- h_1 computes $\mathbb{I}[x_1 + x_2 - 0.5 > 0]$
  - i.e. x_1 OR x_2
- h_2 computes $\mathbb{I}[x_1 + x_2 - 1.5 > 0]$
  - i.e. x_1 AND x_2
- y computes $\mathbb{I}[h_1 - h_2 - 0.5 > 0] \equiv \mathbb{I}[h_1 + (1 - h_2) - 1.5 > 0]$
  - i.e. h_1 AND (NOT h_2) = x_1 XOR x_2

Expressive Power

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

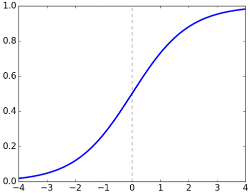
x_1	x_2	x_3	t
	⋮		⋮
-1	-1	1	-1
-1	1	-1	1
-1	1	1	1
	⋮		⋮



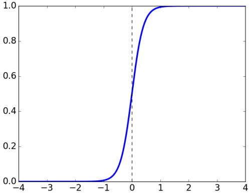
- Only requires one hidden layer, though it needs to be extremely wide.

Expressive Power

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



$$y = \sigma(x)$$



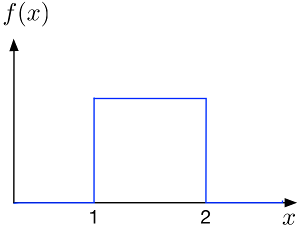
$$y = \sigma(5x)$$

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

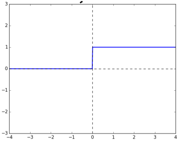
Expressive Power

Let us do some exercises ...

- Q: How can we represent the function that takes value of +1 in $x \in [1, 2]$ and 0 elsewhere using a simple NN with hard threshold activation function?



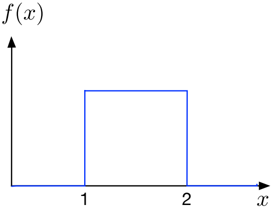
$$f(x) = w_1\phi(x - b_1) + w_2\phi(x - b_2)$$



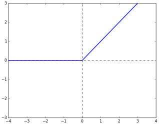
Expressive Power

Let us do some exercises ...

- Q: How can we approximately represent the function that takes value of +1 in $x \in [1, 2]$ and 0 elsewhere using a simple NN with ReLU activation function?



$$f(x) \approx w_1\phi(v_1(x - b_1)) + w_2\phi(v_2(x - b_2)) + ...$$



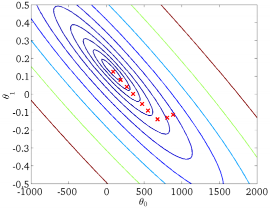
Expressive Power

- Limits of universality
  - You may need to represent an exponentially large network.
  - How can you find the appropriate weights to represent a given function?
  - If you can learn any function, you'll just overfit.
  - We desire a *compact* representation.

Training Neural Networks with Backpropagation

Recap: Gradient Descent

- **Recall:** gradient descent moves in the opposite of the gradient



- 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 have 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

- We have 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:

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

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\
&= ? \qquad \text{(Exercise!)}
\end{aligned}$$

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:

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

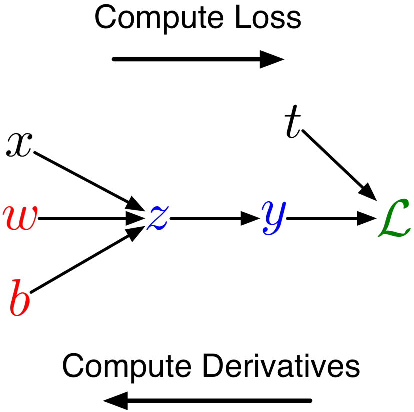
Remember: The goal is not to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Univariate Chain Rule

- We can show 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:

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



Compute Loss

Compute Derivatives

Univariate Chain Rule

A slightly more convenient notation:

- Use $\overline{y}$ to denote the derivative of the loss w.r.t. y (i.e., $\mathrm{d}\mathcal{L}/\mathrm{d}y$), sometimes called the error signal.
  - This is not a standard notation.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).

Computing the loss:

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

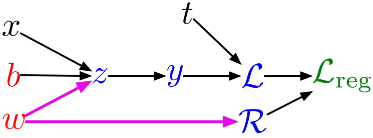
Computing the derivatives:

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

Multivariate Chain Rule

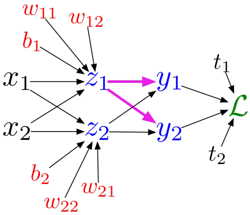
Problem: what if the computation graph has fan-out > 1?
This requires the Multivariate Chain Rule!

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 classifier with the cross-entropy loss

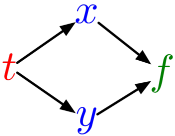


$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$
$$\mathcal{L} = -\sum_k t_k \log y_k$$

Multivariate Chain Rule

- Suppose that 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:

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

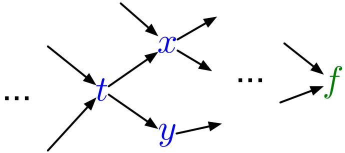
Multivariate Chain Rule

- 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



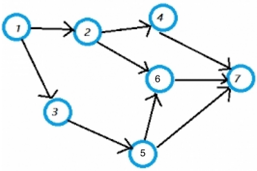
- In our notation:

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

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

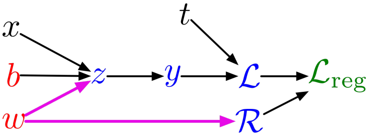
```
For $i = 1, \ldots, N$
    Compute $v_i$ as a function of $\mathrm{Pa}(v_i)$
```

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:

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

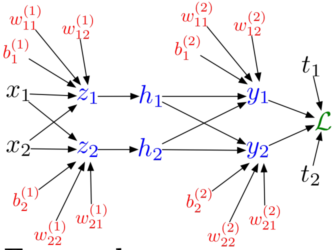
Backward pass:

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

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

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:

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

$$\overline{y_k} = \overline{\mathcal{L}}\,(y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k}\, h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

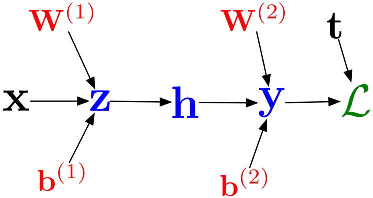
$$\overline{z_i} = \overline{h_i}\, \sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \overline{z_i}\, x_j$$

$$\overline{b_i^{(1)}} = \overline{z_i}$$

Backpropagation

In vectorized form:



Forward pass:

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

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 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.

Conclusion

- Multi-layer feedforward NN addressed the feature learning problem
- Backpropagation as a method to learn NN