STA 414/2104: Statistical Methods of Machine Learning II

Week 6: Neural Networks and Optimization

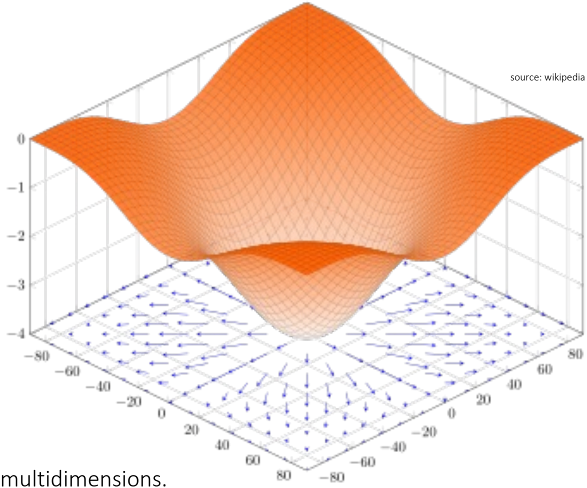
Murat A. Erdogdu

University of Toronto

Outline

1. Basics of Optimization in ML
2. Limits of Linear Classification
3. Introducing Neural Networks
4. Backpropagation

Gradients



source: wikipedia

$$f(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}$$

$$\nabla f(\mathbf{w}) = \begin{bmatrix} \partial f(\mathbf{w})/\partial w_1 \\ \partial f(\mathbf{w})/\partial w_2 \\ \vdots \\ \partial f(\mathbf{w})/\partial w_d \end{bmatrix}$$

- Generalization of derivatives in multidimensions.
- It is a vector representing the slope.
- The direction of the gradient points to the greatest rate of increase of the function.
- Its magnitude is the slope of the graph in its direction.

What is optimization?

- Typical setup (in machine learning, other areas):
  - Formulate a problem
  - Design a solution (usually a model)
  - Use some quantitative measure to determine how good the solution is.

- E.g., classification:
  - Create a system to classify images
  - Model is some classifier, like logistic regression
  - Quantitative measure is misclassification error (lower is better in this case)

- In almost all cases, you end up with a loss minimization problem of the form

$$\text{minimize}_{\mathbf{w}} E(\mathbf{w})$$

- Ex: least squares

$$\text{minimize} \quad E(\mathbf{w}) \quad = \quad \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n^T\mathbf{w} - t_n)^2$$

Error minimization

- Ultimately, training a machine learning model always reduces to solving an optimization problem

$$\text{minimize}_{\mathbf{w}} E(\mathbf{w})$$

Equivalently, we are interested in finding $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} E(\mathbf{w})$

by using an optimization method.

- Standard approach is **Gradient descent** $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t)$

where $\eta \in (0, 1]$ is the step size (or learning rate).

- For the least squares, $\text{minimize}\, E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (\mathbf{x}_n^T \mathbf{w} - t_n)^2$
- we have

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} \mathbf{x}_n (\mathbf{x}_n^T \mathbf{w} - t_n)$$

- We choose an initial point $\mathbf{w}^0$, and perform the following iterations

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t)$$

Gradient descent derivation

- Suppose we are at **w** and we want to pick a direction **d** such that E(**w** + η**d**) is smaller than E(**w**) for a step size η.
- The first-order Taylor series approximation of E(w+d) around w is:

$$E(\mathbf{w} + \eta \mathbf{d}) \approx E(\mathbf{w}) + \eta \nabla E(\mathbf{w})^\top \mathbf{d}$$

- **d** should be in the negative direction of $\nabla E(\mathbf{w})$

- This approximation gets better as η gets smaller since as we zoom in on a differentiable function it will look more and more linear.

Gradient descent derivation

- We need to find a direction for d that minimizes

$$E(\mathbf{w} + \eta \mathbf{d}) \approx E(\mathbf{w}) + \eta \nabla E(\mathbf{w})^\top \mathbf{d}$$

- The best direction is $-\nabla E(\mathbf{w})$

This doesn't affect the problem, but it is common in practice to normalize with N

- For the least squares, $\text{minimize} \quad E(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} (t_n - \mathbf{x}_n^T \mathbf{w})^2$
- we have

$$\nabla E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n (\mathbf{x}_n^T \mathbf{w} - t_n)$$

- We choose an initial point $\mathbf{w}^0$, and do the following iterations

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t)$$

How to choose the step size?

- Step size is referred to as learning rate in machine learning.
- It should be in the interval (0,1).
- The sequence of step sizes is referred to as the learning rate schedule.
- One simple strategy: start with a big η and progressively make it smaller by e.g., halving it until the function decreases.
- There are more formal ways of choosing the step size. But in practice, they are not used for computational reasons.

When does the GD converged?

- When $\|\nabla E(\mathbf{w})\| = 0$

- This is never possible in practice. So we stop iterations if gradient is smaller than a threshold.

- If the function is convex then we have reached a global minimum.

- If the function is not convex, what did we obtain?

- Probably a local minimum or a saddle.

Stochastic Gradient Descent

- Any iteration of a gradient descent method requires that we sum over the entire dataset to compute the gradient.
- SGD idea: at each iteration, sub-sample a small amount of data (even just 1 point can work) and use that to estimate the gradient.
- Each update is noisy, but very fast!
- This is the basis of optimizing ML algorithms with huge datasets (e.g., recent deep learning).
- Computing gradients using the full dataset is called batch learning, using subsets of data is called mini-batch learning.

Stochastic Gradient Descent

- In most cases, the minimization is an average over data points:

$$\text{minimize} E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} L(t_n, y(\mathbf{x}_n, \mathbf{w}))$$

Hard to compute when N is large

Recall that we can write the negative log-likelihood in the above form.

Gradient:

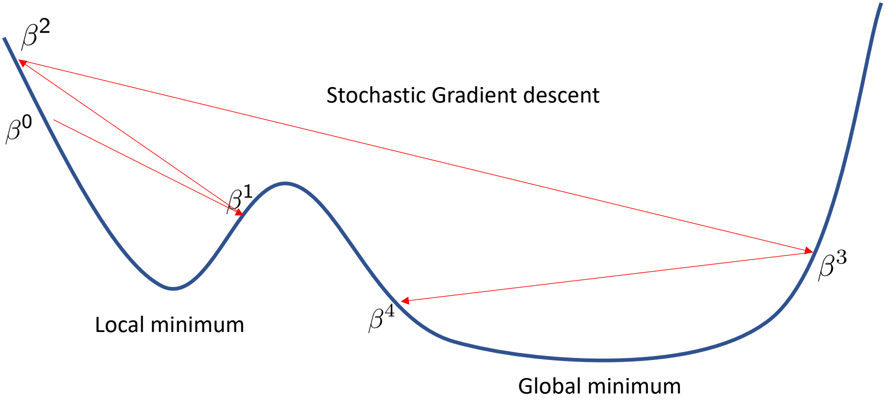
$$\nabla E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \nabla L(t_n, y(\mathbf{x}_n, \mathbf{w}))$$

At each iteration, sub-sample a small amount of data and use that to estimate the gradient.

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta \frac{1}{|S|}\sum_{n\in S} \nabla L(t_n, y(\mathbf{x}_n, \mathbf{w}))$$

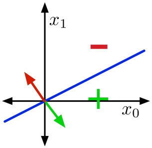
Here, |S| denotes the number of elements in the set S .
Standard SGD has |S|= 1, i.e., randomly samples an index
and takes a step based on that sample. |S|>1 is called mini-batch SGD.

Non-convex optimization



Stochastic methods have higher chance to escape "bad" minima, and converge to favorable regions.

Visualizing NOT

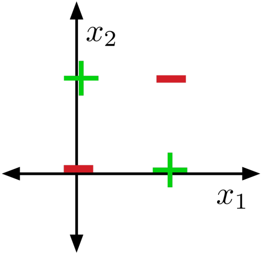


x_0	x_1	t
1	0	1
1	1	0

- Data is linearly separable if a linear decision rule can perfectly separate the training examples.

XOR is Not Linearly Separable

Some datasets are not linearly separable, e.g. **XOR**.



x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

Classifying XOR Using Feature Maps

Sometimes, we can overcome this limitation using feature maps, e.g., for **XOR**.

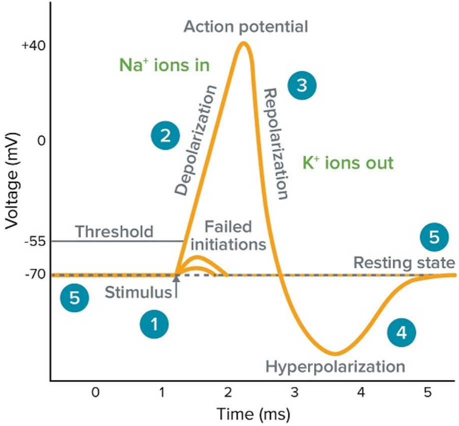
$$\boldsymbol{\psi}(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}$$

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!)
- Designing feature maps can be hard. Can we learn them?

Neurons in the Brain

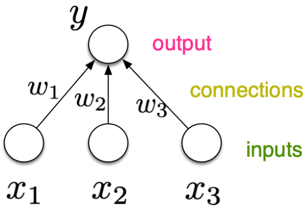
Neurons receive input signals and accumulate voltage.
After some threshold, they will fire spiking responses.

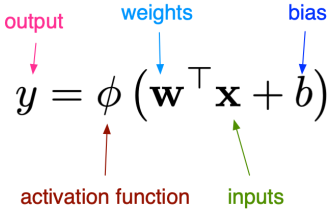


[Pic credit: www.moleculardevices.com]

A Simpler Neuron

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

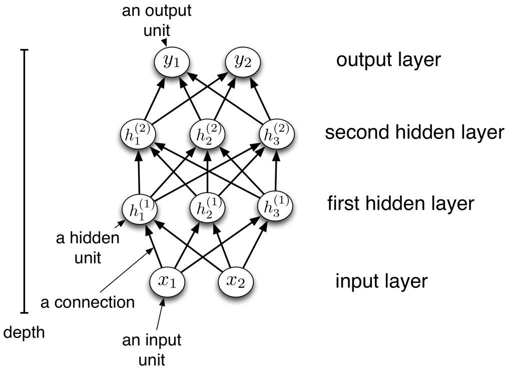




- Same as logistic regression: $y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
- By throwing together lots of these simple neuron-like processing units, we can do some powerful computations!

A Feed-Forward Neural Network

- A directed acyclic graph
- Units are grouped into layers

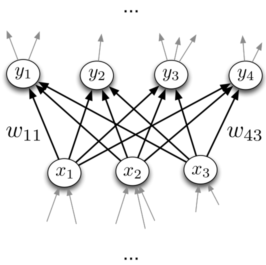


Multilayer Perceptrons

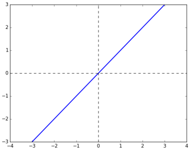
- A multi-layer network consists of fully connected layers.
- In a fully connected layer, all input units are connected to all output units.
- The outputs are a function of the input units:

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

ϕ is applied component-wise.

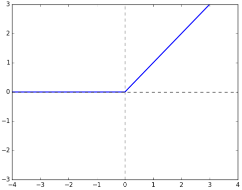


Some Activation Functions



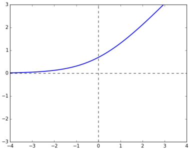
Identity

$$y = z$$



Rectified Linear Unit (ReLU)

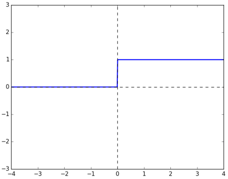
$$y = \max(0, z)$$



Soft ReLU

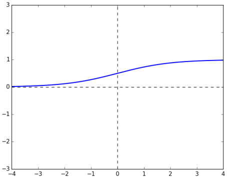
$$y = \log 1 + e^z$$

More Activation Functions



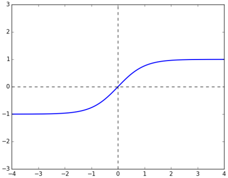
Hard Threshold

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



Logistic

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



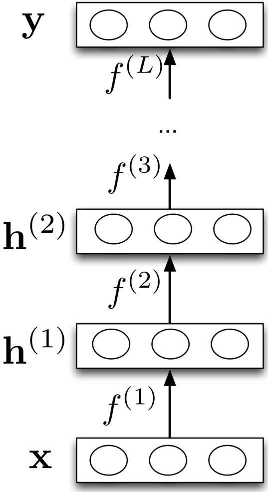
Hyperbolic Tangent (tanh)

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

Computation in Each Layer

Each layer computes a function.

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



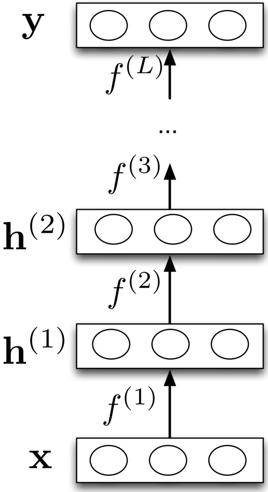
- If task is regression: choose $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = (\mathbf{w}^{(L)})^\top \mathbf{h}^{(L-1)} + b^{(L)}$
- If task is binary classification: choose $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = \sigma((\mathbf{w}^{(L)})^\top \mathbf{h}^{(L-1)} + b^{(L)})$

A Composition of Functions

The network computes
a composition of functions.

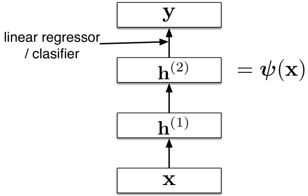
$$\mathbf{y} = f^{(L)} \circ \cdots \circ f^{(1)}(\mathbf{x}).$$

Modularity: We can implement each layer's computations as a black box.

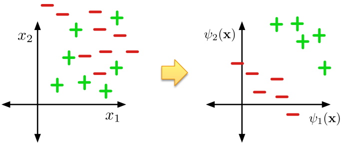


Feature Learning

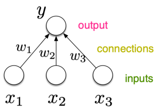
Neural nets can be viewed as a way of learning features:

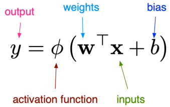


The goal:

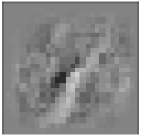


Feature Learning



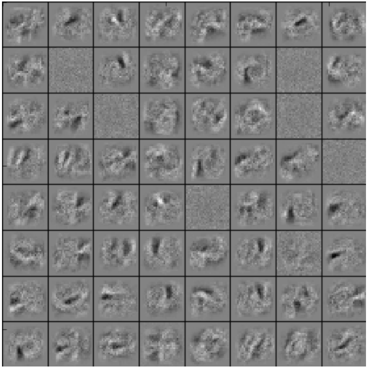


- Suppose we're trying to classify images of handwritten digits.
- Each image is represented as a vector of $28 \times 28 = 784$ pixel values.
- Each hidden unit in the first layer acts as a **feature detector**.
- We can visualize $\mathbf{w}$ by reshaping it into an image.
  Below is an example that responds to a diagonal stroke.



Features for Classifying Handwritten Digits

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.

Expressive Power of Linear Networks

- Consider a linear layer: the activation function was the identity. The layer just computes an affine transformation of the input.
- Any sequence of linear layers is equivalent to a single linear layer.

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

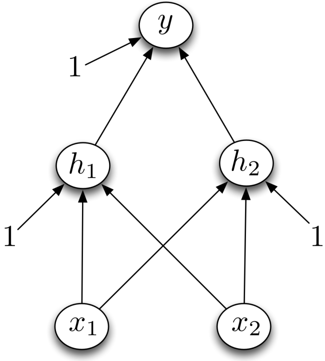
- Deep linear networks can only represent linear functions
— no more expressive than linear regression.

Expressive Power of Non-linear Networks

- Multi-layer feed-forward neural networks
  with non-linear activation functions
- **Universal Function Approximators**:
  They can approximate any function arbitrarily well.
- True for various activation functions
  (e.g. thresholds, logistic, ReLU, etc.)

Designing a Network to Classify XOR

Assume a hard threshold activation function.



Designing a Network to Classify XOR

h_1 is computed as $x_1 \vee x_2$

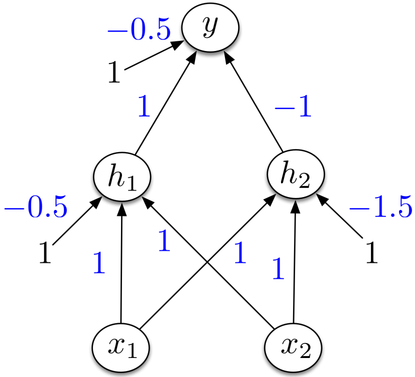
$$h_1 = \mathbb{I}[x_1 + x_2 - 0.5 > 0]$$

h_2 is computed as $x_1 \wedge x_2$

$$h_2 = \mathbb{I}[x_1 + x_2 - 1.5 > 0]$$

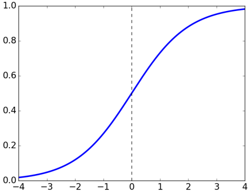
y computes

$$\begin{aligned} y =& \mathbb{I}[h_1 - h_2 - 0.5 > 0] \\ =& x_1 \text{ XOR } x_2 \end{aligned}$$

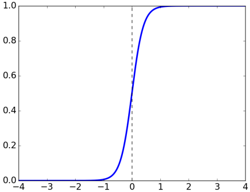


Expressivity of the Logistic Activation Function

- What about the logistic activation function?
- Approximate a hard threshold by scaling up w and b.



$$y = \sigma(x)$$



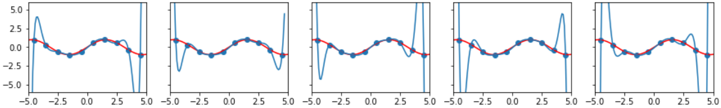
$$y = \sigma(5x)$$

- Logistic units are differentiable, so we can learn weights with gradient descent.

What is Expressivity Good For?

- May need a very large network to represent a function.
- Non-trivial to learn the weights that represent a function.
- If you can learn any function, over-fitting is potentially a serious concern!

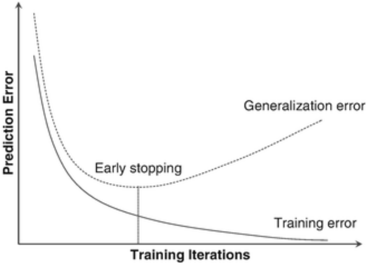
  For the polynomial feature mappings, expressivity increases with the degree M, eventually allowing multiple perfect fits to the training data. This motivated L^2 regularization.



- Do neural networks over-fit and how can we regularize them?

Regularization and Over-fitting for Neural Networks

- The topic of over-fitting (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.
- A common approach is early stopping, or stopping training early, because over-fitting typically increases as training progresses.



Learning Weights in a Neural Network

- Goal is to learn weights in a multi-layer neural network using gradient descent.
- Weight space for a multi-layer neural net: one set of weights for each unit in every layer of the network
- Define a loss $\mathcal{L}$ and compute the gradient of the cost

$$\nabla \mathcal{J}(\mathbf{w}) = \mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{w}$$

  the average loss over all the training examples.
- Let's look at how we can calculate $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$.

Example: Two-Layer Neural Network

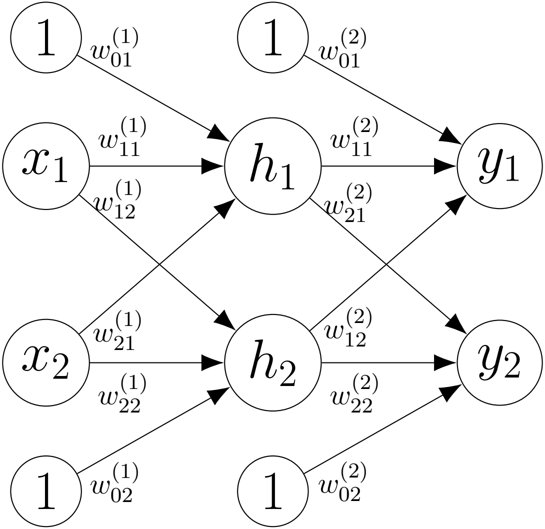


Figure: Two-Layer Neural Network

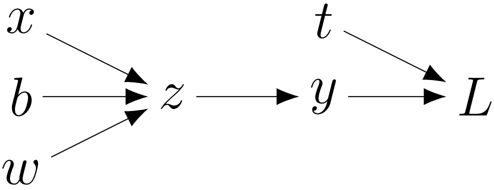
Computations for Two-Layer Neural Network

A neural network computes a composition of functions.

$$
\begin{aligned}
z_1^{(1)} &= w_{01}^{(1)} \cdot 1 + w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2 \\
h_1 &= \sigma(z_1) \\
z_1^{(2)} &= w_{01}^{(2)} \cdot 1 + w_{11}^{(2)} \cdot h_1 + w_{21}^{(2)} \cdot h_2 \\
y_1 &= z_1 \\
z_2^{(1)} &= \\
h_2 &= \\
z_2^{(2)} &= \\
y_2 &= \\
L &= \frac{1}{2}\left((y_1 - t_1)^2 + (y_2 - t_2)^2\right)
\end{aligned}
$$

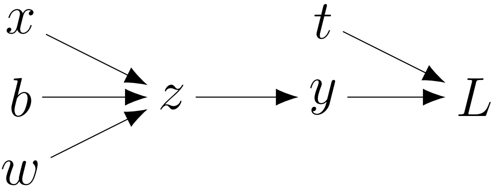
Simplified Example: Logistic Least Squares

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



Computation Graph

- The nodes represent the inputs and computed quantities.
- The edges represent which nodes are computed directly as a function of which other nodes.



Uni-variate Chain Rule

Let $z = f(y)$ and $y = g(x)$ be uni-variate functions.
Then $z = f(g(x))$.

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\,\frac{\mathrm{d}y}{\mathrm{d}x}$$

Logistic Least Squares: Gradient for w

Computing the loss:

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

Computing the gradient for w:

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

Logistic Least Squares: Gradient for b

Computing the loss:

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

Computing the gradient for b:

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

Logistic Least Squares: Gradient for b

Computing the loss:

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

Computing the gradient for b:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial y} \ \frac{\partial y}{\partial b} \\ &= \frac{\partial \mathcal{L}}{\partial y} \ \frac{\partial y}{\partial z} \ \frac{\partial z}{\partial b} \\ &= (y - t) \ \sigma'(z) \ 1 \\ &= (\sigma(wx + b) - t)\sigma'(wx + b)1 \end{aligned}$$

Comparing Gradient Computations for w and b

Computing the loss:

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

Computing the gradient for w:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w} = (y - t)\ \sigma'(z)\ x$$

Computing the gradient for b:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial b} = (y - t)\ \sigma'(z)\ 1$$

Structured Way of Computing Gradients

Computing the loss:

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

Computing the gradients:

$$\frac{\partial \mathcal{L}}{\partial y} = (y - t)$$
$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial y}\, \sigma'(z)$$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\, x \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\, 1$$

Error Signal Notation

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

Computing the loss:

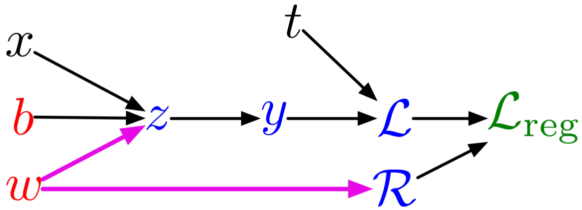
$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\overline{y} = (y - t)$$
$$\overline{z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\,x \qquad \overline{b} = \overline{z}$$

Computation Graph has a Fan-Out > 1

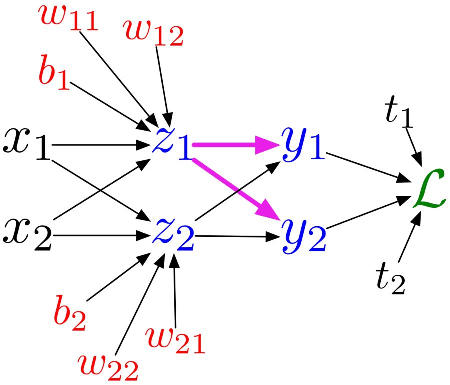
L_2-Regularized Regression



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

Computation Graph has a Fan-Out > 1

Softmax Regression



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

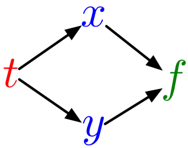
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$

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

Multi-variate Chain Rule

Suppose we have functions $f(x, y)$, $x(t)$, and $y(t)$.

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



Example:

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

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

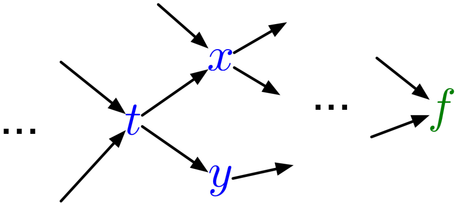
Multi-variate Chain Rule

In the context of back-propagation:

Mathematical expressions to be evaluated

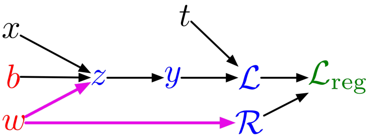
$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed by our program



In our notation:

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



Forward pass:

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

Backward pass:

$$\overline{\mathcal{R}} = \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} = \lambda$$
$$\overline{\mathcal{L}} = \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} = 1$$
$$\overline{y} = \overline{\mathcal{L}}\,\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = \overline{\mathcal{L}}\,(y-t)$$
$$\overline{z} = \overline{y}\,\frac{\mathrm{d}y}{\mathrm{d}z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\,\frac{\partial z}{\partial w} + \overline{\mathcal{R}}\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} = \overline{z}\,x + \overline{\mathcal{R}}\,w$$
$$\overline{b} = \overline{z}\,\frac{\partial z}{\partial b} = \overline{z}$$

Full Backpropagation Algorithm:

Let $v_1, \ldots, v_N$ be an ordering of the computation graph where parents come before children.

v_N denotes the variable for which we're trying to compute gradients.

- forward pass:

$$\text{For } i = 1, \ldots, N,$$
$$\text{Compute } v_i \text{ as a function of Parents}(v_i).$$

- backward pass:

$$\text{For } i = N - 1, \ldots, 1,$$
$$\bar{v_i} = \sum_{j \in \text{Children}(v_i)} \bar{v_j} \frac{\partial v_j}{\partial v_i}$$

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

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

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

- One backward pass is as expensive as two forward passes.

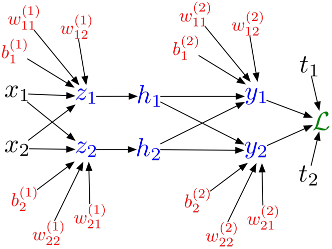
Backpropagation

- 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.

Auto-Differentiation

- Autodifferentiation performs backprop in a completely mechanical and automatic way.
- Many autodiff libraries: PyTorch, Tensorflow, Jax, etc.
- Although autodiff automates the backward pass for you, it's still important to know how things work under the hood.
- In the tutorial, you will use an autodiff framework to build complex neural networks.

Backpropagation for Two-Layer Neural Network



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:

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

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

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

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

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

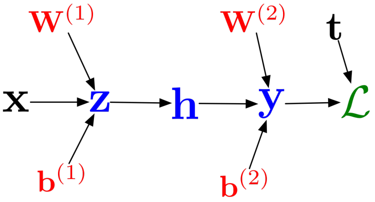
$$\overline{z_i} = \overline{h_i}\,\sigma'(z_i)$$

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

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

Backpropagation for Two-Layer Neural Network

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:

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

Conclusion

- Introduced Neural Networks
- Discuss their expressive power.
  - Can approximate any function.
- Introduced backpropagation.
  - We also work out the updates for a two-layer neural network.