CSC 561: Neural Networks and Deep Learning

Multilayer Perceptron

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

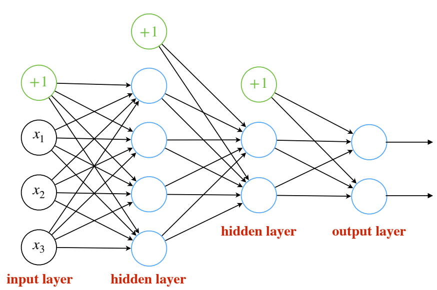
Spring 2024

THINK BIG WE DO℠

Multilayer perceptron

Multilayer perceptron

- A layered network
  - each layer of neurons gets inputs from earlier layer
  - each layer of neurons outputs values to later layers
  - consists of an **input layer**, zero or more **hidden layers**, and an **output layer**



- Input layer receives the input data
- Hidden and output layer(s) perform **computation and learning**

MLP layers

- Input layer
  - fixed-length vector of numbers
  - e.g., pixel values, speech features, embeddings representing text, etc.
- Hidden layer
  - overcoming limitations of linear models — incorporate one or more hidden layers with **nonlinear** activations
- Output layer
  - scalar output => single neuron
  - vector output => many neurons
  - binary classification
    - can use a single neuron with a logistic activation; or
    - can use two neurons with a softmax activation (requires one-hot encoding)

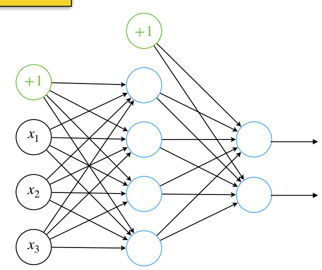
Example

- Asume a minibatch $X \in \mathbb{R}^{n \times 3}$
  - hidden layer weights $W_h \in \mathbb{R}^{4 \times 3}$ and bias $b_h \in \mathbb{R}^4$
  - output layer weights $W_o \in \mathbb{R}^{2 \times 4}$ and bias $b_o \in \mathbb{R}^2$
- Network without nonlinear activations
  - $h(X) = (XW_h^T + b_h)W_o^T + b_o = XW_h^T W_o^T + b_h W_o^T + b_o$

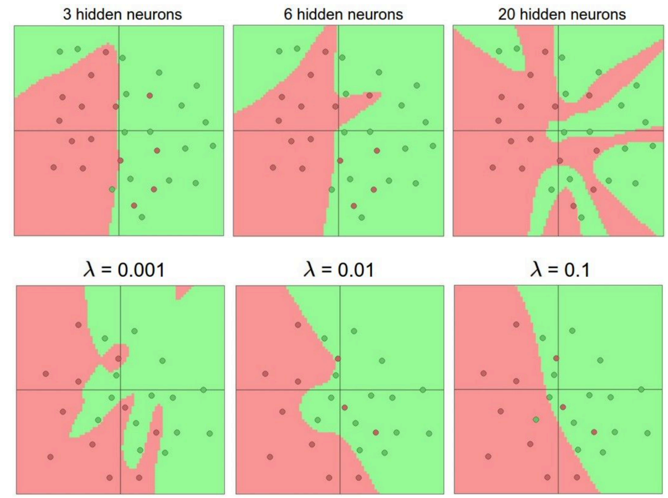
still linear!

- Adding nonlinearities
  - $h(X) = \sigma_o(\sigma_h(XW_h^T + b_h)W_o^T + b_o)$

To build more general MLPs, we can continue stacking such hidden layers, yielding more expressive models



Overfitting / Regularization



https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Training neural nets

- Define the network architecture
  - e.g., $h(X) = \sigma_o(\sigma_h(XW_h^T + b_h)W_o^T + b_o)$
- Define a loss function to "compare" the outputs of the network and the desired targets
  - e.g., cross-entropy loss
- Derive the **gradient** $\nabla_{\mathbf{w}} J(\mathbf{w})$
  - partial derivatives of the loss function with respect to all parameters (weights and biases)
  - **note:** manually deriving the gradient on paper is **not feasible** for complex model — even if possible, minor changes require significant work!
- Use gradient descent to minimize the empirical loss

Numpy code for a 2-layer MLP

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

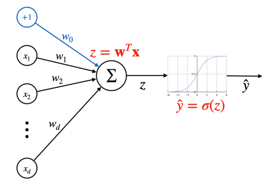
Calculate the analytical gradients

Gradient descent

Stanford University CS231n: Deep Learning for Computer Vision

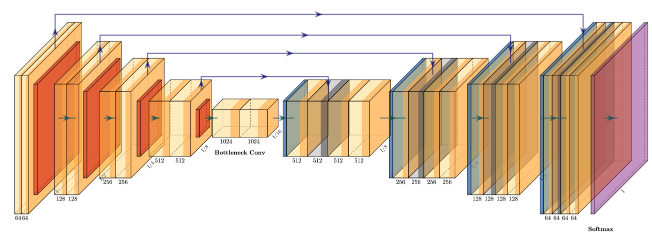
Partial derivatives

Chain rule to the rescue



$$\frac{d\hat{y}}{dz} = \sigma'(z)$$

$$\frac{d\hat{y}}{dw_i} = \frac{d\hat{y}}{dz}\frac{dz}{dw_i} = \sigma'(z)x_i$$



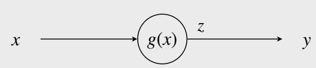
Basic rules

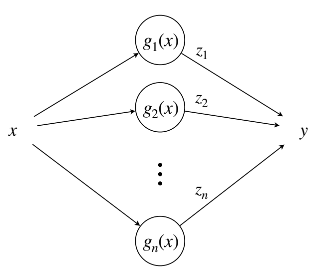
$y = f(x)$ $\quad \frac{dy}{dx}$

$y = f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n)$ $\quad \left[\frac{dy}{dx_1}, \frac{dy}{dx_2}, \ldots, \frac{dy}{dx_n}\right]$

chain rule

$y = f\left(g(x)\right)$
$z = g(x)$ $\quad \frac{dy}{dz}\frac{dz}{dx}$





$y = f\left(g_1(x), g_2(x), \ldots, g_n(x),\right)$
$z_i = g_i(x)$

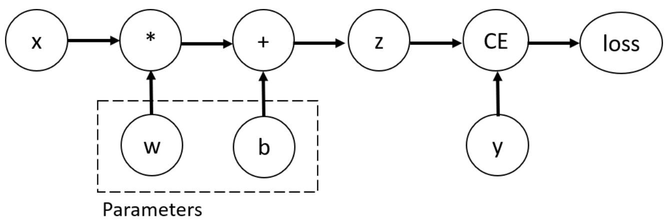
distributed chain rule

$$\frac{dy}{dz_1}\frac{dz_1}{dx} + \frac{dy}{dz_2}\frac{dz_2}{dx} + \ldots + \frac{dy}{dz_n}\frac{dz_n}{dx} = \sum_{i=1}^{n}\frac{dy}{dz_i}\frac{dz_i}{dx}$$

Computational graphs and **backpropagation**

Computational graph

- A **directed acyclic graph** (DAG) that represents a mathematical expression (or algorithm)
  - nodes represent operations or variables
  - (directed) edges indicate the flow of data
- Essential for modern deep learning
  - provide automatic differentiation (partial derivatives)

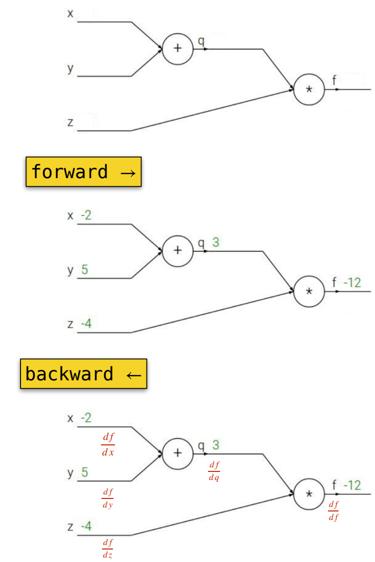


Example 1

$$f(x, y, z) = (x + y)z$$
$$q = x + y$$
$$f = qz$$

$$\frac{dq}{dx} = 1 \qquad \frac{dq}{dy} = 1$$
$$\frac{df}{dq} = z \qquad \frac{df}{dz} = q$$

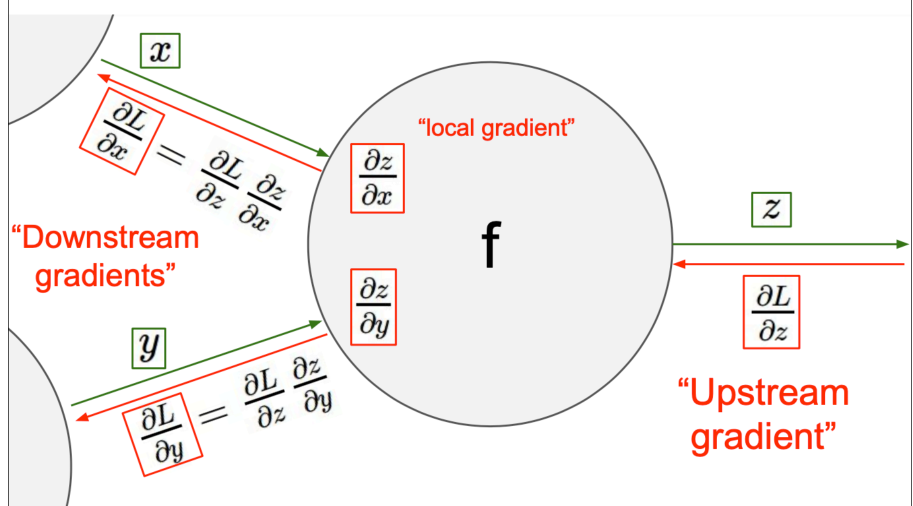
$$\frac{df}{dx} = \frac{df}{dq}\frac{dq}{dx} = z$$
$$\frac{df}{dy} = \frac{df}{dq}\frac{dq}{dy} = z$$



Show me the code

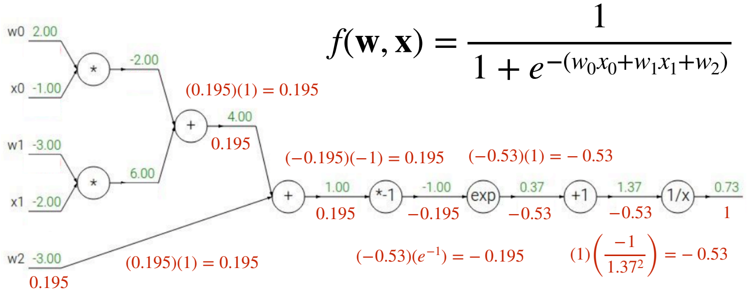
```
import torch

x = torch.tensor(-2., requires_grad=True)
y = torch.tensor(5., requires_grad=True)
z = torch.tensor(-4., requires_grad=True)

# forward pass
f = (x + y) * z

# backward pass
f.backward()

print(x.grad, y.grad, z.grad)


tensor(-4.) tensor(-4.) tensor(3.)
```


Key concepts



Example 2

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



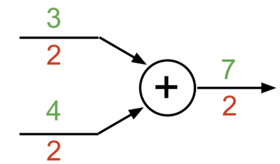
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -\frac{1}{x^2}$$

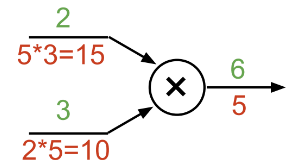
complete the partial derivatives in the graph using (upstream)x(local)

Special patterns

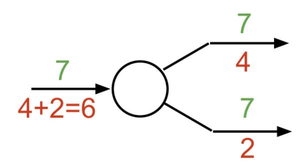
add gate: gradient distributor



mul gate: “swap multiplier”



copy gate: gradient adder



max gate: gradient router