CSC 561: Neural Networks and Deep Learning

Multilayer Perceptron

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

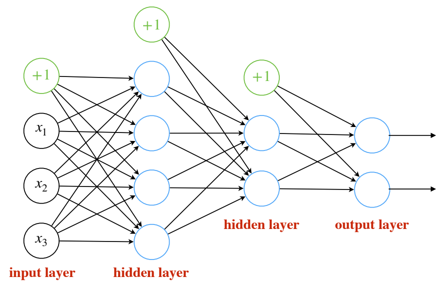
Spring 2024

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

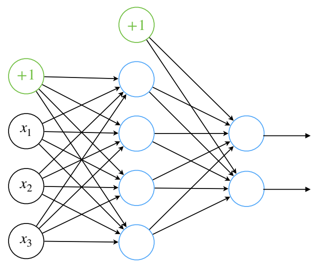
MLP layers

- 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)

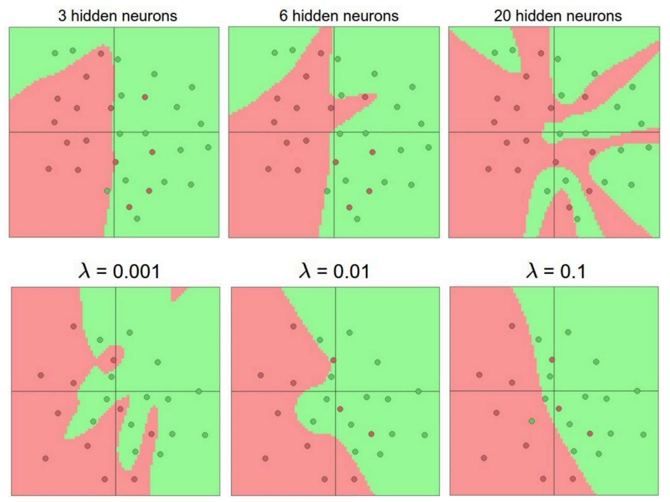
Example

- Asume a minibatch $X \in \mathbb{R}^{n\times3}$
  - hidden layer weights $W_h \in \mathbb{R}^{4\times3}$ and bias $b_h \in \mathbb{R}^4$
  - output layer weights $W_o \in \mathbb{R}^{2\times4}$ and bias $b_o \in \mathbb{R}^2$
- Network without activations `still linear!`
  - $h(X) = (XW_h^T + b_h)W_o^T + b_o = XW_h^TW_o^T + b_hW_o^T + b_o$
- 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



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

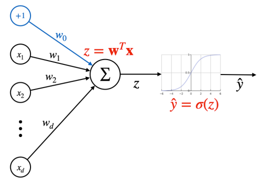
Forward pass

Calculate the analytical gradients

Gradient descent

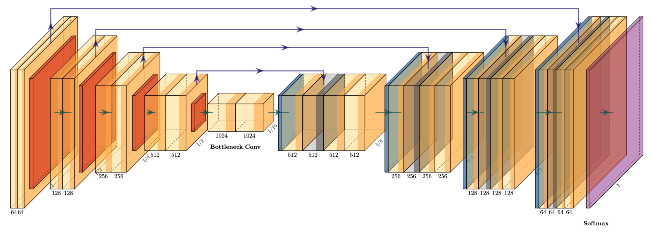
Partial derivatives

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

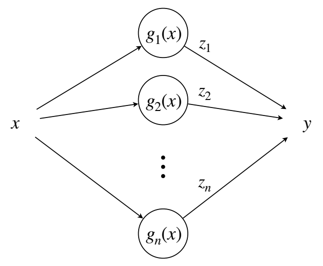
$y = f(x)$ $\qquad \frac{dy}{dx}$

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

chain rule

$$\begin{aligned} y &= f\big(g(x)\big) \\ z &= g(x) \end{aligned} \qquad \frac{dy}{dz}\frac{dz}{dx}$$

$$\begin{tikzcd} x \arrow[r] & g(x) \arrow[r, "z"] & y \end{tikzcd}$$



$$\begin{aligned} y &= f\big(g_1(x), g_2(x), \ldots, g_n(x),\big) \\ z_i &= g_i(x) \end{aligned}$$

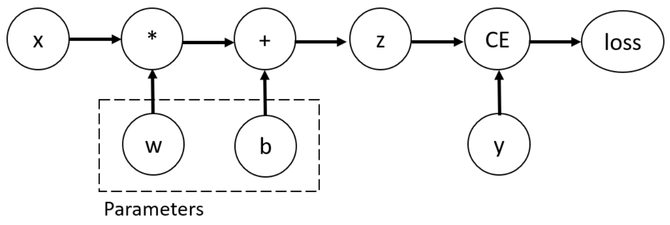
distributed chain rule

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

Computational graphs and **backpropagation**

Computational graph

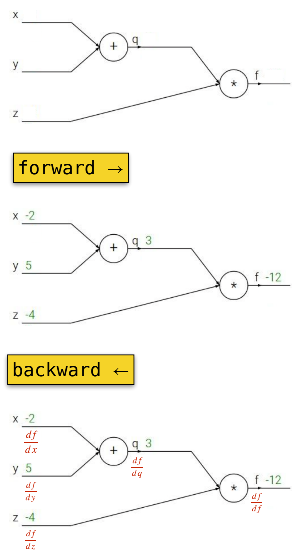
- A directed acyclic graph (DAG) that represents a mathematical expression (or algorithm)
  - nodes represent operations or variables
  - (directed) edges indicate the flow of data
- Essential for modern deep learning
  - provide automatic differentiation (partial derivatives)



Example 1

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

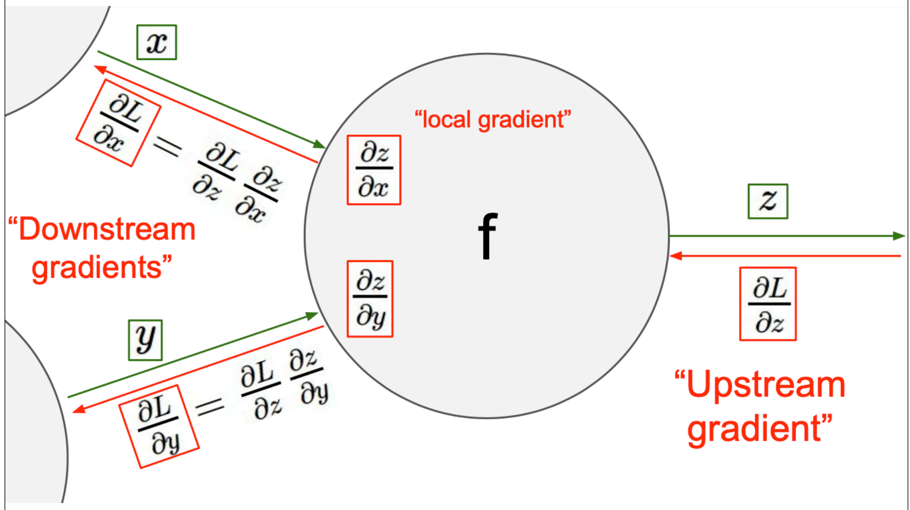
$$\frac{dq}{dx} = 1$$
$$\frac{dq}{dy} = 1$$
$$\frac{df}{dq} = z$$
$$\frac{df}{dz} = q$$
$$\frac{df}{dx} = \frac{df}{dq}\frac{dq}{dx} = z$$
$$\frac{df}{dy} = \frac{df}{dq}\frac{dq}{dy} = z$$



Show me the code

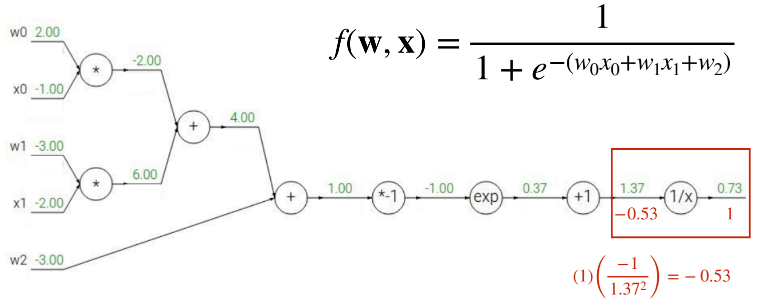
```
import torch

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


x
y
z
f
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
"Downstream gradients"
"Upstream gradient"

Example 2

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



$(1)\left(\frac{-1}{1.37^2}\right) = -0.53$

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

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

complete the partial derivatives in the graph using upstream x local gradients

Show me the code

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = 1 / (1 + torch.exp(-(w@x + b)))

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```

Equivalent computational graph

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = torch.sigmoid(w @ x + b)

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```

Computational graphs

- Pseudo-code for **forward** and **backward** passes

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

PyTorch gates

- Actual code for scalar multiplication

```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to cache some values for use in backward

Upstream gradient

Multiply upstream and local gradients