CSC 311: Introduction to Machine Learning

Lecture 6 - Neural Networks II

Rahul G. Krishnan & Amanjit Singh Kainth

University of Toronto, Fall 2024





Back-Propagation



Autodif

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 $d\mathcal{J}/d\mathbf{w}$, the average loss over all the training examples.
- Let's look at how we can calculate $d\mathcal{L}/d\mathbf{w}$, and then generalize this method to any directed acyclic graph (DAG).

Example: Two-Layer Neural Network



Figure 1: Two-Layer Neural Network

A neural network computes a composition of functions.

$$z_1^{(1)} = w_{10}^{(1)} \cdot 1 + w_{11}^{(1)} \cdot x_1 + w_{12}^{(1)} \cdot x_2$$

$$h_1 = \sigma(z_1^{(1)})$$

$$z_1^{(2)} = w_{10}^{(2)} \cdot 1 + w_{11}^{(2)} \cdot h_1 + w_{12}^{(2)} \cdot h_2$$

$$y_1 = z_1^{(2)}$$

$$z_2^{(1)} =$$

$$h_2 =$$

$$z_2^{(2)} =$$

$$y_2 =$$

$$L = \frac{1}{2} \left((y_1 - t_1)^2 + (y_2 - t_2)^2 \right)$$

Simplified Example: Logistic Least Squares





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



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

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

Univariate Chain Rule

How you would have done it in calculus class

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

What are the disadvantages of this approach?

Logistic Least Squares: Gradient for w

Computing the gradient for w:

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

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^{2}$$

Logistic Least Squares: Gradient for b

Computing the gradient for *b*:

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

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^{2}$$

Logistic Least Squares: Gradient for b

Computing the gradient for *b*:

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

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

13

Comparing Gradient Computations for w and b

Computing the gradient for w: Computing the gradient for b:

$$\begin{array}{ll} \frac{\partial \mathcal{L}}{\partial w} & & \frac{\partial \mathcal{L}}{\partial b} \\ &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w} & & = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial b} \\ &= (y-t) \ \sigma'(z) \ x & & = (y-t) \ \sigma'(z) \ 1 \end{array}$$

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^{2}$$

Structured Way of Computing Gradients

Computing the gradients:

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

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

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

15

- $\cdot\,$ Let $\overline{\mathit{y}}$ denote the derivative $\mathrm{d}\mathcal{L}/\mathrm{d}\mathit{y}$, 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:

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

L₂-Regularized Regression



Softmax Regression





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:

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

Multi-variate Chain Rule

In the context of back-propagation:





In our notation:

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

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 for which we're trying to compute gradients.

forward pass:

For
$$i = 1, ..., N$$
,
Compute v_i as a function of Parents (v_i) .

backward pass:

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

Backpropagation for Regularized Logistic Least Squares



Forward pass:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{reg} = \mathcal{L} + \lambda \mathcal{R}$$

Backward pass:

$$\overline{\mathcal{L}_{\text{reg}}} = 1$$

$$\overline{\mathcal{R}} = \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} \qquad \overline{z} = \overline{y} \frac{dy}{dz}$$

$$= \overline{\mathcal{L}_{\text{reg}}} \lambda \qquad = \overline{y} \sigma'(z)$$

$$\overline{\mathcal{L}} = \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} \qquad \overline{w} = \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{d\mathcal{R}}{dw}$$

$$= \overline{\mathcal{L}_{\text{reg}}} \qquad = \overline{z} x + \overline{\mathcal{R}} w$$

$$\overline{y} = \overline{\mathcal{L}} \frac{d\mathcal{L}}{dy} \qquad = \overline{z}$$

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:

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

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

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

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

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

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

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

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

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:

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

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

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

- One backward pass is as expensive as two forward passes.
- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.

- \cdot 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.
- We need to be careful with network initialization (should not set all weights = 0)
- Even optimization algorithms 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.

Autodiff

Auto-Differentiation

- Suppose we construct our networks out of a series of "primitive" operations (e.g., add, multiply) with specified routines for computing derivatives.
- Automatic-differentiation enables the creation of programs to perform backprop in a mechanical and automatic way.
- Many autodiff libraries: PyTorch, Tensorflow, Jax, etc.
- While autodiff automates the backward pass for you, it's still important to know how things work under the hood.
- We'll learn the basics of how such libraries work under the hood and cover and walk through Autodidact (a simplified numpy-based autograd library)
- https://github.com/mattjj/autodidact/tree/master

- Autograd is *not* finite differences:
 - 1. Finite differences are expensive (need two function evaluations per element of the gradient)
 - 2. Has numerical errors that can propagate if used for gradient-based learning
- The goal of autograd is build a program that for any *given* function, calculates the gradient with respect to some subset of inputs (we can think of parameters of a model as inputs to a function)

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

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\begin{aligned} \overline{\mathcal{L}} &= 1\\ \overline{y} &= (y - t)\\ \overline{z} &= \overline{y} \, \sigma'(z)\\ \overline{w} &= \overline{z} \, x \qquad \overline{b} = \overline{z} \end{aligned}$$

Computing the derivatives:

Reframing program into primitive operations

 We can always break up a program into a set of primitive operations or atomic units (rather than a mathematical operation).

Primitive Operations:

Original program:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^{2}$$

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

$$y = \frac{1}{t_5}$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$\mathcal{L} = t_7/2$$

Computation as a graph



- The evaluation of any function can be represented as a computation graph over primitive operations.
- By traversing the graph in topological order we can represent the evaluation of the function.
- Each node is then **annotated** with a gradient operation with computes a local gradient with special routines.
- Enables us to do backprop mechanically.

Computing gradients



Discuss: how would you create a program for autodiff?

Using computation graphs to trace computation

- Autodiff systems build the computation graph to evaluate a function.
- They create wrappers around the original numpy functions that have, for each function, a gradient operator defined.
- e.g. Node class in tracer.py (https://github.com/mattjj/ autodidact/blob/master/autograd/tracer.py) represents a node using the following attributes:
 - value: the value computed on a given set of inputs
 - ► fun: the operation defining the node
 - ▶ args & kwargs: the arguments to pass into the op
 - parents, parent Node
- During the forward pass, the **value** is kept track of internally so that on the backward pass the gradient function of the corresponding node can be called.

Building computation graphs under the hood

• Autograd's system create primitive ops that simulate the desired mathematical operation but implicitly build a graph.



```
def logistic(z):
    return 1. / (1. + np.exp(-z))
# that is equivalent to:
def logistic2(z):
    return np.reciprocal(np.add(1, np.exp(np.negative(z))))
```

```
z = 1.5
y = logistic(z)
```



• The Jacobian is a matrix of partial derivatives

$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

- For a given node that computes $\mathbf{y} = f(\mathbf{x})$ we can write down the gradient of some downstream loss with respect to \mathbf{x} as: $\overline{x_j} = \sum_i \overline{y_i} \frac{\partial y_i}{\partial x_j}$
- This can be vectorized as $\overline{\mathbf{x}} = \overline{\mathbf{y}}^{\mathbf{T}} \mathbf{J}$
- As a column vector we obtain: $\overline{\mathbf{x}} = \mathbf{J}^{\mathbf{T}}\overline{\mathbf{y}}$

Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \qquad \mathbf{J} = \mathbf{W} \qquad \overline{\mathbf{x}} = \mathbf{W}^T\overline{\mathbf{z}}$$

• Elementwise operations

$$\mathbf{y} = \exp(\mathbf{z}) \mathbf{J} = \begin{pmatrix} \exp(z_1) & 0 & \cdots & 0 \\ 0 & \exp(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \exp(z_n) \end{pmatrix} \, \tilde{\mathbf{z}} = \exp(\mathbf{z}) \odot \bar{\mathbf{y}}$$

Vector-Jacobian Products

- Every primitive operation, y = f(x) in the autograd framework has a defined Vector Jacobian Product function.
- Each vjp is a function.
- Input: (Output gradient \overline{y} , Arguments: x, y), Output: \overline{x}
- defvjp (in core.py) is a routine for registering VJPs (a dict)

<pre>defvjp(negative, defvjp(exp, defvjp(log,</pre>	lambda g, lambda g, lambda g,	ans ans ans	, x: , x: , x:	-g) an: g) s * / x	* (()	3)	
defvjp(add,	lambda	g, (ans,	x,	у	:	g,	
	lambda	g, (ans,	x,	У	:	g)	
<pre>defvjp(multiply,</pre>	lambda	g, (ans,	x,	У	:	у*	g,
	lambda	g, (ans,	x,	у	:	x *	g)
<pre>defvjp(subtract,</pre>	lambda	g, (ans,	x,	У	:	g,	
	lambda	g, (ans,	x,	у	:	-g)	

Putting it all together

- We can write down a computation graph for evaluating the loss function.
- Each node represents computation of an output as a function of the input.
- For each node, we can write down a **local** gradient operation for the loss with respect to the input; this can be expressed as a Vector-Jacobian product.
- Step 1: compute a forward pass to accumulate values in each node
- Step 2: run a backward pass to accumulate gradients at each node and pass the back to their parents recursively
- Take a gradient step and repeat!

Backward pass

• Defined in core.py, g is the error signal for the end node (1 in our case).

```
def backward_pass(g, end_node):
    outgrads = {end_node: g}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        fun, value, args, kwargs, argnums = node.recipe
        for argnum, parent in zip(argnums, node.parents):
            vjp = primitive_vjps[fun][argnum]
            parent_grad = vjp(outgrad, value, *args, **kwargs)
            outgrads[parent] = add_outgrads(outgrads.get(parent), parent_grad)
    return outgrad
```

```
def add_outgrads(prev_g, g):
    if prev_g is None:
        return g
    return prev_g + g
```

Backward pass

 grad (in differential_operators.py) is a wrapper around make_vjp which builds the computational graph and feeds it to backward_pass.

```
def make_vjp(fun, x):
    ""Trace the computation to build the computation graph, and return
    a function which implements the backward pass."""
    start node = Node.new root()
    end_value, end_node = trace(start_node, fun, x)
    def vip(a):
        return backward_pass(g, end_node)
    return vjp, end_value
def grad(fun, argnum=0):
    def gradfun(*args, **kwargs):
        unary_fun = lambda x: fun(*subval(args, argnum, x), **kwargs)
        vjp, ans = make_vjp(unary_fun, args[argnum])
        return vjp(np.ones_like(ans))
    return gradfun
```

- Learned how to manually and programmatically build tools to calculate gradients in computational flow graphs.
- You have the knowledge to build your own neural network know and know exactly whats happening under the hood.
- In CSC413: You will have twelve weeks of learning about different kinds of neural networks, each of them can be thought of as a function with an underlying computational flow graph.
- Autograd is the backbone that enables us to take gradients with respect to all of them to learn via SGD!