CSC 311: Introduction to Machine Learning

Lecture 6 - Neural Networks II

Amanjit Singh Kainth

University of Toronto, Summer 2025

Outline

Back-Propagation

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

Example: Two-Layer Neural Network

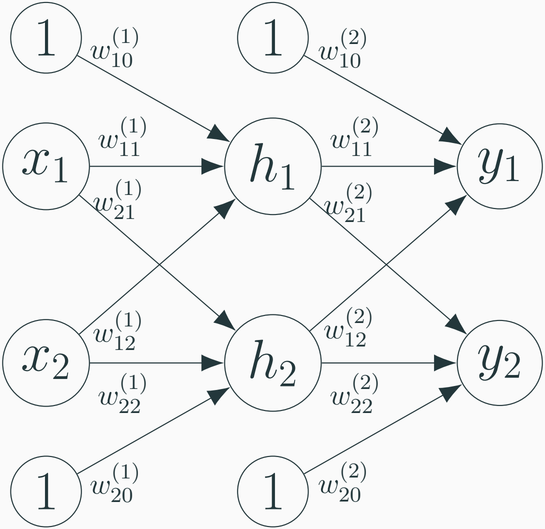


Figure 1: Two-Layer Neural Network

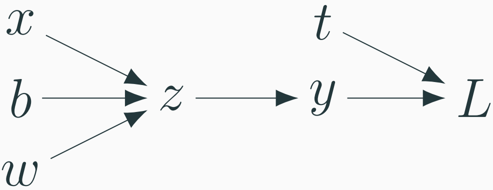
Computations for Two-Layer Neural Network

A neural network computes a composition of functions.

$$
\begin{aligned}
z_1^{(1)} &= w_{10}^{(1)} \cdot 1 + w_{11}^{(1)} \cdot x_1 + w_{12}^{(1)} \cdot x_2 \\
h_1 &= \sigma(z_1^{(1)}) \\
z_1^{(2)} &= w_{10}^{(2)} \cdot 1 + w_{11}^{(2)} \cdot h_1 + w_{12}^{(2)} \cdot h_2 \\
y_1 &= z_1^{(2)} \\
z_2^{(1)} &= \\
h_2 &= \\
z_2^{(2)} &= \\
y_2 &= \\
L &= \frac{1}{2}\left((y_1 - t_1)^2 + (y_2 - t_2)^2\right)
\end{aligned}
$$

Simplified Example: Logistic Least Squares

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



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.

$$
\begin{tikzcd}
x \arrow[dr] & & t \arrow[dr] & \\
b \arrow[r] & z \arrow[r] & y \arrow[r] & L \\
w \arrow[ur] & & &
\end{tikzcd}
$$

Uni-variate Chain Rule

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

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

Univariate Chain Rule

How you would have done it in calculus class

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

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

What are the disadvantages of this approach?

Logistic Least Squares: Gradient for w

Computing the gradient for w:

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

Computing the loss:

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

Logistic Least Squares: Gradient for b

Computing the gradient for b:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b} &= \\
&= \\
&= \\
&=
\end{aligned}
$$

Computing the loss:

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

Logistic Least Squares: Gradient for b

Computing the gradient for b:

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

Computing the loss:

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

Comparing Gradient Computations for w and b

Computing the gradient for w:

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

Computing the gradient for b:

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

Computing the loss:

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

Structured Way of Computing Gradients

Computing the gradients:

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

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

Computing the loss:

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

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:

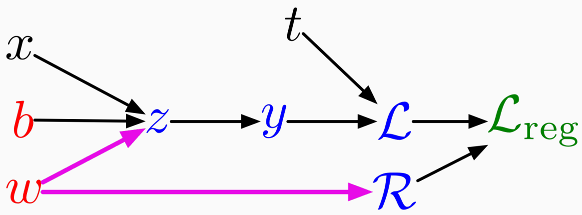
$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2
\end{aligned}
$$

Computing the derivatives:

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

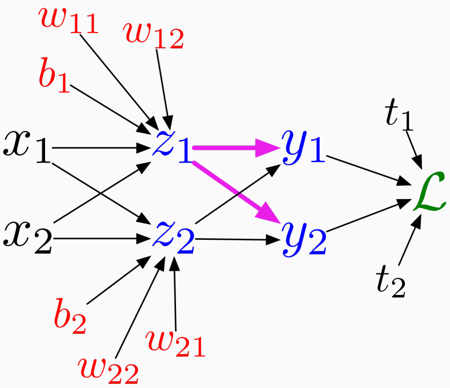
Computation Graph has a Fan-Out > 1

L_2-Regularized Regression



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

Softmax Regression



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

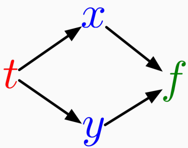
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$

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

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:

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

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

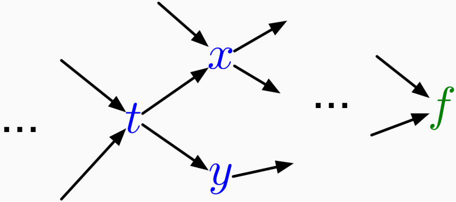
Multi-variate Chain Rule

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:

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

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 for which we're trying to compute gradients.

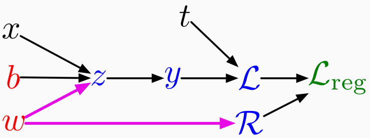
- forward pass:

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

- backward pass:

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

Backpropagation for Regularized Logistic Least Squares



Forward pass:

$$z = wx + b$$

$$y = \sigma(z)$$

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

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

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

Backward pass:

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

$$\begin{aligned}\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\mathrm{reg}}}\,\frac{\mathrm{d}\mathcal{L}_{\mathrm{reg}}}{\mathrm{d}\mathcal{R}} \\ &= \overline{\mathcal{L}_{\mathrm{reg}}}\,\lambda\end{aligned}$$

$$\begin{aligned}\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\mathrm{reg}}}\,\frac{\mathrm{d}\mathcal{L}_{\mathrm{reg}}}{\mathrm{d}\mathcal{L}} \\ &= \overline{\mathcal{L}_{\mathrm{reg}}}\end{aligned}$$

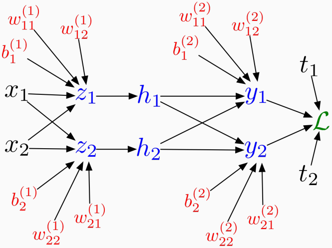
$$\begin{aligned}\overline{y} &= \overline{\mathcal{L}}\,\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} \\ &= \overline{\mathcal{L}}\,(y - t)\end{aligned}$$

$$\begin{aligned}\overline{z} &= \overline{y}\,\frac{\mathrm{d}y}{\mathrm{d}z} \\ &= \overline{y}\,\sigma'(z)\end{aligned}$$

$$\begin{aligned}\overline{w} &= \overline{z}\,\frac{\partial z}{\partial w} + \overline{\mathcal{R}}\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} \\ &= \overline{z}\,x + \overline{\mathcal{R}}\,w\end{aligned}$$

$$\begin{aligned}\overline{b} &= \overline{z}\,\frac{\partial z}{\partial b} \\ &= \overline{z}\end{aligned}$$

Backpropagation for Two-Layer Neural Network



Forward pass:

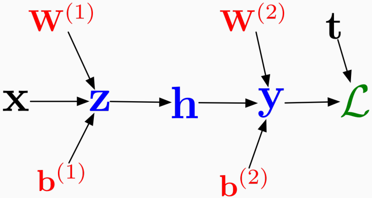
$$
\begin{aligned}
z_i &= \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \\
h_i &= \sigma(z_i) \\
y_k &= \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \\
\mathcal{L} &= \frac{1}{2} \sum_k (y_k - t_k)^2
\end{aligned}
$$

Backward pass:

$$
\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{y_k} &= \overline{\mathcal{L}}\,(y_k - t_k) \\
\overline{w_{ki}^{(2)}} &= \overline{y_k}\, h_i \\
\overline{b_k^{(2)}} &= \overline{y_k} \\
\overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)} \\
\overline{z_i} &= \overline{h_i}\, \sigma'(z_i) \\
\overline{w_{ij}^{(1)}} &= \overline{z_i}\, x_j \\
\overline{b_i^{(1)}} &= \overline{z_i}
\end{aligned}
$$

Backpropagation for Two-Layer Neural Network

In vectorized form:



Forward pass:

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

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

$$\begin{aligned} \overline{w_{ki}^{(2)}} &= \overline{y_k}\, h_i \\ \overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)} \end{aligned}$$

- 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.

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.
- 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`

Starting simple

- 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)

Gradient computation

- 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:

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

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).

Original program:

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

Primitive Operations:

$$t_1 = wx$$
$$z = t_1 + b$$
$$t_3 = -z$$
$$t_4 = \exp(t_3)$$
$$t_5 = 1 + t_4$$
$$y = \frac{1}{t_5}$$
$$t_6 = y - t$$
$$t_7 = t_6^2$$
$$\mathcal{L} = t_7/2$$

Computation as a graph

$$
\begin{tikzcd}
 & t_1 = wx \arrow[rr] & & z = t_1 + b \arrow[r] & t_3 = -z \arrow[rr] & & t_4 = \exp(t_3) \arrow[d] \\
w \arrow[ur] & & x \arrow[ul] & b \arrow[u] & & & t_5 = 1 + t_4 \arrow[d] \\
 & & & & & & y = \frac{1}{t_5} \arrow[d] \\
 & & & & & t \arrow[r] & t_6 = y - t \arrow[d] \\
 & & & & & \mathcal{L} = \frac{t_7}{2} & t_7 = t_6^2 \arrow[l]
\end{tikzcd}
$$

Using computation graphs to trace computation

- 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

$$
\begin{tikzcd}
& t_1 = wx \arrow[dl, "x"'] \arrow[dr, "w"] & & z = t_1 + b \arrow[ll, "1"'] \arrow[d, "1"] & t_3 = -z \arrow[l, "-1"'] & t_4 = \exp(t_3) \arrow[l, "\exp(t_3)"'] \\
w & & x & b & & t_5 = 1 + t_4 \arrow[u, "1"] \\
& & & & & y = \frac{1}{t_5} \arrow[u, "-\frac{1}{y^2}"] \\
& & & & t & t_6 = y - t \arrow[l, "-1"'] \arrow[u, "1"] \\
& & & & \mathcal{L} = \frac{t_7}{2} \arrow[r, "\frac{1}{2}"] & t_7 = t_6^2 \arrow[u, "2t_6"]
\end{tikzcd}
$$

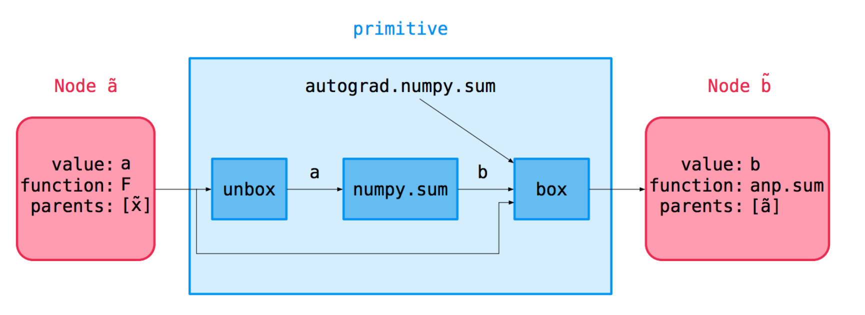
Discuss: how would you create a program for autodiff?

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.

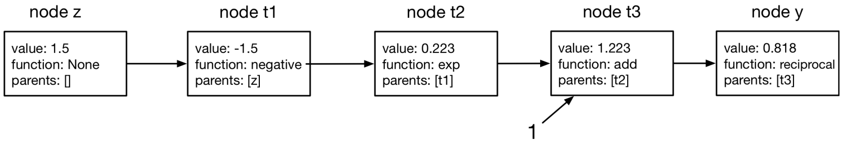


Example graph for a small program

```python
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)
```



Vectorizing gradient operations

- 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: $\overline{\mathbf{x}} = \mathbf{J}^{\mathbf{T}} \overline{\mathbf{y}}$

Vectorizing gradient operations

- Matrix-vector product

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

- 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)

```
defvjp(negative, lambda g, ans, x: -g)
defvjp(exp,      lambda g, ans, x: ans * g)
defvjp(log,      lambda g, ans, x: g / x)

defvjp(add,         lambda g, ans, x, y : g,
                    lambda g, ans, x, y : g)
defvjp(multiply,    lambda g, ans, x, y : y * g,
                    lambda g, ans, x, y : x * g)
defvjp(subtract,    lambda g, ans, x, y : g,
                    lambda g, ans, x, y : -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 vjp(g):
        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
```

Recap

- 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!