

# CSC 311: Introduction to Machine Learning

## Lecture 6 - Neural Networks II

---

Amanjit Singh Kainth

University of Toronto, Summer 2026

# Outline

1 Back-Propagation

2 Autodiff

# Back-Propagation

---

1 Back-Propagation

2 Autodiff

# 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

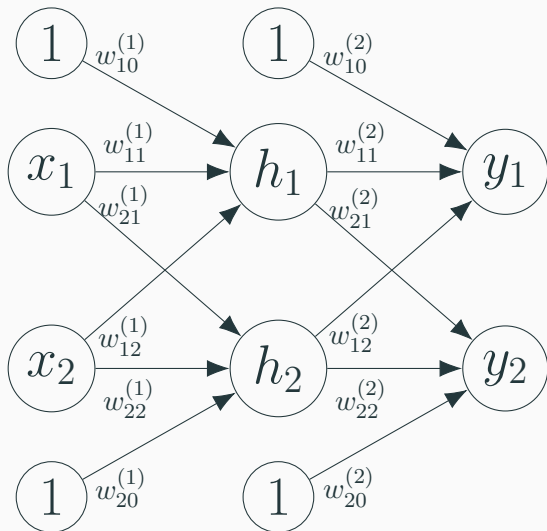


Figure 1: Two-Layer Neural Network

# Computations for Two-Layer Neural Network

A neural network computes a composition of functions.

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

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

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

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

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

$$h_2 =$$

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

$$y_2 =$$

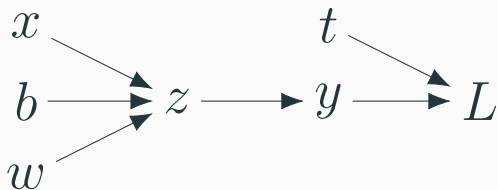
$$L = \frac{1}{2} \left( (y_1 - t_1)^2 + (y_2 - t_2)^2 \right)$$

## Simplified Example: Logistic Least Squares

$$z = wx + b$$

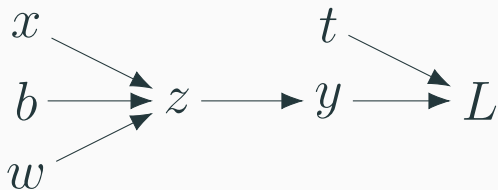
$$y = \sigma(z)$$

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



# 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  $z = f(g(x))$ .

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

# 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:

$$z = wx + b$$

$$y = \sigma(z)$$

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

## Logistic Least Squares: Gradient for $b$

Computing the gradient for  $b$ :

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

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

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

## Comparing Gradient Computations for $w$ and $b$

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

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

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

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

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

# Structured Way of Computing Gradients

Computing the gradients:

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

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{d\mathcal{L}}{dz} \frac{dz}{dw} = \frac{d\mathcal{L}}{dz} x$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{d\mathcal{L}}{dz} \frac{dz}{db} = \frac{d\mathcal{L}}{dz} 1$$

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

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

# Error Signal Notation

- Let  $\bar{y}$  denote the derivative  $d\mathcal{L}/dy$ , 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:

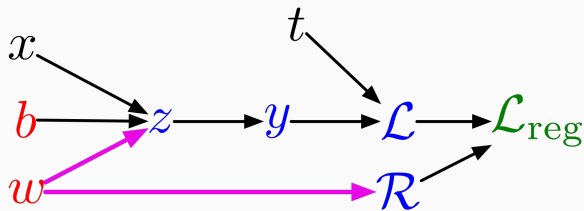
$$\bar{y} = (y - t)$$

$$\bar{z} = \bar{y} \sigma'(z)$$

$$\bar{w} = \bar{z} x \quad \bar{b} = \bar{z}$$

# Computation Graph has a Fan-Out $> 1$

## $L_2$ -Regularized Regression



$$z = wx + b$$

$$y = \sigma(z)$$

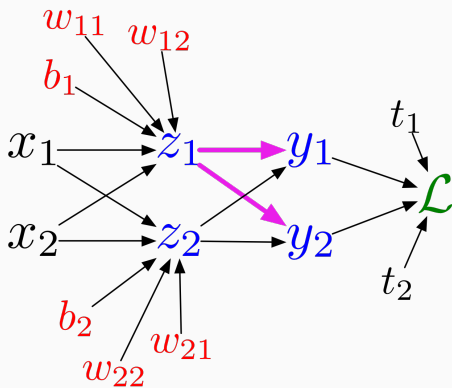
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

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

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

# Computation Graph has a Fan-Out > 1

## Softmax Regression



$$z_l = \sum_j w_{lj} x_j + b_l$$

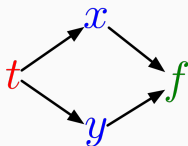
$$y_k = \frac{e^{z_k}}{\sum_l e^{z_l}}$$

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

# Multi-variate Chain Rule

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

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



Example:

$$f(x, y) = y + e^{xy}$$

$$x(t) = \cos t$$

$$y(t) = t^2$$

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

$$= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t$$

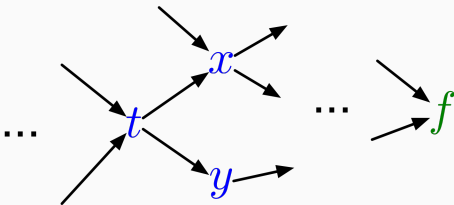
# Multi-variate Chain Rule

In the context of back-propagation:

Mathematical expressions  
to be evaluated

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

Values already computed  
by our program



In our notation:

$$\bar{t} = \bar{x} \frac{dx}{dt} + \bar{y} \frac{dy}{dt}$$

## Full Backpropagation Algorithm:

Let  $v_1, \dots, v_N$  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:

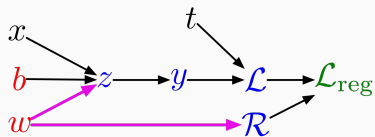
For  $i = 1, \dots, N$ ,  
Compute  $v_i$  as a function of  $\text{Parents}(v_i)$ .

- 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:

$$z = wx + b$$

$$y = \sigma(z)$$

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

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

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

Backward pass:

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

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

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

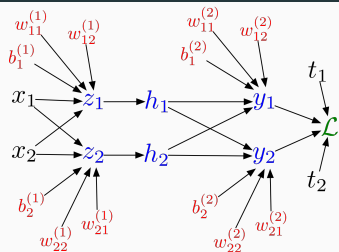
$$\begin{aligned}\overline{y} &= \overline{\mathcal{L}} \frac{d\mathcal{L}}{dy} \\ &= \overline{\mathcal{L}}(y - t)\end{aligned}$$

$$\begin{aligned}\overline{z} &= \overline{y} \frac{dy}{dz} \\ &= \overline{y} \sigma'(z)\end{aligned}$$

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

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

# Backpropagation for Two-Layer Neural Network



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Backward pass:

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

$$\bar{y}_k = \bar{\mathcal{L}} (y_k - t_k)$$

$$\bar{w}_{ki}^{(2)} = \bar{y}_k h_i$$

$$\bar{b}_k^{(2)} = \bar{y}_k$$

$$\bar{h}_i = \sum_k \bar{y}_k w_{ki}^{(2)}$$

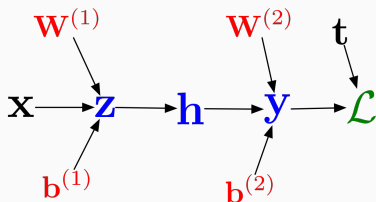
$$\bar{z}_i = \bar{h}_i \sigma'(z_i)$$

$$\bar{w}_{ij}^{(1)} = \bar{z}_i x_j$$

$$\bar{b}_i^{(1)} = \bar{z}_i$$

# 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:

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

$$\bar{\mathbf{y}} = \bar{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$

$$\overline{\mathbf{W}^{(2)}} = \bar{\mathbf{y}}\mathbf{h}^\top$$

$$\overline{\mathbf{b}^{(2)}} = \bar{\mathbf{y}}$$

$$\bar{\mathbf{h}} = \mathbf{W}^{(2)\top}\bar{\mathbf{y}}$$

$$\bar{\mathbf{z}} = \bar{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{\mathbf{W}^{(1)}} = \bar{\mathbf{z}}\mathbf{x}^\top$$

$$\overline{\mathbf{b}^{(1)}} = \bar{\mathbf{z}}$$

# 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  $\bar{y}$  denote the derivative  $d\mathcal{L}/dy$ , 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:

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

$$\bar{y} = (y - t)$$

$$\bar{z} = \bar{y} \sigma'(z)$$

$$\bar{w} = \bar{z} x \quad \bar{b} = \bar{z}$$

## 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:

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

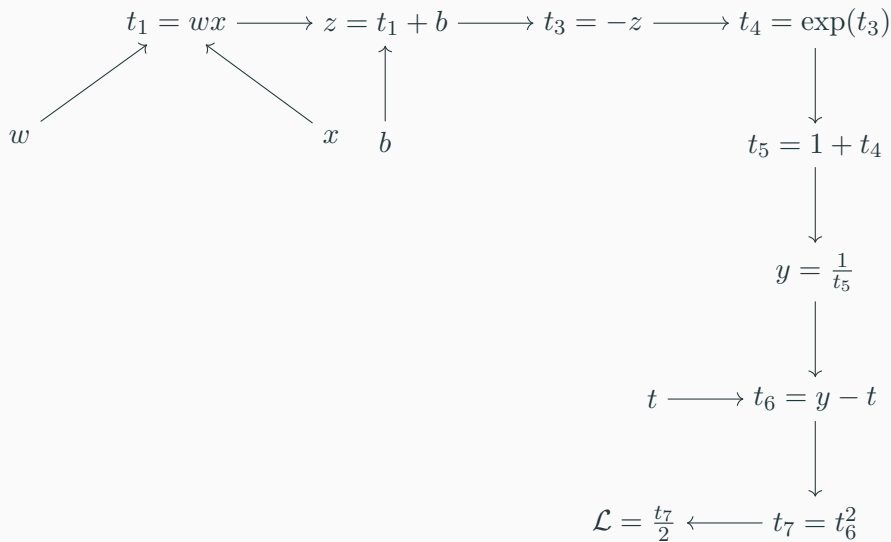
$$y = \frac{1}{t_5}$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

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

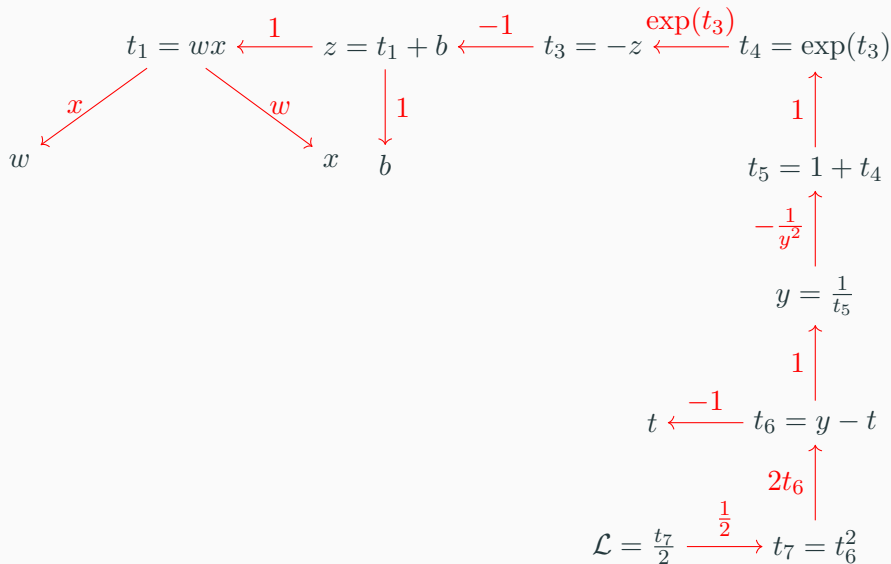
# Computation as a graph



## 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



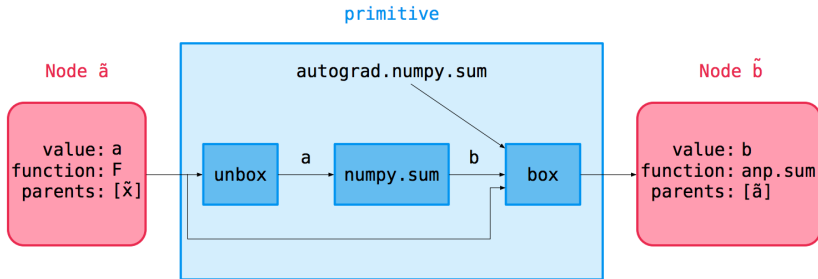
Discuss: how would you create a program for autodiff?

# 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.

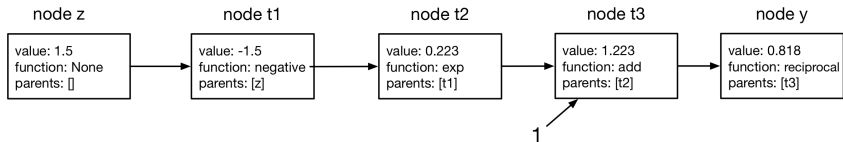
# Building computation graphs under the hood

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



# Example graph for a small program

```
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:

$$\bar{x}_j = \sum_i \bar{y}_i \frac{\partial y_i}{\partial x_j}$$

- This can be vectorized as  $\bar{\mathbf{x}} = \bar{\mathbf{y}}^T \mathbf{J}$
- As a column vector we obtain:  $\bar{\mathbf{x}} = \mathbf{J}^T \bar{\mathbf{y}}$

# Vectorizing gradient operations

- Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \quad \mathbf{J} = \mathbf{W} \quad \bar{\mathbf{x}} = \mathbf{W}^T \bar{\mathbf{z}}$$

- Elementwise operations

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

## 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  $\bar{y}$ , Arguments:  $x, y$ ), Output:  $\bar{x}$
- 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!