ECE 1508: Applied Deep Learning

Chapter 2: Feedforward Neural Networks

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Fall 2025

Fully-Connected FNNs: *Training*

+ *So, what do we do with forward pass?*
– Say the FNN is fixed and all the weights and biases are given. Forward pass determines the label of a given data-point x.
+ *Well! But we need to train the network! Right?!*
– Yes! We define the loss and then train it via gradient descent
+ *Then, we need to determine the gradient! It sounds complicated!*
– Well! there is an efficient algorithm for that called *backpropagation*

Let's see what backpropagation is!

Fully-Connected FNNs: *Training*

Let's recall how we train the network: in our FNN, we considered M outputs; thus, we could assume that the dataset is of the form

$$\mathbb{D} = \{(\boldsymbol{x}_b, \boldsymbol{v}_b) \text{ for } b = 1, \ldots, B\}$$

Here, we have denoted the true labels by $\boldsymbol{v}_b = [v_{b,1}, \ldots, v_{b,M}]^{\mathsf{T}}$ to avoid confusion with the FNN's outputs. Now, let's denote the forward pass by $\mathbf{y}_b = \mathit{PassF}(\boldsymbol{x}_b|\mathbf{w})$ with $\mathbf{w}$ is a vector collecting $\{\mathbf{W}_\ell\}$ for $\ell = 1, \ldots, L+1$

> *Given data-point $\boldsymbol{x}_b$, by forward pass we get $\mathbf{y}_b$ at the output layer of the FNN with weights $\mathbf{w}$. This output is desired to be the true label $\boldsymbol{v}_b$*

How do we do the training?

$$\mathbf{w}^\star = \operatorname*{argmin}_{\mathbf{w}} \hat{R}(\mathbf{w}) = \operatorname*{argmin}_{\mathbf{w}} \frac{1}{B} \sum_{b=1}^{B} \mathcal{L}(\mathbf{y}_b, \boldsymbol{v}_b) \qquad \text{(Training)}$$

Fully-Connected FNNs: *Training*

Let's recall how we train the network

$$\mathbf{w}^\star = \operatorname*{argmin}_{\mathbf{w}} \hat{R}(\mathbf{w}) = \operatorname*{argmin}_{\mathbf{w}} \frac{1}{B} \sum_{b=1}^{B} \mathcal{L}(\mathbf{y}_b, \boldsymbol{v}_b)$$

$$= \operatorname*{argmin}_{\mathbf{w}} \frac{1}{B} \sum_{b=1}^{B} \mathcal{L}(\textit{PassF}(\boldsymbol{x}_b | \mathbf{w}), \boldsymbol{v}_b)$$

1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ and η, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
5: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
6: **end while**

In this algorithm, the main challenge is to calculate $\nabla \hat{R}(\mathbf{w}^{(t-1)})$

Fully-Connected FNNs: *Training*

The main challenge is to calculate $\nabla \hat{R}(\mathbf{w})$

First, let's see what are the entries of $\mathbf{w}$: $\mathbf{w}$ *contains all weights and biases. Following our notations, we can say*

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}_1[1,:] \\ \vdots \\ \mathbf{w}_1[\mathcal{W}_1,:] \\ \vdots \\ \mathbf{w}_{L+1}[1,:] \\ \vdots \\ \mathbf{w}_{L+1}[\mathcal{W}_{L+1},:] \end{bmatrix} \begin{matrix} \text{layer } \ell = 1 \\ \\ \text{layer } \ell = L+1 \end{matrix}$$

$$\mathbf{w}_\ell[j,:] = \begin{bmatrix} w_\ell[j,0] \\ w_\ell[j,1] \\ \vdots \\ w_\ell[j,\mathcal{W}_{\ell-1}] \end{bmatrix}$$

So, the entries of $\mathbf{w}$ *are* $w_\ell[j,i]$ *for different choices of* i, j *and* ℓ

Fully-Connected FNNs: *Training*

The main challenge is to calculate $\nabla \hat{R}(\mathbf{w})$

Let's try to open up the gradient: *we need partial derivatives of* $\hat{R}(\mathbf{w})$ *with respect to* $w_\ell[j, i]$ *for* $i = 0 : \mathcal{W}_{\ell-1}$, $j = 1 : \mathcal{W}_\ell$, *and* $\ell = 1 : L + 1$

$$\frac{\partial}{\partial w_\ell[j, i]} \hat{R}(\mathbf{w}) = \frac{\partial}{\partial w_\ell[j, i]} \frac{1}{B} \sum_{b=1}^{B} \mathcal{L}(\mathbf{y}_b, \boldsymbol{v}_b)$$

$$= \frac{1}{B} \sum_{b=1}^{B} \boxed{\frac{\partial}{\partial w_{i,j}[\ell]} \mathcal{L}(\mathbf{y}_b, \boldsymbol{v}_b)}$$

So, it's enough to develop an algorithm that computes partial derivative for a *single data-point.* *Derivative of the risk is then average of these point-wise derivatives.*

Fully-Connected FNNs: *Training*

Let's make an agreement: *we consider a data-point x with label v and write*

$$\frac{\partial}{\partial w_\ell [j,i]} \mathcal{L}\left(\textit{PassF}\left(x | \mathbf{w}\right), v\right) = \frac{\partial}{\partial w_\ell [j,i]} \mathcal{L}\left(\mathbf{y}, v\right)$$

while keeping in mind that $\mathbf{y}$ is a function of $\mathbf{w}$

To determine the partial derivatives, we note that

$\mathbf{y}$ *is a nested function of* $\mathbf{w}$

so, we can determine the derivative via *chain rule*. Let's recall the *chain rule* and see how we can *apply it on a graph*

Review: *Chain Rule*

Assume $z = g(x)$ and $y = f(z)$: y *is a nested function of* x*, as we can write*

$$y = f(g(x))$$

Intuitively, we can say: *if at point* x *we move with tiny step* $\mathrm{d}x$, z *varies as*

$$\mathrm{d}z = \dot{g}(x)\mathrm{d}x$$

This variation also varies y: *moving from* $z = g(x)$ *with tiny step* $\mathrm{d}z$ *leads to*

$$\mathrm{d}y = \dot{f}(z)\mathrm{d}z$$

So, we have

$$\mathrm{d}y = \dot{f}(z)\dot{g}(x)\mathrm{d}x$$

Review: *Chain Rule*

We have concluded that by moving x with $\mathrm{d}x$, we get

$$\mathrm{d}y = \dot{f}(z)\dot{g}(x)\mathrm{d}x$$

On the other hand, we know that

$$\mathrm{d}y = \frac{\mathrm{d}}{\mathrm{d}x} f(g(x))\mathrm{d}x$$

This concludes the *chain rule*

Chain Rule: *Scalar Form*

The derivative of nested function $y = f(g(x))$ with respect to x is given by

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = \dot{f}(z)\dot{g}(x) = \frac{\mathrm{d}y}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}x}$$

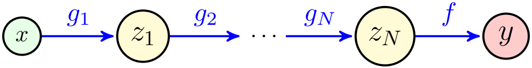
Computation Graph

We can extend this idea to deeper nested functions:

Let $z_1 = g_1(x)$ *and* $z_{n+1} = g_{n+1}(z_n)$ *for* $n = 1, \dots, N-1$; *then, derivative of* $y = f(z_N)$ *with respect to* x *is given by*

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z_N} \left(\prod_{n=1}^{N-1} \frac{\mathrm{d}z_{n+1}}{\mathrm{d}z_n} \right) \frac{\mathrm{d}z_1}{\mathrm{d}x} = \dot{f}(z_N) \left(\prod_{n=1}^{N-1} \dot{g}_{n+1}(z_n) \right) \dot{g}_1(x)$$

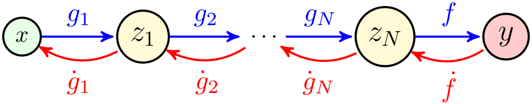
We can represent the chain rule, using a *computation graph*: *for the deep nested function given above, the computation graph is given by*



In this graph, we start from x *and pass forward to* $z_1 \to z_2 \to \dots$ *until we get to* y. *In each pass, we determine next variable via the function on the link*

Computation Graph

The derivative of y with respect to any variable on this graph is determined by a *backward pass from* y *towards the variable*



$$\frac{\mathrm{d}y}{\mathrm{d}z_N} = \dot{f}\left(z_N\right)$$

$$\frac{\mathrm{d}y}{\mathrm{d}z_{N-1}} = \frac{\mathrm{d}y}{\mathrm{d}z_N}\frac{\mathrm{d}z_N}{\mathrm{d}z_{N-1}} = \dot{f}\left(z_N\right)\dot{g}_N\left(z_{N-1}\right)$$

$$\vdots$$

$$\frac{\mathrm{d}y}{\mathrm{d}z_1} = \frac{\mathrm{d}y}{\mathrm{d}z_N}\frac{\mathrm{d}z_N}{\mathrm{d}z_{N-1}}\dots\frac{\mathrm{d}z_2}{\mathrm{d}z_1} = \dot{f}\left(z_N\right)\dot{g}_N\left(z_{N-1}\right)\dots\dot{g}_2\left(z_2\right)$$

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z_N}\frac{\mathrm{d}z_N}{\mathrm{d}z_{N-1}}\dots\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}x} = \dot{f}\left(z_N\right)\dot{g}_N\left(z_{N-1}\right)\dots\dot{g}_2\left(z_2\right)\dot{g}_1\left(x\right)$$

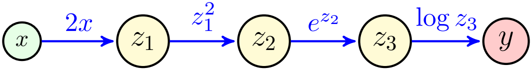
Computation Graph: *Example*

Example: *y is a nested function of x through the following chain of functions:*

$$z_1 = 2x \qquad z_2 = z_1^2 \qquad z_3 = e^{z_2} \qquad y = \log z_3$$

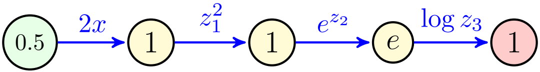
Determine the derivative of y with respect to x at $x = 0.5$.

Let's first plot the computation graph



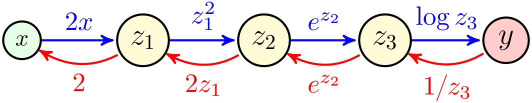
During the *forward pass* we get

$$z_1 = 2 \times 0.5 = 1 \rightarrow z_2 = 1^2 = 1 \rightarrow z_3 = e^1 = e \rightarrow y = \log e = 1$$

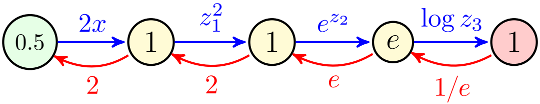


Computation Graph: *Example*

Now, we *pass backward* to determine the derivative

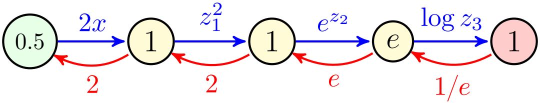


Let's first enter the values into the backward links



We now *navigate backward* to each variable that we want to determine the derivative y with respect to it

Computation Graph: *Example*



The derivatives are easily determined *recursively*

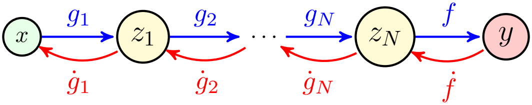
$$\frac{\mathrm{d}y}{\mathrm{d}z_3} = \frac{1}{e}$$

$$\frac{\mathrm{d}y}{\mathrm{d}z_2} = \frac{\mathrm{d}y}{\mathrm{d}z_3}\frac{\mathrm{d}z_3}{\mathrm{d}z_2} = \frac{e}{e} = 1$$

$$\frac{\mathrm{d}y}{\mathrm{d}z_1} = \frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1} = 1 \times 2 = 2$$

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}x} = 2 \times 2 = 4$$

Computation Graph



In the example, we had to *first compute the value of each variable*, in order to be able to compute the *values of the backward links*

This is an important fact that we should remember

Backward pass is only possible if we have already taken the forward pass

Review: *Chain Rule*

The nested function can be a multivariate: *assume for* $n = 1, \ldots, N$

$$z_n = g_n(x)$$

and let the nested function be

$$y = f(z_1, \ldots, z_N)$$

Let's follow the same logic: *starting from point* x, *we move with tiny step* $\mathrm{d}x$

$$\mathrm{d}z_n = \dot{g}_n(x)\mathrm{d}x$$

These variations lead to variation $\mathrm{d}y$ *in the nested function*

$$\mathrm{d}y = \nabla f(\mathbf{z})^{\mathsf{T}} \mathrm{d}\mathbf{z} = \sum_{n=1}^{N} \frac{\partial y}{\partial z_n} \mathrm{d}z_n = \sum_{n=1}^{N} \dot{f}_n(\mathbf{z}) \mathrm{d}z_n$$

Review: *Chain Rule*

We can hence write

$$\mathrm{d}y = \sum_{n=1}^{N} \dot{f}_n(\mathbf{z})\, \mathrm{d}z_n = \sum_{n=1}^{N} \dot{f}_n(\mathbf{z})\, \dot{g}_n(x) \mathrm{d}x$$

This concludes the *vector form* of the chain rule

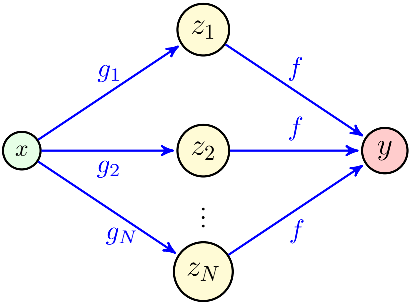
Chain Rule: *Vector Form*

Let $\mathbf{z} = [z_1, \ldots, z_N]^\mathsf{T}$ and $z_n = g_n(x)$. The derivative of nested function $y = f(\mathbf{z})$ with respect to x is given by

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \sum_{n=1}^{N} \frac{\partial y}{\partial z_n} \frac{\mathrm{d}z_n}{\mathrm{d}x} = \sum_{n=1}^{N} \dot{f}_n(\mathbf{z}) \dot{g}_n(x)$$

Computation Graph

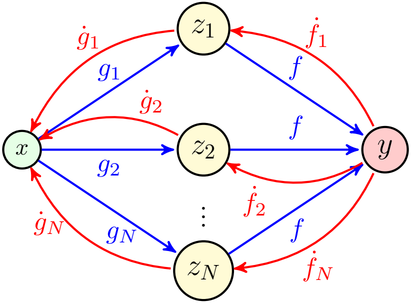
We can again represent the function via its vectorized *computation graph* and compute it by *passing forward* on this graph



We start from x and pass forward to $\mathbf{z} = [z_1, z_2, \dots, z_N]$. We then pass forward $\mathbf{z}$ to y. In each pass, we compute next variable via function on the link.

Computation Graph

The derivative with respect to any node is then given by *backward pass towards the node on the computation graph*

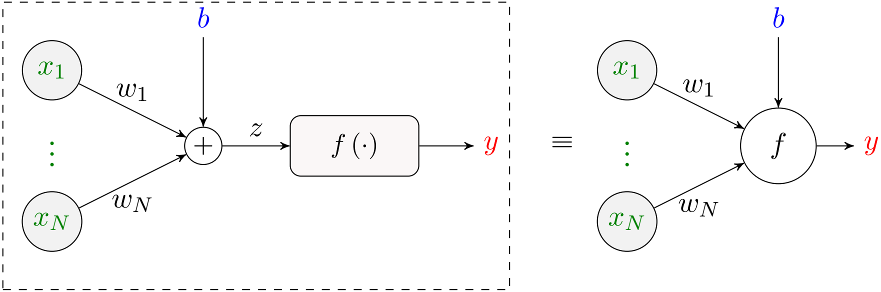


We add all backward passes towards x to determine the derivative

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \sum_{n=1}^{N} \frac{\partial y}{\partial z_n} \frac{\mathrm{d}z_n}{\mathrm{d}x} = \sum_{n=1}^{N} \dot{f}_n(\mathbf{z}) \dot{g}_n(x)$$

Computation Graph: *Single Neuron*

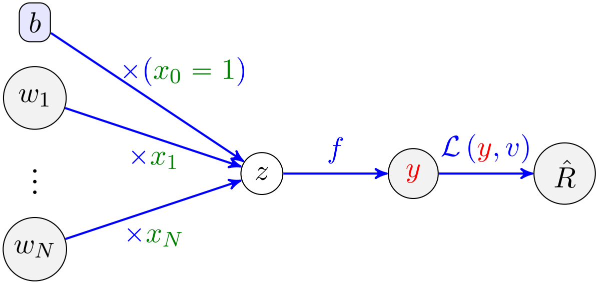
Let's now plot the *computation graph* of a single neuron and determine the gradient of the loss by *backward pass*



After passing the data-point $\boldsymbol{x}$ through the neuron, we get y and we calculate the loss for the *true label* v as $\mathcal{L}(y, v)$

Computation Graph: *Single Neuron*

The *computation graph* is hence given by

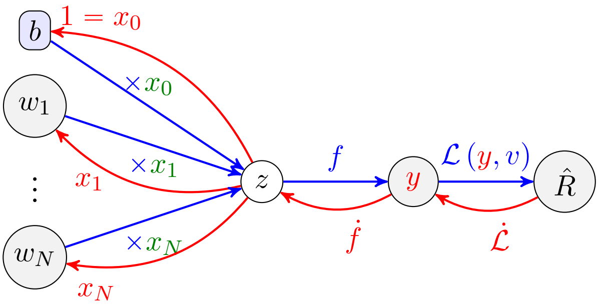


Here, the computation nodes are the *weights and bias of the neuron*

once we fix them, we can pass forward and get to the loss $\hat{R}$

Computation Graph: *Single Neuron*

Once passed forward, we can move backward to determine the derivatives

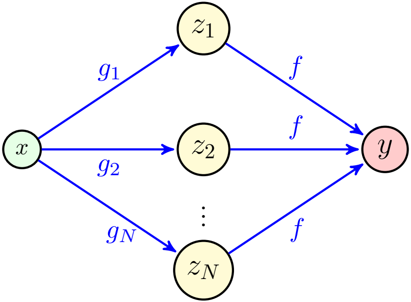


For a particular weight w_n, we can write (we drop arguments whenever clear)

$$\frac{\partial \hat{R}}{\partial w_n} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y} \frac{\mathrm{d}y}{\mathrm{d}z} \frac{\partial z}{\partial w_n} = \dot{\mathcal{L}} \dot{f} x_n$$

Computation Graph: *Multivariate Form*

Let's get back to the following *computation graph*



We define *vector-valued functions*, and show the graph *compactly*: *let's define*

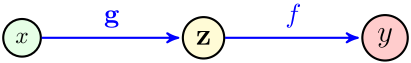
$$\mathbf{g}(x) = \begin{bmatrix} g_1(x) \\ \vdots \\ g_N(x) \end{bmatrix}$$

Computation Graph: *Multivariate Form*

Function $\mathbf{g}(\cdot)$ *gets* x *as the input and returns all* z_n*'s in a vector* $\mathbf{z}$*, i.e.,*

$$\mathbf{g}(x) = \begin{bmatrix} g_1(x) \\ \vdots \\ g_N(x) \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_N \end{bmatrix} = \mathbf{z}$$

We now use this vectorized notation to simplify the *computation graph* as



The forward pass on this graph is exactly the same: *we give* x *to the vectorized function* $\mathbf{g}(\cdot)$ *to get* $\mathbf{z}$ *which is then passed forward to* $f(\cdot)$ *to get* y

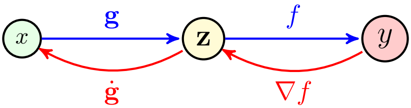
How does the backward pass look like then?

Computation Graph: *Multivariate Form*

We could define the derivative $\dot{\mathbf{g}}(\cdot)$ *as the vector of derivatives* $\dot{g}(\cdot)$

$$\dot{\mathbf{g}}(x) = \begin{bmatrix} \dot{g}_1(x) \\ \vdots \\ \dot{g}_N(x) \end{bmatrix} = \begin{bmatrix} \frac{\mathrm{d}z_1}{\mathrm{d}x} \\ \vdots \\ \frac{\mathrm{d}z_N}{\mathrm{d}x} \end{bmatrix} = \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}x}$$

Let's show this vectorized derivative and gradient of f *on the backward links*

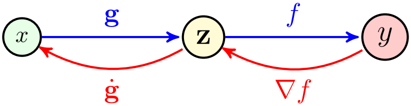


Well, we can pass backward as follows

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \sum_{n=1}^{N} \frac{\partial y}{\partial z_n} \frac{\mathrm{d}z_n}{\mathrm{d}x} = \nabla f(\mathbf{z})^{\mathsf{T}} \dot{\mathbf{g}}(x)$$

Computation Graph: *Multivariate Form*

+ *What can we conclude then?*
- We can sketch the computation graph very compactly using *vectorized derivatives* and *gradients*
+ *Does it mean that we should then pass backward exactly the same as in a computation graph with scalar variables and derivatives?*
- Pretty much Yes! Only one delicate detail: *we should know how to multiply those gradients and vectorized derivatives!*



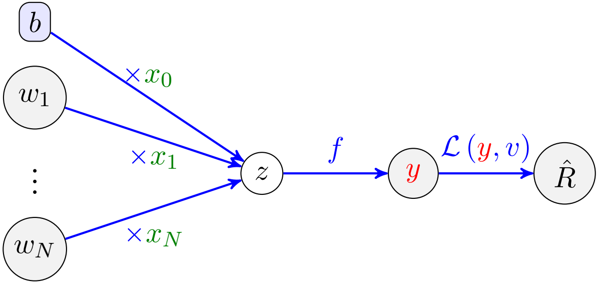
In our example, we determined *the inner product*

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \nabla f^{\mathsf{T}} \dot{\mathbf{g}}$$

Computation Graph: *Multivariate Form*

+ *How do we know which type of product we should use?*
– Well! If you were in doubt, we could always do it by expanding in terms of entries; however, *we are going to practice all key functions that appear in NN computation graphs!*

Before we start with all key functions, let's get back to a *single neuron*



Computation Graph: *Multivariate Form*

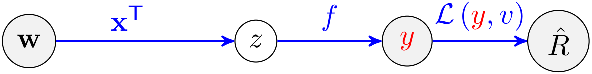
Let's define, as we did earlier, the following vectors

$$\mathbf{x} = \begin{bmatrix} x_0 = 1 \\ x_1 \\ \vdots \\ x_N \end{bmatrix} \qquad \textit{and} \qquad \mathbf{w} = \begin{bmatrix} w_0 = b \\ w_1 \\ \vdots \\ w_N \end{bmatrix}$$

Recall that output of the neuron is determined as $y = f(z)$ for

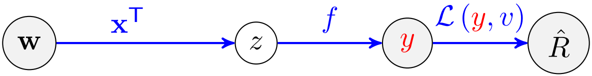
$$z = \mathbf{x}^\mathsf{T}\mathbf{w}$$

So, we can show the computation graph compactly as



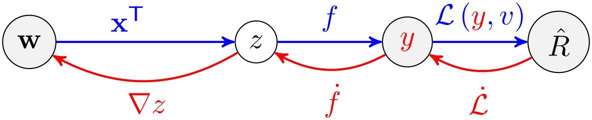
Computation Graph: *Multivariate Form*

Let's look at each link carefully: *we pass backward, so we start with last link*

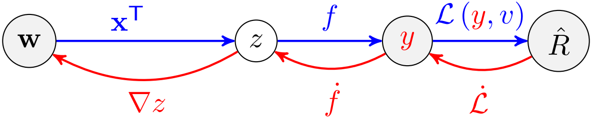


- $\hat{R}$ is a *scalar* function of *scalar* y, i.e., $\hat{R} = \mathcal{L}(y, v)$
  - ↳ *the backward link contains the scalar derivative* $\dot{\mathcal{L}}$
- y is a *scalar* function of *scalar* z, i.e., $y = f(z)$
  - ↳ *the backward link contains the scalar derivative* $\dot{f}$
- z is a *scalar* function of *vector* $\mathbf{w}$, i.e., $z = \mathbf{x}^\mathsf{T}\mathbf{w}$
  - ↳ *the backward link contains the gradient* ∇z

So, the graph with the backward links looks like



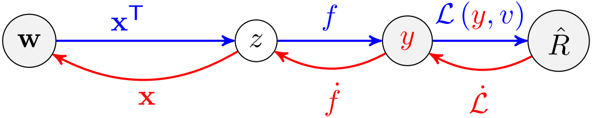
Computation Graph: *Multivariate Form*



We are almost complete; only *we need to calculate* ∇z

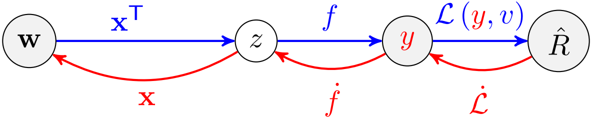
$$z = w_0 + w_1 x_1 + \ldots + w_N x_N \rightsquigarrow \nabla z = \begin{bmatrix} \partial z / \partial w_0 \\ \partial z / \partial w_1 \\ \vdots \\ \partial z / \partial w_1 \end{bmatrix} = \begin{bmatrix} 1 = x_0 \\ x_1 \\ \vdots \\ x_N \end{bmatrix} = \mathbf{x}$$

So, we are complete! Here is the *vectorized computation graph of the neuron*



Computation Graph: *Multivariate Form*

Now, how do we pass backward on this graph?



We arrived at y at the end of forward pass: *at this point, we can determine*

$$\frac{\mathrm{d}\hat{R}}{\mathrm{d}y} = \dot{\mathcal{L}}\left(y, v\right) = \dot{\mathcal{L}}$$

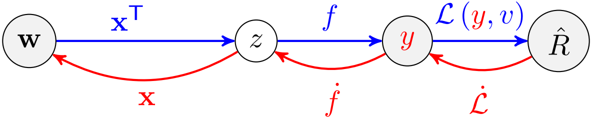
and we are at the computing node y. We then pass backward $\dot{\mathcal{L}}$. At node z, we can compute $\dot{f}\left(z\right)$, and use what we received from y to compute

$$\frac{\mathrm{d}\hat{R}}{\mathrm{d}z} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z} = \dot{f}\dot{\mathcal{L}}$$

and pass it backward

Computation Graph: *Multivariate Form*

Now, how do we pass backward on this graph?



Arriving at $\mathbf{w}$, we can determine $\nabla z = \mathbf{x}$ and use what we received from z to compute what we want

$$\nabla \hat{R} \text{ w.r.t. } \mathbf{w} \equiv \nabla_{\mathbf{w}} \hat{R} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}z} \nabla z = \dot{f} \dot{\mathcal{L}} \mathbf{x}$$

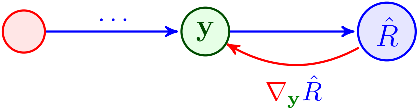
+ *Well! That seems easier!*
– Right! Let's now try some important cases

Backpropagation: *Local Operations*

Let's consider a general problem: *an objective scalar $\hat{R}$ is a function of K-dimensional vector $\mathbf{y} \in \mathbb{R}^K$. Clearly in this case, we have a gradient*

$$\nabla_{\mathbf{y}} \hat{R} = \begin{bmatrix} \partial \hat{R} / \partial y_1 \\ \vdots \\ \partial \hat{R} / \partial y_K \end{bmatrix}$$

Assume that we know this gradient. The vector $\mathbf{y}$ is also function of another variable. We want to compute gradient of $\hat{R}$ with respect to this other variable

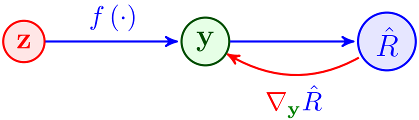


We now consider different cases for the other variable and its link to $\mathbf{y}$

Backpropagation: *Local Operation 1*

Entry-wise Functional Operation

$\mathbf{y} \in \mathbb{R}^K$ *is a function of* $\mathbf{z} \in \mathbb{R}^K$ *as* $\mathbf{y} = f(\mathbf{z})$ *with* $f(\cdot)$ *operating entry-wise*



For this case, we note that y_k is *only a function of* z_k; thus we have

$$\frac{\partial \hat{R}}{\partial z_k} = \frac{\partial \hat{R}}{\partial y_k} \frac{\partial y_k}{\partial z_k} = \frac{\partial \hat{R}}{\partial y_k} \dot{f}(z_k)$$

So, we can use *entry-wise product* $\odot$ to get from $\nabla_{\mathbf{y}} \hat{R}$ to $\nabla_{\mathbf{z}} \hat{R}$

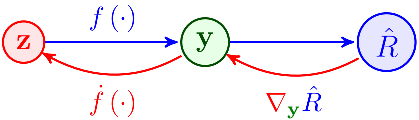
$$\nabla_{\mathbf{z}} \hat{R} = \nabla_{\mathbf{y}} \hat{R} \odot \dot{f}(\mathbf{z})$$

Backpropagation: *Local Operation 1*

Reminder: *Entry-wise product of two vectors of the same size is*

$$\mathbf{z} \odot \mathbf{y} \begin{bmatrix} z_1 \\ \vdots \\ z_K \end{bmatrix} \odot \begin{bmatrix} y_1 \\ \vdots \\ y_K \end{bmatrix} = \begin{bmatrix} y_1 z_1 \\ \vdots \\ y_K z_K \end{bmatrix}$$

So, we can compactly perform this local operation as follows



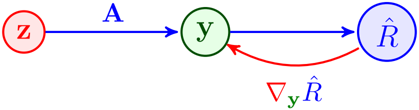
with the backward step

$$\nabla_{\mathbf{z}} \hat{R} = \nabla_{\mathbf{y}} \hat{R} \odot \dot{f}(\mathbf{z})$$

Backpropagation: *Local Operation 2*

Linear Vector-to-Vector Operation

$\mathbf{y} \in \mathbb{R}^K$ *is a function of* $\mathbf{z} \in \mathbb{R}^N$ *as* $\mathbf{y} = \mathbf{A}\mathbf{z}$ *with* $\mathbf{A} \in \mathbb{R}^{K \times N}$



Here, y_k is *a linear function of* $z_1, \ldots, z_N$

$$y_k = \sum_{n=1}^{N} \mathbf{A}[k,n] \, z_n$$

where $\mathbf{A}[k,n]$ is entry of $\mathbf{A}$ at row k and column n. We thus can write

$$\frac{\partial \hat{R}}{\partial z_n} = \sum_{k=1}^{K} \frac{\partial \hat{R}}{\partial y_k} \frac{\partial y_k}{\partial z_n} = \sum_{k=1}^{K} \frac{\partial \hat{R}}{\partial y_k} \mathbf{A}[k,n]$$

Backpropagation: *Local Operation 2*

Let's denote column n of $\mathbf{A}$ by notation $\mathbf{A}[:, n]$; so, we can write

$$\frac{\partial \hat{R}}{\partial z_n} = \sum_{k=1}^{K} \frac{\partial \hat{R}}{\partial y_k} \mathbf{A}[k, n] = \mathbf{A}[:, n]^{\mathsf{T}} \nabla_{\mathbf{y}} \hat{R}$$

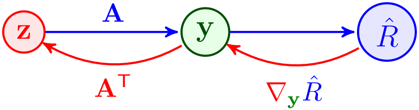
Now, if we collect them in a vector form we get

$$\nabla_{\mathbf{z}} \hat{R} = \begin{bmatrix} \partial \hat{R} / \partial z_1 \\ \vdots \\ \partial \hat{R} / \partial z_N \end{bmatrix} = \begin{bmatrix} \mathbf{A}[:, 1]^{\mathsf{T}} \\ \vdots \\ \mathbf{A}[:, N]^{\mathsf{T}} \end{bmatrix} \nabla_{\mathbf{y}} \hat{R} = \mathbf{A}^{\mathsf{T}} \nabla_{\mathbf{y}} \hat{R}$$

This makes sense! Since we are changing dimensions fro K to N, we need a product that does such dimensionality change for us

Backpropagation: *Local Operation 2*

Long story short . . .



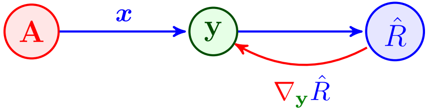
with backward step

$$\nabla_{\mathbf{z}} \hat{R} = \mathbf{A}^\top \nabla_{\mathbf{y}} \hat{R}$$

Backpropagation: *Local Operation 3*

Linear Matrix-to-Vector Operation

$\mathbf{y} \in \mathbb{R}^K$ *is a function of* $\mathbf{A} \in \mathbb{R}^{K \times N}$ *as* $\mathbf{y} = \mathbf{A}\boldsymbol{x}$ *with* $\boldsymbol{x} \in \mathbb{R}^N$



+ *Wait a moment! The other variable is a matrix! How do we define* $\nabla_{\mathbf{A}} \hat{R}$*?*
– Right! Let's first extend the definition

Backpropagation: *Local Operation 3*

Assume scalar $\hat{R}$ is a function of matrix $\mathbf{A} \in \mathbb{R}^{K \times N}$, we define

$$\nabla_{\mathbf{A}} \hat{R} = \begin{bmatrix} \partial \hat{R} / \partial \mathbf{A}[1,1] & \dots & \partial \hat{R} / \partial \mathbf{A}[1,N] \\ \vdots & & \vdots \\ \partial \hat{R} / \partial \mathbf{A}[K,1] & \dots & \partial \hat{R} / \partial \mathbf{A}[K,N] \end{bmatrix}$$

with $\mathbf{A}[k,n]$ being the entry of $\mathbf{A}$ at row k and column n

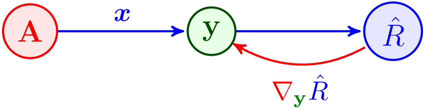
It is worth to also think of gradient descent in this case: *assume we are minimizing $\hat{R}$ over $\mathbf{A}$ using gradient descent with learning rate η. At iteration t we got point $\mathbf{A}^{(t)}$; now, in the next iteration we can readily write*

$$\mathbf{A}^{(t+1)} = \mathbf{A}^{(t)} - \nabla_{\mathbf{A}} \hat{R}|_{\mathbf{A}=\mathbf{A}^{(t)}}$$

so apparently everything is as before!

Backpropagation: *Local Operation 3*

Back to our problem, we can write



Entry k of $\mathbf{y}$ is a linear function of the k-th row of $\mathbf{A}$, i.e.,

$$y_k = \sum_{n=1}^{N} x_n \mathbf{A}[k, n]$$

So, we can write

$$\frac{\partial \hat{R}}{\partial \mathbf{A}[j, n]} = \sum_{k=1}^{K} \frac{\partial \hat{R}}{\partial y_k} \frac{\partial y_k}{\partial \mathbf{A}[j, n]} = \sum_{k=1}^{K} \frac{\partial \hat{R}}{\partial y_k} \mathbf{1}\{k = j\} x_n = \frac{\partial \hat{R}}{\partial y_j} x_n$$

Backpropagation: *Local Operation 3*

Let's now put them in a matrix

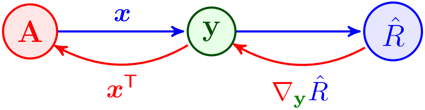
$$\nabla_{\mathbf{A}} \hat{R} = \begin{bmatrix} \frac{\partial \hat{R}}{\partial y_1} x_1 & \dots & \frac{\partial \hat{R}}{\partial y_1} x_N \\ \vdots & & \vdots \\ \frac{\partial \hat{R}}{\partial y_K} x_1 & \dots & \frac{\partial \hat{R}}{\partial y_K} x_N \end{bmatrix} = \nabla_{\mathbf{y}} \hat{R} \boldsymbol{x}^{\mathsf{T}}$$

So, we should now apply outer product! This again makes sense!

We have a K-dimensional gradient $\nabla_{\mathbf{y}} \hat{R}$ and an N-dimensional vector $\boldsymbol{x}$, we need an outer product to convert it into the $K \times N$ matrix $\nabla_{\mathbf{A}} \hat{R}$

Backpropagation: *Local Operation 3*

So, we could conclude



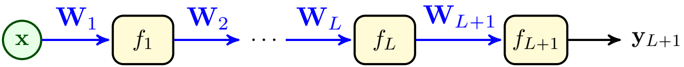
with the backward step

$$\nabla_{\mathbf{A}} \hat{R} = \nabla_{\mathbf{y}} \hat{R} \boldsymbol{x}^\mathsf{T}$$

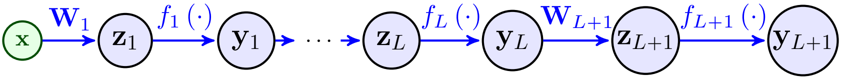
Now, we are ready to "backpropagate" over an FNN

Backpropagation: *Algorithm*

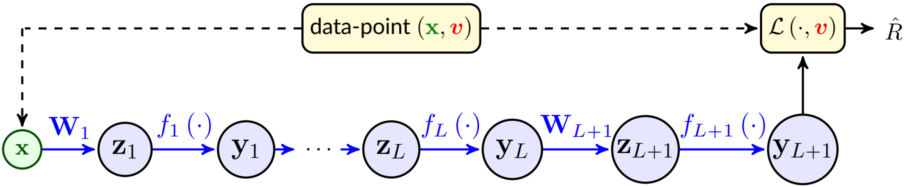
Let's recall the compact diagram of an FNN with L hidden layers



We can easily expand it into a *computation graph*

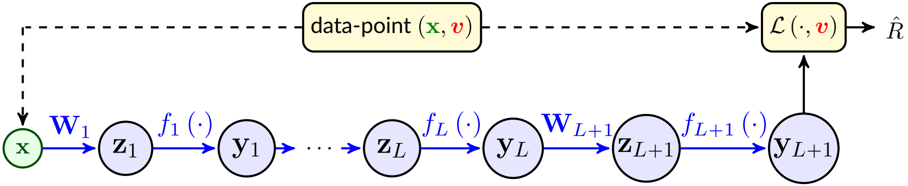


Our objective is *the empirical risk*; so let's include it also in the graph



Backpropagation: *Algorithm*

Given data-point $\mathbf{x}$ and its true label v, we once complete a forward pass



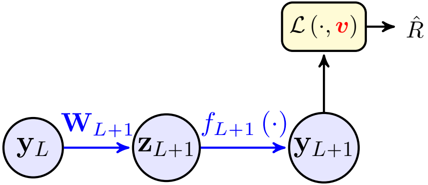
At the end of forward pass,

we know the value of all variables, i.e., $\mathbf{z}_\ell$ and $\mathbf{y}_\ell$ for all ℓ

Now, let's assume we want to find $\nabla_{\mathbf{W}_{L+1}} \hat{R}$

Backpropagation: *Algorithm*

We now cut the graph at the link $\mathbf{W}_{L+1}$



Let's recall . . .

+ *what is the variable here?*
– It's $\mathbf{W}_{L+1}$
+ *Can we modify the graph such that it becomes a node?*
– Sure! We note that $\mathbf{z}_{L+1} = \mathbf{W}_{L+1}\mathbf{y}_L$. We can look at it as a linear *matrix-to-vector* operation; so, we could modify the graph as

Backpropagation: *Algorithm*

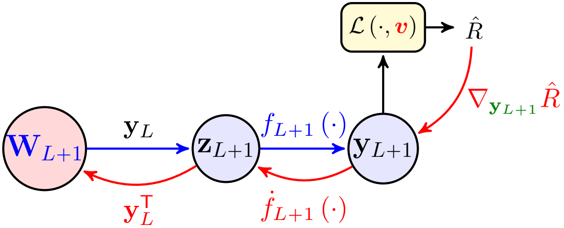
Let's now move backward to $\mathbf{W}_{L+1}$

1. *We have* $\mathbf{y}_{L+1}$*, so we compute* $\nabla_{\mathbf{y}_{L+1}} \hat{R}$*, and pass it*
2. *We have* $\mathbf{z}_{L+1}$*, so we compute* $\dot{f}_{L+1}(\mathbf{z}_{L+1})$*, and then we get*

$$\nabla_{\mathbf{z}_{L+1}} \hat{R} = \nabla_{\mathbf{y}_{L+1}} \hat{R} \odot \dot{f}_{L+1}(\mathbf{z}_{L+1})$$

3. *We have* $\mathbf{y}_L$*, so we compute* $\nabla_{\mathbf{W}_{L+1}} \hat{R}$ *from the last pass as*

$$\nabla_{\mathbf{W}_{L+1}} \hat{R} = \nabla_{\mathbf{z}_{L+1}} \hat{R} \, \mathbf{y}_L^{\mathsf{T}}$$

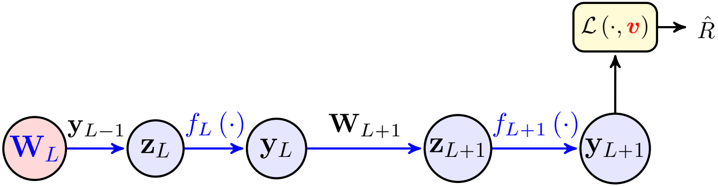


Backpropagation: *Algorithm*

We can *propagate* backward deeper and deeper

- *We cut at the link that we want to compute the gradient with respect to*
- *We exchange the linear vector-to-vector function at that particular link to a linear matrix-to-vector function*
- *We move backwards till we get to the source of this graph*

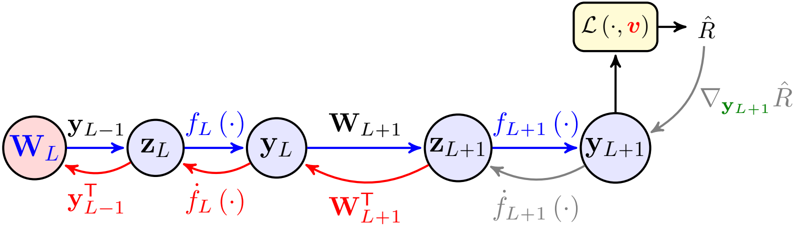
Let's see the example for $\mathbf{W}_L$



Backpropagation: *Algorithm*

We already have passed backward messages till $\mathbf{z}_{L+1}$: we have $\nabla_{\mathbf{z}_{L+1}} \hat{R}$

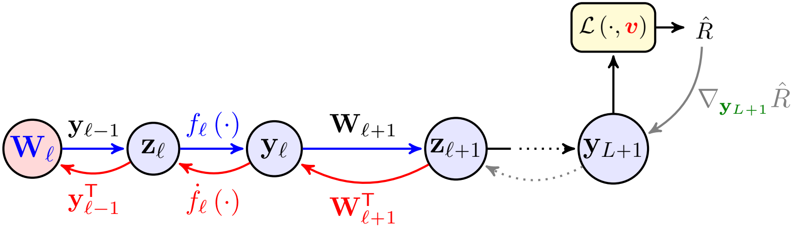
1. *We now pass* $\nabla_{\mathbf{z}_{L+1}} \hat{R}$ *to* $\mathbf{y}_L$: $\nabla_{\mathbf{y}_L} \hat{R} = \mathbf{W}_{L+1}^{\mathsf{T}} \nabla_{\mathbf{z}_{L+1}} \hat{R}$
2. *We then pass* $\nabla_{\mathbf{y}_L} \hat{R}$ *to* $\mathbf{z}_L$: $\nabla_{\mathbf{z}_L} \hat{R} = \nabla_{\mathbf{y}_L} \hat{R} \odot \dot{f}_L$
   - ↳ *We should remove entry of* $\nabla_{\mathbf{y}_L} \hat{R}$ *at index* 0: *we don't want* $\partial \hat{R} / \partial y_L [0]$
3. *We finally pass* $\nabla_{\mathbf{z}_L} \hat{R}$ *to* $\mathbf{W}_L$: $\nabla_{\mathbf{W}_L} \hat{R} = \nabla_{\mathbf{z}_L} \hat{R}\, \mathbf{y}_{L-1}^{\mathsf{T}}$



Backpropagation: *Algorithm*

As we arrive backward at layer ℓ, we already have messages till $\mathbf{z}_{\ell+1}$

1. *We pass* $\nabla_{\mathbf{z}_{\ell+1}} \hat{R}$ *to* $\mathbf{y}_\ell$: $\nabla_{\mathbf{y}_\ell} \hat{R} = \mathbf{W}_{\ell+1}^{\mathsf{T}} \nabla_{\mathbf{z}_{\ell+1}} \hat{R}$
2. *We then pass* $\nabla_{\mathbf{y}_\ell} \hat{R}$ *to* $\mathbf{z}_\ell$: $\nabla_{\mathbf{z}_\ell} \hat{R} = \nabla_{\mathbf{y}_\ell} \hat{R} \odot \dot{f}_\ell$
   - ↳ *We should remove entry of* $\nabla_{\mathbf{y}_\ell} \hat{R}$ *at index* 0: *we don't want* $\partial \hat{R} / \partial y_\ell [0]$
3. *We finally pass* $\nabla_{\mathbf{z}_\ell} \hat{R}$ *to* $\mathbf{W}_\ell$: $\nabla_{\mathbf{W}_\ell} \hat{R} = \nabla_{\mathbf{z}_\ell} \hat{R} \mathbf{y}_{\ell-1}^{\mathsf{T}}$



Once we propagate back to the input, i.e., $\ell = 1$, *then we have all gradients!*

Backpropagation: *Few Notations*

To formally present backpropagation, *let us define a few notations*

For $\ell = 1, \ldots, L+1$, we define

$$\overleftarrow{\mathbf{y}}_\ell = \nabla_{\mathbf{y}_\ell} \hat{R}$$
$$\overleftarrow{\mathbf{z}}_\ell = \nabla_{\mathbf{z}_\ell} \hat{R}$$

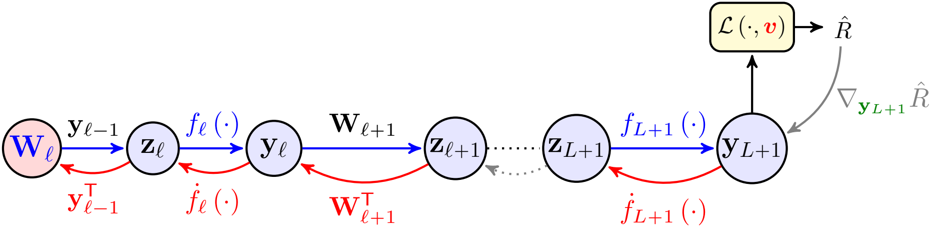
and keep in mind that

- $\mathbf{y}_\ell$ *and* $\overleftarrow{\mathbf{y}}_\ell$ *are totally different things*
- $\mathbf{z}_\ell$ *and* $\overleftarrow{\mathbf{z}}_\ell$ *are totally different things*

Backpropagation: *Pseudo Code*

```
Initiate with ←y_{L+1} = ∇L(y_{L+1}, v) and ←z_{L+1} = ←y_{L+1} ⊙ ḟ_{L+1}(z_{L+1})
for ℓ = L, ..., 1 do
    Determine ←y_ℓ = W^T_{ℓ+1} ←z_{ℓ+1} and drop ←y_ℓ[0]          # backward affine
    Determine ←z_ℓ = ḟ_ℓ(z_ℓ) ⊙ ←y_ℓ                              # backward activation
end for
for ℓ = 1, ..., L + 1 do
    Return ∇_{W_ℓ} R̂ = ←z_ℓ y^T_{ℓ−1}
end for
```

Where:
- $\overleftarrow{\mathbf{y}}_{L+1} = \nabla \mathcal{L}(\mathbf{y}_{L+1}, \boldsymbol{v})$ and $\overleftarrow{\mathbf{z}}_{L+1} = \overleftarrow{\mathbf{y}}_{L+1} \odot \dot{f}_{L+1}(\mathbf{z}_{L+1})$
- $\overleftarrow{\mathbf{y}}_{\ell} = \mathbf{W}_{\ell+1}^{\mathsf{T}} \overleftarrow{\mathbf{z}}_{\ell+1}$, then drop $\overleftarrow{y}_{\ell}[0]$
- $\overleftarrow{\mathbf{z}}_{\ell} = \dot{f}_{\ell}(\mathbf{z}_{\ell}) \odot \overleftarrow{\mathbf{y}}_{\ell}$
- $\nabla_{\mathbf{W}_\ell} \hat{R} = \overleftarrow{\mathbf{z}}_{\ell} \mathbf{y}_{\ell-1}^{\mathsf{T}}$



Backpropagation: *Pseudo Code*

Initiate with $\overleftarrow{\mathbf{y}}_{L+1} = \nabla \mathcal{L}(\mathbf{y}_{L+1}, \boldsymbol{v})$ and $\overleftarrow{\mathbf{z}}_{L+1} = \overleftarrow{\mathbf{y}}_{L+1} \odot \dot{f}_{L+1}(\mathbf{z}_{L+1})$
for $\ell = L, \ldots, 1$ **do**
    Determine $\overleftarrow{\mathbf{y}}_{\ell} = \mathbf{W}_{\ell+1}^{\mathsf{T}} \overleftarrow{\mathbf{z}}_{\ell+1}$ and drop $\overleftarrow{y}_{\ell}[0]$ `# backward affine`
    Determine $\overleftarrow{\mathbf{z}}_{\ell} = \dot{f}_{\ell}(\mathbf{z}_{\ell}) \odot \overleftarrow{\mathbf{y}}_{\ell}$ `# backward activation`
end for
for $\ell = 1, \ldots, L+1$ **do**
    Return $\nabla_{\mathbf{W}_{\ell}} \hat{R} = \overleftarrow{\mathbf{z}}_{\ell} \mathbf{y}_{\ell-1}^{\mathsf{T}}$
end for

+ *This looks very similar to forward propagation! Right?!*
– Yeah! Just we go *backward*! That's the whole point of backpropagation

You need to go once forth and then back to determine all gradients

Let's put them next to each other

Backpropagation: *Pseudo Code*

`ForwardProp():`

Initiate with $\mathbf{y}_0 = \mathbf{x}$
for $\ell = 0, \ldots, L$ **do**
    Add $y_\ell[0] = 1$ and determine $\mathbf{z}_{\ell+1} = \mathbf{W}_{\ell+1}\mathbf{y}_\ell$ `# forward affine`
    Determine $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$ `# forward activation`
end for
for $\ell = 1, \ldots, L+1$ **do**
    Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
end for

`BackProp():`

Initiate with $\overleftarrow{\mathbf{y}}_{L+1} = \nabla \mathcal{L}(\mathbf{y}_{L+1}, \boldsymbol{v})$ and $\overleftarrow{\mathbf{z}}_{L+1} = \overleftarrow{\mathbf{y}}_{L+1} \odot \dot{f}_{L+1}(\mathbf{z}_{L+1})$
for $\ell = L, \ldots, 1$ **do**
    Determine $\overleftarrow{\mathbf{y}}_\ell = \mathbf{W}_{\ell+1}^{\mathsf{T}} \overleftarrow{\mathbf{z}}_{\ell+1}$ and drop $\overleftarrow{y}_\ell[0]$ `# backward affine`
    Determine $\overleftarrow{\mathbf{z}}_\ell = \dot{f}_\ell(\mathbf{z}_\ell) \odot \overleftarrow{\mathbf{y}}_\ell$ `# backward activation`
end for
for $\ell = 1, \ldots, L+1$ **do**
    Return $\nabla_{\mathbf{W}_\ell} \hat{R} = \overleftarrow{\mathbf{z}}_\ell \mathbf{y}_{\ell-1}^{\mathsf{T}}$
end for

Complete Training Loop via *Gradient Descent*

+ *Say we use backpropagation; then, how does gradient descent look?*
– Well! We should go back and forth with all data-points

Say we have dataset

$$\mathbb{D} = \{(\boldsymbol{x}_b, \boldsymbol{v}_b) \text{ for } b = 1, \dots, B\}$$

`GradientDescent():`

Initiate with some initial values $\{\mathbf{W}_\ell^{(0)}\}$ and set a learning rate η
while weights not converged **do**
  for $b = 1, \dots, B$ **do**
    $\texttt{NN.values} \leftarrow \texttt{ForwardProp}\left(\boldsymbol{x}_b, \{\mathbf{W}_\ell^{(t)}\}\right)$ `# forward`
    $\{\nabla_{\mathbf{W}_\ell{}^{(t)}} \hat{R}_b\} \leftarrow \texttt{BackProp}\left(\boldsymbol{x}_b, \boldsymbol{v}_b, \{\mathbf{W}_\ell^{(t)}\}, \texttt{NN.values}\right)$ `# backward`
  end for
  for $\ell = 1, \dots, L+1$ **do**
    $\mathbf{W}_\ell{}^{(t+1)} \leftarrow \mathbf{W}_\ell{}^{(t)} - \eta \; \texttt{mean}(\nabla_{\mathbf{W}_\ell{}^{(t)}} \hat{R}_1, \dots, \nabla_{\mathbf{W}_\ell{}^{(t)}} \hat{R}_B)$
  end for
end while