ECE 1508S2: Applied Deep Learning

Chapter 2: Feedforward Neural Networks

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2024

Various Architectures for NNs

Let's abbreviate the term *Neural Network* from now on with *NN*

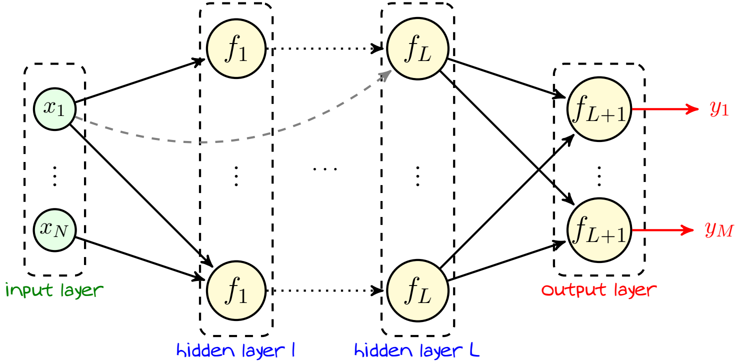
Now that we know *what they are* and *how to train them*, we are going through the famous architectures for NNs which are

1. *Feedforward NNs* abbreviated as FNNs
   ↳ Some people call them also *Multi-Layer Perceptrons (MLPs)*: *you may say that this is a misnomer and you are right! Check the wikipedia page*
2. *Convolutional NNs* abbreviated as CNNs
3. *Recurrent NNs* abbreviated as RNNs

In this chapter, we start with FNNs which are known to be *vanilla NNs*, *i.e.*, *the most basic architecture we could think for a NN*

FNNs: *Architecture*

In FNNs, the *inputs flow in one direction:* each layer's output is connected to the next layers, and thus *we do not have any feedback*



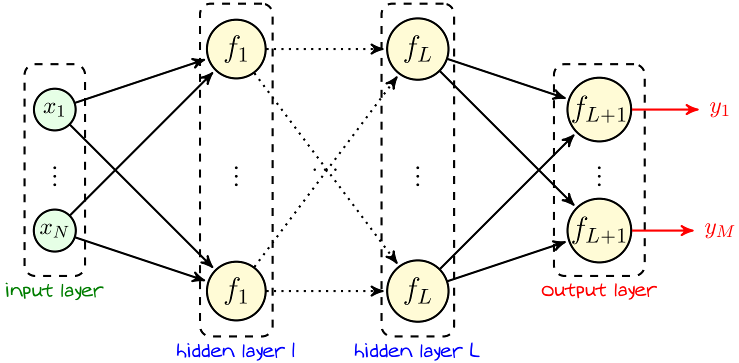
Though it is not a must, we usually use same activation for all neurons in a layer

Fully-Connected FNNs

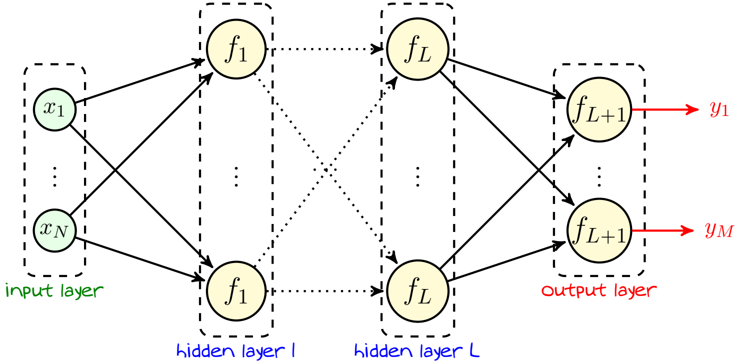
We start with the most straightforward FNNs: *fully-connected FNNs*

Fully-Connected FNNs

In a fully-connected FNN, each node is connected to all nodes in the next layer



Fully-Connected FNNs: *Few Definitions*



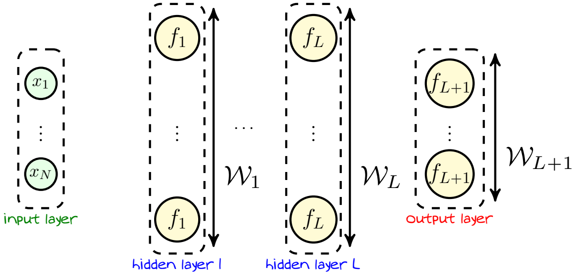
In this FNN, we have L hidden layers; thus, its depth is $L+1$

***Recall:** this network is Deep if $L > 1$*

Fully-Connected FNNs: *Few Definitions*

Width of a Layer

The width of layer ℓ is the number of neurons in layer ℓ



Some people call the largest width, the width of the network, i.e.,

$$\mathcal{W} = \max_{\ell \in \{1,\ldots,L+1\}} \mathcal{W}_\ell$$

Fully-Connected FNNs: *Looking as a Model*

+ *We should look at a fully-connected FNN as a model. Then, what are the hyperparameters and learnable parameters?*
– I am glad that you ask! Let's take a look

Assume that someone tells us that we should *use a fully-connected FNN* with only *ReLU activation*. Now, we could say

- To write down the model, we need to know *the number of hidden layers* L and *width of each layer* $\mathcal{W}_\ell$: these are the *hyperparameters*
- If we set the L and $\mathcal{W}_\ell$, we can specify the *learnable parameters*
  - *in hidden layer* 1*, we have* $\mathcal{W}_1$ *neurons each having* N *weights and a bias*
  - *in hidden layer* 2*, we have* $\mathcal{W}_2$ *neurons each having* $\mathcal{W}_1$ *weights and a bias*

  $\vdots$
  - *in output layer, we have* $\mathcal{W}_{L+1}$ *neurons each having* $\mathcal{W}_L$ *weights and a bias*

$$\text{\# model parameters} = (N+1)\,\mathcal{W}_1 + \sum_{\ell=1}^{L} (\mathcal{W}_\ell + 1)\,\mathcal{W}_{\ell+1}$$

Fully-Connected FNNs: *Forward Pass*

Let us first see *how a given data-point propagates through the FNN*: we want to write the outputs $y_1, \dots, y_M$ when inputs $x_1, \dots, x_N$ are given

this is called *forward propagation* through the network

or simply

the *forward pass*

which tracks values *passed* through the NN from the *input* to output layer

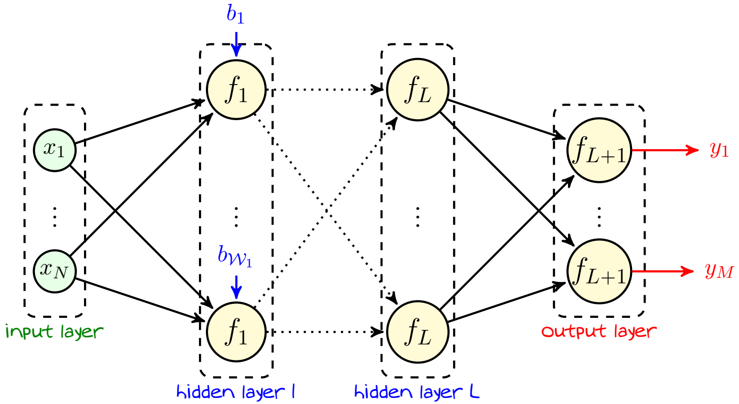
---

To present the forward pass compactly, we need to *define some notations* and apply *some modifications* in the network

Fully-Connected FNNs: *Few Definitions*

We can get rid of biases by defining a new constant node in each layer

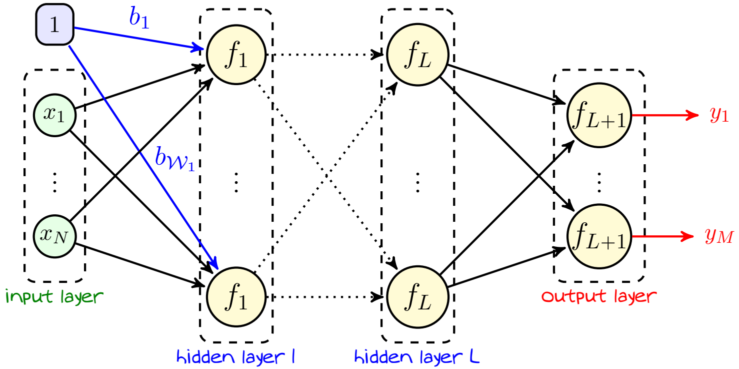
Let's look at the first layer: *we have $\mathcal{W}_1$ neurons and each has a bias*



Fully-Connected FNNs: *Few Definitions*

We can get rid of biases by defining a new constant node in each layer

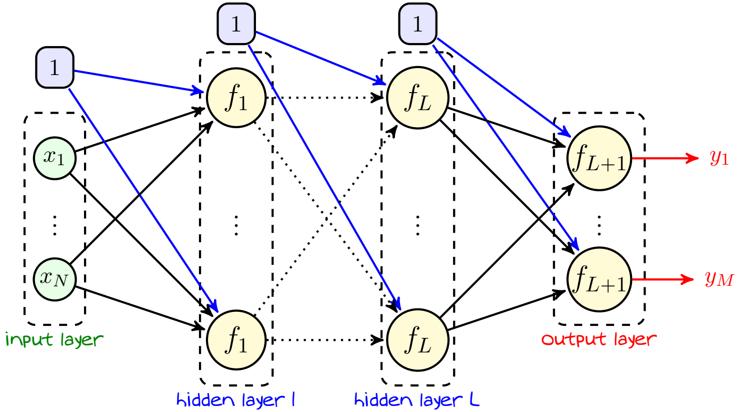
We introduce a constant input and let these biases being the weights of its links



Fully-Connected FNNs: *Few Definitions*

We can get rid of biases by defining a new constant node in each layer

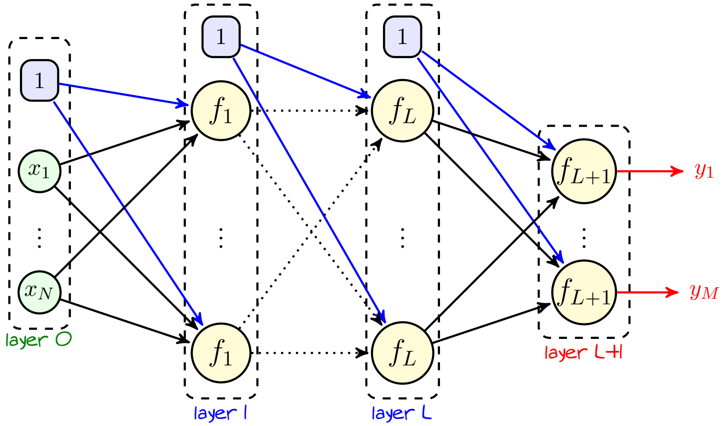
We do the same in all layers: now neurons have no biases



Fully-Connected FNNs: *Few Definitions*

We next give an index to each layer each layer

- Input layer is *layer 0*
- Hidden layer ℓ is *layer* ℓ
- Output layer is *layer* $L+1$

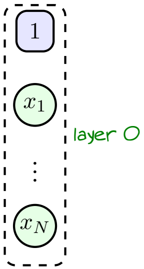


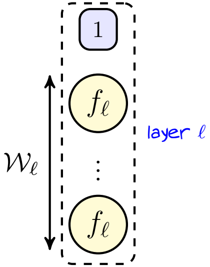
Fully-Connected FNNs: *Few Definitions*

We next give an index to each layer each layer

So, our layers are indexed by $\ell \in \{0, \dots, L+1\}$

- We denote the width of layer ℓ with $\mathcal{W}_\ell$
  - ↳ *For $\ell \geqslant 1$ this is exactly the layer width $\equiv$ # of neurons in the layer*
  - ↳ *For $\ell = 0$ this is the number of inputs, i.e., $\mathcal{W}_0 = N$*
- In layer ℓ, we have $\mathcal{W}_\ell + 1$ nodes
  - ↳ *$\mathcal{W}_\ell$ neurons (or inputs if $\ell = 0$)*
  - ↳ *One constant node that always returns 1*





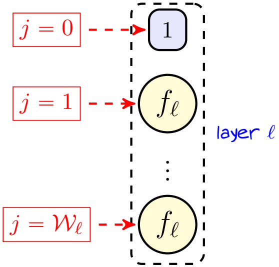
Fully-Connected FNNs: *Few Definitions*

We next index the nodes in each layer

In layer ℓ: *we have* $\mathcal{W}_\ell + 1$ *nodes*

↳ *One constant node* $\equiv$ *node* $j = 0$

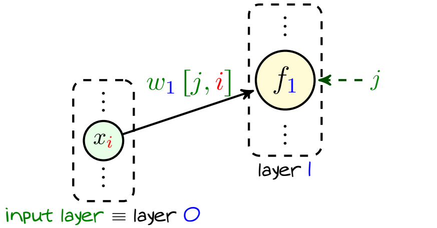
↳ $\mathcal{W}_\ell$ *neurons/inputs* $\equiv$ *node* $j = 1, \ldots, \mathcal{W}_\ell$

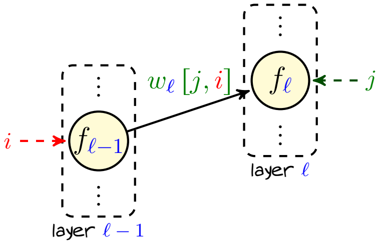


Fully-Connected FNNs: *Forward Pass*

We next give weights to the links

Weight of the link connecting node i in layer $\ell - 1$ to node j in layer ℓ is denoted by $w_\ell [j, i]$





Fully-Connected FNNs: *Forward Pass*

We next give weights to the links

Weight of the link connecting node i in layer $\ell - 1$ to node j in layer ℓ is denoted by $w_\ell[j, i]$

↳ The weights coming out of $i = 0$ are *biases*

$$\boxed{w_\ell[j, 0] \text{ is the bias of neuron } j \text{ in layer } \ell}$$

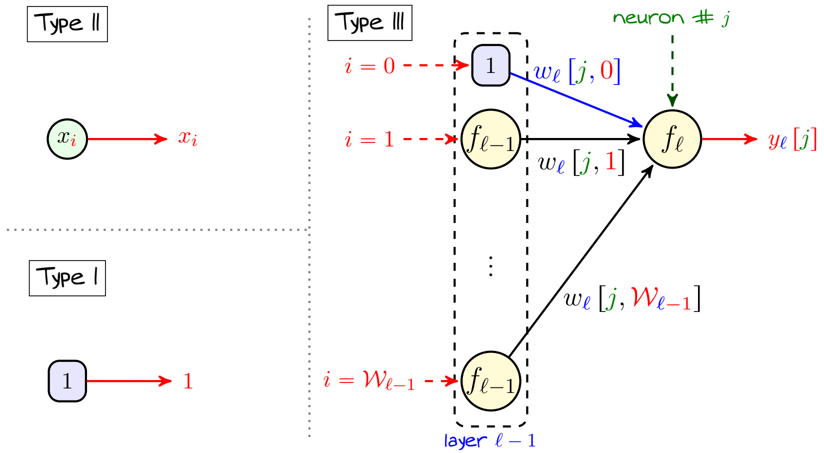
↳ There is no link from a node to a *constant node*: $w_\ell[j, i]$ *exists for*

- $i = 0, \ldots, \mathcal{W}_{\ell-1}$
- $j = 1, \ldots, \mathcal{W}_\ell$

This means that *there exists no such a weight* $w_\ell[0, i]$

Fully-Connected FNNs: *Forward Pass*

We finally specify the output of each node



Fully-Connected FNNs: *Forward Pass*

We finally specify the output of each node

We represent the output of node j in layer ℓ with $y_\ell\left[j\right]$

↳ Since $j = 0$ is the *constant node*: $y_\ell\left[0\right] = 1$ for $\ell = 0, \ldots, L+1$

↳ Since $\ell = 0$ is the *input layer*: $y_0\left[j\right] = x_j$ for $j = 1, \ldots, N$

↳ For neuron j in layer ℓ we can write

$$y_\ell\left[j\right] = f_\ell(z_\ell\left[j\right])$$

where $z_\ell\left[j\right]$ is the *output of the affine function* in neuron j

Fully-Connected FNNs: *Forward Pass*

We finally specify the output of each node

We represent the output of node j in layer ℓ with $y_\ell [j]$

↳ Since $j = 0$ is the *constant node*: $y_\ell [0] = 1$ for $\ell = 0, \dots, L+1$

↳ Since $\ell = 0$ is the *input layer*: $y_0 [j] = x_j$ for $j = 1, \dots, N$

↳ For neuron j in layer ℓ we can write

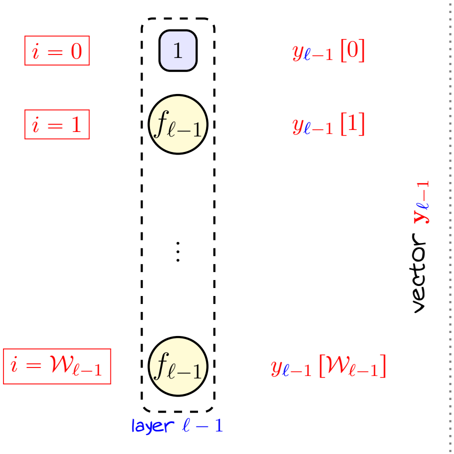
$$y_\ell [j] = f_\ell(z_\ell [j])$$

where $z_\ell [j]$ is the *output of the affine function* in neuron j

$$\begin{aligned} z_\ell [j] &= w_\ell [j, 0] + \sum_{i=1}^{\mathcal{W}_{\ell-1}} w_\ell [j, i] \, y_{\ell-1} [i] \\ &= \sum_{i=0}^{\mathcal{W}_{\ell-1}} w_\ell [j, i] \, y_{\ell-1} [i] \end{aligned}$$

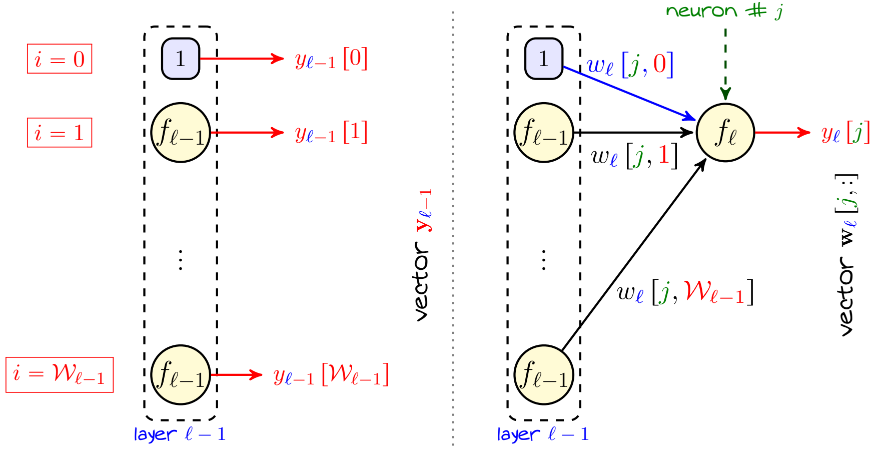
Fully-Connected FNNs: *Forward Pass*

We finally specify the output of each node


$i = 0$
$i = 1$
$i = \mathcal{W}_{\ell-1}$
1
$f_{\ell-1}$
$f_{\ell-1}$
layer $\ell - 1$
$y_{\ell-1}[0]$
$y_{\ell-1}[1]$
$y_{\ell-1}[\mathcal{W}_{\ell-1}]$
vector $\mathbf{y}_{\ell-1}$

Fully-Connected FNNs: *Forward Pass*

We finally specify the output of each node


$i = 0$
$i = 1$
$i = \mathcal{W}_{\ell-1}$
1
$f_{\ell-1}$
$y_{\ell-1}[0]$
$y_{\ell-1}[1]$
$y_{\ell-1}[\mathcal{W}_{\ell-1}]$
layer $\ell - 1$
vector $\mathbf{y}_{\ell-1}$
neuron # j
$w_\ell[j,0]$
$w_\ell[j,1]$
$w_\ell[j,\mathcal{W}_{\ell-1}]$
f_ℓ
$y_\ell[j]$
vector $\mathbf{w}_\ell[j,:]$
layer $\ell - 1$

Fully-Connected FNNs: *Forward Pass*

We finally specify the output of each node

We can represent $z_j[\ell]$ *more compactly via vectorized notation*

$$
\begin{aligned}
z_\ell[j] &= \sum_{i=0}^{\mathcal{W}_{\ell-1}} w_\ell[i,j]\, y_{\ell-1}[i] \\
&= \underbrace{\begin{bmatrix} w_\ell[j,0] & w_\ell[j,1] & \dots & w_\ell[j,\mathcal{W}_{\ell-1}] \end{bmatrix}}_{\mathbf{w}_\ell[j,:]^\mathsf{T}} \underbrace{\begin{bmatrix} y_{\ell-1}[0] \\ y_{\ell-1}[1] \\ \vdots \\ y_{\ell-1}[\mathcal{W}_{\ell-1}] \end{bmatrix}}_{\mathbf{y}_\ell} \\
&= \mathbf{w}_\ell[j,:]^\mathsf{T}\, \mathbf{y}_{\ell-1}
\end{aligned}
$$

Fully-Connected FNNs: *Forward Pass*

We finally specify the output of each node

We can further extend vectorized notation by defining

$$\mathbf{z}_\ell = \begin{bmatrix} z_\ell[1] \\ \vdots \\ z_\ell[\mathcal{W}_\ell] \end{bmatrix}$$

and thus writing $\mathbf{y}_\ell$ as

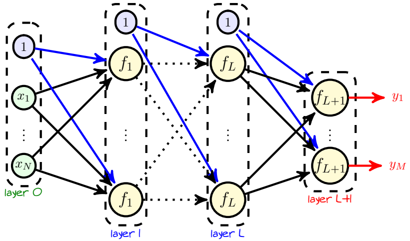
$$\mathbf{y}_\ell = f_\ell(\mathbf{z}_\ell)$$

where $f_\ell(\cdot)$ *is applied entry-wise*

and don't forget to add the dummy input 1, *i.e.,*

$$\mathbf{y}_\ell \leftarrow \begin{bmatrix} 1 \\ \mathbf{y}_\ell \end{bmatrix}$$

Fully-Connected FNNs: *Forward Pass*



```
 1: Initiate the output of the first layer as $\mathbf{y}_0 = \boldsymbol{x}$
 2: for $\ell = 0, \dots, L$ do
 3:     for $j = 1, \dots, \mathcal{W}_\ell$ do
 4:         Add $y_\ell[0] = 1$ and set $z_{\ell+1}[j] = \mathbf{w}_{\ell+1}[j,:]^\mathsf{T} \mathbf{y}_\ell$    # affine function
 5:     end for
 6:     Compute $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$                                    # activation
 7: end for
 8: for $\ell = 1, \dots, L+1$ do
 9:     Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
10: end for
```

Fully-Connected FNNs: *Forward Pass*

We can present everything even more compactly: *let's write down* $\mathbf{z}_\ell$

$$\mathbf{z}_\ell = \begin{bmatrix} z_\ell[1] \\ \vdots \\ z_\ell[\mathcal{W}_\ell] \end{bmatrix} = \begin{bmatrix} \mathbf{w}_\ell[1,:]^\mathsf{T}\, \mathbf{y}_{\ell-1} \\ \vdots \\ \mathbf{w}_\ell[\mathcal{W}_\ell,:]^\mathsf{T}\, \mathbf{y}_{\ell-1} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_\ell[1,:]^\mathsf{T} \\ \vdots \\ \mathbf{w}_\ell[\mathcal{W}_\ell,:]^\mathsf{T} \end{bmatrix} \mathbf{y}_{\ell-1}$$

Now, we can define the matrix $\mathbf{W}_\ell$ *as*

$$\mathbf{W}_\ell = \begin{bmatrix} \mathbf{w}_\ell[1,:]^\mathsf{T} \\ \vdots \\ \mathbf{w}_\ell[\mathcal{W}_\ell,:]^\mathsf{T} \end{bmatrix} = \begin{bmatrix} w_\ell[1,0] & \dots & w_\ell[1,\mathcal{W}_{\ell-1}] \\ \vdots & & \vdots \\ w_\ell[\mathcal{W}_\ell,0] & \dots & w_\ell[\mathcal{W}_\ell,\mathcal{W}_{\ell-1}] \end{bmatrix}$$

This matrix collects all learning parameters of layer ℓ

Note that $\mathbf{W}_\ell$ *has* $\mathcal{W}_\ell$ *rows and* $\mathcal{W}_{\ell-1}+1$ *columns*

Forward Propagation: *Pseudo Code*

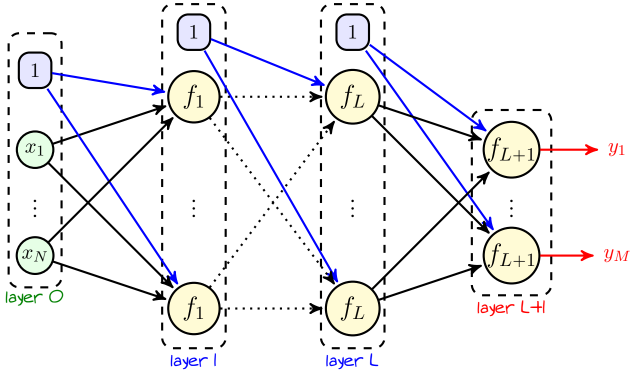
So, we can compactly present the forward propagation algorithm as follow

```
ForwardProp():
1: Initiate with $\mathbf{y}_0 = \boldsymbol{x}$
2: for $\ell = 0, \ldots, L$ do
3:     Add $y_\ell[0] = 1$ and determine $\mathbf{z}_{\ell+1} = \mathbf{W}_{\ell+1}\mathbf{y}_\ell$        # forward affine
4:     Determine $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$                 # forward activation
5: end for
6: for $\ell = 1, \ldots, L+1$ do
7:     Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
8: end for
```

After getting data-point $\boldsymbol{x}$, we convert it to $\mathbf{x}$ by adding an entry 1 at its index zero. We then pass it through a linear layer whose weights are learnable and a nonlinear transform that is specified by activation. The output of this layer passes forward to the next layer till we get to the output.

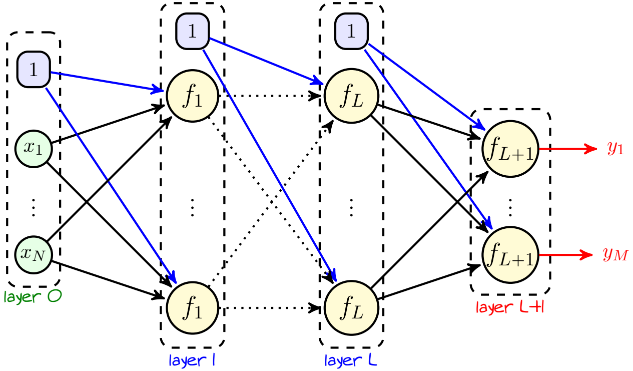
Forward Propagation: *Compact Diagram*

Inspired by forward propagation, we can represent the FNN



Forward Propagation: *Compact Diagram*

Inspired by forward propagation, we can represent the FNN



by the following compact diagram

$$\begin{tikzcd} \mathbf{x} \arrow[r, "\mathbf{W}_1"] & \boxed{f_1} \arrow[r, dotted] & \boxed{f_L} \arrow[r, "\mathbf{W}_{L+1}"] & \boxed{f_{L+1}} \arrow[r] & \mathbf{y}_{L+1} \end{tikzcd}$$

Forward Propagation: *Compact Diagram*

$$\begin{tikzcd} \mathbf{x} \arrow[r, "\mathbf{W}_1"] & \boxed{f_1} \arrow[r, dotted] & \boxed{f_L} \arrow[r, "\mathbf{W}_{L+1}"] & \boxed{f_{L+1}} \arrow[r] & \mathbf{y}_{L+1} \end{tikzcd}$$

Here, we compactly represent layer ℓ as

$$\begin{tikzcd} \mathbf{y}_{\ell-1} \arrow[r, "\mathbf{W}_\ell \quad \mathbf{z}_\ell"] & \boxed{f_\ell} \arrow[r] & \mathbf{y}_\ell \end{tikzcd}$$

- *The link $\mathbf{W}_\ell$ represents the affine function of layer ℓ*
- *The block f_ℓ represents the activation of layer ℓ and adding $y_\ell[0] = 1$*
  - ↳ The input of this block can be considered $\mathbf{z}_\ell$
  - ↳ The output of this block can be considered $\mathbf{y}_\ell$

This compact diagram will come in handy when we derive backpropagation!

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.

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 *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 output PassF $(\boldsymbol{x}_b|\mathbf{w})$ from FNN with weights $\mathbf{w}$. This output is desired to be the true label $\boldsymbol{v}_b$

How do we do the training?

$$\mathbf{w}^\star = \underset{\mathbf{w}}{\operatorname{argmin}}\, \hat{R}(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}}\, \frac{1}{B} \sum_{b=1}^{B} \mathcal{L}\left(\textit{PassF}\left(\boldsymbol{x}_b|\mathbf{w}\right), \boldsymbol{v}_b\right) \qquad \text{(Training)}$$

Fully-Connected FNNs: *Training*

Let's recall how we train the network

$$\mathbf{w}^{\star} = \underset{\mathbf{w}}{\operatorname{argmin}}\, \hat{R}\left(\mathbf{w}\right) = \underset{\mathbf{w}}{\operatorname{argmin}}\, \frac{1}{B}\sum_{b=1}^{B} \mathcal{L}\left(\mathit{PassF}\left(\boldsymbol{x}_b | \mathbf{w}\right), \boldsymbol{v}_b\right) \qquad \text{(Training)}$$

Also, we recall the gradient descent algorithm

```
1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small $\epsilon$ and $\eta$, 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*

$$\mathbf{w} = \left[\begin{array}{c} \mathbf{w}_1[1,:] \\ \vdots \\ \mathbf{w}_1[\mathcal{W}_1,:] \\ \vdots \\ \mathbf{w}_{L+1}[1,:] \\ \vdots \\ \mathbf{w}_{L+1}[\mathcal{W}_{L+1},:] \end{array}\right] \begin{array}{l} \\ \text{layer } \ell = 1 \\ \\ \\ \\ \text{layer } \ell = L+1 \\ \\ \end{array} \dashrightarrow \mathbf{w}_\ell[j,:] = \begin{bmatrix} w_\ell[j,0] \\ w_\ell[j,1] \\ \vdots \\ w_\ell[j,\mathcal{W}_{\ell-1}] \end{bmatrix}$$

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* $i = 0, \ldots, \mathcal{W}_{\ell-1}$ *and* $j = 1, \ldots, \mathcal{W}_\ell$ *as* ℓ *runs over* $\ell = 1, \ldots, L+1$

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

$$= \frac{1}{B} \sum_{b=1}^{B} \boxed{\frac{\partial}{\partial w_{i,j}[\ell]} \mathcal{L}\left(\text{PassF}\left(\boldsymbol{x}_b | \mathbf{w}\right), \boldsymbol{v}_b\right)}$$

So, it's enough to develop an algorithm that determined the partial derivative for a $\boxed{\textit{single data-point.}}$ *The partial derivative is then the average of these point-wise derivatives.*

Fully-Connected FNNs: *Training*

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

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

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*

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

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*

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 $n = 1, \dots, N$; then, derivative of $y = f(z_N)$ with respect to x is given by

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

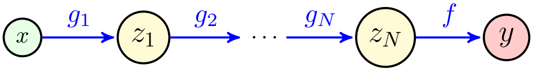
Computation Graph

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 $n = 1, \dots, N$; then, derivative of $y = f(z_N)$ with respect to x is given by

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

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



$$\begin{tikzcd} x \arrow[r, "g_1"] & z_1 \arrow[r, "g_2"] & \cdots \arrow[r, "g_N"] & z_N \arrow[r, "f"] & y \end{tikzcd}$$

In this graph, we start from x and pass forward to $z_1 \rightarrow z_2 \rightarrow \dots$ 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*

$$\begin{tikzcd} x \arrow[r, "g_1"] & z_1 \arrow[l, bend left, "\dot{g}_1"] \arrow[r, "g_2"] & \cdots \arrow[l, bend left, "\dot{g}_2"] \arrow[r, "g_N"] & z_N \arrow[l, bend left, "\dot{g}_N"] \arrow[r, "f"] & y \arrow[l, bend left, "\dot{f}"] \end{tikzcd}$$

Let's start from the last node

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

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*

$$\begin{tikzcd}
x \arrow[r, "g_1"] & z_1 \arrow[l, bend left, "\dot{g}_1"] \arrow[r, "g_2"] & \cdots \arrow[l, bend left, "\dot{g}_2"] \arrow[r, "g_N"] & z_N \arrow[l, bend left, "\dot{g}_N"] \arrow[r, "f"] & y \arrow[l, bend left, "\dot{f}"]
\end{tikzcd}$$

Let's start from the last node

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

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

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*

$$\begin{tikzcd}
x \arrow[r, "g_1"] & z_1 \arrow[l, bend left, "\dot{g}_1"] \arrow[r, "g_2"] & \cdots \arrow[l, bend left, "\dot{g}_2"] \arrow[r, "g_N"] & z_N \arrow[l, bend left, "\dot{g}_N"] \arrow[r, "f"] & y \arrow[l, bend left, "\dot{f}"]
\end{tikzcd}$$

Let's start from the last node

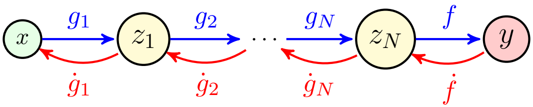
$$\frac{\mathrm{d}y}{\mathrm{d}z_N} = \dot{f}\left(z_N\right)$$

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

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

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*



$$\begin{tikzcd} x \arrow[r, "g_1"] & z_1 \arrow[l, bend left, "\dot{g}_1"] \arrow[r, "g_2"] & \cdots \arrow[l, bend left, "\dot{g}_2"] \arrow[r, "g_N"] & z_N \arrow[l, bend left, "\dot{g}_N"] \arrow[r, "f"] & y \arrow[l, bend left, "\dot{f}"] \end{tikzcd}$$

Let's start from the last node

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

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

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

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

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 $x = 0.5$.

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

$$\begin{tikzcd} x \arrow[r, "2x"] & z_1 \arrow[r, "z_1^2"] & z_2 \arrow[r, "e^{z_2}"] & z_3 \arrow[r, "\log z_3"] & y \end{tikzcd}$$

During the *forward pass* we get

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

$$\begin{tikzcd} 0.5 \arrow[r, "2x"] & 1 \arrow[r, "z_1^2"] & 1 \arrow[r, "e^{z_2}"] & e \arrow[r, "\log z_3"] & 1 \end{tikzcd}$$

Computation Graph: *Example*

Now, we *pass backward* to determine the derivative

$$\begin{tikzcd}
0.5 \arrow[r, "2x"] & 1 \arrow[r, "z_1^2"] \arrow[l, bend left, "2"] & 1 \arrow[r, "e^{z_2}"] \arrow[l, bend left, "2z_1"] & e \arrow[r, "\log z_3"] \arrow[l, bend left, "e^{z_2}"] & 1 \arrow[l, bend left, "1/z_3"]
\end{tikzcd}$$

Computation Graph: *Example*

Now, we *pass backward* to determine the derivative

$$\begin{tikzcd}
0.5 \arrow[r, "2x"] & 1 \arrow[r, "z_1^2"] \arrow[l, bend left, "2"] & 1 \arrow[r, "e^{z_2}"] \arrow[l, bend left, "2z_1"] & e \arrow[r, "\log z_3"] \arrow[l, bend left, "e^{z_2}"] & 1 \arrow[l, bend left, "1/z_3"]
\end{tikzcd}$$

Let's first enter the values into the backward links

$$\begin{tikzcd}
0.5 \arrow[r, "2x"] & 1 \arrow[r, "z_1^2"] \arrow[l, bend left, "2"] & 1 \arrow[r, "e^{z_2}"] \arrow[l, bend left, "2"] & e \arrow[r, "\log z_3"] \arrow[l, bend left, "e"] & 1 \arrow[l, bend left, "1/e"]
\end{tikzcd}$$

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

Computation Graph: *Example*

$$\begin{tikzcd}
0.5 \arrow[r, "2x"] & 1 \arrow[l, bend left, "2"] \arrow[r, "z_1^2"] & 1 \arrow[l, bend left, "2"] \arrow[r, "e^{z_2}"] & e \arrow[l, bend left, "e"] \arrow[r, "\log z_3"] & 1 \arrow[l, bend left, "1/e"]
\end{tikzcd}$$

The derivatives are easily determined *recursively*

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

Computation Graph: *Example*

$$\begin{tikzcd} 0.5 \arrow[r, "2x"] & 1 \arrow[l, bend left, "2"] \arrow[r, "z_1^2"] & 1 \arrow[l, bend left, "2"] \arrow[r, "e^{z_2}"] & e \arrow[l, bend left, "e"] \arrow[r, "\log z_3"] & 1 \arrow[l, bend left, "1/e"] \end{tikzcd}$$

The derivatives are easily determined *recursively*

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

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

Computation Graph: *Example*

$$\begin{tikzcd}
0.5 \arrow[r, "2x"] & 1 \arrow[l, bend left, "2"] \arrow[r, "z_1^2"] & 1 \arrow[l, bend left, "2"] \arrow[r, "e^{z_2}"] & e \arrow[l, bend left, "e"] \arrow[r, "\log z_3"] & 1 \arrow[l, bend left, "1/e"]
\end{tikzcd}$$

The derivatives are easily determined *recursively*

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

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

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

Computation Graph: *Example*

$$\begin{tikzcd}
0.5 \arrow[r, "2x"] & 1 \arrow[l, bend left, "2"] \arrow[r, "z_1^2"] & 1 \arrow[l, bend left, "2"] \arrow[r, "e^{z_2}"] & e \arrow[l, bend left, "e"] \arrow[r, "\log z_3"] & 1 \arrow[l, bend left, "1/e"]
\end{tikzcd}$$

The derivatives are easily determined *recursively*

$$\begin{aligned}
\frac{\mathrm{d}y}{\mathrm{d}z_3} &= \frac{1}{e} \\
\frac{\mathrm{d}y}{\mathrm{d}z_2} &= \frac{\mathrm{d}y}{\mathrm{d}z_3}\frac{\mathrm{d}z_3}{\mathrm{d}z_2} = \frac{e}{e} = 1 \\
\frac{\mathrm{d}y}{\mathrm{d}z_1} &= \frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1} = 1 \times 2 = 2 \\
\frac{\mathrm{d}y}{\mathrm{d}x} &= \frac{\mathrm{d}y}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}x} = 2 \times 2 = 4
\end{aligned}$$

Computation Graph

$$\begin{tikzcd}
x \arrow[r, "g_1"] & z_1 \arrow[l, bend left, "\dot{g}_1"] \arrow[r, "g_2"] & \cdots \arrow[l, bend left, "\dot{g}_2"] \arrow[r, "g_N"] & z_N \arrow[l, bend left, "\dot{g}_N"] \arrow[r, "f"] & y \arrow[l, bend left, "\dot{f}"]
\end{tikzcd}$$

In the example, we had to *first determine the value of each variable*, in order to be able to determine 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* $n = 1, \dots, N$

$$z_n = g_n(x)$$

and let the nested function be

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

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

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

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

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

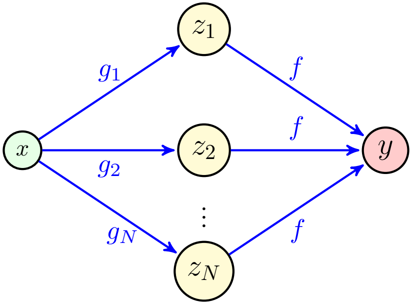
Chain Rule: *Vector Form*

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 vector form via its *computation graph*

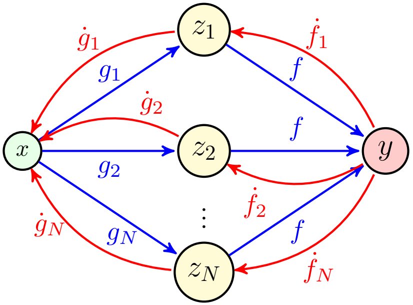


$$\begin{tikzcd}
 & z_1 \arrow[dr, "f"] & \\
x \arrow[ur, "g_1"] \arrow[r, "g_2"'] \arrow[dr, "g_N"'] & z_2 \arrow[r, "f"] & y \\
 & \vdots \\
 & z_N \arrow[uur, "f"'] &
\end{tikzcd}$$

In 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 determine next variable via the 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*

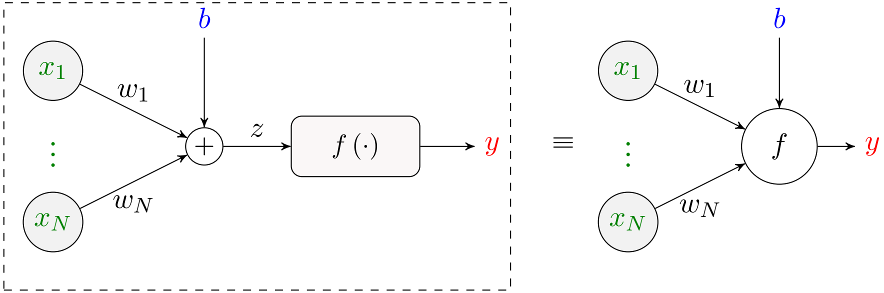


We add all backward passes towards x to determine the derivative

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

Computation Graph: *Single Neuron*

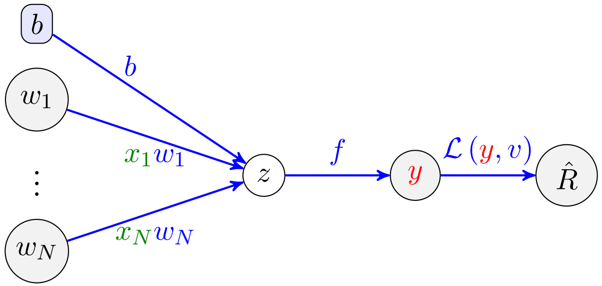
Let's now plot the *computation graph* of a single neuron and determine the gradient of the loss by *backward pass*



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

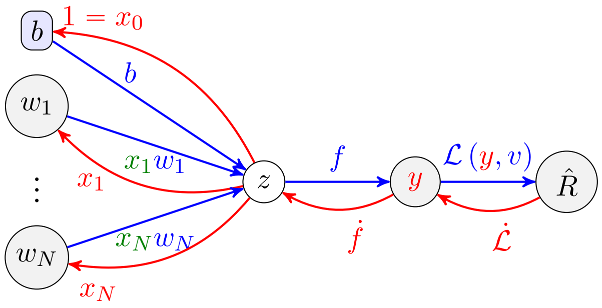


Here, the computation nodes are the *weights and bias of the neuron*

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



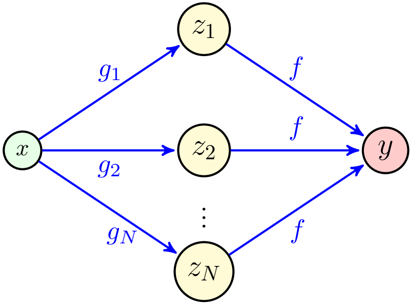
For a particular weight w_n, we can write (we drop arguments whenever clear)

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

We can extend it to a deeper and wider network

Computation Graph: *Multivariate Form*

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


$$\begin{tikzcd} & z_1 \arrow[dr, "f"] & \\ x \arrow[ur, "g_1"] \arrow[r, "g_2"] \arrow[dr, "g_N"] & z_2 \arrow[r, "f"] & y \\ & \vdots & \\ & z_N \arrow[uur, "f"] & \end{tikzcd}$$

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

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

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.,*

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

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

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

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*

$$\begin{tikzcd} x \arrow[r, "\mathbf{g}"] & \mathbf{z} \arrow[l, bend left, "\dot{\mathbf{g}}"] \arrow[r, "f"] & y \arrow[l, bend left, "\nabla f"] \end{tikzcd}$$

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

$$\begin{tikzcd} x \arrow[r, "\mathbf{g}"] & \mathbf{z} \arrow[r, "f"] \arrow[l, bend left, "\dot{\mathbf{g}}"] & y \arrow[l, bend left, "\nabla f"] \end{tikzcd}$$

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!*

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!*

$$\begin{tikzcd} x \arrow[r, "\mathbf{g}"] & \mathbf{z} \arrow[l, bend left, "\dot{\mathbf{g}}"] \arrow[r, "f"] & y \arrow[l, bend left, "\nabla f"] \end{tikzcd}$$

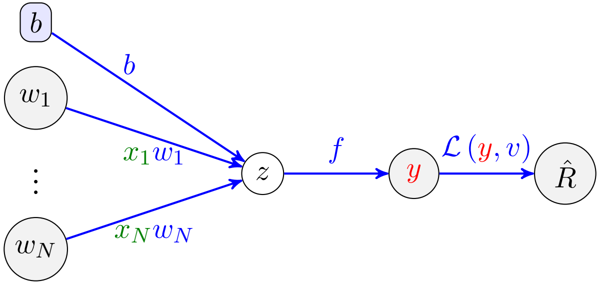
In our example, we determined *the inner product*

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

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

$$\begin{tikzcd} \mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] & y \arrow[r, "\mathcal{L}(y, v)"] & \hat{R} \end{tikzcd}$$

Computation Graph: *Multivariate Form*

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

$$\begin{tikzcd} \mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] & y \arrow[r, "\mathcal{L}(y,v)"] & \hat{R} \end{tikzcd}$$

- $\hat{R}$ is a *scalar* function of *scalar* y, i.e., $\hat{R} = \mathcal{L}(y,v)$
  - ↳ *the backward link contains the scalar derivative* $\dot{\mathcal{L}}$

Computation Graph: *Multivariate Form*

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

$$\begin{tikzcd} \mathbf{w} \arrow[r, "\mathbf{x}^\mathsf{T}"] & z \arrow[r, "f"] & y \arrow[r, "\mathcal{L}(y, v)"] & \hat{R} \end{tikzcd}$$

- $\hat{R}$ 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* $\dot{f}$

Computation Graph: *Multivariate Form*

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

$$\begin{tikzcd} \mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] & y \arrow[r, "\mathcal{L}(y, v)"] & \hat{R} \end{tikzcd}$$

- $\hat{R}$ 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

Computation Graph: *Multivariate Form*

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

$$\begin{tikzcd} \mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] & y \arrow[r, "\mathcal{L}(y,v)"] & \hat{R} \end{tikzcd}$$

- $\hat{R}$ 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

$$\begin{tikzcd} \mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] \arrow[l, bend left, "\nabla z"] & y \arrow[r, "\mathcal{L}(y,v)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"] \end{tikzcd}$$

Computation Graph: *Multivariate Form*

$$\begin{tikzcd} \mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] \arrow[l, bend left, "\nabla z"] & y \arrow[r, "\mathcal{L}(y, v)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"] \end{tikzcd}$$

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

Computation Graph: *Multivariate Form*

$$\begin{tikzcd}
\mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] \arrow[l, bend left, "\nabla z"] & y \arrow[r, "\mathcal{L}(y, v)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"]
\end{tikzcd}$$

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

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

Computation Graph: *Multivariate Form*

$$\begin{tikzcd}
\mathbf{w} \arrow[r, "\mathbf{x}^\mathsf{T}"] & z \arrow[r, "f"] \arrow[l, bend left, "\nabla z"] & y \arrow[r, "\mathcal{L}(y, v)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"]
\end{tikzcd}$$

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*

$$\begin{tikzcd}
\mathbf{w} \arrow[r, "\mathbf{x}^\mathsf{T}"] & z \arrow[r, "f"] \arrow[l, bend left, "\mathbf{x}"] & y \arrow[r, "\mathcal{L}(y, v)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"]
\end{tikzcd}$$

Computation Graph: *Multivariate Form*

Now, how do we pass backward on this graph?

$$\begin{tikzcd}
\mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] \arrow[l, bend left, "\mathbf{x}"] & y \arrow[r, "\mathcal{L}\left(y,v\right)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"]
\end{tikzcd}$$

Computation Graph: *Multivariate Form*

Now, how do we pass backward on this graph?

$$\begin{tikzcd}
\mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] \arrow[l, bend left, "\mathbf{x}"] & y \arrow[r, "\mathcal{L}(y,v)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"]
\end{tikzcd}$$

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

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

and we are at the computing node y. We then pass backward $\dot{\mathcal{L}}$.

Computation Graph: *Multivariate Form*

Now, how do we pass backward on this graph?

$$\begin{tikzcd}
\mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] \arrow[l, bend left, "\mathbf{x}"] & y \arrow[r, "\mathcal{L}\left(y, v\right)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"]
\end{tikzcd}$$

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?

$$\begin{tikzcd}
\mathbf{w} \arrow[r, "\mathbf{x}^{\mathsf{T}}"] & z \arrow[r, "f"] \arrow[l, bend left, "\mathbf{x}"] & y \arrow[r, "\mathcal{L}(y, v)"] \arrow[l, bend left, "\dot{f}"] & \hat{R} \arrow[l, bend left, "\dot{\mathcal{L}}"]
\end{tikzcd}$$

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*

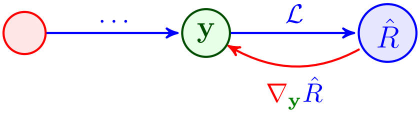
$$\nabla_{\mathbf{y}} \hat{R} = \begin{bmatrix} \partial \hat{R} / \partial y_1 \\ \vdots \\ \partial \hat{R} / \partial y_K \end{bmatrix}$$

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

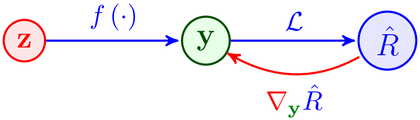


We now consider different cases for the other variable and its link to $\mathbf{y}$

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*



$$\begin{tikzcd} \mathbf{z} \arrow[r, "f(\cdot)"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

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*

$$\begin{tikzcd} \mathbf{z} \arrow[r, "f(\cdot)"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

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

$$\begin{tikzcd} \mathbf{z} \arrow[r, "f(\cdot)"] & \mathbf{y} \arrow[l, bend left, "\dot{f}(\cdot)"] \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

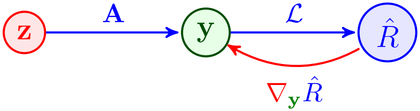
with the backward step

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

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* $\mathbf{A} \in \mathbb{R}^{K \times N}$

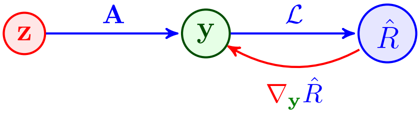


$$\begin{tikzcd} \mathbf{z} \arrow[r, "\mathbf{A}"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

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* $\mathbf{A} \in \mathbb{R}^{K \times N}$


$$\begin{tikzcd} \mathbf{z} \arrow[r, "\mathbf{A}"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

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

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

Backpropagation: *Local Operation 2*

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

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

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 . . .

$$\begin{tikzcd}
\mathbf{z} \arrow[r, "\mathbf{A}"] & \mathbf{y} \arrow[l, bend left, "\mathbf{A}^{\top}"] \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"]
\end{tikzcd}$$

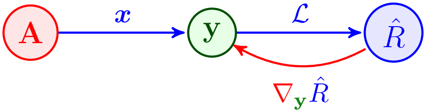
with backward step

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

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* $\boldsymbol{x} \in \mathbb{R}^N$


$$\begin{tikzcd}
\mathbf{A} \arrow[r, "\boldsymbol{x}"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"]
\end{tikzcd}$$

+ *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

$$\begin{tikzcd}
\mathbf{A} \arrow[r, "\boldsymbol{x}"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"]
\end{tikzcd}$$

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

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

Backpropagation: *Local Operation 3*

Let's now put them in a matrix

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

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

$$\begin{tikzcd}
\mathbf{A} \arrow[r, "\boldsymbol{x}"] & \mathbf{y} \arrow[l, bend left, "\boldsymbol{x}^\mathsf{T}"] \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"]
\end{tikzcd}$$

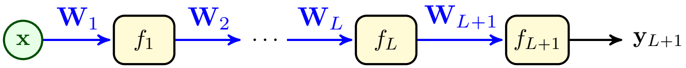
with the backward step

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

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

Backpropagation: *Algorithm*

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



$$\begin{tikzcd} \mathbf{x} \arrow[r, "\mathbf{W}_1"] & \boxed{f_1} \arrow[r, "\mathbf{W}_2"] & \cdots \arrow[r, "\mathbf{W}_L"] & \boxed{f_L} \arrow[r, "\mathbf{W}_{L+1}"] & \boxed{f_{L+1}} \arrow[r] & \mathbf{y}_{L+1} \end{tikzcd}$$

Backpropagation: *Algorithm*

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

$$\begin{tikzcd}
\mathbf{x} \arrow[r, "\mathbf{W}_1"] & \boxed{f_1} \arrow[r, "\mathbf{W}_2"] & \cdots \arrow[r, "\mathbf{W}_L"] & \boxed{f_L} \arrow[r, "\mathbf{W}_{L+1}"] & \boxed{f_{L+1}} \arrow[r] & \mathbf{y}_{L+1}
\end{tikzcd}$$

We can easily expand it into a *computation graph*

$$\begin{tikzcd}
\mathbf{x} \arrow[r, "\mathbf{W}_1"] & \mathbf{z}_1 \arrow[r, "f_1(\cdot)"] & \mathbf{y}_1 \arrow[r] & \cdots \arrow[r] & \mathbf{z}_L \arrow[r, "f_L(\cdot)"] & \mathbf{y}_L \arrow[r, "\mathbf{W}_{L+1}"] & \mathbf{z}_{L+1} \arrow[r, "f_{L+1}(\cdot)"] & \mathbf{y}_{L+1}
\end{tikzcd}$$

Backpropagation: *Algorithm*

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

$$\begin{tikzcd}
\mathbf{x} \arrow[r, "\mathbf{W}_1"] & \boxed{f_1} \arrow[r, "\mathbf{W}_2"] & \cdots \arrow[r, "\mathbf{W}_L"] & \boxed{f_L} \arrow[r, "\mathbf{W}_{L+1}"] & \boxed{f_{L+1}} \arrow[r] & \mathbf{y}_{L+1}
\end{tikzcd}$$

We can easily expand it into a *computation graph*

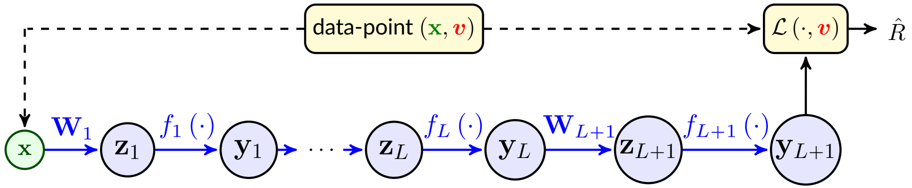
$$\begin{tikzcd}
\mathbf{x} \arrow[r, "\mathbf{W}_1"] & \mathbf{z}_1 \arrow[r, "f_1(\cdot)"] & \mathbf{y}_1 \arrow[r] & \cdots \arrow[r] & \mathbf{z}_L \arrow[r, "f_L(\cdot)"] & \mathbf{y}_L \arrow[r, "\mathbf{W}_{L+1}"] & \mathbf{z}_{L+1} \arrow[r, "f_{L+1}(\cdot)"] & \mathbf{y}_{L+1}
\end{tikzcd}$$

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

$$\begin{tikzcd}
& & & \boxed{\text{data-point } (\mathbf{x}, \boldsymbol{v})} \arrow[dlll, dashed] \arrow[rrrr, dashed] & & & & \boxed{\mathcal{L}(\cdot, \boldsymbol{v})} \arrow[r] & \hat{R} \\
\mathbf{x} \arrow[r, "\mathbf{W}_1"] & \mathbf{z}_1 \arrow[r, "f_1(\cdot)"] & \mathbf{y}_1 \arrow[r] & \cdots \arrow[r] & \mathbf{z}_L \arrow[r, "f_L(\cdot)"] & \mathbf{y}_L \arrow[r, "\mathbf{W}_{L+1}"] & \mathbf{z}_{L+1} \arrow[r, "f_{L+1}(\cdot)"] & \mathbf{y}_{L+1} \arrow[u]
\end{tikzcd}$$

Backpropagation: *Algorithm*

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



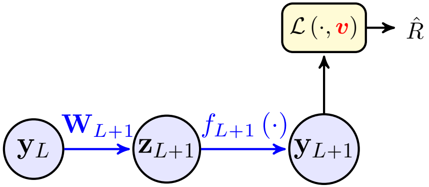
At the end of forward pass,

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 $\mathbf{W}_{L+1}$



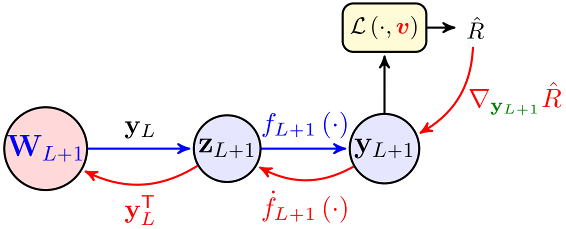
Let's recall . . .

+ *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 to* $\mathbf{z}_{L+1}$
2. *We have* $\mathbf{z}_{L+1}$*, so we compute* $\dot{f}_{L+1}(\mathbf{z}_{L+1})$*, and then we get*
$$\nabla_{\mathbf{z}_{L+1}} \hat{R} = \nabla_{\mathbf{y}_{L+1}} \hat{R} \odot \dot{f}_{L+1}$$
We now pass $\nabla_{\mathbf{z}_{L+1}} \hat{R}$ to $\mathbf{W}_{L+1}$
3. *We have* $\mathbf{y}_L$*, so we compute* $\nabla_{\mathbf{W}_{L+1}} \hat{R}$ *from the last pass as*
$$\nabla_{\mathbf{W}_{L+1}} \hat{R} = \nabla_{\mathbf{z}_{L+1}} \hat{R}\, \mathbf{y}_L^{\mathsf{T}}$$

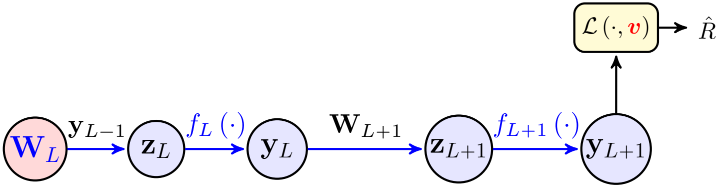


Backpropagation: *Algorithm*

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 liner 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*

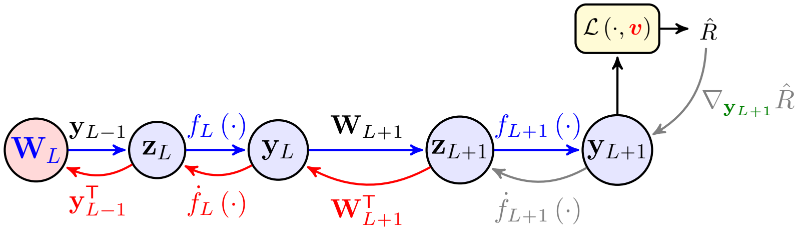
Let's see the example for $\mathbf{W}_L$



Backpropagation: *Algorithm*

We already have passed backward messages till $\mathbf{z}_{L+1}$

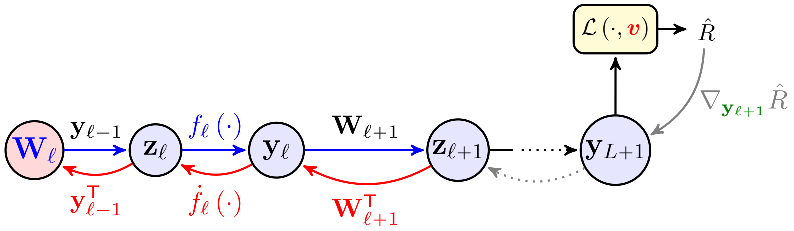
1. *We now pass* $\nabla_{\mathbf{z}_{L+1}} \hat{R}$ *to* $\mathbf{y}_L$: $\nabla_{\mathbf{y}_L} \hat{R} = \mathbf{W}_{L+1}^{\mathsf{T}} \nabla_{\mathbf{z}_{L+1}} \hat{R}$
2. *We then pass* $\nabla_{\mathbf{y}_L} \hat{R}$ *to* $\mathbf{z}_L$: $\nabla_{\mathbf{z}_L} \hat{R} = \nabla_{\mathbf{y}_L} \hat{R}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

```
1: Initiate with $\overleftarrow{\mathbf{y}}_{L+1} = \nabla\mathcal{L}(\mathbf{y}_{L+1}, \boldsymbol{v})$ and $\overleftarrow{\mathbf{z}}_{L+1} = \overleftarrow{\mathbf{y}}_{L+1} \odot \dot{f}_{L+1}(\mathbf{z}_{L+1})$
2: for $\ell = L, \ldots, 1$ do
3:     Determine $\overleftarrow{\mathbf{y}}_{\ell} = \mathbf{W}_{\ell+1}^{\mathsf{T}} \overleftarrow{\mathbf{z}}_{\ell+1}$ and drop $\overleftarrow{y}_{\ell}[0]$      # backward affine
4:     Determine $\overleftarrow{\mathbf{z}}_{\ell} = \dot{f}_{\ell}(\mathbf{z}_{\ell}) \odot \overleftarrow{\mathbf{y}}_{\ell}$      # backward activation
5: end for
6: for $\ell = 1, \ldots, L+1$ do
7:     Return $\nabla_{\mathbf{W}_\ell} \hat{R} = \overleftarrow{\mathbf{z}}_{\ell} \mathbf{y}_{\ell-1}^{\mathsf{T}}$
8: 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():
1: Initiate with $\mathbf{y}_0 = \mathbf{x}$
2: for $\ell = 0, \ldots, L$ do
3:     Add $y_\ell[0] = 1$ and determine $\mathbf{z}_{\ell+1} = \mathbf{W}_{\ell+1}\mathbf{y}_\ell$          # forward affine
4:     Determine $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$          # forward activation
5: end for
6: for $\ell = 1, \ldots, L+1$ do
7:     Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
8: end for
```

```
BackProp():
1: Initiate with $\overleftarrow{\mathbf{y}}_{L+1} = \nabla\mathcal{L}(\mathbf{y}_{L+1}, \boldsymbol{v})$ and $\overleftarrow{\mathbf{z}}_{L+1} = \overleftarrow{\mathbf{y}}_{L+1} \odot \dot{f}_{L+1}(\mathbf{z}_{L+1})$
2: for $\ell = L, \ldots, 1$ do
3:     Determine $\overleftarrow{\mathbf{y}}_\ell = \mathbf{W}_{\ell+1}^{\mathsf{T}}\overleftarrow{\mathbf{z}}_{\ell+1}$ and drop $\overleftarrow{y}_\ell[0]$          # backward affine
4:     Determine $\overleftarrow{\mathbf{z}}_\ell = \dot{f}_\ell(\mathbf{z}_\ell) \odot \overleftarrow{\mathbf{y}}_\ell$          # backward activation
5: end for
6: for $\ell = 1, \ldots, L+1$ do
7:     Return $\nabla_{\mathbf{W}_\ell}\hat{R} = \overleftarrow{\mathbf{z}}_\ell \mathbf{y}_{\ell-1}^{\mathsf{T}}$
8: 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 foo all data-points

Say we have dataset

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

```
GradientDescent():
 1: Initiate with some initial values $\{\mathbf{W}_\ell^{(0)}\}$ and set a learning rate $\eta$
 2: while weights not converged do
 3:     for $b = 1, \ldots, B$ do
 4:         NN.values ← ForwardProp $(\boldsymbol{x}_b, \{\mathbf{W}_\ell^{(t)}\})$            # forward
 5:         $\{\nabla_{\mathbf{W}_\ell{}^{(t)}} \hat{R}_b\}$ ← BackProp $(\boldsymbol{x}_b, \boldsymbol{v}_b, \{\mathbf{W}_\ell^{(t)}\}, \text{NN.values})$   # backward
 6:     end for
 7:     for $\ell = 1, \ldots, L+1$ do
 8:         $\mathbf{W}_\ell{}^{(t+1)} \leftarrow \mathbf{W}_\ell{}^{(t)} - \eta$ mean$(\nabla_{\mathbf{W}_\ell{}^{(t)}} \hat{R}_1, \ldots, \nabla_{\mathbf{W}_\ell{}^{(t)}} \hat{R}_B)$
 9:     end for
10: end while
```

Binary Classification via FNN

Let's now design a deep FNN for *binary classification*

We have a set of images of hand-written numbers, something like this[a]

We intend to train a fully-connected FNN that given a new hand-written image, it finds out whether it is "2" or not

[a] Source: *Wikipedia*

This is a *binary classification!*

Binary Classification via FNN: *Data*

Let's get clear about the data: *our dataset looks like*

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

where in this set each component is defined as follows:

- B is the number of images we have
  - ↳ *we also call the set of images: a batch of images*
- $\boldsymbol{x}_b \in \mathbb{R}^N$ is the *pixel vector* of image b
  - ↳ *N is the number of pixels in the image*
- $v_b \in \{0, 1\}$ is a binary label indicating *whether it is "2" or not*
  - ↳ *if the image is a hand-written "2" we set $v_b = 1$*
  - ↳ *if the image is not a hand-written "2" we set $v_b = 0$*

Binary Classification via FNN: *Model*

Let's now set the model: *we use a fully-connected FNN*

What are the hyperparameters?

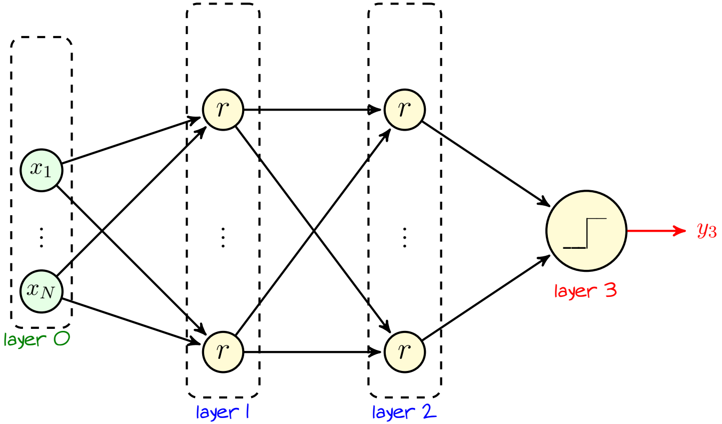
- We want it to be deep; so, we consider *2 hidden layers*
  - ↳ *the depth is hence* 3
- We specify the width of each hidden layer
  - ↳ *first hidden layer has width* K
  - ↳ *second hidden layer has width* J
- All hidden neurons use ReLU activation
  - ↳ $f_1(\cdot) = f_2(\cdot) = \mathrm{ReLU}(\cdot)$: *let's show* ReLU *by* r, *i.e.*,

$$r(x) = \mathrm{ReLU}(x)$$

- Output layer has *a single perceptron*

We can now write down the model!

Binary Classification via FNN: *Model*



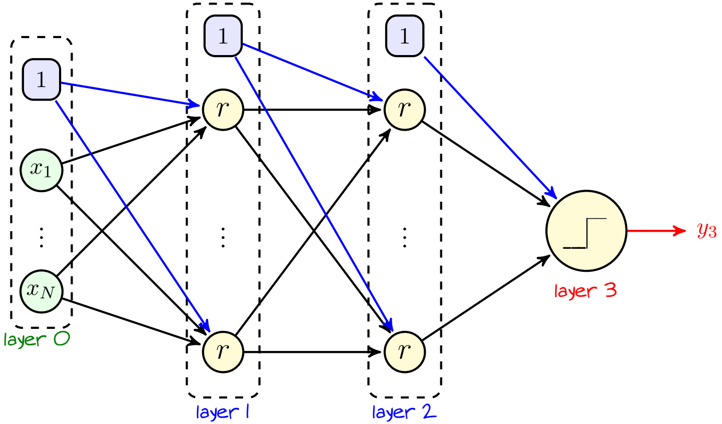
Binary Classification via FNN: *Model*

Let's now set the model: *we use a fully-connected FNN*

What are the learnable parameters?

- Layer 1 has $(N+1)K$ links
  - ↳ NK *of them are weights*
  - ↳ K *of them are biases* $\equiv$ *weights of dummy node* $x_0 = 1$
- Layer 2 has $(K+1)J$ links
  - ↳ KJ *of them are weights*
  - ↳ J *of them are biases* $\equiv$ *weights of dummy node* $y_1[0] = 1$
- Output layer has $J+1$ links
  - ↳ J *of them are weights*
  - ↳ *one is bias* $\equiv$ *weight of dummy node* $y_2[0] = 1$

Binary Classification via FNN: *Model*



Binary Classification via FNN: *Loss*

How to calculate the loss? *Let's do what we did before*

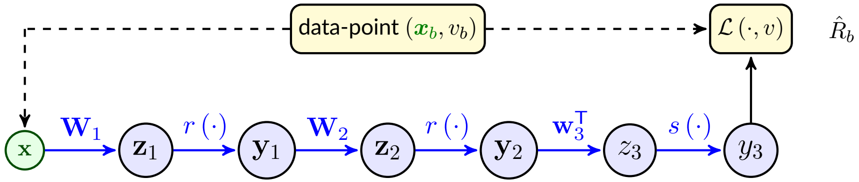
We use the error indicator as the loss function

$$\mathcal{L}(y_b, v_b) = \mathbb{1}\{y_b \neq v_b\} = \begin{cases} 1 & y_b \neq v_b \\ 0 & y_b = v_b \end{cases}$$

+ *Wait a moment! Didn't you say that this was a bad choice?*
- Yeah! So said I also for the *perceptron's activation!* Let's try it out to find out really why they are bad! We should be able to understand it now

Binary Classification via FNN: *Training*

Let's look at the computation graph: *for a given data-point* $(\boldsymbol{x}_b, v_b)$, *we have*

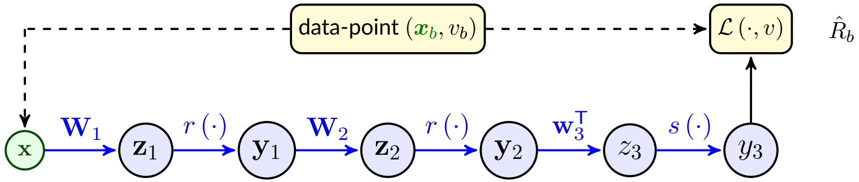


Here, we have *3 linear operations*

- First operation is $\mathbf{z}_1 = \mathbf{W}_1 \mathbf{x}$ with $\mathbf{W}_1 \in \mathbb{R}^{K \times (N+1)}$
  - ↳ *first column of* $\mathbf{W}_1$ *is bias and the remaining columns are weights*
- Second operation is $\mathbf{z}_2 = \mathbf{W}_2 \mathbf{y}_1$ with $\mathbf{W}_2 \in \mathbb{R}^{J \times (K+1)}$
  - ↳ *first column of* $\mathbf{W}_2$ *is bias and the remaining columns are weights*
- Last operation is $z_3 = \mathbf{w}_3^\mathsf{T} \mathbf{x}$ with $\mathbf{w}_3 \in \mathbb{R}^{J+1}$
  - ↳ *first entry of* $\mathbf{w}_3$ *is bias and the remaining entries are weights*

Binary Classification via FNN: *Training*

Let's look at the computation graph: *for a given data-point* $(\boldsymbol{x}_b, v_b)$, *we have*



We have *3 functional operations*

- The first two are $\mathbf{y}_1 = r(\mathbf{z}_1)$ and $\mathbf{y}_2 = r(\mathbf{z}_2)$
- The last one is $y_3 = s(z_3)$, and recall that $s(\cdot)$ is the step function

$$y_3 = s(z_3) = \begin{cases} 1 & z_3 \geqslant 0 \\ 0 & z_3 < 0 \end{cases}$$

Binary Classification via FNN: *Training*

Let's write gradient descent for training of our model

```
GradientDescent():
1: Initiate with some initial values $\{\mathbf{W}_1^{(0)}, \mathbf{W}_2^{(0)}, \mathbf{w}_3^{(0)}\}$ and set a learning rate $\eta$
2: while weights not converged do
3:     for $b = 1, \dots, B$ do
4:         NN.values ← ForwardProp $(\boldsymbol{x}_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{w}_3^{(t)}\})$
5:         $\{\mathbf{G}_{1,b}, \mathbf{G}_{2,b}, \mathbf{g}_{3,b}\}$ ← BackProp $(\boldsymbol{x}_b, v_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{w}_3^{(t)}\}, \text{NN.values})$
6:     end for
7:     Update
           $\mathbf{W}_1^{(t+1)} \leftarrow \mathbf{W}_1^{(t)} - \eta \ \mathtt{mean}(\mathbf{G}_{1,1}, \dots, \mathbf{G}_{1,B})$
           $\mathbf{W}_2^{(t+1)} \leftarrow \mathbf{W}_2^{(t)} - \eta \ \mathtt{mean}(\mathbf{G}_{2,1}, \dots, \mathbf{G}_{2,B})$
           $\mathbf{w}_3^{(t+1)} \leftarrow \mathbf{w}_3^{(t)} - \eta \ \mathtt{mean}(\mathbf{g}_{3,1}, \dots, \mathbf{g}_{3,B})$
8: end while
```

Let's look at forward and backward propagation!

Binary Classification via FNN: *Forward Pass*

Forward pass is very straightforward: say we are at iteration t

1. For each pixel vector $\boldsymbol{x}_b$, we determine $\mathbf{z}_1$ as

$$\mathbf{x} \leftarrow \begin{bmatrix} 1 \\ \boldsymbol{x}_b \end{bmatrix} \rightsquigarrow \mathbf{z}_1 = \mathbf{W}_1^{(t)} \mathbf{x}$$

The output of first layer is then given by $\mathbf{y}_1 = r(\mathbf{z}_1)$: $r(\cdot)$ *is ReLU, so*

$$\boxed{\textit{we keep positive entries of } \mathbf{z}_1 \textit{ and replace negative ones with zero}}$$

Binary Classification via FNN: *Forward Pass*

Forward pass is very straightforward: say we are at iteration t

1. For each pixel vector $\boldsymbol{x}_b$, we determine $\mathbf{z}_1$ as

$$\mathbf{x} \leftarrow \begin{bmatrix} 1 \\ \boldsymbol{x}_b \end{bmatrix} \rightsquigarrow \mathbf{z}_1 = \mathbf{W}_1^{(t)} \mathbf{x}$$

   The output of first layer is then given by $\mathbf{y}_1 = r(\mathbf{z}_1)$: $r(\cdot)$ *is ReLU, so*

$$\boxed{\text{we keep positive entries of } \mathbf{z}_1 \text{ and replace negative ones with zero}}$$

2. We add 1 at index 0 of $\mathbf{y}_1$ and determine $\mathbf{z}_2$ as

$$\mathbf{y}_1 \leftarrow \begin{bmatrix} 1 \\ \mathbf{y}_1 \end{bmatrix} \rightsquigarrow \mathbf{z}_2 = \mathbf{W}_2^{(t)} \mathbf{y}_1$$

   The output of second layer is given by $\mathbf{y}_2 = r(\mathbf{z}_2)$

Binary Classification via FNN: *Forward Pass*

3. We add 1 at index 0 of $\mathbf{y}_2$ and determine z_3 as

$$\mathbf{y}_2 \leftarrow \begin{bmatrix} 1 \\ \mathbf{y}_2 \end{bmatrix} \rightsquigarrow z_3 = \mathbf{w}_3^{(t)^\mathsf{T}} \mathbf{y}_2$$

The network output is given by $y_3 = s(z_3)$: $s(\cdot)$ *is step function, so*

$$\boxed{\text{it's } 0 \text{ if } z_3 \text{ is negative, and } 1 \text{ if it is not negative}}$$

Binary Classification via FNN: *Forward Pass*

3. We add 1 at index 0 of $\mathbf{y}_2$ and determine z_3 as

$$\mathbf{y}_2 \leftarrow \begin{bmatrix} 1 \\ \mathbf{y}_2 \end{bmatrix} \rightsquigarrow z_3 = \mathbf{w}_3^{(t)^{\mathsf{T}}} \mathbf{y}_2$$

The network output is given by $y_3 = s(z_3)$: *$s(\cdot)$ is step function, so*

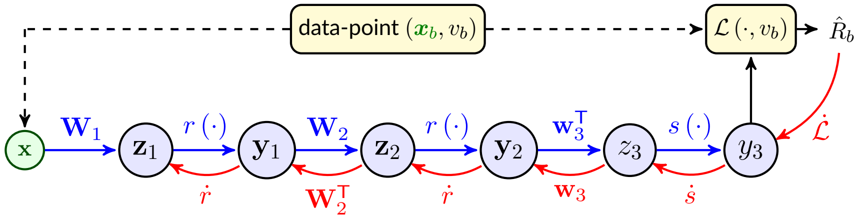
$$\boxed{\text{it's } 0 \text{ if } z_3 \text{ is negative, and } 1 \text{ if it is not negative}}$$

At this point, we have all that we need, i.e.,

$$\mathbf{x}, \mathbf{z}_1, \mathbf{y}_1, \mathbf{z}_2, \mathbf{y}_2, z_3 \text{ and } y_3$$

Binary Classification via FNN: *Training*

How does the graph look like on the backward pass?



Let's move backward!

Binary Classification via FNN: *Backward Pass*

We know all the derivatives, i.e.,

$$\dot{\mathcal{L}}\left(y, v_b\right) = \frac{\mathrm{d}}{\mathrm{d}y} \mathbb{1}\left\{y \neq v_b\right\} \qquad \dot{s}\left(z\right) = \frac{\mathrm{d}}{\mathrm{d}z} s\left(z\right) \qquad \dot{r}\left(z\right) = \frac{\mathrm{d}}{\mathrm{d}z} r\left(z\right)$$

Binary Classification via FNN: *Backward Pass*

We know all the derivatives, i.e.,

$$\dot{\mathcal{L}}\left(y, v_b\right) = \frac{\mathrm{d}}{\mathrm{d}y} \mathbb{1}\left\{y \neq v_b\right\} \qquad \dot{s}\left(z\right) = \frac{\mathrm{d}}{\mathrm{d}z} s\left(z\right) \qquad \dot{r}\left(z\right) = \frac{\mathrm{d}}{\mathrm{d}z} r\left(z\right)$$

For backward pass we start at *node* y_3:

1. We find derivative w.r.t. output $\overleftarrow{y}_3 = \dot{\mathcal{L}}(y_3, v_b)$ and set

$$\overleftarrow{z}_3 = \overleftarrow{y}_3 \dot{s}(z_3)$$

Binary Classification via FNN: *Backward Pass*

We know all the derivatives, i.e.,

$$\dot{\mathcal{L}}\left(y, v_b\right) = \frac{\mathrm{d}}{\mathrm{d}y} \mathbb{1}\left\{y \neq v_b\right\} \qquad \dot{s}\left(z\right) = \frac{\mathrm{d}}{\mathrm{d}z} s\left(z\right) \qquad \dot{r}\left(z\right) = \frac{\mathrm{d}}{\mathrm{d}z} r\left(z\right)$$

For backward pass we start at *node* y_3:

1. We find derivative w.r.t. output $\overleftarrow{y}_3 = \dot{\mathcal{L}}(y_3, v_b)$ and set

$$\overleftarrow{z}_3 = \overleftarrow{y}_3 \dot{s}(z_3)$$

2. We compute $\overleftarrow{\mathbf{y}}_2 = \mathbf{w}_3 \overleftarrow{z}_3$ and drop its first entry; then, compute

$$\overleftarrow{\mathbf{y}}_2 \leftarrow \begin{bmatrix} \cancel{\overleftarrow{y}_2[0]} \\ \overleftarrow{\mathbf{y}}_2[1:] \end{bmatrix} \rightsquigarrow \overleftarrow{\mathbf{z}}_2 = \dot{r}(\mathbf{z}_2) \odot \overleftarrow{\mathbf{y}}_2$$

Binary Classification via FNN: *Backward Pass*

3. We compute $\overleftarrow{\mathbf{y}}_1 = \mathbf{W}_2^\top \overleftarrow{\mathbf{z}}_2$ and drop its first entry; then, compute

$$\overleftarrow{\mathbf{y}}_1 \leftarrow \begin{bmatrix} \cancel{\overleftarrow{y}_1[0]} \\ \overleftarrow{\mathbf{y}}_1[1:] \end{bmatrix} \rightsquigarrow \overleftarrow{\mathbf{z}}_1 = \dot{r}(\mathbf{z}_1) \odot \overleftarrow{\mathbf{y}}_1$$

Binary Classification via FNN: *Backward Pass*

3. We compute $\overleftarrow{\mathbf{y}}_1 = \mathbf{W}_2^\mathsf{T} \overleftarrow{\mathbf{z}}_2$ and drop its first entry; then, compute

$$\overleftarrow{\mathbf{y}}_1 \leftarrow \begin{bmatrix} \cancel{\overleftarrow{y}_1[0]} \\ \overleftarrow{\mathbf{y}}_1[1:] \end{bmatrix} \rightsquigarrow \overleftarrow{\mathbf{z}}_1 = \dot{r}(\mathbf{z}_1) \odot \overleftarrow{\mathbf{y}}_1$$

At this point, we can calculate all gradients

$$\mathbf{G}_{1,b} = \nabla_{\mathbf{W}_1} \hat{R}_b = \overleftarrow{\mathbf{z}}_1 \mathbf{y}_0^\mathsf{T} = \overleftarrow{\mathbf{z}}_1 \mathbf{x}^\mathsf{T}$$
$$\mathbf{G}_{2,b} = \nabla_{\mathbf{W}_2} \hat{R}_b = \overleftarrow{\mathbf{z}}_2 \mathbf{y}_1^\mathsf{T}$$
$$\mathbf{g}_{3,b}^\mathsf{T} = \nabla_{\mathbf{w}_3^\mathsf{T}} \hat{R}_b = \overleftarrow{z}_3 \mathbf{y}_2^\mathsf{T}$$

All done! We repeat it for every image in the batch and then average gradients. Finally, we move one step in gradient descent and find the weights of the next iteration

Binary Classification via FNN: *Differentiability Issue*

+ *Where is then the issue with perceptron and indicator error?*
– $\dot{\mathcal{L}}(y, v_b)$ and $\dot{s}(z)$ are not well-defined!
  ↳ *Recall that they are discontinuous*

In fact, the empirical risk is not a smooth function of the weights and biases; therefore, using gradient descent we do not end up with a well-trained network

+ *How can we get over it?*
– Well! There is a very well-established trick!

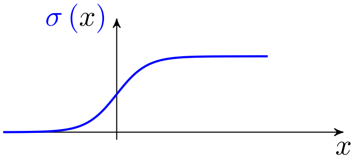
Binary Classification via FNN: *Differentiability Issue*

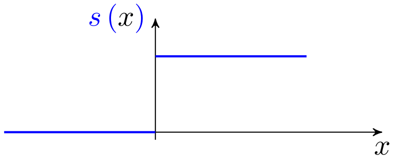
We first replace the perceptron with a neuron whose activation is a good approximation of *step function* and *differentiable*[1]

We already have seen the sigmoid function

$$\sigma\left(x\right) = \frac{1}{1+e^{-x}}$$

which looks pretty close to step function





Using sigmoid instead of step function resolves the differentiability issue

[1]Or at least, we can easily calculate a sub-gradient for it

Binary Classification via FNN: *Differentiability Issue*

But, replacing *perceptron* by *sigmoid-activated neuron* makes a new problem

The output of the network is now not binary!

How can we address this problem?

We now interpret the output as probability, i.e.,

y_3 *is the probability of the label being 1*

Binary Classification via FNN: *Differentiability Issue*

But, replacing *perceptron* by *sigmoid-activated neuron* makes a new problem

The output of the network is now not binary!

How can we address this problem?

We now interpret the output as probability, i.e.,

y_3 *is the probability of the label being 1*

+ *OK! But how can we define the loss now?*
– Well! We could look at the true label from the same point of view

Say $v \in \{0, 1\}$ *is true label: if* $v = 1$ *then the true label is* 1 *with probability* 1*; if* $v = 0$ *then the true label is* 1 *with probability* 0*. So, we could say*

the true label is 1 *with probability* v

Binary Classification via FNN: *Differentiability Issue*

true label is 1 with probability v $\leftrightsquigarrow$ y_3 is probability of the label being 1

Apparently, v and y_3 are of the same nature: *we can still define a loss that evaluates the difference between y_3 and v*

+ *What should be the loss then?*
– Definitely not the indicator error!

Binary Classification via FNN: *Differentiability Issue*

true label is 1 with probability v $\leftrightsquigarrow$ *y_3 is probability of the label being 1*

Apparently, v and y_3 are of the same nature: *we can still define a loss that evaluates the difference between y_3 and v*

+ *What should be the loss then?*
- Definitely not the indicator error!

Indicator error is not suitable because

1. *we already now that it is not differentiable*
2. *more importantly, with sigmoid activation becomes useless*

$$\mathbb{1}\left\{\sigma\left(z_3\right) \neq 1\right\} = \mathbb{1}\left\{\sigma\left(z_3\right) \neq 0\right\} = 1$$

Binary Classification via FNN: *MSE*

One may suggest that we use the squared error, i.e.,

$$\mathcal{L}\left(y_3, v\right) = \left(y_3 - v\right)^2$$

in this case the empirical risk is called

Mean Squared Error (MSE)

This loss is differentiable

$$\dot{\mathcal{L}}\left(y_3, v\right) = 2\left(y_3 - v\right)$$

and proportional to the distance between y_3 and v

It's a good choice but not best

Binary Classification via FNN: *Cross-Entropy*

A better choice is to determine the cross-entropy loss

$$\begin{aligned}\mathcal{L}(y_3, v) = \mathrm{CE}(y_3, v) &= -v \log y_3 - (1-v) \log (1 - y_3) \\ &= \begin{cases} \log \frac{1}{y_3} & v = 1 \\ \log \frac{1}{(1-y_3)} & v = 0 \end{cases}\end{aligned}$$

This loss function is sometimes wrongly called KL-divergence: it is proportional to the Kullback-Leibler divergence but it's different

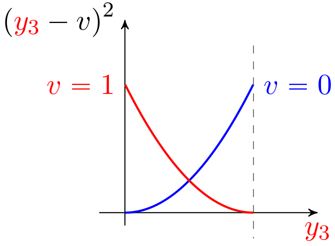
This loss is again differentiable

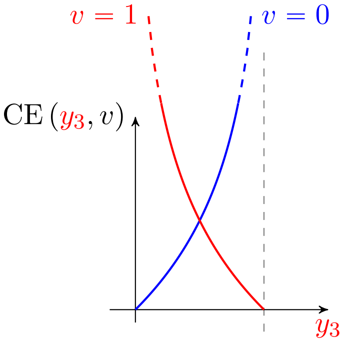
$$\dot{\mathcal{L}}(y_3, v) = \dot{\mathrm{CE}}(y_3, v) = -\frac{v}{y_3} + \frac{1-v}{1-y_3}$$

Note: *The logarithm is usually in natural base, i.e.,* $\log x = \ln x$

Binary Classification via FNN: *Cross-Entropy*

+ *But why cross-entropy is a better loss?*
– It pushes y_3 more towards the edges of interval $[0,1]$





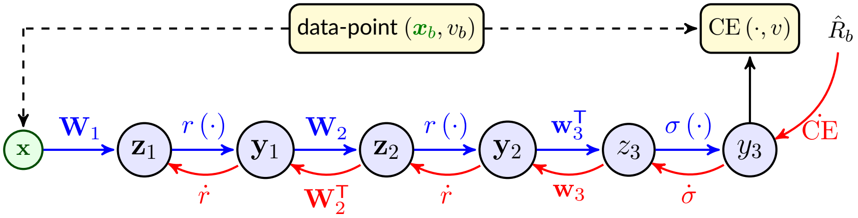
Cross entropy returns much higher loss when y_3 is different from v

Binary Classification via FNN: *Training with Cross-Entropy*

+ *What changes in the training loop in this case?*
– Pretty much *nothing*! Just replace
  - $\mathcal{L}(y_3, v)$ with $\mathrm{CE}(y_3, v)$
  - $\dot{\mathcal{L}}(y_3, v)$ with $\dot{\mathrm{CE}}(y_3, v)$
  - $s(z_3)$ with $\sigma(z_3)$
  - $\dot{s}(z_3)$ with $\dot{\sigma}(z_3)$

Binary Classification via FNN: *Training with Cross-Entropy*

+ *What changes in the training loop in this case?*
– Pretty much *nothing*! Just replace
  - $\mathcal{L}(y_3, v)$ with $\mathrm{CE}(y_3, v)$
  - $\dot{\mathcal{L}}(y_3, v)$ with $\dot{\mathrm{CE}}(y_3, v)$
  - $s(z_3)$ with $\sigma(z_3)$
  - $\dot{s}(z_3)$ with $\dot{\sigma}(z_3)$



Binary Classification via FNN: *Training with Cross-Entropy*

+ *How do we use the output of network then, when we give a new image to it for classification? It's not binary!*

– Just follow the interpretation

y_3 *gives the probability of the image being hand-written "2"*; therefore, $(1 - y_3)$ *gives the probability of image being any other hand-written number*. So, we select the outcome with higher chance, i.e.,

- *if* $y_3 \geqslant 0.5$, *we label the new image as a hand-written "2"*
- *if* $y_3 < 0.5$, *we label the new image as not being a hand-written "2"*

+ *Can't we classify more classes? Like hand-written "0", "1", . . ., "9"?*

– Now that we have this nice interpretation: *Yes! We can!*

Multiclass Classification

We initially saw that any multiclass classification can be seen as *a sequence of binary classifications*; however, for that, we need multiple NNs!

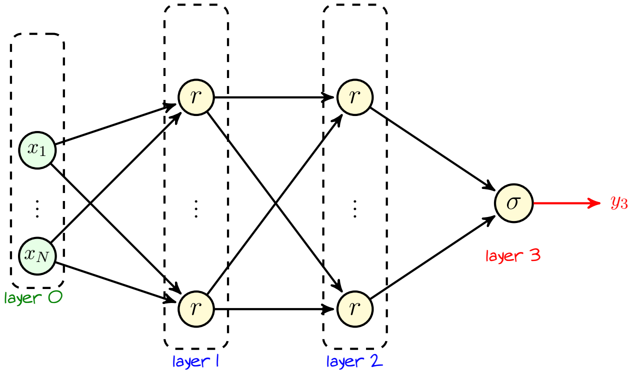
+ *Why not follow the same idea and determine the probability of input belonging to each class?*
– Yes! That's actually the effective way!

Let's get back to our image recognition, but now with multiple classes!

We have images of hand-written numbers from "0" to "9" and want to train a NN that recognizes any hand-written number

We first draw our earlier FNN

Multiclass Classification via FNN

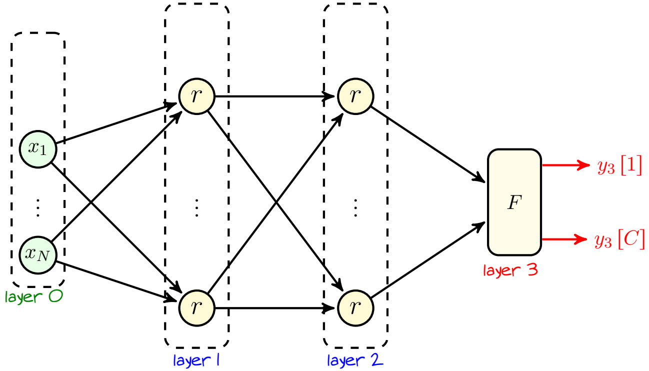


In this FNN, y_3 *is interpreted as a probability of label being* 1

probability of label being 0 *is hence* $1 - y_3$

This was done by a standard single-output neuron, since we had only 2 classes

Multiclass Classification via FNN



With C classes, we need *a module that computes probabilities of all C classes*

this module can be seen as a neuron with vector output

Multiclass Classification: *Vector-Activated Neuron*

Vector-Activated Neuron

A vector-activated neuron is an artificial neuron with multivariate activation function: let $\boldsymbol{x} \in \mathbb{R}^N$ be the input to this neuron and C be its output dimension; then, the output vector $\mathbf{y} \in \mathbb{R}^C$ is given by

$$\mathbf{y} = F\left(\tilde{\mathbf{W}}\boldsymbol{x} + \mathbf{b}\right)$$

for weight matrix $\tilde{\mathbf{W}} \in \mathbb{R}^{C \times N}$, bias $\mathbf{b} \in \mathbb{R}^C$ and activation $F(\cdot) : \mathbb{R}^C \mapsto \mathbb{R}^C$

Multiclass Classification: *Vector-Activated Neuron*

Vector-Activated Neuron

A vector-activated neuron is an artificial neuron with multivariate activation function: let $\boldsymbol{x} \in \mathbb{R}^N$ be the input to this neuron and C be its output dimension; then, the output vector $\mathbf{y} \in \mathbb{R}^C$ is given by

$$\mathbf{y} = F\left(\tilde{\mathbf{W}}\boldsymbol{x} + \mathbf{b}\right)$$

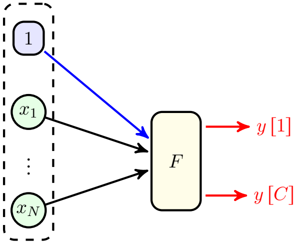
for weight matrix $\tilde{\mathbf{W}} \in \mathbb{R}^{C \times N}$, bias $\mathbf{b} \in \mathbb{R}^C$ and activation $F(\cdot) : \mathbb{R}^C \mapsto \mathbb{R}^C$

First thing first: *let's get rid of the bias before we go on*

$$\mathbf{y} = F(\begin{bmatrix} \mathbf{b} & \tilde{\mathbf{W}} \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix}) = F(\mathbf{W}\mathbf{x})$$

So, we keep on with our dummy 1 input here as well

Multiclass Classification: *Vector-Activated Neuron*



Next, *let's see how its computation graph looks*

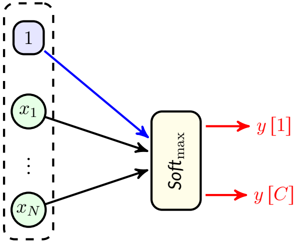
$$\begin{tikzcd} \mathbf{x} \arrow[r, "\mathbf{W}"] & \mathbf{z} \arrow[r, "F(\cdot)"] & \mathbf{y} \end{tikzcd}$$

This looks exactly like a standard layer with a minor difference

$F(\cdot)$ does not necessarily perform entry-wise

Multiclass Classification: *Softmax*

A very well-known example of vector activation is *softmax*



Softmax Function

For $\mathbf{z} \in \mathbb{R}^C$, softmax function returns $\text{Soft}_{\max}(\mathbf{z}) = \mathbf{y} \in \mathbb{R}^C$ *whose entry c is*

$$y[c] = \frac{e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}}$$

Multiclass Classification: *Softmax*

Softmax always returns a probability distribution on the set of classes

$$\sum_{c=1}^{C} y\left[c\right] =$$

Multiclass Classification: *Softmax*

Softmax always returns a probability distribution on the set of classes

$$\sum_{c=1}^{C} y[c] = \sum_{c=1}^{C} \frac{e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}}$$

Multiclass Classification: *Softmax*

Softmax always returns a probability distribution on the set of classes

$$\sum_{c=1}^{C} y[c] = \sum_{c=1}^{C} \frac{e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}} = \frac{\sum_{c=1}^{C} e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}}$$

Multiclass Classification: *Softmax*

Softmax always returns a probability distribution on the set of classes

$$\sum_{c=1}^{C} y[c] = \sum_{c=1}^{C} \frac{e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}} = \frac{\sum_{c=1}^{C} e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}} = 1$$

Multiclass Classification: *Softmax*

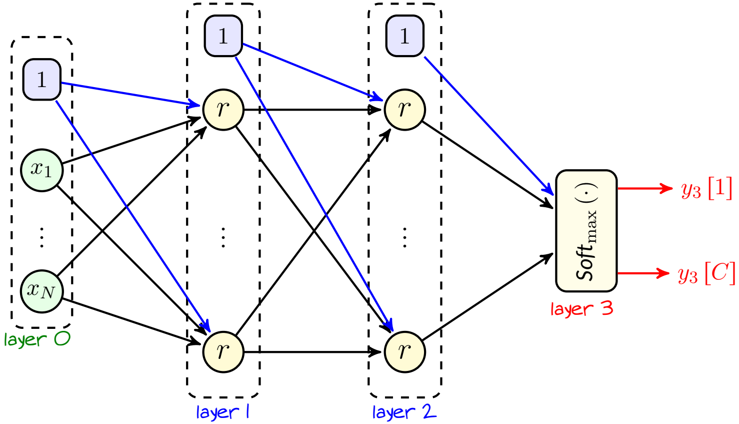
Softmax always returns a probability distribution on the set of classes

$$\sum_{c=1}^{C} y[c] = \sum_{c=1}^{C} \frac{e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}} = \frac{\sum_{c=1}^{C} e^{z[c]}}{\sum_{j=1}^{C} e^{z[j]}} = 1$$

We can hence use it to extend our FNN to a *multiclass classifier*

We replace layer 3 with a softmax-activated multivariate neuron and treat its outcome as the chance of input belonging to each class; then, we select the class with highest chance

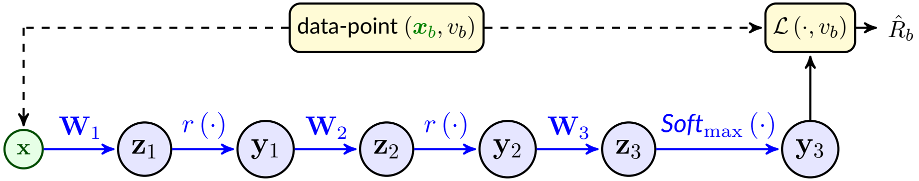
Multiclass Classification via FNN: *Softmax Activation*



Let's try again the forward pass

Multiclass Classification via FNN: *Softmax Activation*

Let's look at the computation graph: *for a given data-point* $(\boldsymbol{x}_b, v_b)$, *we have*

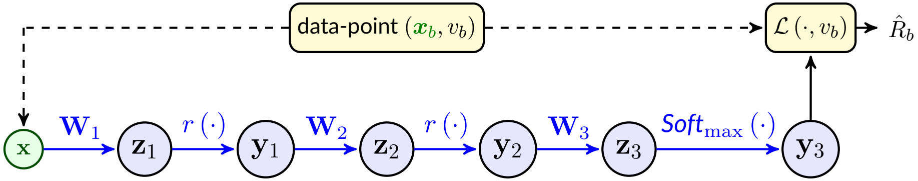


Note that the output layer has been changed

- *We now have a vector* $\mathbf{z}_3 \in \mathbb{R}^C$
- *We now have a vector* $\mathbf{y}_3 \in \mathbb{R}^C$
- *We get from* $\mathbf{z}_3$ *to* $\mathbf{y}_3$ *via softmax*

Multiclass Classification via FNN: *Softmax Activation*

Let's look at the computation graph: *for a given data-point* $(\boldsymbol{x}_b, v_b)$, *we have*

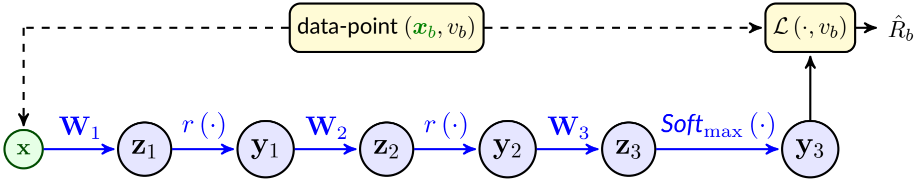


Note that the output layer has been changed

- *We now have a vector* $\mathbf{z}_3 \in \mathbb{R}^C$
  ↳ *So we have a matrix of weights* $\mathbf{W}_3 \in \mathbb{R}^{C\times(J+1)}$
- *We now have a vector* $\mathbf{y}_3 \in \mathbb{R}^C$
- *We get from* $\mathbf{z}_3$ *to* $\mathbf{y}_3$ *via softmax*

Multiclass Classification via FNN: *Softmax Activation*

Let's look at the computation graph: *for a given data-point* $(\boldsymbol{x}_b, v_b)$, *we have*


$$\begin{tikzcd}
 & & & \text{data-point } (\boldsymbol{x}_b, v_b) \arrow[llld, dashed] \arrow[rrrrr, dashed] & & & & & \mathcal{L}(\cdot, v_b) \arrow[r] & \hat{R}_b \\
\mathbf{x} \arrow[r, "\mathbf{W}_1"] & \mathbf{z}_1 \arrow[r, "r(\cdot)"] & \mathbf{y}_1 \arrow[r, "\mathbf{W}_2"] & \mathbf{z}_2 \arrow[r, "r(\cdot)"] & \mathbf{y}_2 \arrow[r, "\mathbf{W}_3"] & \mathbf{z}_3 \arrow[rrr, "\text{Soft}_{\max}(\cdot)"] & & & \mathbf{y}_3 \arrow[u] &
\end{tikzcd}$$

Note that the output layer has been changed

- *We now have a vector* $\mathbf{z}_3 \in \mathbb{R}^C$
  - ↳ *So we have a matrix of weights* $\mathbf{W}_3 \in \mathbb{R}^{C \times (J+1)}$
- *We now have a vector* $\mathbf{y}_3 \in \mathbb{R}^C$
- *We get from* $\mathbf{z}_3$ *to* $\mathbf{y}_3$ *via softmax*
  - ↳ *This is not an entry-wise activation anymore!*

Multiclass Classification via FNN: *Softmax Activation*

1. For each pixel vector $\boldsymbol{x}_b$, we determine $\mathbf{z}_1$ as

$$\mathbf{x} \leftarrow \begin{bmatrix} 1 \\ \boldsymbol{x}_b \end{bmatrix} \rightsquigarrow \mathbf{z}_1 = \mathbf{W}_1^{(t)} \mathbf{x}$$

The output of first layer is then given by $\mathbf{y}_1 = r(\mathbf{z}_1)$

Multiclass Classification via FNN: *Softmax Activation*

1. For each pixel vector $\boldsymbol{x}_b$, we determine $\mathbf{z}_1$ as
$$\mathbf{x} \leftarrow \begin{bmatrix} 1 \\ \boldsymbol{x}_b \end{bmatrix} \rightsquigarrow \mathbf{z}_1 = \mathbf{W}_1^{(t)} \mathbf{x}$$
The output of first layer is then given by $\mathbf{y}_1 = r(\mathbf{z}_1)$

2. We add 1 at index 0 of $\mathbf{y}_1$ and determine $\mathbf{z}_2$ as
$$\mathbf{y}_1 \leftarrow \begin{bmatrix} 1 \\ \mathbf{y}_1 \end{bmatrix} \rightsquigarrow \mathbf{z}_2 = \mathbf{W}_2^{(t)} \mathbf{y}_1$$
The output of second layer is given by $\mathbf{y}_2 = r(\mathbf{z}_2)$

Multiclass Classification via FNN: *Softmax Activation*

1. For each pixel vector $\boldsymbol{x}_b$, we determine $\mathbf{z}_1$ as
$$\mathbf{x} \leftarrow \begin{bmatrix} 1 \\ \boldsymbol{x}_b \end{bmatrix} \rightsquigarrow \mathbf{z}_1 = \mathbf{W}_1^{(t)} \mathbf{x}$$
The output of first layer is then given by $\mathbf{y}_1 = r(\mathbf{z}_1)$
2. We add 1 at index 0 of $\mathbf{y}_1$ and determine $\mathbf{z}_2$ as
$$\mathbf{y}_1 \leftarrow \begin{bmatrix} 1 \\ \mathbf{y}_1 \end{bmatrix} \rightsquigarrow \mathbf{z}_2 = \mathbf{W}_2^{(t)} \mathbf{y}_1$$
The output of second layer is given by $\mathbf{y}_2 = r(\mathbf{z}_2)$
3. We add 1 at index 0 of $\mathbf{y}_2$ and determine $\mathbf{z}_3$ as
$$\mathbf{y}_2 \leftarrow \begin{bmatrix} 1 \\ \mathbf{y}_2 \end{bmatrix} \rightsquigarrow \mathbf{z}_3 = \mathbf{W}_3^{(t)\mathsf{T}} \mathbf{y}_2$$
The network output is given by $\mathbf{y}_3 = \mathit{Soft}_{\max}(\mathbf{z}_3)$

Multiclass Classification: *Loss*

+ *How can we define the loss now? On one side we have a vector of probabilities; one the other side an integer label!*
– Again we need to *convert true labels* to *probabilities*

Multiclass Classification: *Loss*

+ *How can we define the loss now? On one side we have a vector of probabilities; one the other side an integer label!*
– Again we need to *convert* true labels *to* probabilities

Let's say we have C classes: *the vector of probabilities contains C entries*

$$\mathbf{p} = \begin{bmatrix} p_1 \\ \vdots \\ p_C \end{bmatrix} = \begin{bmatrix} \Pr\{\textit{image belongs to class } 1\} \\ \vdots \\ \Pr\{\textit{image belongs to class } C\} \end{bmatrix}$$

Multiclass Classification: *Loss*

+ *How can we define the loss now? On one side we have a vector of probabilities; one the other side an integer label!*
  - Again we need to *convert* *true labels* to *probabilities*

Let's say we have C classes: *the vector of probabilities contains C entries*

$$\mathbf{p} = \begin{bmatrix} p_1 \\ \vdots \\ p_C \end{bmatrix} = \begin{bmatrix} \Pr\{\textit{image belongs to class } 1\} \\ \vdots \\ \Pr\{\textit{image belongs to class } C\} \end{bmatrix}$$

If we know that the image b belongs to class v_b, we could say that

$$\mathbf{p} \textit{ of image } b = \begin{bmatrix} p_1 \\ \vdots \\ p_{v_b} \\ \vdots \\ p_C \end{bmatrix} = \begin{bmatrix} \Pr\{\textit{image } b \textit{ belongs to class } 1\} \\ \vdots \\ \Pr\{\textit{image } b \textit{ belongs to class } v_b\} \\ \vdots \\ \Pr\{\textit{image } b \textit{ belongs to class } C\} \end{bmatrix}$$

Multiclass Classification: *Loss*

+ *How can we define the loss now? On one side we have a vector of probabilities; one the other side an integer label!*
– Again we need to *convert* true labels to *probabilities*

Let's say we have C classes: *the vector of probabilities contains C entries*

$$\mathbf{p} = \begin{bmatrix} p_1 \\ \vdots \\ p_C \end{bmatrix} = \begin{bmatrix} \Pr\{\textit{image belongs to class } 1\} \\ \vdots \\ \Pr\{\textit{image belongs to class } C\} \end{bmatrix}$$

If we know that the image b belongs to class v_b, we could say that

$$\mathbf{p} \textit{ of image } b = \begin{bmatrix} p_1 \\ \vdots \\ p_{v_b} \\ \vdots \\ p_C \end{bmatrix} = \begin{bmatrix} \Pr\{\textit{image } b \textit{ belongs to class } 1\} = 0 \\ \vdots \\ \Pr\{\textit{image } b \textit{ belongs to class } v_b\} = 1 \\ \vdots \\ \Pr\{\textit{image } b \textit{ belongs to class } C\} = 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Multiclass Classification: *Loss*

So, we could say that *label v is corresponding to a vector of size C whose entry v is 1 and the remaining entries are all 0*: this vector is called a *one-hot vector*

Multiclass Classification: *Loss*

So, we could say that *label v is corresponding to a vector of size C whose entry v is 1 and the remaining entries are all 0*: this vector is called a *one-hot vector*

One-hot Vector

The one-hot vector $\mathbf{1}_v \in \{0,1\}^C$ is a C-dimensional vector whose entry v is 1 and all remaining entries are 0

For instance: *say $C = 3$; then, we have*

$$\mathbf{1}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{1}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \mathbf{1}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Multiclass Classification: *Loss*

So, we could say that *label v is corresponding to a vector of size C whose entry v is 1 and the remaining entries are all 0*: this vector is called a *one-hot vector*

One-hot Vector

The one-hot vector $\mathbf{1}_v \in \{0,1\}^C$ is a C-dimensional vector whose entry v is 1 and all remaining entries are 0

For instance: *say $C = 3$; then, we have*

$$\mathbf{1}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{1}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \mathbf{1}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Moral of Story

We can interpret true label v as a probability vector $\mathbf{1}_v$

Multiclass Classification: *Loss*

We can interpret true label v as a probability vector $\mathbf{1}_v$

Now for image b with label v_b we compare network's output $\mathbf{y}_3$ to $\mathbf{1}_{v_b}$

$$\hat{R}_b = \mathcal{L}\left(\mathbf{y}_3, \mathbf{1}_{v_b}\right)$$

for loss $\mathcal{L}(\cdot)$ that determines distance between two *probability vectors*

Multiclass Classification: *Loss*

We can interpret true label v as a probability vector $\mathbf{1}_v$

Now for image b with label v_b we compare network's output $\mathbf{y}_3$ to $\mathbf{1}_{v_b}$

$$\hat{R}_b = \mathcal{L}\left(\mathbf{y}_3, \mathbf{1}_{v_b}\right)$$

for loss $\mathcal{L}(\cdot)$ that determines distance between two *probability vectors*

+ *What kind of loss functions do we use usually?*
– Like binary case: squared error is good, cross-entropy is the best
+ *How do we define them in this case?*
– Just extend them to multi-dimensional vectors

Multiclass Classification: *Loss*

We can extend squared error to vector form as

$$\mathcal{L}(\mathbf{y}, \mathbf{1}_v) = \|\mathbf{y} - \mathbf{1}_v\|^2 = \sum_{c=1}^{C} (y[c] - \mathbb{1}\{c = v\})^2$$
$$= \sum_{c=1, c\neq v}^{C} y[c]^2 + (y[v] - 1)^2$$

This gradient of this loss is

$$\nabla\mathcal{L}(\mathbf{y}, \mathbf{1}_v) = 2 \begin{bmatrix} y[1] \\ \vdots \\ y[v] - 1 \\ \vdots \\ y[C] \end{bmatrix} = 2(\mathbf{y} - \mathbf{1}_v)$$

Multiclass Classification: *Loss*

Cross entropy can also be extended as follows

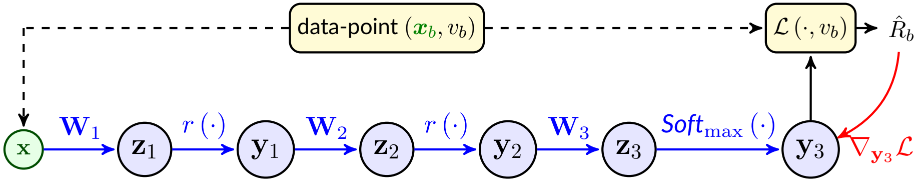
$$\mathcal{L}\left(\mathbf{y}, \mathbf{1}_v\right) = \mathrm{CE}\left(\mathbf{y}, \mathbf{1}_v\right) = -\sum_{c=1}^{C} \mathbb{1}\left\{c = v\right\} \log y\left(c\right)$$
$$= -\log y\left[v\right]$$

The gradient of this loss is

$$\nabla\mathcal{L}\left(\mathbf{y}, \mathbf{1}_v\right) = \nabla\mathrm{CE}\left(\mathbf{y}, \mathbf{1}_v\right) = \begin{bmatrix} 0 \\ \vdots \\ -1/y\left[v\right] \\ \vdots \\ 0 \end{bmatrix} = -\frac{1}{y\left[v\right]}\mathbf{1}_v$$

Backpropagation Through *Vector Activation*

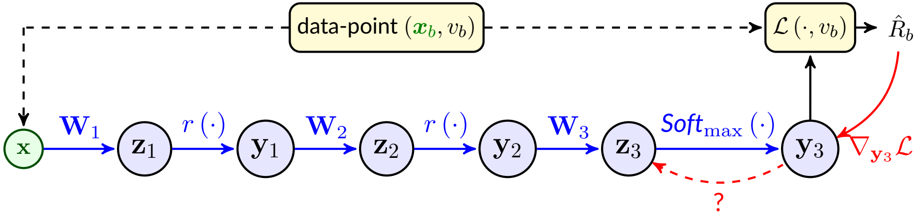
How can we backpropagate through this neural network?



1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$

Backpropagation Through *Vector Activation*

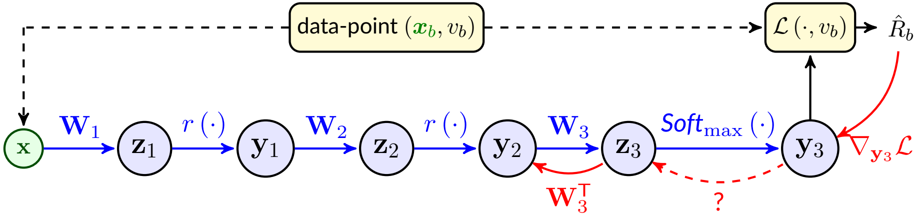
How can we backpropagate through this neural network?



1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$
2. Compute $\nabla_{\mathbf{z}_3}\hat{R}_b$ from $\nabla_{\mathbf{y}_3}\hat{R}_b$ *by its backward link that* $\boxed{\textit{we don't know yet}}$

Backpropagation Through *Vector Activation*

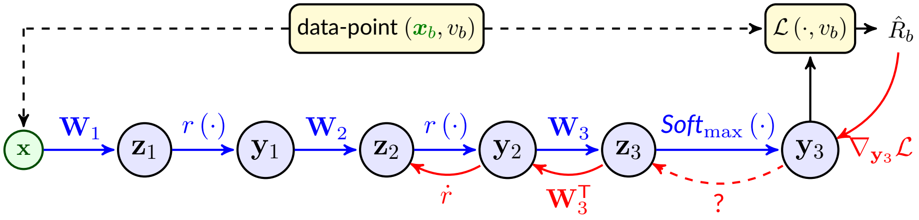
How can we backpropagate through this neural network?



1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$
2. Compute $\nabla_{\mathbf{z}_3}\hat{R}_b$ from $\nabla_{\mathbf{y}_3}\hat{R}_b$ *by its backward link that* $\boxed{\textit{we don't know yet}}$
3. Compute $\nabla_{\mathbf{y}_2}\hat{R}_b = \mathbf{W}_3^\mathsf{T}\nabla_{\mathbf{z}_3}\hat{R}_b$

Backpropagation Through *Vector Activation*

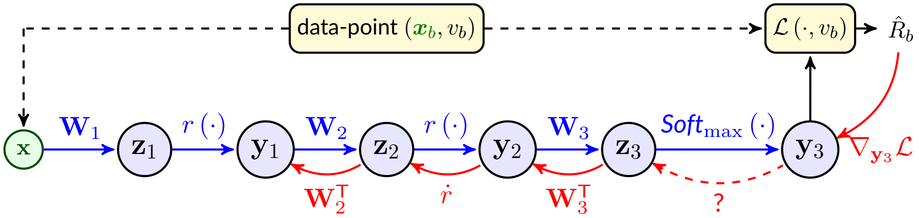
How can we backpropagate through this neural network?



1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$
2. Compute $\nabla_{\mathbf{z}_3}\hat{R}_b$ from $\nabla_{\mathbf{y}_3}\hat{R}_b$ *by its backward link that* *we don't know yet*
3. Compute $\nabla_{\mathbf{y}_2}\hat{R}_b = \mathbf{W}_3^\top \nabla_{\mathbf{z}_3}\hat{R}_b$
4. Remove entry at index 0 of $\nabla_{\mathbf{y}_2}\hat{R}_b$ and compute $\nabla_{\mathbf{z}_2}\hat{R}_b = \dot{r}(\mathbf{z}_2) \odot \nabla_{\mathbf{y}_2}\hat{R}_b$

Backpropagation Through *Vector Activation*

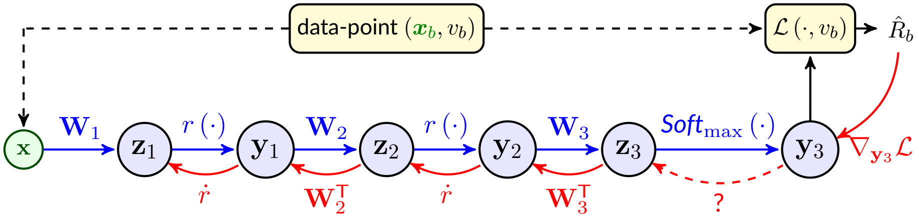
How can we backpropagate through this neural network?



1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$
2. Compute $\nabla_{\mathbf{z}_3}\hat{R}_b$ from $\nabla_{\mathbf{y}_3}\hat{R}_b$ *by its backward link that* $\boxed{\textit{we don't know yet}}$
3. Compute $\nabla_{\mathbf{y}_2}\hat{R}_b = \mathbf{W}_3^\top\nabla_{\mathbf{z}_3}\hat{R}_b$
4. Remove entry at index 0 of $\nabla_{\mathbf{y}_2}\hat{R}_b$ and compute $\nabla_{\mathbf{z}_2}\hat{R}_b = \dot{r}(\mathbf{z}_2) \odot \nabla_{\mathbf{y}_2}\hat{R}_b$
5. Compute $\nabla_{\mathbf{y}_1}\hat{R}_b = \mathbf{W}_2^\top\nabla_{\mathbf{z}_2}\hat{R}_b$

Backpropagation Through *Vector Activation*

How can we backpropagate through this neural network?

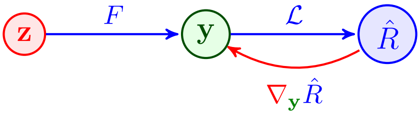


1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$
2. Compute $\nabla_{\mathbf{z}_3}\hat{R}_b$ from $\nabla_{\mathbf{y}_3}\hat{R}_b$ *by its backward link that* $\boxed{\textit{we don't know yet}}$
3. Compute $\nabla_{\mathbf{y}_2}\hat{R}_b = \mathbf{W}_3^\top \nabla_{\mathbf{z}_3}\hat{R}_b$
4. Remove entry at index 0 of $\nabla_{\mathbf{y}_2}\hat{R}_b$ and compute $\nabla_{\mathbf{z}_2}\hat{R}_b = \dot{r}(\mathbf{z}_2) \odot \nabla_{\mathbf{y}_2}\hat{R}_b$
5. Compute $\nabla_{\mathbf{y}_1}\hat{R}_b = \mathbf{W}_2^\top \nabla_{\mathbf{z}_2}\hat{R}_b$
6. Remove entry at index 0 of $\nabla_{\mathbf{y}_1}\hat{R}_b$ and compute $\nabla_{\mathbf{z}_1}\hat{R}_b = \dot{r}(\mathbf{z}_1) \odot \nabla_{\mathbf{y}_1}\hat{R}_b$

Backpropagation Through *Vector Activation*

How is $\nabla_{\mathbf{z}_3} \hat{R}_b$ related to $\nabla_{\mathbf{y}_3} \hat{R}_b$? Let's do what we did before

*In this graph, $F(\cdot)$ is a **vector activation**. We know $\nabla_{\mathbf{y}} \hat{R}$*



$$\begin{tikzcd} \mathbf{z} \arrow[r, "F"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

We want to find $\nabla_{\mathbf{z}} \hat{R}$

As mentioned before: *we can always extend things entry-wise*

With vector activation, we need to use the notion of Jacobian

Recap: *Jacobian Matrix*

Consider vector activation $F(\cdot)$ that maps C-dimensional $\mapsto$ C-dimensional

$$\begin{bmatrix} y_1 \\ \vdots \\ y_C \end{bmatrix} = F\left(\begin{bmatrix} z_1 \\ \vdots \\ z_C \end{bmatrix}\right)$$

When we use this function, we can say

Any entry y_j is function of all[a] $z_1, \ldots, z_C$, *so we have*

$$\nabla_{\mathbf{z}} y_j = \begin{bmatrix} \partial y_j / \partial z_1 \\ \vdots \\ \partial y_j / \partial z_C \end{bmatrix}$$

[a] It is not any more an entry-wise functional operation

Recap: *Jacobian Matrix*

Consider vector activation $F(\cdot)$ that maps C-dimensional $\mapsto$ C-dimensional

$$\begin{bmatrix} y_1 \\ \vdots \\ y_C \end{bmatrix} = F\left(\begin{bmatrix} z_1 \\ \vdots \\ z_C \end{bmatrix}\right)$$

When we use this function, we can say

We can collect all these gradients into a matrix

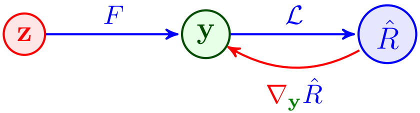
$$\mathbf{J}_{\mathbf{z}}\mathbf{y} = \mathbf{J}_{\mathbf{z}}F = \begin{bmatrix} \nabla_{\mathbf{z}} y_1^{\mathsf{T}} \\ \vdots \\ \nabla_{\mathbf{z}} y_C^{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \partial y_1 / \partial z_1 & \dots & \partial y_1 / \partial z_C \\ \vdots & & \vdots \\ \partial y_C / \partial z_1 & \dots & \partial y_C / \partial z_C \end{bmatrix}$$

and we call it the Jacobian matrix

Backpropagation Through *Vector Activation*

Now, let's get back to our problem

In this graph, $F(\cdot)$ is a **vector activation**. *We know $\nabla_{\mathbf{y}} \hat{R}$*


$$\begin{tikzcd} \mathbf{z} \arrow[r, "F"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

We want to find $\nabla_{\mathbf{z}} \hat{R}$: let's write down a partial derivative $\hat{R}$ w.r.t. z_c

$$\frac{\partial \hat{R}}{\partial z_c} = \sum_{j=1}^{C} \frac{\partial \hat{R}}{\partial y_j} \frac{\partial y_j}{\partial z_c} =$$

Backpropagation Through *Vector Activation*

Now, let's get back to our problem

In this graph, $F(\cdot)$ is a ***vector activation****. We know $\nabla_{\mathbf{y}} \hat{R}$*

$$\begin{tikzcd}
\mathbf{z} \arrow[r, "F"] & \mathbf{y} \arrow[r, "\mathcal{L}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"]
\end{tikzcd}$$

We want to find $\nabla_{\mathbf{z}} \hat{R}$: let's write down a partial derivative $\hat{R}$ w.r.t. z_c

$$\frac{\partial \hat{R}}{\partial z_c} = \sum_{j=1}^{C} \frac{\partial \hat{R}}{\partial y_j} \frac{\partial y_j}{\partial z_c} = \underbrace{\begin{bmatrix} \frac{\partial y_1}{\partial z_c} & \cdots & \frac{\partial y_C}{\partial z_c} \end{bmatrix}}_{\text{transpose of column } c \text{ of } \mathbf{J}_{\mathbf{z}\mathbf{y}} \equiv \text{ row } c \text{ of } \mathbf{J}_{\mathbf{z}\mathbf{y}}^{\mathsf{T}}} \nabla_{\mathbf{y}} \hat{R}$$

Backpropagation Through *Vector Activation*

So, the gradient of $\hat{R}$ w.r.t. $\mathbf{z}$ is given by

$$\nabla_{\mathbf{z}} \hat{R} \begin{bmatrix} \partial \hat{R} / \partial z_1 \\ \vdots \\ \partial \hat{R} / \partial z_C \end{bmatrix} = \begin{bmatrix} \text{row } 1 \text{ of } \mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y} \nabla_{\mathbf{y}} \hat{R} \\ \vdots \\ \text{row } C \text{ of } \mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y} \nabla_{\mathbf{y}} \hat{R} \end{bmatrix} = \left(\mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y} \right) \nabla_{\mathbf{y}} \hat{R}$$

So, we can complete the computation graph as follows

$$\begin{tikzcd} \mathbf{z} \arrow[r, "F"] & \mathbf{y} \arrow[r, "\mathcal{L}"] \arrow[l, bend left, "\mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"] \end{tikzcd}$$

Backpropagation Through *Vector Activation*

So, the gradient of $\hat{R}$ w.r.t. $\mathbf{z}$ is given by

$$\nabla_{\mathbf{z}} \hat{R} \begin{bmatrix} \partial \hat{R} / \partial z_1 \\ \vdots \\ \partial \hat{R} / \partial z_C \end{bmatrix} = \begin{bmatrix} \text{row } 1 \text{ of } \mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y} \nabla_{\mathbf{y}} \hat{R} \\ \vdots \\ \text{row } C \text{ of } \mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y} \nabla_{\mathbf{y}} \hat{R} \end{bmatrix} = \left(\mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y} \right) \nabla_{\mathbf{y}} \hat{R}$$

So, we can complete the computation graph as follows

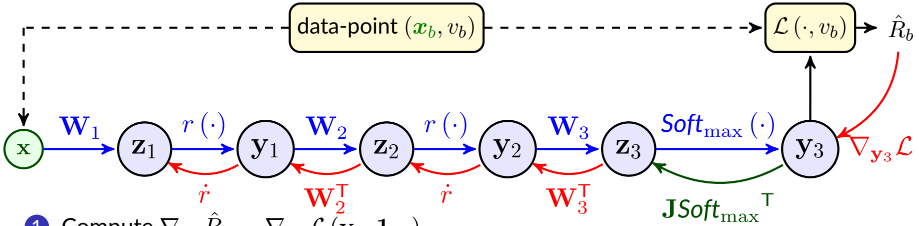
$$\begin{tikzcd}
\mathbf{z} \arrow[r, "F"] & \mathbf{y} \arrow[r, "\mathcal{L}"] \arrow[l, bend left, "\mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"]
\end{tikzcd}$$

Backward Pass of Vector Activation

To pass backward on a vector activation, we use the transpose of its Jacobian

Backpropagation Through *Vector Activation*

How can we backpropagate through this neural network? Let's complete


data-point $(\boldsymbol{x}_b, v_b)$
$\mathcal{L}(\cdot, v_b)$
$\hat{R}_b$
$\mathbf{x}$
$\mathbf{W}_1$
$\mathbf{z}_1$
$r(\cdot)$
$\mathbf{y}_1$
$\mathbf{W}_2$
$\mathbf{z}_2$
$r(\cdot)$
$\mathbf{y}_2$
$\mathbf{W}_3$
$\mathbf{z}_3$
$\mathit{Soft}_{\max}(\cdot)$
$\mathbf{y}_3$
$\nabla_{\mathbf{y}_3}\mathcal{L}$
$\dot{r}$
$\mathbf{W}_2^\mathsf{T}$
$\dot{r}$
$\mathbf{W}_3^\mathsf{T}$
$\mathbf{J}\mathit{Soft}_{\max}{}^\mathsf{T}$

1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$
2. $\boxed{\text{Compute } \nabla_{\mathbf{z}_3}\hat{R}_b = (\mathbf{J}\mathit{Soft}_{\max})^\mathsf{T} \nabla_{\mathbf{y}_3}\hat{R}_b}$
3. Compute $\nabla_{\mathbf{y}_2}\hat{R}_b = \mathbf{W}_3^\mathsf{T}\nabla_{\mathbf{z}_3}\hat{R}_b$
4. Remove entry at index 0 of $\nabla_{\mathbf{y}_2}\hat{R}_b$ and compute $\nabla_{\mathbf{z}_2}\hat{R}_b = \dot{r}(\mathbf{z}_2) \odot \nabla_{\mathbf{y}_2}\hat{R}_b$
5. Compute $\nabla_{\mathbf{y}_1}\hat{R}_b = \mathbf{W}_2^\mathsf{T}\nabla_{\mathbf{z}_2}\hat{R}_b$
6. Remove entry at index 0 of $\nabla_{\mathbf{y}_1}\hat{R}_b$ and compute $\nabla_{\mathbf{z}_1}\hat{R}_b = \dot{r}(\mathbf{z}_1) \odot \nabla_{\mathbf{y}_1}\hat{R}_b$

Multiclass Classification via FNN: *Training*

Let's now recall gradient descent for training of multiclass classifier

```
GradientDescent():
1: Initiate with some initial values $\{\mathbf{W}_1^{(0)}, \mathbf{W}_2^{(0)}, \mathbf{W}_3^{(0)}\}$ and set a learning rate $\eta$
2: while weights not converged do
3:     for $b = 1, \ldots, B$ do
4:         NN.values ← ForwardProp $(\boldsymbol{x}_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\})$
5:         $\{\mathbf{G}_{1,b}, \mathbf{G}_{2,b}, \mathbf{G}_{3,b}\}$ ← BackProp $(\boldsymbol{x}_b, v_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\}$, NN.values$)$
6:     end for
7:     Update
           $\mathbf{W}_1{}^{(t+1)} \leftarrow \mathbf{W}_1{}^{(t)} - \eta \ \mathtt{mean}\left(\mathbf{G}_{1,1}, \ldots, \mathbf{G}_{1,B}\right)$
           $\mathbf{W}_2{}^{(t+1)} \leftarrow \mathbf{W}_2{}^{(t)} - \eta \ \mathtt{mean}\left(\mathbf{G}_{2,1}, \ldots, \mathbf{G}_{2,B}\right)$
           $\mathbf{W}_3{}^{(t+1)} \leftarrow \mathbf{W}_3{}^{(t)} - \eta \ \mathtt{mean}\left(\mathbf{G}_{3,1}, \ldots, \mathbf{G}_{3,B}\right)$
8: end while
```

We call this form of gradient descent full-batch

Full-Batch Training

Batch $\equiv$ *the dataset reserved for training*

In full-batch training, we calculate the gradient for all data-points in the batch: *so, we need to wait till forward and backward pass are over for all B data-points*

This can be a huge burdensome!

Full-Batch Training

Batch $\equiv$ *the dataset reserved for training*

In full-batch training, we calculate the gradient for all data-points in the batch: *so, we need to wait till forward and backward pass are over for all B data-points*

This can be a huge burdensome!

+ *Wait a moment! Don't we use all the dataset for training?*
– No! As you may have noticed in the assignments, *we reserve a part of it for testing*
+ *And, why should it be a burdensome?*
– OK! Let's get more into datasets!

Public Datasets

Let's consider our example of image recognition: *we want to recognize the hand-written number in an image*. For this, we need to have access *to images of hand-written numbers*. This has been done before by people at *National Institute of Standards and Technology* and collected in a database called

Modified National Institute of Standards and Technology (MNIST)

that is available for public on internet

Public Datasets

Let's consider our example of image recognition: *we want to recognize the hand-written number in an image*. For this, we need to have access *to images of hand-written numbers*. This has been done before by people at *National Institute of Standards and Technology* and collected in a database called

Modified National Institute of Standards and Technology (MNIST)

that is available for public on internet

There are several of such public databases; *some well-known examples are*

- *CIFAR-10 and CIFAR-100 by Canadian Institute For Advanced Research*
- *ImageNet initiated by Fei-Fei Li at Princeton University*
- *Caltech-101 and Caltech-256 compiled at Caltech*
- *Fashion MNIST that collects fashion images and labels them*

You can find out more about public datasets online

Public Datasets: *Accessing via PyTorch*

PyTorch provides us a simple tool to access these public datasets, e.g.,

```
>> import torchvision.datasets as DataSets

>> dataset = DataSets.MNIST( ... )
>> dataset = DataSets.CIFAR10( ... )
```

Public Datasets: *Accessing via PyTorch*

PyTorch provides us a simple tool to access these public datasets, e.g.,

```
>> import torchvision.datasets as DataSets

>> dataset = DataSets.MNIST( ... )
>> dataset = DataSets.CIFAR10( ... )
```

In the example of MNIST, we load the dataset which contains the pixel vectors of the images of size 28×28. This means that we load a *list of pairs were each pair contains*

a 784-dimensional vector of pixel values and a label that is in $\{0, 1, \dots, 9\}$

Public Datasets: *How Do They Look?*

Public datasets include a large amount of data-points with their labels

MNIST includes 70,000 images of hand-written numbers with their true labels: from these 70,000 we use 60,000 for training and 10,000 for test

This means that once we load the MNIST dataset, we make a batch of 60,000 images to train our FNN. Once the training is over, we test the performance of the trained FNN on the remaining 10,000 images

Public Datasets: *How Do They Look?*

Public datasets include a large amount of data-points with their labels

MNIST includes 70,000 images of hand-written numbers with their true labels: from these 70,000 we use 60,000 for training and 10,000 for test

This means that once we load the MNIST dataset, we make a batch of 60,000 images to train our FNN. Once the training is over, we test the performance of the trained FNN on the remaining 10,000 images

---

Back to our problem, this means that *our full-batch training performs each iteration of the gradient descent after*

60,000 forward and backward passes over the FNN

which sounds a lot!

Full-Batch Training: *Complexity*

Given the example of MNIST, let's see roughly how long it takes to do a full-batch training: *if we need 100 iterations of gradient descent, we need to pass back and forth for 6×10^6 times!*

+ *But do we really need to do this much? This sounds impossible in large NNs!*
- No! We really don't need! We can do the training much faster

Full-Batch Training: *Complexity*

Given the example of MNIST, let's see roughly how long it takes to do a full-batch training: *if we need 100 iterations of gradient descent, we need to pass back and forth for* 6×10^6 *times!*

+ *But do we really need to do this much? This sounds impossible in large NNs!*
– No! We really don't need! We can do the training much faster

---

The full-batch training is really not practical: *in practice, we use stochastic (mini-batch) gradient descent to train our NN with feasible complexity*

Let's take a look at these approaches!

Sample-Level Training

The most primary idea is to apply one step of gradient descent after each forward and backward pass: *in our FNN this means that we do the following*

Sample-Level Training

The most primary idea is to apply one step of gradient descent after each forward and backward pass: *in our FNN this means that we do the following*

```
SymbLevel_GradientDescent():
 1: Initiate with some initial values $\{\mathbf{W}_1^{(0)}, \mathbf{W}_2^{(0)}, \mathbf{W}_3^{(0)}\}$ and set a learning rate $\eta$
 2: Start at $b = 1$
 3: while weights not converged do
 4:     if $b > B$ then
 5:         Update $b \leftarrow 1$                         # start over with the dataset
 6:     end if
 7:     NN.values $\leftarrow$ ForwardProp $(\boldsymbol{x}_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\})$
 8:     $\{\mathbf{G}_{1,b}, \mathbf{G}_{2,b}, \mathbf{G}_{3,b}\}$ $\leftarrow$ BackProp $(\boldsymbol{x}_b, v_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\}$, NN.values$)$
 9:     Update $\mathbf{W}_\ell^{(t+1)} \leftarrow \mathbf{W}_\ell^{(t)} - \eta\, \mathbf{G}_{\ell,b}$          # symbol_level update
10:     Update $b \leftarrow b + 1$                      # go for next data-point
11: end while
```

We call this approach *symbol-level training*

Sample-Level Training: *Meaning*

+ *But what does it mean in the sense of empirical risk minimization? Aren't we now doing something different from the standard gradient descent?!*
– Yes! We are in fact performing an approximative gradient descent

Sample-Level Training: *Meaning*

+ *But what does it mean in the sense of empirical risk minimization? Aren't we now doing something different from the standard gradient descent?!*
– Yes! We are in fact performing an approximative gradient descent

---

Consider the an ideal scenario in which

$$\mathbf{G}_{\ell,1} = \mathbf{G}_{\ell,2} = \ldots = \mathbf{G}_{\ell,B}$$

In this case, we do not need to wait for the batch to be fully over, since

$$\mathbf{G}_{\ell,1} = \texttt{mean}\ (\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \ldots, \mathbf{G}_{\ell,B})$$

In other words, in this case

symbol-level training $\equiv$ *full-batch training*

Sample-Level Training: *Meaning*

In practice, at each data-point we calculate a noisy-version of full-batch gradient[a]*. In other words, we can think of* $\mathbf{G}_{\ell,b}$ *for each* b *as*

$$\mathbf{G}_{\ell,b} = \texttt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \ldots, \mathbf{G}_{\ell,B}) + \text{Noise}$$

[a] Full-batch gradient means the average of gradients over the whole batch.

Sample-Level Training: *Meaning*

In practice, at each data-point we calculate a noisy-version of full-batch gradient[a]*. In other words, we can think of $\mathbf{G}_{\ell,b}$ for each b as*

$$\mathbf{G}_{\ell,b} = \texttt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \dots, \mathbf{G}_{\ell,B}) + \text{Noise}$$

[a] Full-batch gradient means the average of gradients over the whole batch.

If this noise is small enough, we can say that

$$\mathbf{G}_{\ell,b} \approx \texttt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \dots, \mathbf{G}_{\ell,B})$$

and therefore, we can conclude that

symbol-level training $\approx$ full-batch training

In this case, we say that $\mathbf{G}_{\ell,b}$ is an estimator of the full-batch gradient

Sample-Level Training: *Repetitive Cycle Issue*

Naive sample-level update can trap us into a repetitive cycle: *in simple words, we can end up with our initial point at the end of the batch.*

Sample-Level Training: *Repetitive Cycle Issue*

Naive sample-level update can trap us into a repetitive cycle: *in simple words, we can end up with our initial point at the end of the batch. For instance, consider the following dummy (but possible) scenario in our three-layer FNN*

We start with $\mathbf{W}_\ell^{(0)}$ and get into the batch for the first time

- *We update $\mathbf{W}_\ell^{(0)}$ after the first data-point to $\mathbf{W}_\ell^{(1)}$*
- *We update $\mathbf{W}_\ell^{(1)}$ after the second data-point to $\mathbf{W}_\ell^{(2)}$*
- $\ldots$
- *We update $\mathbf{W}_\ell^{(B-1)}$ after the last data-point to $\mathbf{W}_\ell^{(B)}$*

Now, assume that $\mathbf{W}_\ell^{(B)} = \mathbf{W}_\ell^{(0)}$ for all layers again!

Sample-Level Training: *Repetitive Cycle Issue*

Naive sample-level update can trap us into a repetitive cycle: *in simple words, we can end up with our initial point at the end of the batch. For instance, consider the following dummy (but possible) scenario in our three-layer FNN*

We start with $\mathbf{W}_\ell^{(0)}$ and get into the batch for the first time

- *We update $\mathbf{W}_\ell^{(0)}$ after the first data-point to $\mathbf{W}_\ell^{(1)}$*
- *We update $\mathbf{W}_\ell^{(1)}$ after the second data-point to $\mathbf{W}_\ell^{(2)}$*
- $\ldots$
- *We update $\mathbf{W}_\ell^{(B-1)}$ after the last data-point to $\mathbf{W}_\ell^{(B)}$*

Now, assume that $\mathbf{W}_\ell^{(B)} = \mathbf{W}_\ell^{(0)}$ for all layers again!

In the above dummy example, further looping over the batch is *useless*, since *we always get back to the initial point: this is the most basic example of the repetitive cycle issue*

Stochastic Sample-Level Training: *SGD*

+ *How can we avoid such cyclic behaviors?*
– We can use *Stochastic Gradient Descent (SGD)*

Stochastic Sample-Level Training: *SGD*

+ *How can we avoid such cyclic behaviors?*
- We can use *Stochastic Gradient Descent (SGD)*

Each time we are to loop over our training batch, *we shuffle the data-points randomly: this way we avoid next loop behave like the previous one*

This idea is called Stochastic Gradient Descent (SGD)

SGD is the most common algorithm for training of NNs!

Stochastic Sample-Level Training: *SGD*

+ *How can we avoid such cyclic behaviors?*

– We can use *Stochastic Gradient Descent (SGD)*

Each time we are to loop over our training batch, *we shuffle the data-points randomly: this way we avoid next loop behave like the previous one*

This idea is called Stochastic Gradient Descent (SGD)

SGD is the most common algorithm for training of NNs!

What does random shuffling mean?

It means randomly permuting the data-points

Stochastic Gradient Descent

```
SGD():
 1: Initiate with some initial values $\{\mathbf{W}_1^{(0)}, \mathbf{W}_2^{(0)}, \mathbf{W}_3^{(0)}\}$ and set a learning rate $\eta$
 2: Randomly shuffle the batch and start at $b = 1$
 3: while weights not converged do
 4:     if $b > B$ then
 5:         Randomly shuffle the batch and set $b \leftarrow 1$          # random shuffling
 6:     end if
 7:     NN.values $\leftarrow$ ForwardProp $(\boldsymbol{x}_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\})$
 8:     $\{\mathbf{G}_{1,b}, \mathbf{G}_{2,b}, \mathbf{G}_{3,b}\} \leftarrow$ BackProp $(\boldsymbol{x}_b, v_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\}$, NN.values$)$
 9:     Update $\mathbf{W}_\ell^{(t+1)} \leftarrow \mathbf{W}_\ell^{(t)} - \eta\ \mathbf{G}_{\ell,b}$          # symbol_level update
10:     Update $b \leftarrow b + 1$          # go for next data-point
11: end while
```

Stochastic Gradient Descent

```
SGD():
 1: Initiate with some initial values $\{\mathbf{W}_1^{(0)}, \mathbf{W}_2^{(0)}, \mathbf{W}_3^{(0)}\}$ and set a learning rate $\eta$
 2: Randomly shuffle the batch and start at $b = 1$
 3: while weights not converged do
 4:     if $b > B$ then
 5:         Randomly shuffle the batch and set $b \leftarrow 1$          # random shuffling
 6:     end if
 7:     NN.values ← ForwardProp $(\boldsymbol{x}_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\})$
 8:     $\{\mathbf{G}_{1,b}, \mathbf{G}_{2,b}, \mathbf{G}_{3,b}\}$ ← BackProp $(\boldsymbol{x}_b, v_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\}$, NN.values)
 9:     Update $\mathbf{W}_\ell^{(t+1)} \leftarrow \mathbf{W}_\ell^{(t)} - \eta\, \mathbf{G}_{\ell,b}$          # symbol_level update
10:     Update $b \leftarrow b + 1$          # go for next data-point
11: end while
```

+ *But, doesn't sample-level training lead to any drawback?*
– Sure! But we accept this drawback as a cost we pay for less complexity

Let's see how this trade-off looks like

Recap: *Variance*

For random variable x with mean μ, the variance is defined as

$$\mathbb{Var}\left\{x\right\} = \mathbb{E}\left\{(x-\mu)^2\right\} = \mathbb{E}\left\{x^2\right\} - \mu^2$$

Clearly, when x is zero-mean, we can say $\mathbb{Var}\left\{x\right\} = \mathbb{E}\left\{x^2\right\}$

Recap: *Variance*

For random variable x with mean μ, the variance is defined as

$$\mathbb{V}\mathrm{ar}\left\{x\right\} = \mathbb{E}\left\{(x-\mu)^2\right\} = \mathbb{E}\left\{x^2\right\} - \mu^2$$

Clearly, when x is zero-mean, we can say $\mathbb{V}\mathrm{ar}\left\{x\right\} = \mathbb{E}\left\{x^2\right\}$

Properties of Variance

For any random variable x and constant c, we have

$$\mathbb{V}\mathrm{ar}\left\{cx\right\} = c^2 \mathbb{V}\mathrm{ar}\left\{x\right\}$$

Let $x_1, \ldots, x_N$ be N independent random variables; then, we have

$$\mathbb{V}\mathrm{ar}\left\{\sum_{n=1}^{N} x_n\right\} = \sum_{n=1}^{N} \mathbb{V}\mathrm{ar}\left\{x_n\right\}$$

Recap: *Variance*

Now, assume $x_1, \ldots, x_N$ are N independent zero-mean random variables all with variance σ^2: *let* $\bar{x}$ *be the arithmetic average of* $x_1, \ldots, x_N$, *i.e.*,

$$\bar{x} = \texttt{mean}\left(x_1, \ldots, x_N\right) = \frac{1}{N} \sum_{n=1}^{N} x_n$$

We could then say

$$\mathbb{Var}\left\{\bar{x}\right\} = \mathbb{Var}\left\{\frac{1}{N} \sum_{n=1}^{N} x_n\right\}$$

Recap: *Variance*

Now, assume $x_1, \dots, x_N$ are N independent zero-mean random variables all with variance σ^2: *let $\bar{x}$ be the arithmetic average of $x_1, \dots, x_N$, i.e.,*

$$\bar{x} = \texttt{mean}\left(x_1, \dots, x_N\right) = \frac{1}{N} \sum_{n=1}^{N} x_n$$

We could then say

$$\mathbb{Var}\left\{\bar{x}\right\} = \mathbb{Var}\left\{\frac{1}{N} \sum_{n=1}^{N} x_n\right\} = \frac{1}{N^2} \mathbb{Var}\left\{\sum_{n=1}^{N} x_n\right\}$$

Recap: *Variance*

Now, assume $x_1, \dots, x_N$ are N independent zero-mean random variables all with variance σ^2: *let $\bar{x}$ be the arithmetic average of* $x_1, \dots, x_N$, *i.e.,*

$$\bar{x} = \mathtt{mean}\left(x_1, \dots, x_N\right) = \frac{1}{N}\sum_{n=1}^{N} x_n$$

We could then say

$$\mathbb{Var}\left\{\bar{x}\right\} = \mathbb{Var}\left\{\frac{1}{N}\sum_{n=1}^{N} x_n\right\} = \frac{1}{N^2}\mathbb{Var}\left\{\sum_{n=1}^{N} x_n\right\}$$
$$= \frac{1}{N^2}\sum_{n=1}^{N} \underbrace{\mathbb{Var}\left\{x_n\right\}}_{\sigma^2}$$

Recap: *Variance*

Now, assume $x_1, \ldots, x_N$ are N independent zero-mean random variables all with variance σ^2: *let $\bar{x}$ be the arithmetic average of $x_1, \ldots, x_N$, i.e.,*

$$\bar{x} = \texttt{mean}\left(x_1, \ldots, x_N\right) = \frac{1}{N} \sum_{n=1}^{N} x_n$$

We could then say

$$\begin{aligned} \mathbb{Var}\left\{\bar{x}\right\} &= \mathbb{Var}\left\{\frac{1}{N} \sum_{n=1}^{N} x_n\right\} = \frac{1}{N^2} \mathbb{Var}\left\{\sum_{n=1}^{N} x_n\right\} \\ &= \frac{1}{N^2} \sum_{n=1}^{N} \underbrace{\mathbb{Var}\left\{x_n\right\}}_{\sigma^2} = \frac{1}{N^2} \left(N\sigma^2\right) \\ &= \frac{\sigma^2}{N} \end{aligned}$$

Recap: *Variance*

Now, assume $x_1, \ldots, x_N$ are N independent zero-mean random variables all with variance σ^2: *let $\bar{x}$ be the arithmetic average of $x_1, \ldots, x_N$, i.e.,*

$$\bar{x} = \mathtt{mean}\left(x_1, \ldots, x_N\right) = \frac{1}{N} \sum_{n=1}^{N} x_n$$

We could then say

$$\begin{aligned} \mathbb{Var}\left\{\bar{x}\right\} &= \mathbb{Var}\left\{\frac{1}{N} \sum_{n=1}^{N} x_n\right\} = \frac{1}{N^2} \mathbb{Var}\left\{\sum_{n=1}^{N} x_n\right\} \\ &= \frac{1}{N^2} \sum_{n=1}^{N} \underbrace{\mathbb{Var}\left\{x_n\right\}}_{\sigma^2} = \frac{1}{N^2}\left(N\sigma^2\right) \\ &= \frac{\sigma^2}{N} \end{aligned}$$

variance of average drops by $1/\#$ samples

Complexity-Accuracy Trade-off of SGD

Now, let's get back to our problem: *when we talked about the meaning of symbol level update, we said*

> *In practice, at each data-point we calculate a noisy-version of full-batch gradient. In other words, we can think of $\mathbf{G}_{\ell,b}$ for each b as*
>
> $$\mathbf{G}_{\ell,b} = \mathtt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \ldots, \mathbf{G}_{\ell,B}) + \text{Noise}$$

and called $\mathbf{G}_{\ell,b}$ an estimator of the mean.

Complexity-Accuracy Trade-off of SGD

Now, let's get back to our problem: *when we talked about the meaning of symbol level update, we said*

> *In practice, at each data-point we calculate a noisy-version of full-batch gradient. In other words, we can think of $\mathbf{G}_{\ell,b}$ for each b as*
>
> $$\mathbf{G}_{\ell,b} = \mathtt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \ldots, \mathbf{G}_{\ell,B}) + \mathsf{Noise}$$

and called $\mathbf{G}_{\ell,b}$ an estimator of the mean.

Let's make the above statement a bit more formal: *we assume that Noise for each b is a matrix with independent zero-mean entries all with variance σ^2, i.e.,*

$$\mathbf{G}_{\ell,b} = \bar{\mathbf{G}}_{\ell} + \mathbf{N}_{\ell,b}$$

where we define $\bar{\mathbf{G}}_{\ell}$ to be the gradient of the true risk

Complexity-Accuracy Trade-off of SGD

What is the true risk? If you remember, when we started with training

Our goal was to minimize the risk $R(\mathbf{w})$

However, we could not do this: *since we did not know (1) the true function, and (2) the data distribution.* Thus,

we approximated the true risk $R(\mathbf{w})$ *with empirical risk* $\hat{R}(\mathbf{w})$

Complexity-Accuracy Trade-off of SGD

What is the true risk? If you remember, when we started with training

Our goal was to minimize the risk $R(\mathbf{w})$

However, we could not do this: *since we did not know (1) the true function, and (2) the data distribution*. Thus,

we approximated the true risk $R(\mathbf{w})$ *with empirical risk* $\hat{R}(\mathbf{w})$

We assume that $\bar{\mathbf{G}}_\ell$ is the gradient of true risk with respect to $\mathbf{W}_\ell$, i.e.,

$$\bar{\mathbf{G}}_\ell = \nabla_{\mathbf{W}_\ell} R(\mathbf{w})$$

+ *Can we determine this gradient?*
– Of course not! We can only approximate it with $\nabla_{\mathbf{W}_\ell} \hat{R}(\mathbf{w})$

Complexity-Accuracy Trade-off of SGD

Let's see accurate the gradient is approximated, when we do full-batch training

In full-batch training, we determine the gradient as

$$\hat{\mathbf{G}}_\ell^{\text{batch}} = \texttt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \ldots, \mathbf{G}_{\ell,B}) = \frac{1}{B} \sum_{b=1}^{B} \bar{\mathbf{G}}_\ell + \mathbf{N}_{\ell,b}$$

Complexity-Accuracy Trade-off of SGD

Let's see accurate the gradient is approximated, when we do full-batch training

In full-batch training, we determine the gradient as

$$\hat{\mathbf{G}}_\ell^{\text{batch}} = \texttt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \ldots, \mathbf{G}_{\ell,B}) = \frac{1}{B}\sum_{b=1}^{B} \bar{\mathbf{G}}_\ell + \mathbf{N}_{\ell,b}$$

$$= \bar{\mathbf{G}}_\ell + \underbrace{\frac{1}{B}\sum_{b=1}^{B} \mathbf{N}_{\ell,b}}_{\hat{\mathbf{N}}_\ell^{\text{batch}}}$$

Complexity-Accuracy Trade-off of SGD

Let's see accurate the gradient is approximated, when we do full-batch training

In full-batch training, we determine the gradient as

$$\hat{\mathbf{G}}_\ell^{\text{batch}} = \texttt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \ldots, \mathbf{G}_{\ell,B}) = \frac{1}{B}\sum_{b=1}^{B} \bar{\mathbf{G}}_\ell + \mathbf{N}_{\ell,b}$$

$$= \bar{\mathbf{G}}_\ell + \underbrace{\frac{1}{B}\sum_{b=1}^{B} \mathbf{N}_{\ell,b}}_{\hat{\mathbf{N}}_\ell^{\text{batch}}} = \bar{\mathbf{G}}_\ell + \hat{\mathbf{N}}_\ell^{\text{batch}}$$

Complexity-Accuracy Trade-off of SGD

Let's see accurate the gradient is approximated, when we do full-batch training

In full-batch training, we determine the gradient as

$$\hat{\mathbf{G}}_\ell^{\text{batch}} = \texttt{mean}(\mathbf{G}_{\ell,1}, \mathbf{G}_{\ell,2}, \dots, \mathbf{G}_{\ell,B}) = \frac{1}{B}\sum_{b=1}^{B} \bar{\mathbf{G}}_\ell + \mathbf{N}_{\ell,b}$$

$$= \bar{\mathbf{G}}_\ell + \underbrace{\frac{1}{B}\sum_{b=1}^{B} \mathbf{N}_{\ell,b}}_{\hat{\mathbf{N}}_\ell^{\text{batch}}} = \bar{\mathbf{G}}_\ell + \hat{\mathbf{N}}_\ell^{\text{batch}}$$

Recall that by *arithmetic averaging variance drops by $1/\#$ of samples*

In full-batch training the approximated gradient is different from the true gradient by an error whose variance drops as σ^2/B

Complexity-Accuracy Trade-off of SGD

In full-batch training the approximated gradient is different from the true gradient by an error whose variance drops as σ^2/B

Now, let's compare it to SGD

In SGD, we approximate the gradient with a sample gradient, i.e.,

$$\hat{\mathbf{G}}_\ell^{\mathrm{SGD}} = \mathbf{G}_{\ell,b} = \bar{\mathbf{G}}_\ell + \mathbf{N}_{\ell,b}$$

Complexity-Accuracy Trade-off of SGD

In full-batch training the approximated gradient is different from the true gradient by an error whose variance drops as σ^2/B

Now, let's compare it to SGD

In SGD, we approximate the gradient with a sample gradient, i.e.,

$$\hat{\mathbf{G}}_\ell^{\mathrm{SGD}} = \mathbf{G}_{\ell,b} = \bar{\mathbf{G}}_\ell + \mathbf{N}_{\ell,b}$$

As we see, SGD is still approximating the true gradient but with much larger variance: *entries of $\mathbf{N}_{\ell,b}$ have all variance σ^2*

For instance, consider MNIST with $60,000$ samples: *by full-batch training we get gradient values whose difference from the entries of the true gradient is approximately 1.67×10^{-5} times smaller that those gradient entries calculated by SGD!*

Complexity-Accuracy Trade-off of SGD

In the context of ML, we often say: *in the analyses of last slides,*

SGD and full-batch training are both unbiased estimators of $\bar{\mathbf{G}}_\ell$

We call them unbiased, since $\mathbb{E}\left\{\hat{\mathbf{G}}_\ell^{\mathrm{SGD}}\right\} = \mathbb{E}\left\{\hat{\mathbf{G}}_\ell^{\mathrm{batch}}\right\} = \bar{\mathbf{G}}_\ell$

Complexity-Accuracy Trade-off of SGD

In the context of ML, we often say: *in the analyses of last slides,*

SGD and full-batch training are both unbiased estimators of $\bar{\mathbf{G}}_\ell$

We call them unbiased, since $\mathbb{E}\left\{\hat{\mathbf{G}}_\ell^{\mathrm{SGD}}\right\} = \mathbb{E}\left\{\hat{\mathbf{G}}_\ell^{\mathrm{batch}}\right\} = \bar{\mathbf{G}}_\ell$

Complexity-Accuracy Trade-off

Assume that a forward and backward pass takes time T and that gradient of the risk at each sample be an unbiased estimators of the true gradient; then,

1. *each step of SGD takes time T while each step of full-batch training takes BT with B being the batch size*
2. *if we denote the variance of estimation given by SGD by σ^2, the variance of full-batch estimator is σ^2/B*

Training via Mini-Batches

+ *But, can't we play with this trade-off? For instance, increase a bit the complexity to improve the accuracy!*
- Yes! This is the idea of mini-batch training

Training via Mini-Batches

+ *But, can't we play with this trade-off? For instance, increase a bit the complexity to improve the accuracy!*
– Yes! This is the idea of mini-batch training

In mini-batch training, we divide the whole batch of data into mini-batches:

- *after each mini-batch is over, we average the gradients over the mini-batch*
- *we apply one step of gradient descent using this averaged gradient*

To avoid cyclic behavior, we still *shuffle the dataset randomly each time we start a new loop over it*. This training approach is hence often called

Mini-Batch Stochastic Gradient Descent = *Mini-Batch SGD*

Mini-Batch SGD

```
mBatchSGD():
 1: Initiate with some initial values $\{\mathbf{W}_\ell^{(0)}\}$ and set a learning rate $\eta$
 2: Randomly shuffle the batch and divide it into mini-batches of size $\Omega$
 3: Denote the number of mini-batches by $\Xi = \lceil B/\Omega \rceil$ and start at $\xi = 1$
 4: while weights not converged do
 5:     if $\xi > \Xi$ then
 6:         Randomly shuffle the batch and divide it into mini-batches of size $\Omega$
 7:         Set $\xi \leftarrow 1$                          # start over with the dataset
 8:     end if
 9:     for $\omega = 1, \ldots, \Omega$ do
10:         NN.values $\leftarrow$ ForwardProp $(\boldsymbol{x}_\omega, \{\mathbf{W}_\ell^{(t)}\})$
11:         $\{\mathbf{G}_{\ell,\omega}\} \leftarrow$ BackProp $(\boldsymbol{x}_\omega, v_\omega, \{\mathbf{W}_\ell^{(t)}\},$ NN.values$)$
12:     end for
13:     Update $\mathbf{W}_\ell^{(t+1)} \leftarrow \mathbf{W}_\ell^{(t)} - \eta$ mean $(\mathbf{G}_{\ell,1}, \ldots, \mathbf{G}_{\ell,\Omega})$
14:     Update $\xi \leftarrow \xi + 1$                       # go for next mini-batch
15: end while
```

Complexity-Accuracy Trade-off

It is easy to see that

- *mini-batch training reduces to full-batch training when we set the size of mini-batches* to B, i.e., $\Omega = B$
- *mini-batch training reduces to SGD when we set the size of mini-batches to* 1, *i.e.*, $\Omega = 1$

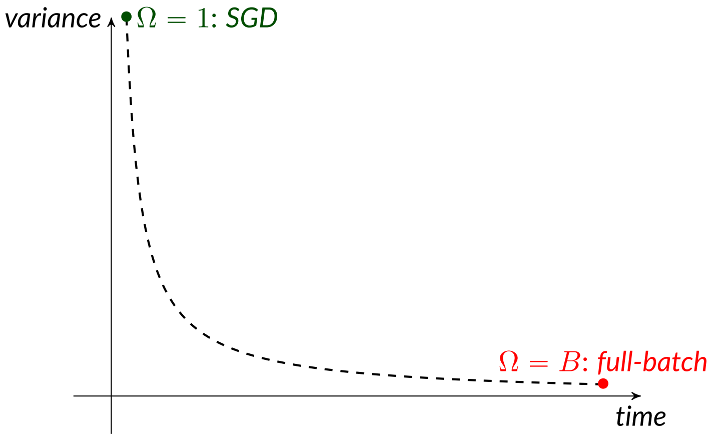
Complexity-Accuracy Trade-off

Assume that a forward and backward pass takes time T and that gradient of the risk at each sample be an unbiased estimators of the true gradient; then,

1. *each step of SGD takes time T while each step of mini-batch* training takes ΩT with Ω being the mini-batch size
2. *if we denote the variance of estimation given by SGD* by σ^2, the variance *of mini-batch* estimator is σ^2/Ω

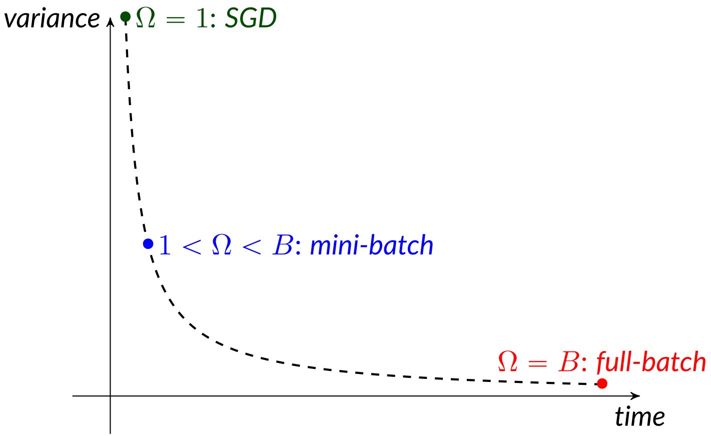
Complexity-Accuracy Trade-off

The complete trade-off can be visualized as



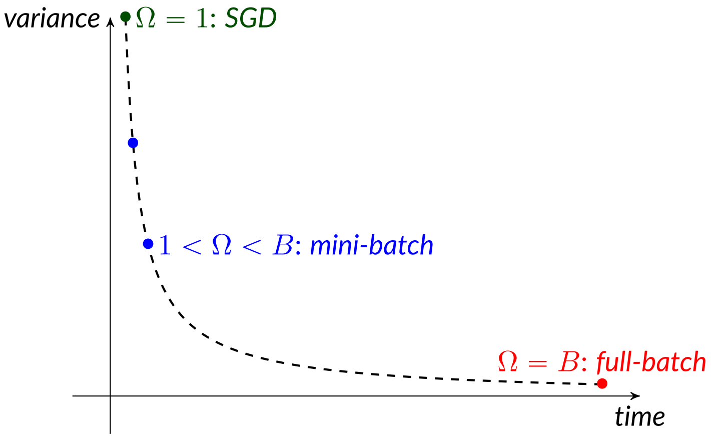
Complexity-Accuracy Trade-off

The complete trade-off can be visualized as



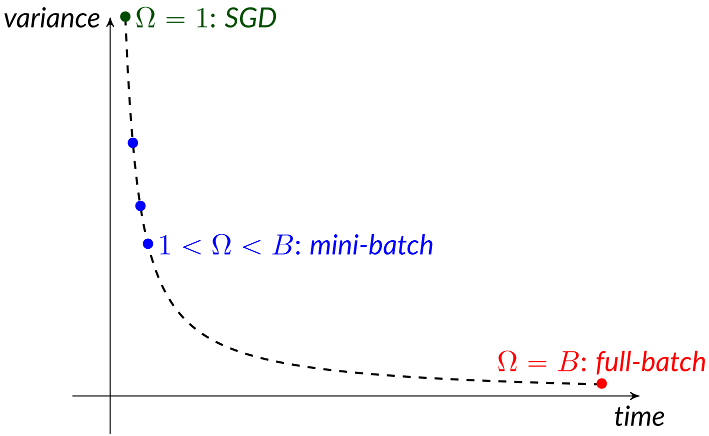
Complexity-Accuracy Trade-off

The complete trade-off can be visualized as



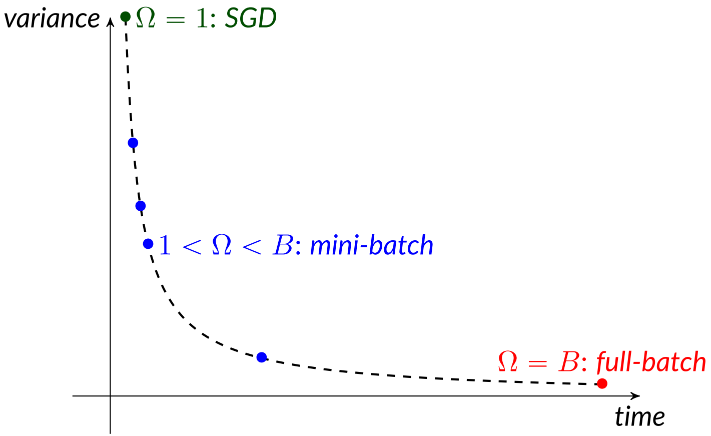
Complexity-Accuracy Trade-off

The complete trade-off can be visualized as



Complexity-Accuracy Trade-off

The complete trade-off can be visualized as



$\boxed{\textit{Mini-batch size is what specifies the trade-off point}}$

Few Definitions: *Epoch and Iteration*

In the language of deep learning there are few terms that we must know

Batch Size

Through time, the term mini-batch has been transformed to batch, and the complete batch is referred to as training dataset or the full batch. People hence call the size of each mini-batch, i.e., Ω, the batch size

Iteration

When we take one step of gradient descent, we take one iteration. So, one iteration is over when we finish with a mini-batch

Epoch

An epoch is over when we finish once with the whole training dataset

Few Definitions: *Epoch and Iteration*

We can annotate these definitions in our algorithm

```
mBatchSGD():
 1: Initiate with some initial values $\{\mathbf{W}_\ell^{(0)}\}$ and set a learning rate $\eta$
 2: Randomly shuffle training dataset and make mini-batches of size $\Omega \equiv$ batch-size
 3: Denote the number of mini-batches by $\Xi = \lceil B/\Omega \rceil$ and start at $\xi = 1$
 4: while weights not converged do
 5:     if $\xi > \Xi$ then
 6:         Randomly shuffle the batch and divide it into mini-batches of size $\Omega$
 7:         Set $\xi \leftarrow 1$          $\leftarrow$ one epoch is over, we start another epoch
 8:     end if
 9:     for $\omega = 1, \ldots, \Omega$ do
10:         NN.values $\leftarrow$ ForwardProp $(\boldsymbol{x}_\omega, \{\mathbf{W}_\ell^{(t)}\})$ going through a min-batch
11:         $\{\mathbf{G}_{\ell,\omega}\} \leftarrow$ BackProp $(\boldsymbol{x}_\omega, v_\omega, \{\mathbf{W}_\ell^{(t)}\}$, NN.values$)$
12:     end for
13:     Update $\mathbf{W}_\ell^{(t+1)} \leftarrow \mathbf{W}_\ell^{(t)} - \eta$ mean $(\mathbf{G}_{\ell,1}, \ldots, \mathbf{G}_{\ell,\Omega})$ $\leftarrow$ one iteration
14:     Update $\xi \leftarrow \xi + 1$                    # go for next mini-batch
15: end while
```

Few Definitions: *Epoch and Iteration*

We can consider a simple example: *say we train our FNN over MNIST using mini-batch SGD with batch size $\Omega = 100$. Our training dataset has $60,000$ data-points; thus, we have*

$$\Xi = \frac{60,000}{100} = 600$$

mini-batches. Each time we finish with a mini-bacth, we do one iteration of gradient descent. After 600 iterations, we finish with a single epoch

So, if we have trained the FNN for 10 epochs, it means that

we have done $600 \times 10 = 6000$ iterations of gradient descent

Testing NNs with New Data-Point

+ *Say we are over with the training; then, what should we do?*
– We need to test it with the data we reserved for testing

After training, we need to test our trained NN: *say we get a new data-point $\boldsymbol{x}_{\text{new}}$ with label $\boldsymbol{v}_{\text{new}}$. We can test our NN for this new test data-point by evaluating classical metrics*

1. **Test Risk** also called **Test Loss**: *we pass $\boldsymbol{x}_{\text{new}}$ forward through our trained NN and get $\boldsymbol{y}_{\text{new}}$. We then calculate the test loss as $\mathcal{L}\left(\boldsymbol{y}_{\text{new}}, \boldsymbol{v}_{\text{new}}\right)$ using the same loss function $\mathcal{L}$ we used for training*
2. **Test Accuracy**: *we use $\boldsymbol{y}_{\text{new}}$ to classify $\boldsymbol{x}_{\text{new}}$. We then compare it to the true class of $\boldsymbol{x}_{\text{new}}$. If they are the same; then, the test accuracy is 1, if not, it is 0*

Testing NNs over Test Dataset

Testing for a single new point is not reliable: *this is why we had reserved the test dataset*.

Given the test dataset, we go through every single test data-point

- *we pass the data-point forward through the trained NN*
- *we compute the test loss and test accuracy*
- *we average them over the whole test dataset*

Therefore, we get

- *an average loss that approximates the risk*
- *a test accuracy between 0 and 1 that says how accurate our trained NN is*

Learning Curves

+ *What you said gives us two numbers! But, I have seen curves!*
– Yes! They are learning curves

In practice, the SGD can take very long to converge, i.e., to stop iterating

it needs too many iterations to get too close to the minimum

But, it might be not really needed to get that close! So,

we test our NN once every epoch

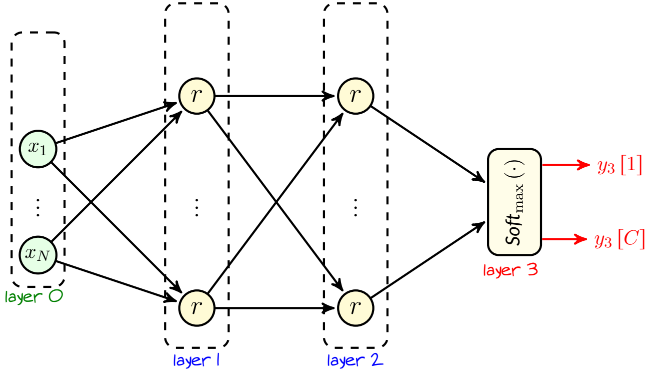
We then plot the test risk and test accuracy against number of epochs in a curve: *these curves are often called learning curves*

if we see that learning curves are saturating, we can stop the training

In practice: $\boxed{\text{we always perform the training for a fixed number of epochs}}$

Learning Curves: *Example*

Let's see an example: *recall our three-layer FNN. Say, we train it for image classification over MNIST which has 60,000 data-points for and 10,000 for test*



In MNIST, we have 10 classes, so $C = 10$. We use cross-entropy as loss function

Learning Curves: *Example*

We agree to do the following: *we use mini-batch SGD with batch size* $\Omega = 100$ *and train the FNN for 100 epochs*.

In epoch $\xi = 1, \ldots, 100$

Learning Curves: *Example*

We agree to do the following: *we use mini-batch SGD with batch size $\Omega = 100$ and train the FNN for 100 epochs*.

In epoch $\xi = 1, \dots, 100$

1. *we perform 600 iterations of gradient descent*

Learning Curves: *Example*

We agree to do the following: *we use mini-batch SGD with batch size* $\Omega = 100$ *and train the FNN for 100 epochs*.

In epoch $\xi = 1, \ldots, 100$

1. *we perform* 600 *iterations of gradient descent*
2. *we fix the weights to what we computed at last iteration of the epoch*

Learning Curves: *Example*

We agree to do the following: *we use mini-batch SGD with batch size $\Omega = 100$ and train the FNN for 100 epochs*.

In epoch $\xi = 1, \dots, 100$

1. *we perform 600 iterations of gradient descent*
2. *we fix the weights to what we computed at last iteration of the epoch*
3. *for each test data-point:* we pass it forward and determine $\mathbf{y}_3$
    1. we compute $\mathrm{CE}\left(\mathbf{y}_3, \mathbf{1}_v\right)$, where v is the true class of test data-point
    2. we find the index of maximum term in $\mathbf{y}_3$ and compare it to v

        ↳ *if they are the same, we set accuracy to 1; otherwise, we set it 0*

Learning Curves: *Example*

We agree to do the following: *we use mini-batch SGD with batch size* $\Omega = 100$ *and train the FNN for 100 epochs*.

In epoch $\xi = 1, \ldots, 100$

1. *we perform* 600 *iterations of gradient descent*
2. *we fix the weights to what we computed at last iteration of the epoch*
3. *for each test data-point:* we pass it forward and determine $\mathbf{y}_3$
   1. we compute $\mathrm{CE}\left(\mathbf{y}_3, \mathbf{1}_v\right)$, where v is the true class of test data-point
   2. we find the index of maximum term in $\mathbf{y}_3$ and compare it to v

      ↳ *if they are the same, we set accuracy to* 1; *otherwise, we set it* 0
4. *we average test loss and accuracy*

Learning Curves: *Example*

We agree to do the following: *we use mini-batch SGD with batch size* $\Omega = 100$ *and train the FNN for 100 epochs*.

In epoch $\xi = 1, \ldots, 100$

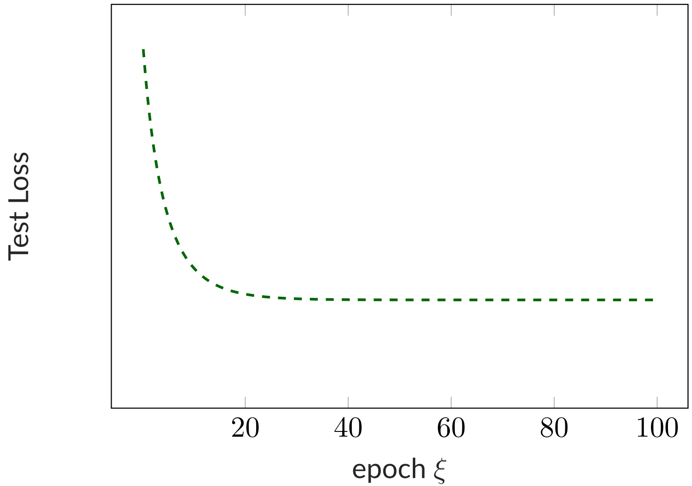
1. *we perform* 600 *iterations of gradient descent*
2. *we fix the weights to what we computed at last iteration of the epoch*
3. *for each test data-point*: we pass it forward and determine $\mathbf{y}_3$
   1. we compute $\mathrm{CE}\left(\mathbf{y}_3, \mathbf{1}_v\right)$, where v is the true class of test data-point
   2. we find the index of maximum term in $\mathbf{y}_3$ and compare it to v
      - ↳ *if they are the same, we set accuracy to* 1; *otherwise, we set it* 0
4. *we average test loss and accuracy*

Now, for each epoch

we have a test loss and test accuracy: *we plot them against* ξ

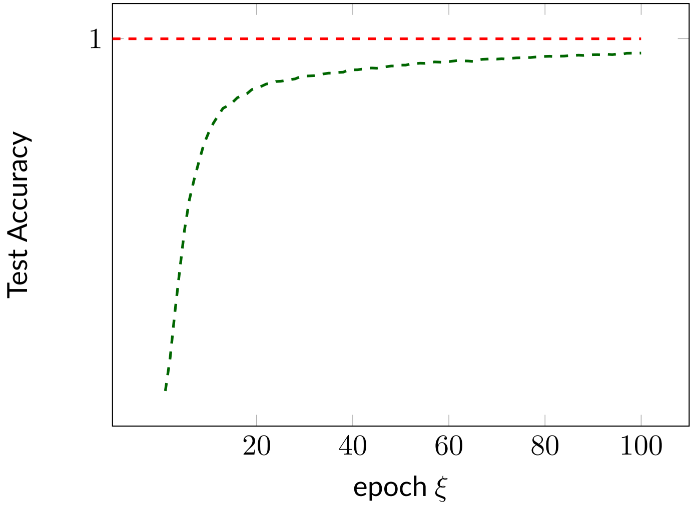
Learning Curves: *Example*

How should the learning curves look? A typical curve for test loss is



Learning Curves: *Example*

How should the learning curves look? A typical curve for test accuracy is



Summary of This Chapter

- To train a NN we need *gradients*
  - ↳ *We can calculate gradient by forward and backpropagation over the NN*
  - ↳ *In FNNs, forward propagation uses simple linear and nonlinear operations*
  - ↳ *Backpropagation is readily derived using computation graph*

Summary of This Chapter

- To train a NN we need *gradients*
  - ↳ *We can calculate gradient by forward and backpropagation over the NN*
  - ↳ *In FNNs, forward propagation uses simple linear and nonlinear operations*
  - ↳ *Backpropagation is readily derived using computation graph*
- We tried Classification via FNNs
  - ↳ *Better to work with probabilities instead of exact labels*
  - ↳ *For multiclass classification, we should use vector-activated neurons*

Summary of This Chapter

- To train a NN we need *gradients*
  - ↳ *We can calculate gradient by forward and backpropagation over the NN*
  - ↳ *In FNNs, forward propagation uses simple linear and nonlinear operations*
  - ↳ *Backpropagation is readily derived using computation graph*
- We tried Classification via FNNs
  - ↳ *Better to work with probabilities instead of exact labels*
  - ↳ *For multiclass classification, we should use vector-activated neurons*
- To minimize the exact empirical risk, we have to do full-batch training
  - ↳ *This requires huge computation complexity*
  - ↳ *We can hugely reduce this cost by SGD which does sample-level training*
  - ↳ *SGD versus full-batch describes a complexity-accuracy trade-off*
  - ↳ *We can tune this trade-off by mini-batch SGD*