ECE 1508: Applied Deep Learning

Chapter 2: Feedforward Neural Networks

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Fall 2025

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?

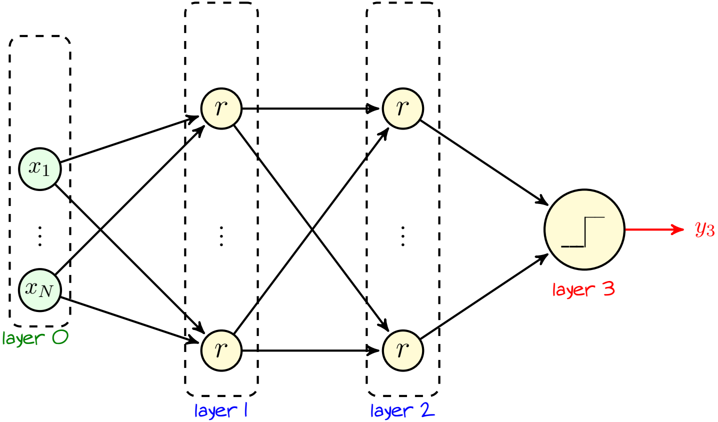
- We want it to be deep; so, we consider *2 hidden layers*
  - ↳ *the depth is hence* 3
- We specify the width of each hidden layer
  - ↳ *first hidden layer has width* K
  - ↳ *second hidden layer has width* J
- All hidden neurons use ReLU activation
  - ↳ $f_1(\cdot) = f_2(\cdot) = \mathrm{ReLU}(\cdot)$: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



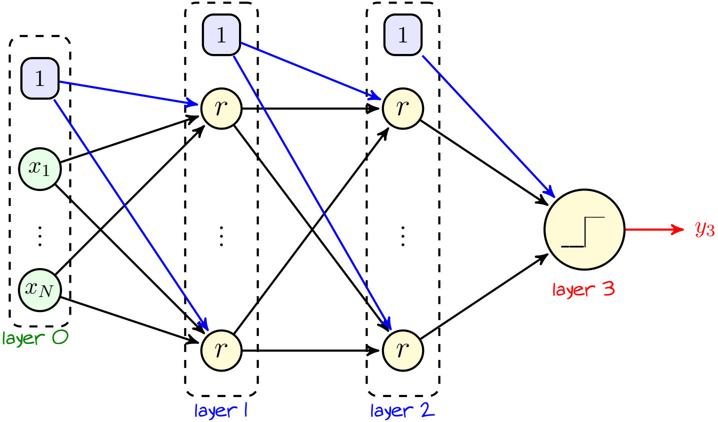
Binary Classification via FNN: *Model*

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*

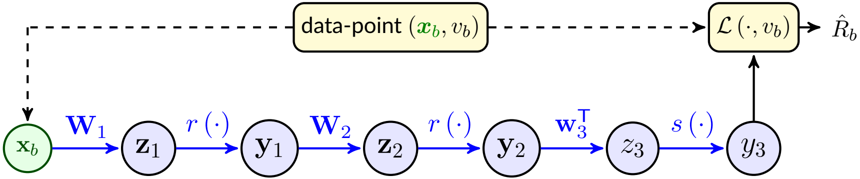
We use the error indicator as the loss function

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

+ *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*

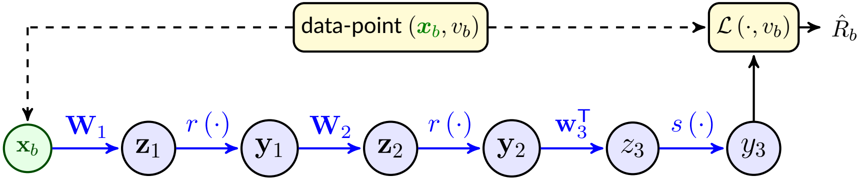


Here, we have *3 linear operations*

- First operation is $\mathbf{z}_1 = \mathbf{W}_1 \mathbf{x}_b$ 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 η

2: **while** weights not converged **do**

3:     **for** $b = 1, \dots, B$ **do**

4:         $\texttt{NN.values} \leftarrow \texttt{ForwardProp}\left(\boldsymbol{x}_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{w}_3^{(t)}\}\right)$

5:         $\{\mathbf{G}_{1,b}, \mathbf{G}_{2,b}, \mathbf{g}_{3,b}\} \leftarrow \texttt{BackProp}\left(\boldsymbol{x}_b, v_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{w}_3^{(t)}\}, \texttt{NN.values}\right)$

6:     **end for**

7:     Update

$$\mathbf{W}_1^{(t+1)} \leftarrow \mathbf{W}_1^{(t)} - \eta \ \texttt{mean}\left(\mathbf{G}_{1,1}, \dots, \mathbf{G}_{1,B}\right)$$

$$\mathbf{W}_2^{(t+1)} \leftarrow \mathbf{W}_2^{(t)} - \eta \ \texttt{mean}\left(\mathbf{G}_{2,1}, \dots, \mathbf{G}_{2,B}\right)$$

$$\mathbf{w}_3^{(t+1)} \leftarrow \mathbf{w}_3^{(t)} - \eta \ \texttt{mean}\left(\mathbf{g}_{3,1}, \dots, \mathbf{g}_{3,B}\right)$$

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*

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

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*

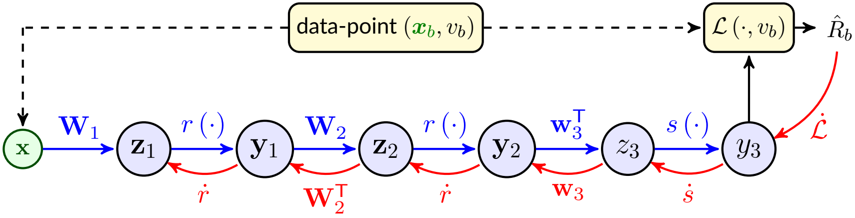
$\boxed{\textit{it's } 0 \textit{ if } z_3 \textit{ is negative, and } 1 \textit{ if it is not negative}}$

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.,

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

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 $\overleftarrow{\mathbf{y}}_1 = \mathbf{W}_2^\mathsf{T} \overleftarrow{\mathbf{z}}_2$ 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!

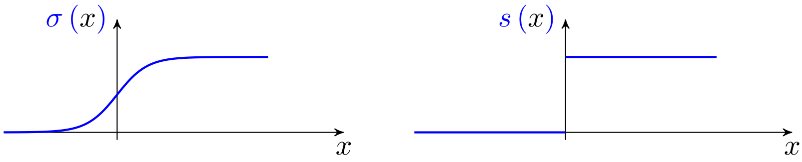
Binary Classification via FNN: *Differentiability Issue*

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*

+ *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!

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.,

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

in this case the empirical risk is called

Mean Squared Error (MSE)

This loss is differentiable

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

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

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

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

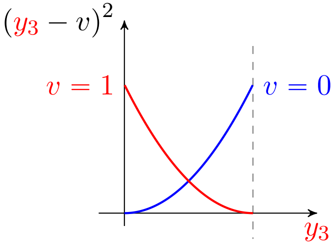
This loss is again differentiable

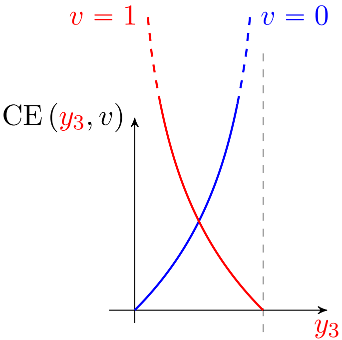
$$\dot{\mathcal{L}}(y_3, v) = \dot{\mathrm{CE}}(y_3, v) = -\frac{v}{y_3} + \frac{1-v}{1-y_3}$$

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 $[0, 1]$

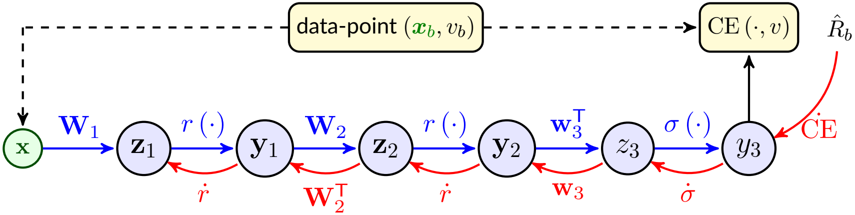




Cross entropy returns much higher loss when y_3 is different from v

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*

+ *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!

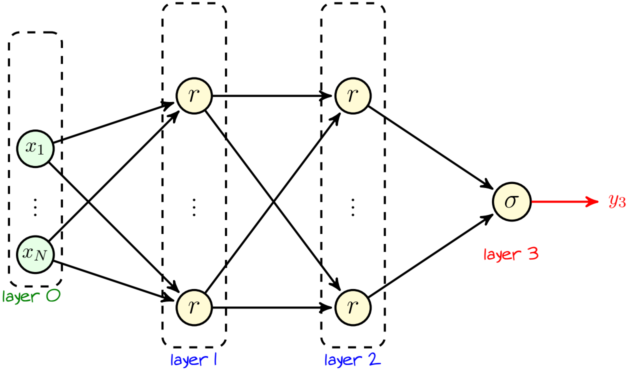
+ *Why not follow the same idea and determine the probability of input belonging to each class?*
- Yes! That's actually the effective way!

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

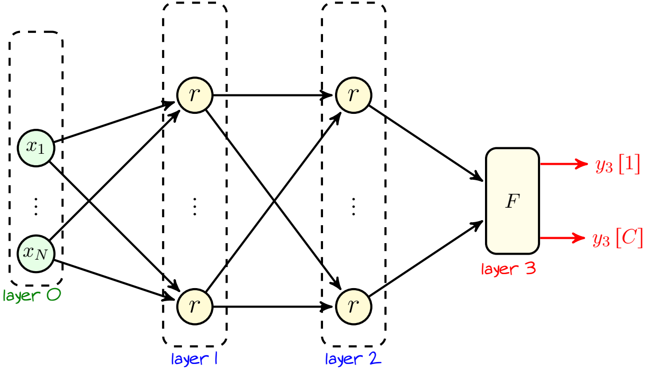


In this FNN, y_3 *is interpreted as a probability of label being* 1

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

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

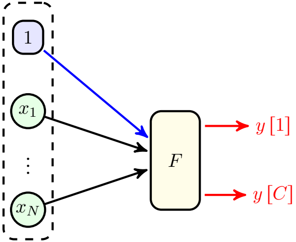
for weight matrix $\tilde{\mathbf{W}} \in \mathbb{R}^{C \times N}$, bias $\mathbf{b} \in \mathbb{R}^C$ and activation $F(\cdot) : \mathbb{R}^C \mapsto \mathbb{R}^C$

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

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

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*

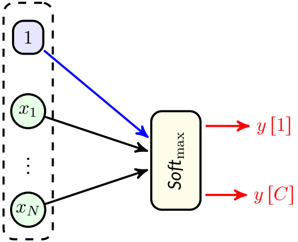
$$\mathbf{x} \xrightarrow{\mathbf{W}} \mathbf{z} \xrightarrow{F(\cdot)} \mathbf{y}$$

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* $\mathsf{Soft}_{\max}(\mathbf{z}) = \mathbf{y} \in \mathbb{R}^C$ *whose entry* c *is*

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

Multiclass Classification: *Softmax*

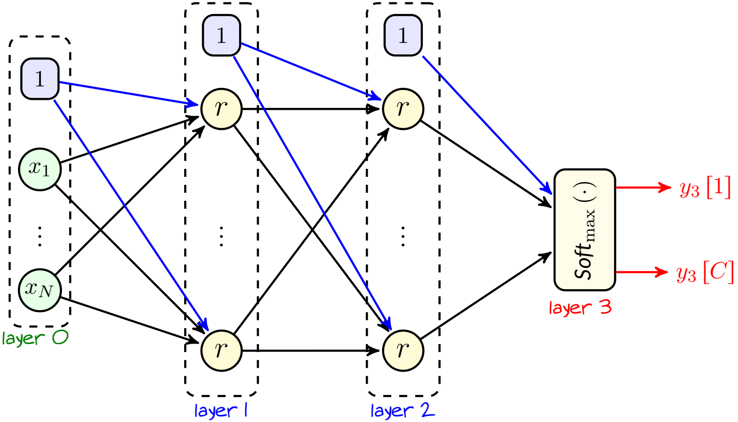
Softmax always returns a probability distribution on the set of classes

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

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

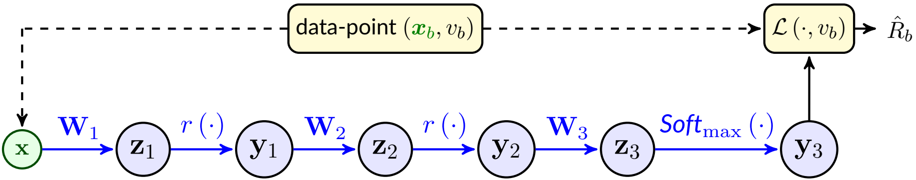
Multiclass Classification via FNN: *Softmax Activation*



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* $\mathbf{z}_3 \in \mathbb{R}^C$
  - ↳ *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 $\mathbf{y}_1 = r(\mathbf{z}_1)$

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

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*

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*

+ *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

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

This gradient of this loss is

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

Multiclass Classification: *Loss*

Cross entropy can also be extended as follows

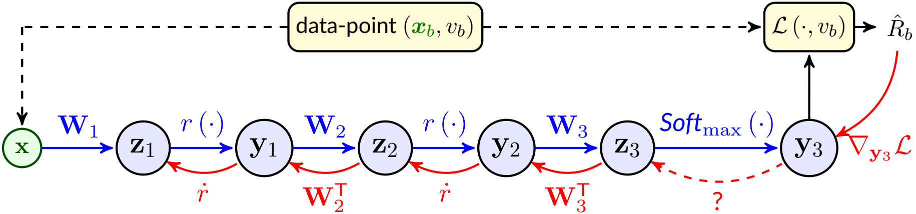
$$\mathcal{L}(\mathbf{y}, \mathbf{1}_v) = \mathrm{CE}(\mathbf{y}, \mathbf{1}_v) = -\sum_{c=1}^{C} \mathbb{1}\{c = v\} \log y(c)$$
$$= -\log y[v]$$

The gradient of this loss is

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

Backpropagation Through *Vector Activation*

How can we backpropagate through this neural network?

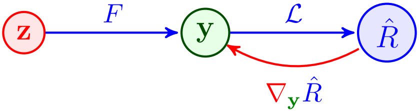


1. Compute $\nabla_{\mathbf{y}_3}\hat{R}_b = \nabla_{\mathbf{y}_3}\mathcal{L}(\mathbf{y}_3, \mathbf{1}_{v_b})$
2. Compute $\nabla_{\mathbf{z}_3}\hat{R}_b$ from $\nabla_{\mathbf{y}_3}\hat{R}_b$ *by its backward link that* *we don't know yet*
3. Compute $\nabla_{\mathbf{y}_2}\hat{R}_b = \mathbf{W}_3^\top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



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

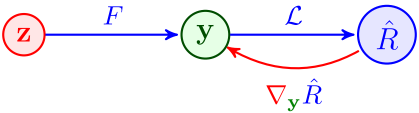
$$\mathbf{J_z y} = \mathbf{J_z} F = \begin{bmatrix} \nabla_{\mathbf{z}} y_1^{\mathsf{T}} \\ \vdots \\ \nabla_{\mathbf{z}} y_C^{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \partial y_1 / \partial z_1 & \dots & \partial y_1 / \partial z_C \\ \vdots & & \vdots \\ \partial y_C / \partial z_1 & \dots & \partial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

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 $\nabla_{\mathbf{y}} \hat{R}$*



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

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

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

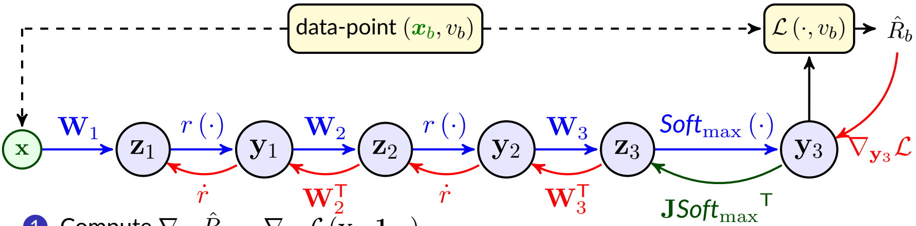
$$\begin{tikzcd}
\mathbf{z} \arrow[r, "F"] & \mathbf{y} \arrow[r, "\mathcal{L}"] \arrow[l, bend left, "\mathbf{J}_{\mathbf{z}}^{\mathsf{T}} \mathbf{y}"] & \hat{R} \arrow[l, bend left, "\nabla_{\mathbf{y}} \hat{R}"]
\end{tikzcd}$$

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



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 η
2: **while** weights not converged **do**
3:     **for** $b = 1, \ldots, B$ **do**
4:         $\texttt{NN.values} \leftarrow \texttt{ForwardProp}\left(\boldsymbol{x}_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\}\right)$
5:         $\{\mathbf{G}_{1,b}, \mathbf{G}_{2,b}, \mathbf{G}_{3,b}\} \leftarrow \texttt{BackProp}\left(\boldsymbol{x}_b, v_b, \{\mathbf{W}_1^{(t)}, \mathbf{W}_2^{(t)}, \mathbf{W}_3^{(t)}\}, \texttt{NN.values}\right)$
6:     **end for**
7:     Update

$$\mathbf{W}_1{}^{(t+1)} \leftarrow \mathbf{W}_1{}^{(t)} - \eta \ \texttt{mean}\left(\mathbf{G}_{1,1}, \ldots, \mathbf{G}_{1,B}\right)$$
$$\mathbf{W}_2{}^{(t+1)} \leftarrow \mathbf{W}_2{}^{(t)} - \eta \ \texttt{mean}\left(\mathbf{G}_{2,1}, \ldots, \mathbf{G}_{2,B}\right)$$
$$\mathbf{W}_3{}^{(t+1)} \leftarrow \mathbf{W}_3{}^{(t)} - \eta \ \texttt{mean}\left(\mathbf{G}_{3,1}, \ldots, \mathbf{G}_{3,B}\right)$$

8: **end while**

We call this form of gradient descent full-batch