ECE 1513: Introduction to Machine Learning

Lecture 8: Neural Networks II

Ali Bereyhi

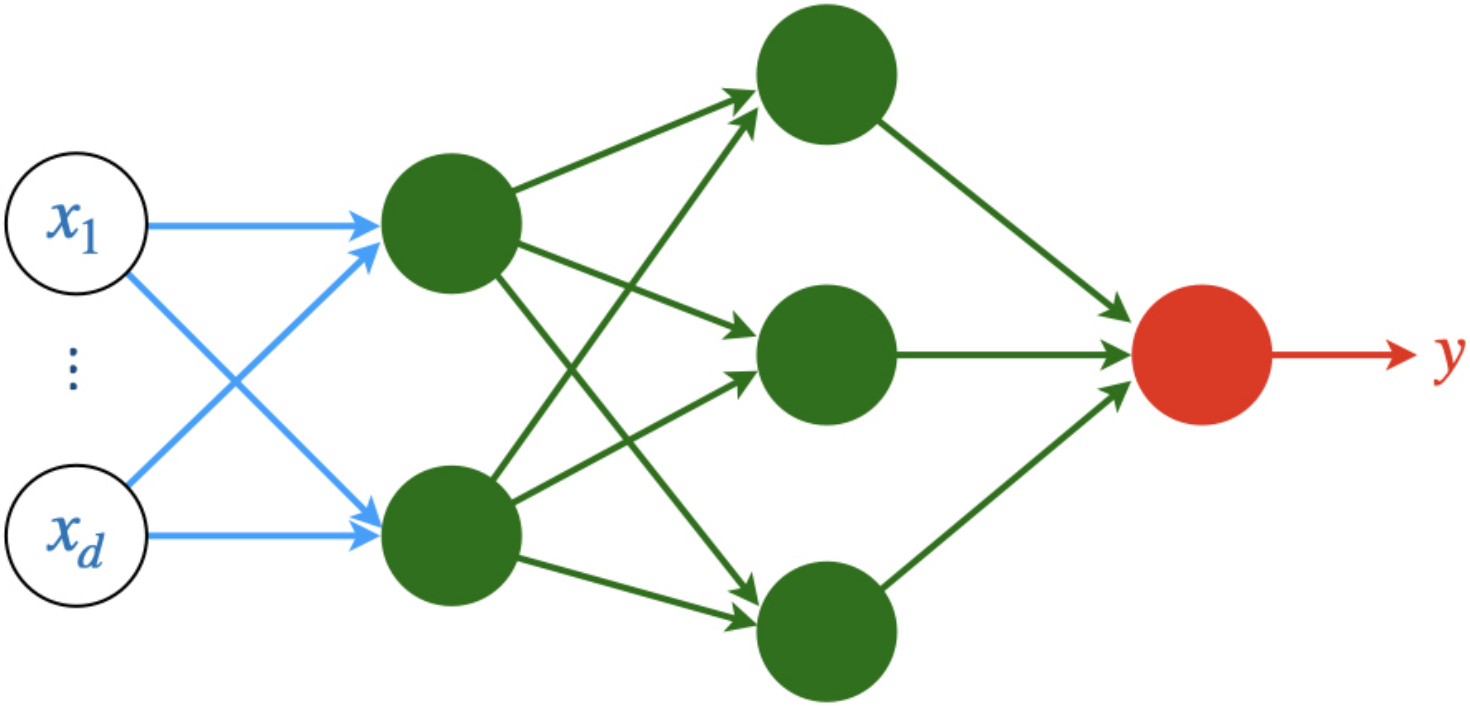
ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

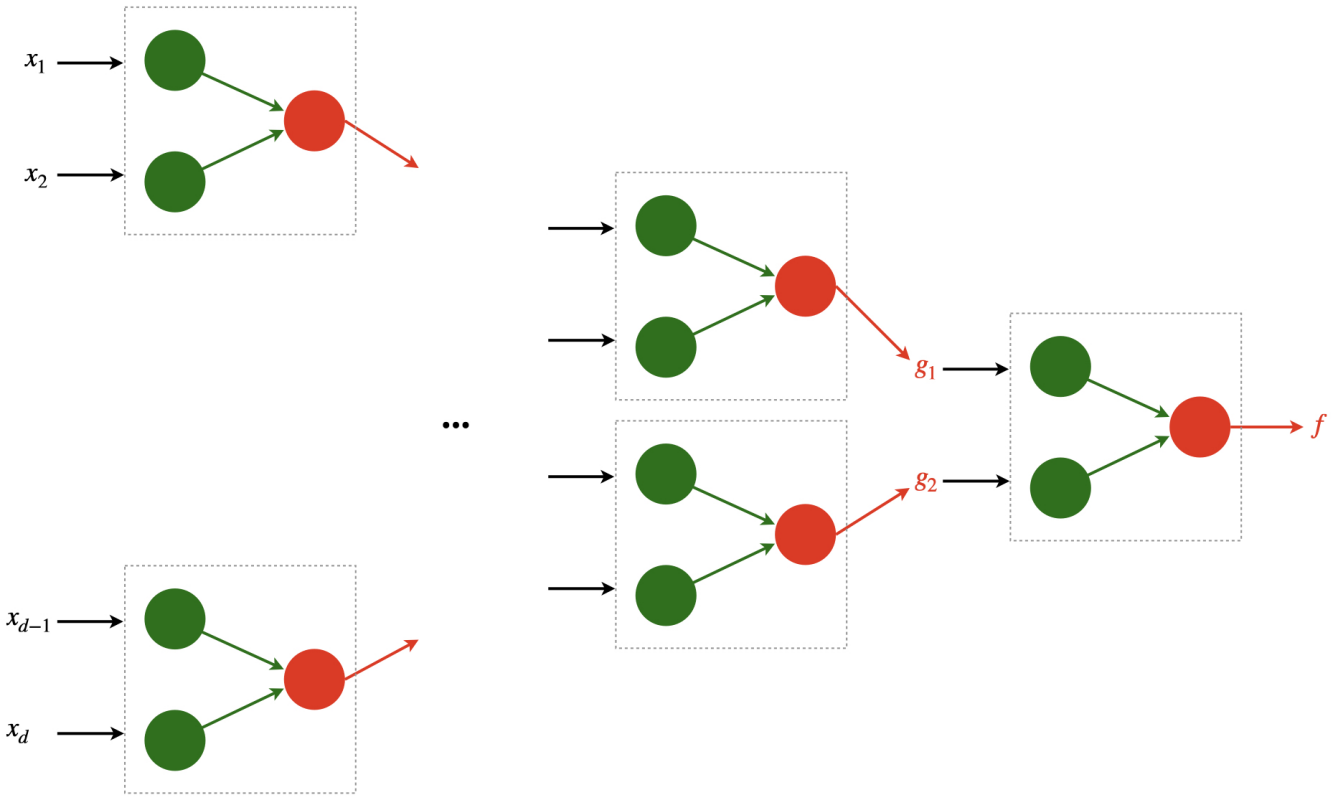
University of Toronto

Winter 2025

Quick Recap: *Neural Networks*



Quick Recap: *Expressive Power of Deep NNs*



As we go deep, our expressive power increases exponentially

Today's Agenda: *Training and Inference via NNs*

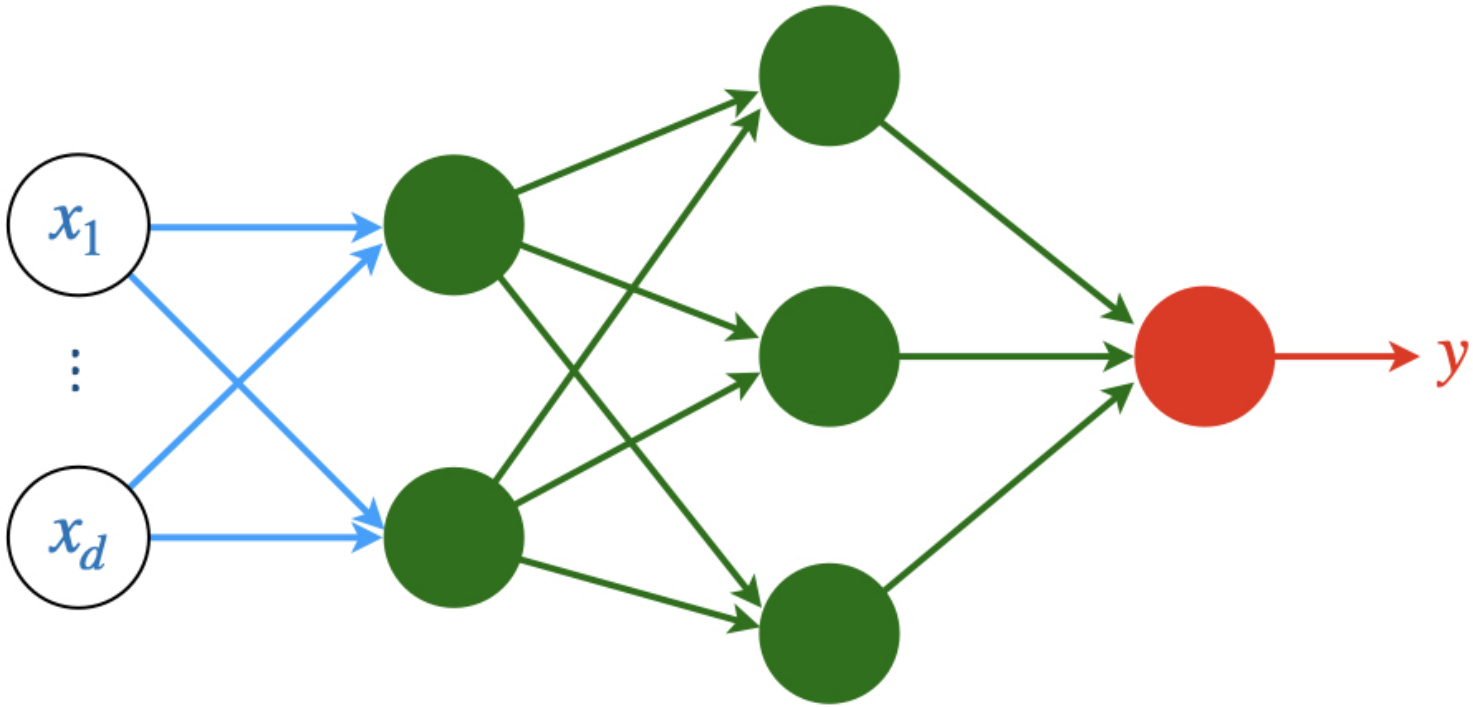
Today, we learn

How to Train Neural Networks

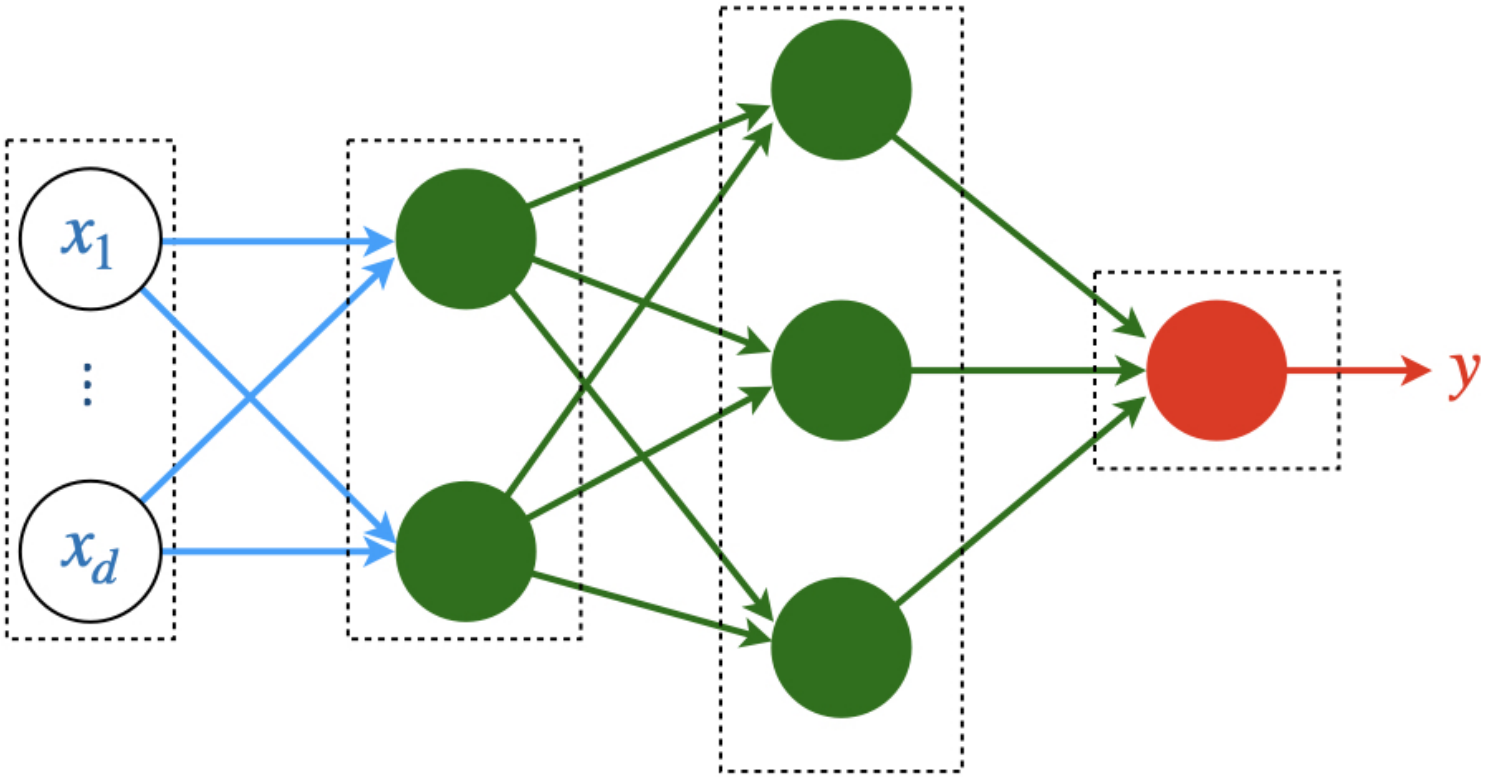
In this way, we discuss the following topics

- *Understand Inference by NNs*
- *Backpropagation*
- *Complete training and test loop*

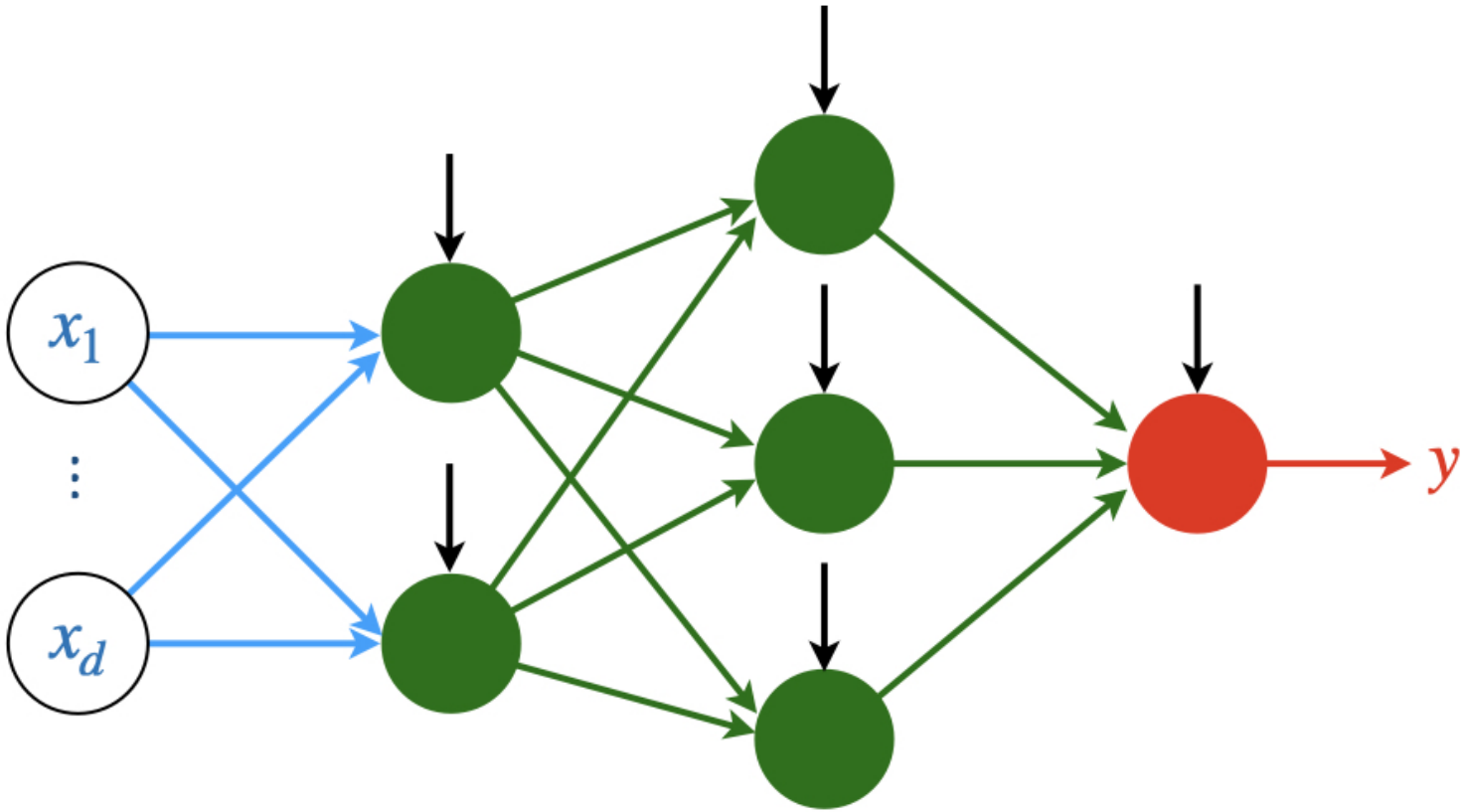
NN in a Box



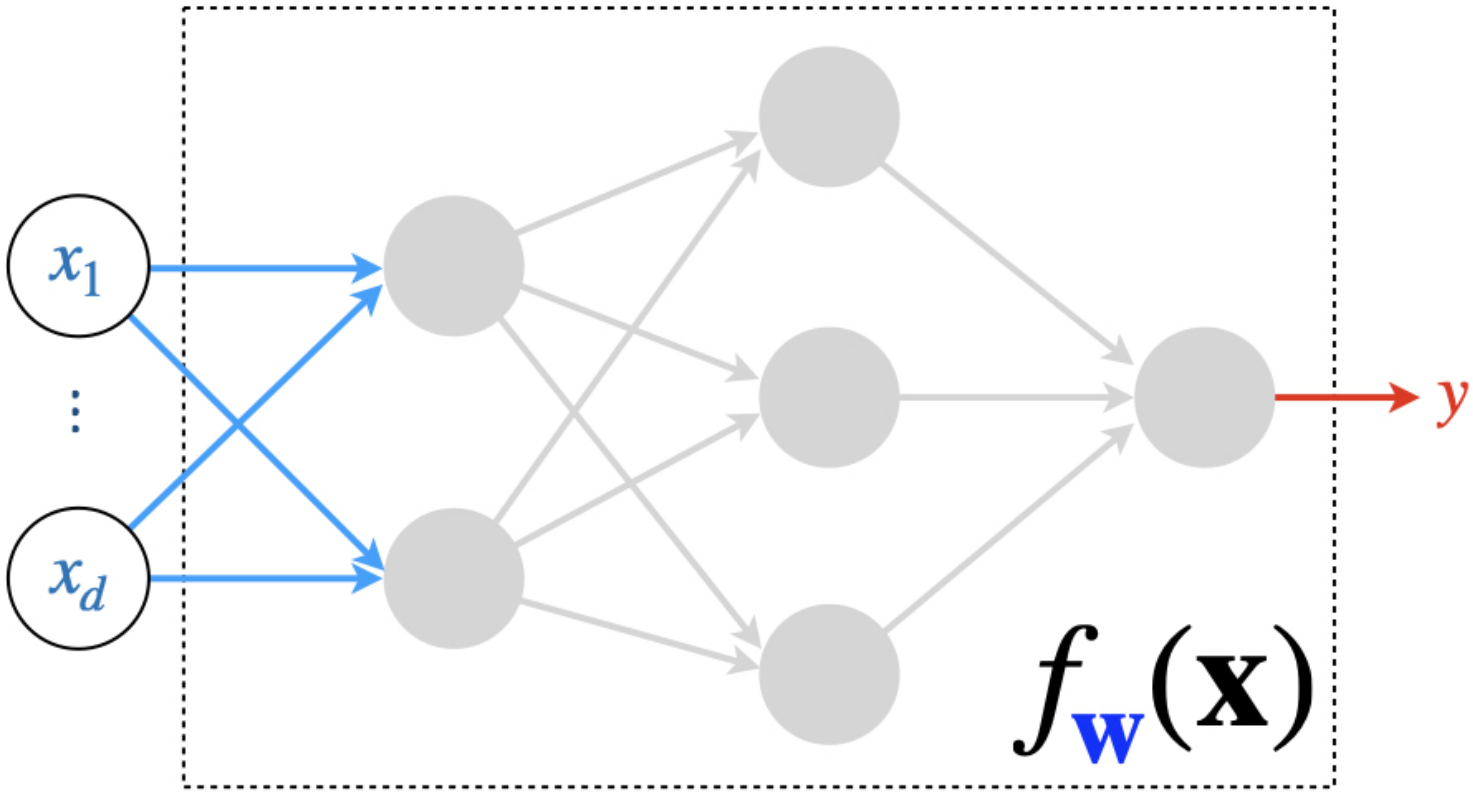
NN in a Box



NN in a Box



NN in a Box



Recall *Supervised Learning*

We have three components

- *A labeled dataset*

$$\mathbb{D} = \{(\boldsymbol{x}_n \in \mathbb{X}, \boldsymbol{v}_n \in \mathbb{V}) : n = 1, \ldots, N\}$$

- *Model that relates data samples and their labels*

$$f_{\mathbf{w}} : \mathbb{X} \mapsto \mathbb{V}$$

  ↳ $f_{\mathbf{w}} \in \mathbb{H}$ *is now a NN with* m *model parameters, i.e.,*

$$\mathbb{H} = \{f_{\mathbf{w}}(\boldsymbol{x}) \text{ for all } \mathbf{w} \in \mathbb{R}^m\}$$

  ↳ *Hypothesis is specified with the NN architecture*

- *Learning algorithm finds optimal weights for the NN*
  ↳ *This is what we call training a NN*

Using NNs for Classification

For classification, the NN needs to give us information about the class

- *Naive approach is to set*

$$f_{\mathbf{w}}(\boldsymbol{x}) \in \{1, \ldots, k\}$$

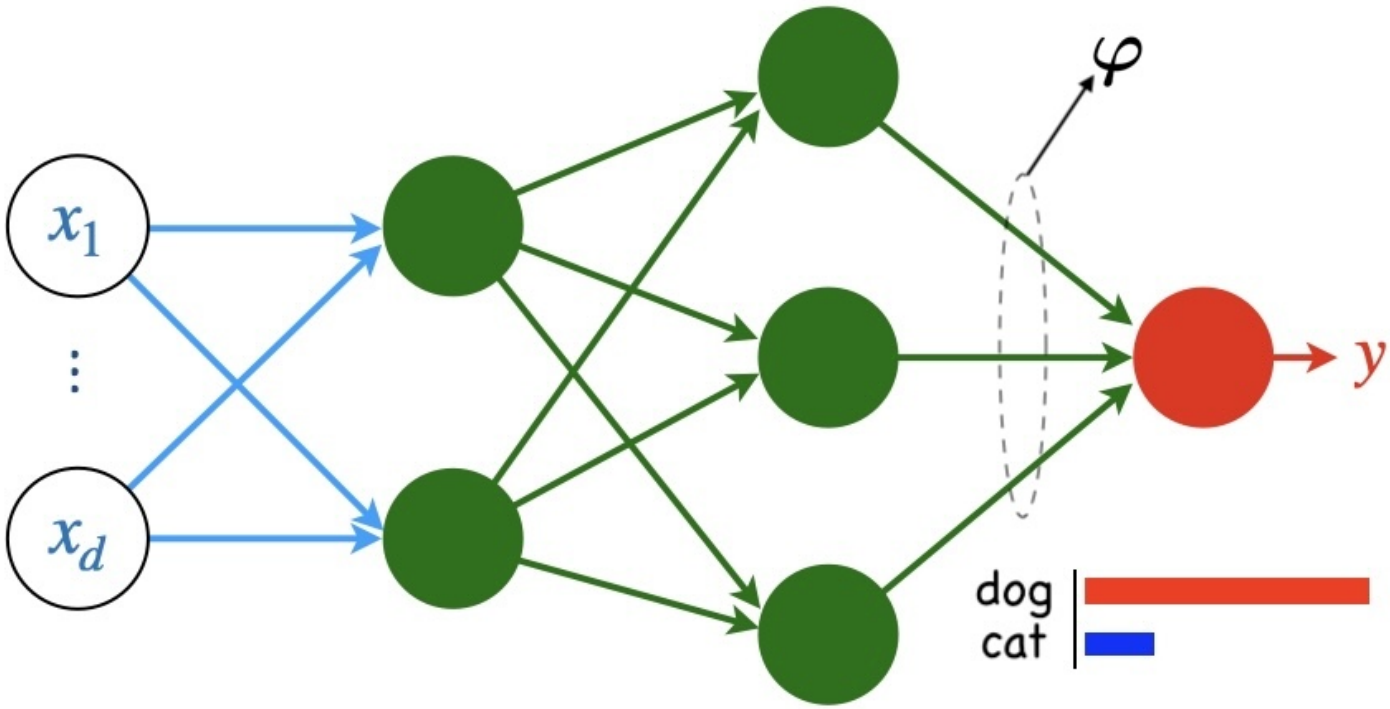
for k-class classification

↳ *This is not efficient for training as we will see*

- *In practice we set*

$$f_{\mathbf{w}}(\boldsymbol{x}) \propto \Pr\{\boldsymbol{x} \in \textit{ Class } i\}$$

Binary Classification via NNs



Binary Classification via NNs: *Sigmoid Activation*

We need a single neuron at the output layer

- *Compute single linear combination and activate it by Sigmoid*

$$\mathbf{w}\boldsymbol{\varphi} + b = z \rightsquigarrow y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

- *We treat output as probability of one of the classes*

$$f_{\mathbf{w}}(\boldsymbol{x}) = \Pr\{\boldsymbol{x} \in \textit{ Class } 1\}$$

  ↳ *No need to compute another probability then*

$$\Pr\{\boldsymbol{x} \in \textit{ Class } 2\} = 1 - f_{\mathbf{w}}(\boldsymbol{x})$$

- *For inference, we take the class with highest probability*

Multiclass Classification via NNs

- ? *What if we have more than two classes?*
- ! *We compute the complete distribution*

We need an output layer with k outputs

$$\boldsymbol{y} = f_{\mathbf{w}}(\boldsymbol{x}) = \begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} \in [0,1]^k$$

such that $y_i = \Pr\{\boldsymbol{x} \in \textit{ Class } i\}$

$$\sum_{i=1}^{k} y_i = 1$$

Multiclass Classification via NNs: *Softmax*

Classic choice is to

- *Compute k linear combinations in the output layer*

$$\mathbf{W}\boldsymbol{\varphi} + \mathbf{b} = \boldsymbol{z} \in \mathbb{R}^k$$

- *Compute a Softmax of $\boldsymbol{z}$*

$$\begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} = \mathit{Soft}_{\max}(\boldsymbol{z}) = \frac{1}{\sum_{j=1}^{k} e^{z_j}} \begin{bmatrix} e^{z_1} \\ \vdots \\ e^{z_k} \end{bmatrix}$$

Softmax

Softmax computes a distribution: larger z_i has higher probability

Using NNs for Regression

For regression, the NN needs to give us real values

- *We may have a single real output*

$$y = f_{\mathbf{w}}(\boldsymbol{x}) \in \mathbb{R}$$

- *We may have multiple real outputs*

$$\boldsymbol{y} = f_{\mathbf{w}}(\boldsymbol{x}) \in \mathbb{R}^k$$

- *We may have restricted outputs*

$$\boldsymbol{y} = f_{\mathbf{w}}(\boldsymbol{x}) \in [A, B]^k$$

Regression via NNs: *Linear Output*

We can consider the NN output as the real label

- *We can use directly the linear output*

$$y = z = \mathbf{w}\boldsymbol{\varphi} + b \rightsquigarrow \hat{v} = y$$

  ↳ *The output layer is activated by linear function*

- *We may activate to restrict the output*
  ↳ *Say we need the output to be between -2 and 5*

$$y = \sigma(z) \rightsquigarrow \hat{v} = 7y - 2$$

Recall: *Empirical Risk Minimization*

Empirical Risk

Say $\boldsymbol{y}_n = f_{\mathbf{w}}(\boldsymbol{x}_n)$ *for every* $(\boldsymbol{x}_n, \boldsymbol{v}_n) \in \mathbb{D}$*; then, the empirical risk is*

$$\hat{R}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}(\boldsymbol{y}_n, \boldsymbol{v}_n) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}(f_{\mathbf{w}}(\boldsymbol{x}_n), \boldsymbol{v}_n)$$

Optimal model is the one which minimizes the empirical risk

$$\begin{aligned}\mathbf{w}^\star &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^m} \hat{R}(\mathbf{w}) \\ &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^m} \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}(f_{\mathbf{w}}(\boldsymbol{x}_n), \boldsymbol{v}_n)\end{aligned}$$

Choosing Loss: *Binary Classification*

Say we infer the label of a sample from model output y as

$$\hat{v} = \begin{cases} 1 & y \geqslant 0 \\ 0 & y < 0 \end{cases}$$

We can use the indicator function to compute loss

$$\mathcal{L}(y, v) = \begin{cases} 1 & y \geqslant 0 \text{ and } v = 0 \text{ or } y < 0 \text{ and } v = 1 \\ 0 & \text{otherwise} \end{cases}$$

But, it's not something we can easily work with! *We may also use*

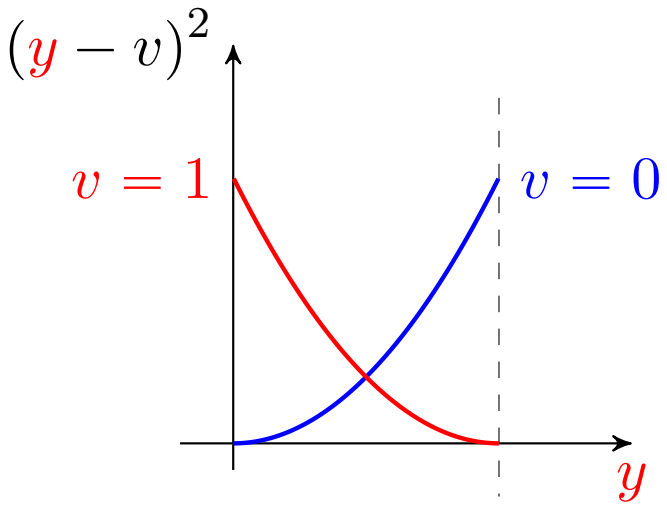
$$\mathcal{L}(y, v) = (y - v)^2$$

It still does what we want!

Binary Classification via FNN: *Cross-Entropy*

? *What do we want?*

! *We want it to convince NN return right label!*

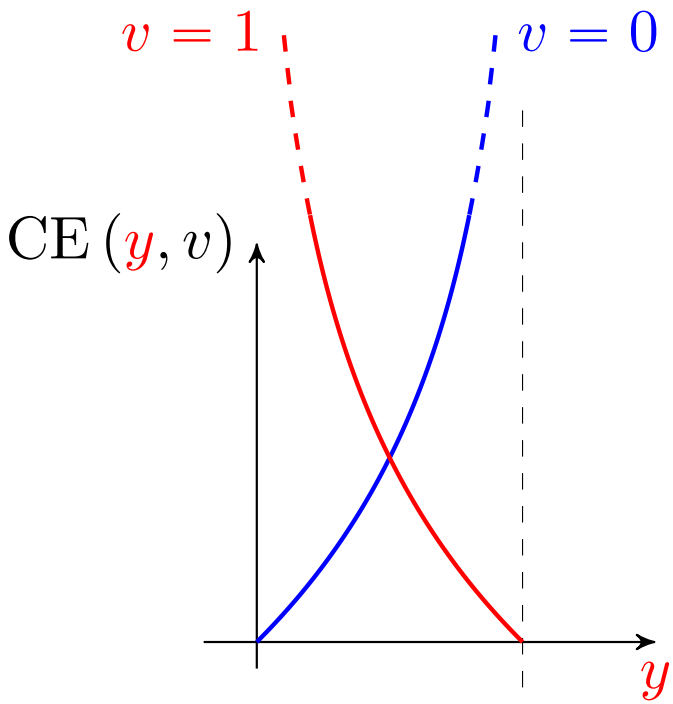


Choosing Loss: *Binary Cross Entropy*

We know y is a probability: *it's better to use maximum likelihood*

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

CE is again doing what we want! This time better



Choosing Loss: *Multiclass Classification*

For k-class classification, we can extend the idea of CE by setting

$$\mathrm{CE}\left(\boldsymbol{y}, v\right) = -\log y_v$$

In this case, we see that

$$\boldsymbol{y} = \begin{matrix} 1 \to \\ 2 \to \\ \vdots \\ v \to \\ \vdots \\ k \to \end{matrix} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \rightsquigarrow \mathrm{CE}\left(\boldsymbol{y}, v\right) = 0 \qquad \boldsymbol{y} \neq \begin{matrix} 1 \to \\ 2 \to \\ \vdots \\ v \to \\ \vdots \\ k \to \end{matrix} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \rightsquigarrow \mathrm{CE}\left(\boldsymbol{y}, v\right) > 0$$

Choosing Loss: *Generic Cross Entropy*

We can write it down more elegantly

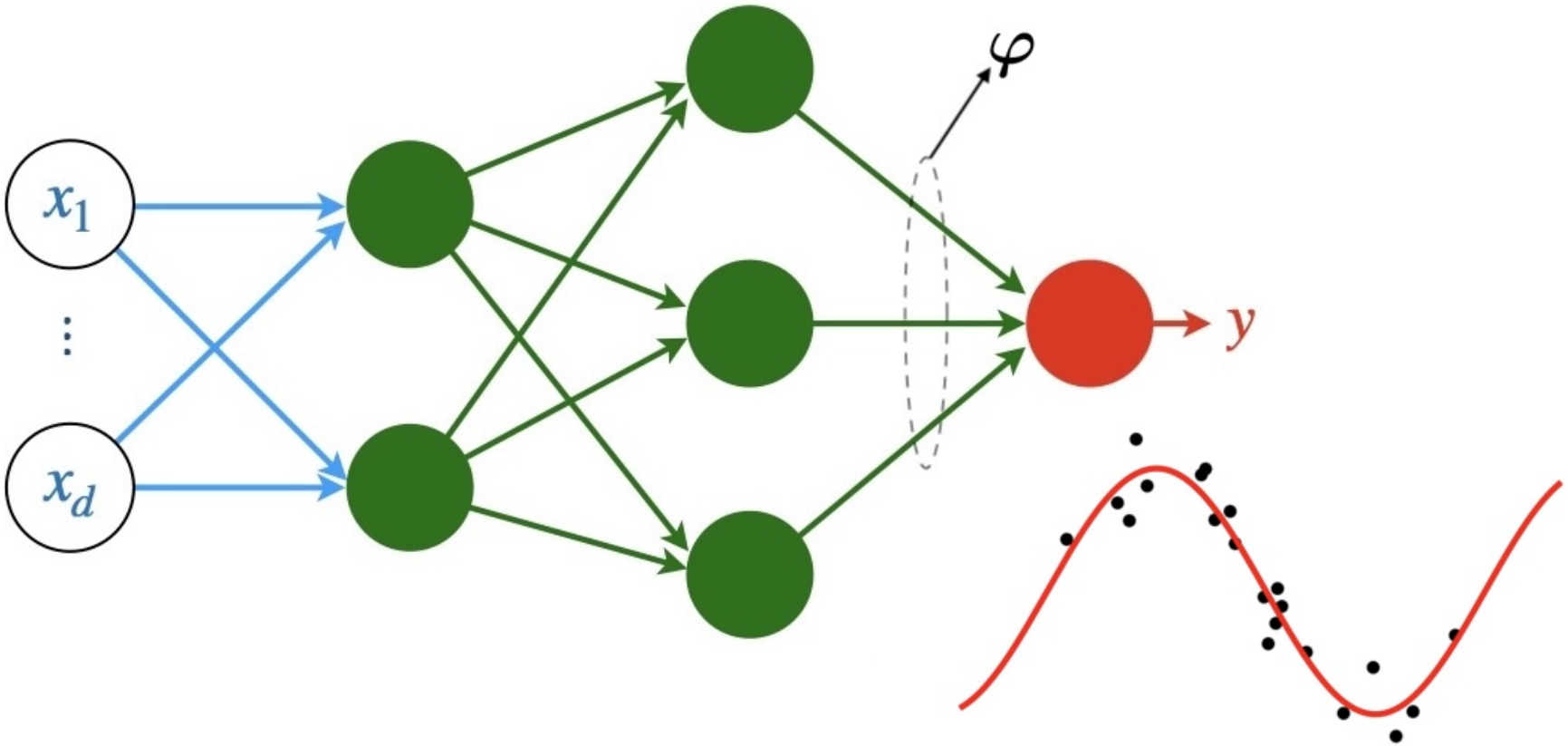
$$\mathrm{CE}\left(\boldsymbol{y}, v\right) = -\sum_{i=1}^{k} \mathbf{1}\left\{i = v\right\} \log y_i$$

Define one-hot vector for v

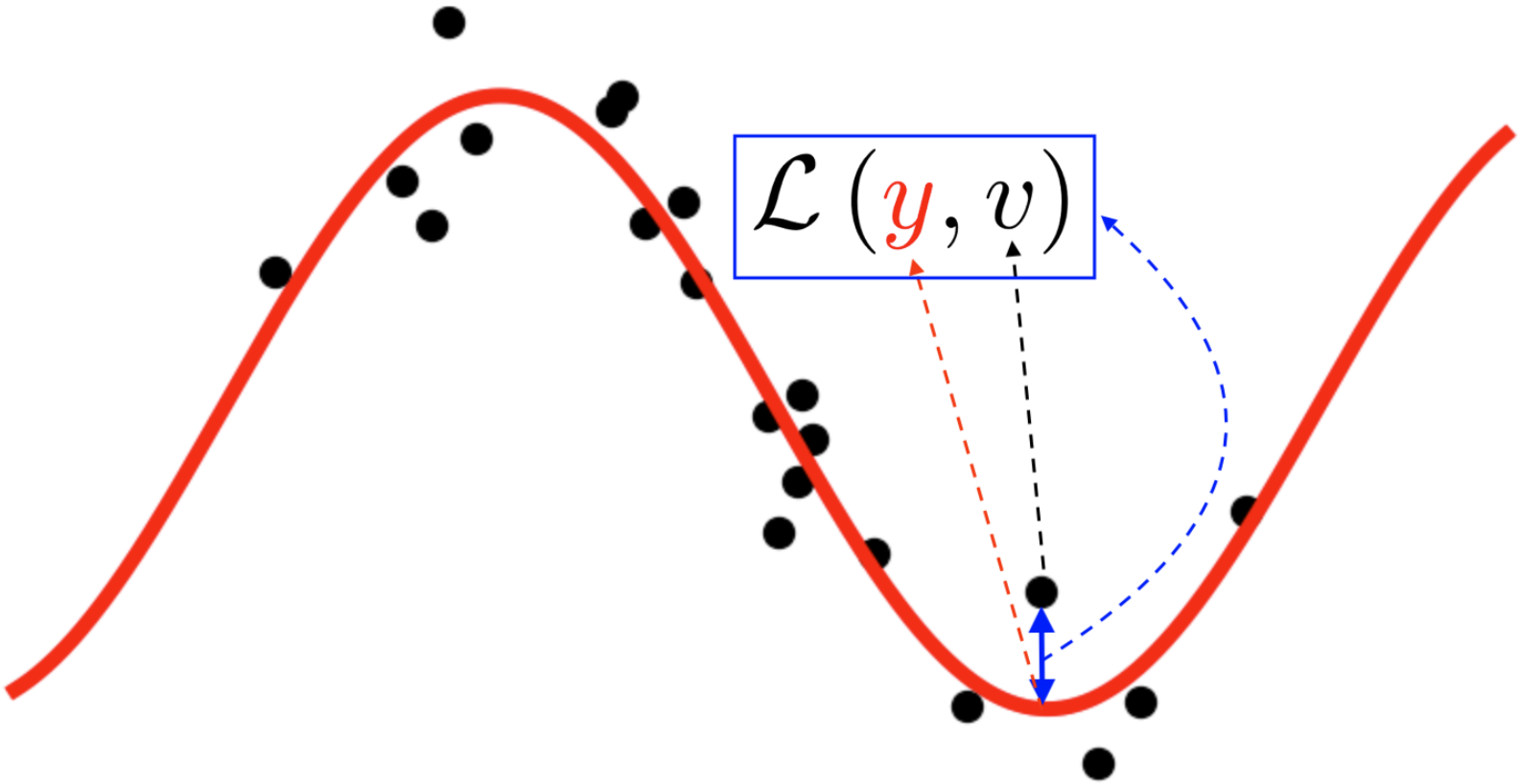
$$\mathbf{v} = \begin{matrix} 1 \to \\ 2 \to \\ \vdots \\ v \to \\ \vdots \\ k \to \end{matrix} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \rightsquigarrow \mathrm{CE}\left(y, v\right) = -\sum_{i=1}^{k} \mathrm{v}_i \log y_i = \mathrm{CE}\left(\boldsymbol{y}, \mathbf{v}\right)$$

This is the general cross-entropy between distributions $\boldsymbol{y}$ and $\mathbf{v}$

Choosing Loss: *Regression*



Choosing Loss: *Regression*



Choosing Loss: *MSE*

With k-dimensional output, we can set

$$\mathcal{L}(\boldsymbol{y}, \boldsymbol{v}) = \|\boldsymbol{y} - \boldsymbol{v}\|^2 = \sum_{i=1}^{k} (y_i - v_i)^2$$

It again does what we want

↳ *It gets minimize when we get close to true labels* $\boldsymbol{v}$

Empirical risk in this case is often referred to as

Mean Squared Error $\equiv$ *MSE*

Recall: *Gradient Descent*

`GradientDescent():`

1: Choose some *learning rate* η
2: Start with an arbitrary $\mathbf{w}^0$
3: **while** *not converged* **do**
4: Compute the gradient at $\mathbf{w}^t$
5: Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R}\left(\mathbf{w}^t\right)$
6: Update $t \leftarrow t + 1$
7: **end while**
8: Return final $\mathbf{w}^t$

So, we need to find a way to compute $\nabla \hat{R}\left(\mathbf{w}\right)$ for a given $\mathbf{w}$

Gradient of Empirical Risk

Let's define $\hat{R}_n(\mathbf{w}) = \mathcal{L}\left(f_{\mathbf{w}}(\boldsymbol{x}_n), \boldsymbol{v}_n\right)$

this is a sample loss, i.e., the loss computed for single sample $\boldsymbol{x}_n$

We can write

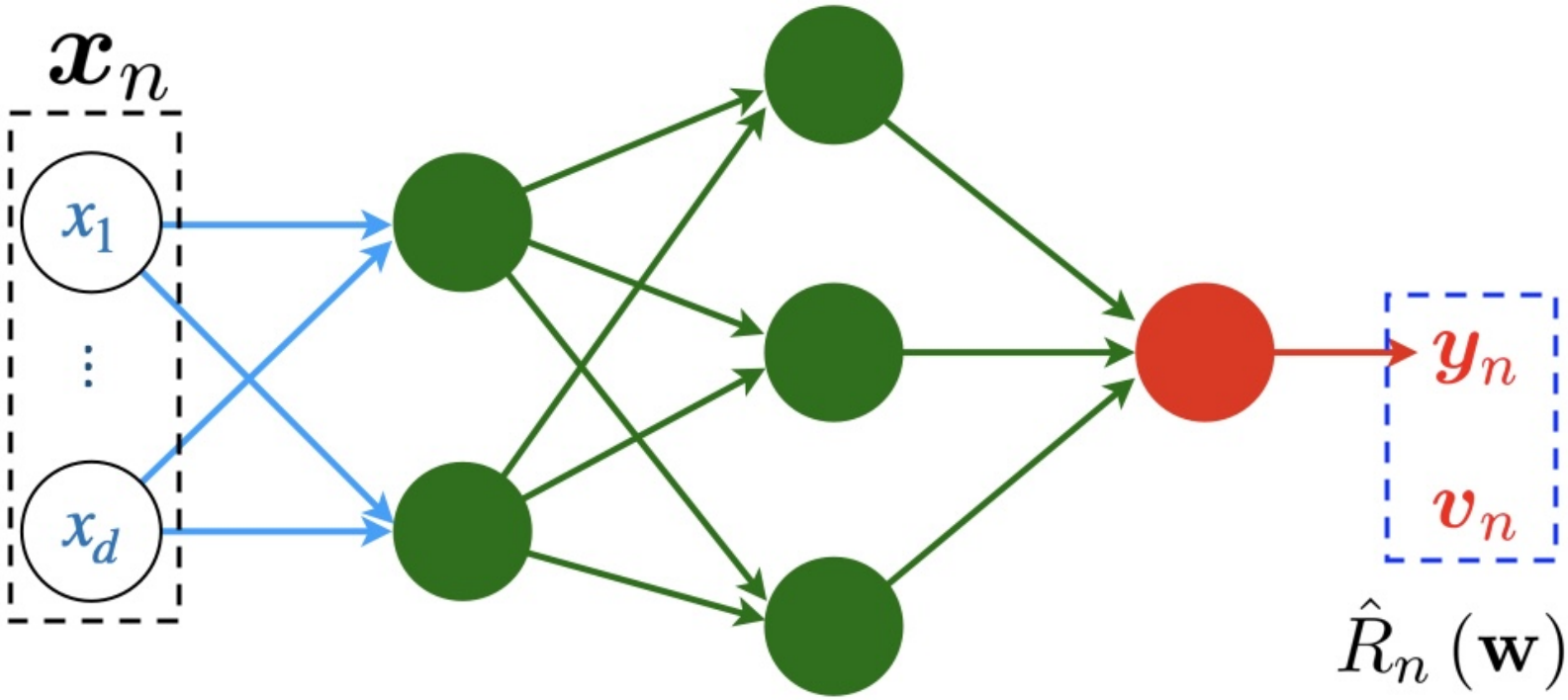
$$\nabla \hat{R}(\mathbf{w}) = \nabla \frac{1}{N} \sum_{n=1}^{N} \hat{R}_n(\mathbf{w})$$

$$= \frac{1}{N} \sum_{n=1}^{N} \nabla \hat{R}_n(\mathbf{w})$$

Conclusion

We need to find out a way to compute gradient of a sample loss

Gradient of Empirical Risk $\equiv$ *Average of Sample Gradients*



Computing Sample Gradients

The sample loss is a *nested function* of model parameters

$$\nabla \hat{R}_n(\mathbf{w}) = \nabla \mathcal{L}\left(f_{\mathbf{w}}(\boldsymbol{x}_n), \boldsymbol{v}_n\right)$$

We need to compute the gradient of a nested function

Nested Function

Let $y = f(z)$ and $z = g(x)$; then, y is a nested function of x

Example: *Say $y = z^2$ and $z = 3x$; then,*

$$y = z^2 = (3x)^2 = 9x^2$$

Review: *Chain Rule*

Problem

Let $y = f(z)$ and $z = g(x)$: what is the derivative of y with respect to x?

If we change x with a tiny change $\mathrm{d}x$: z changes with

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

The change $\mathrm{d}z$ will change y by

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

So, we have

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

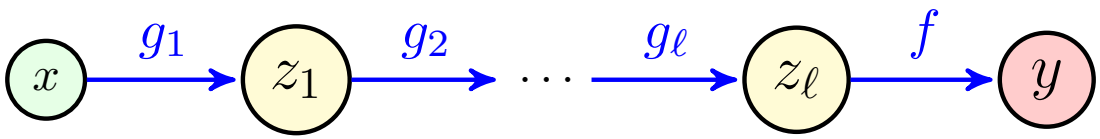
Review: *Chain Rule*

Chain Rule

Let $z_1 = g_1(x)$ *and* $z_2 = g_2(z_1)$ *and* $\ldots$ *and* $y = f(z_\ell)$*: then, we have*

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z_\ell}\frac{\mathrm{d}z_\ell}{\mathrm{d}z_{\ell-1}}\cdots\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}x}$$

We can show this on the so-called *computation graph*



Review: *Vectorized Chain Rule*

Problem

$y = f(z_1, \ldots, z_\ell)$ *with* $z_i = g_i(x)$*: what is derivative of* y *with respect to* x*?*

If we change x *with a tiny change* $\mathrm{d}x$*, each* z_i *changes with*

$$\mathrm{d}z_i = g_i'(x)\,\mathrm{d}x$$

The changes $\mathrm{d}z_1, \ldots, \mathrm{d}z_\ell$ *will change* y *by*

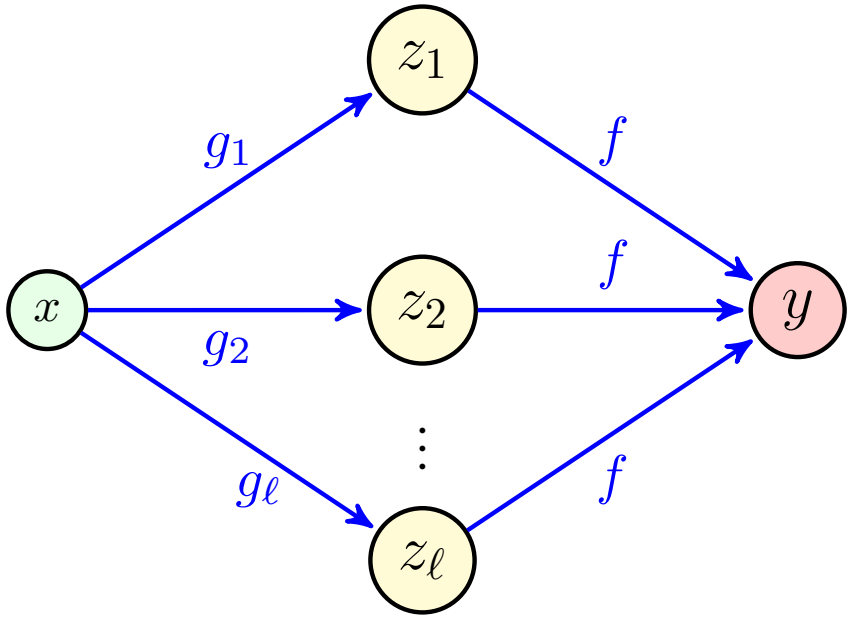
$$\mathrm{d}y = \frac{\partial y}{\partial z_1}\mathrm{d}z_1 + \ldots + \frac{\partial y}{\partial z_\ell}\mathrm{d}z_\ell = \frac{\partial y}{\partial z_1}g_1'(x)\,\mathrm{d}x + \ldots + \frac{\partial y}{\partial z_\ell}g_\ell'(x)\,\mathrm{d}x$$

So, we have

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\partial y}{\partial z_1}\frac{\mathrm{d}z_1}{\mathrm{d}x} + \ldots + \frac{\partial y}{\partial z_\ell}\frac{\mathrm{d}z_\ell}{\mathrm{d}x}$$

Review: *Vectorized Chain Rule*

We can show this again on a *computation graph*



How *Compute* Derivatives on Computer: *Example*

Consider the following example

$$z_1 = x^2 \qquad z_2 = x^3 \qquad y = 2z_1^3 + z_2^2$$

We look for derivative of y w.r.t. x at $x = 1$

Our analytic brain does the following

$$y = 2z_1^3 + z_2^2 = 2\left(x^2\right)^3 + \left(x^3\right)^2 = 3x^6$$

Then, we would say

$$\frac{\mathrm{d}y}{\mathrm{d}x}|_{x=1} = 18x^5|_{x=1} = 18$$

How *Compute* Derivatives on Computer: *Example*

Consider the following example

$$z_1 = x^2 \qquad z_2 = x^3 \qquad y = 2z_1^3 + z_2^2$$

We look for derivative of y w.r.t. x at $x = 1$

The computer can only follow the computation graph

```
» z1 = x**2          z2 = x**3
» y = 2*(z1**3) + (z2**2)
» dy_dz1 = 6*(z1**2)          dy_dz2 = 2*z2
» dz1_dx = 2*x          dz2_dx = 3*x
» dy_dx = dy_dz1 * dz1_dx + dy_dz2 * dz2_dx
```

Back to Computing Sample Gradients

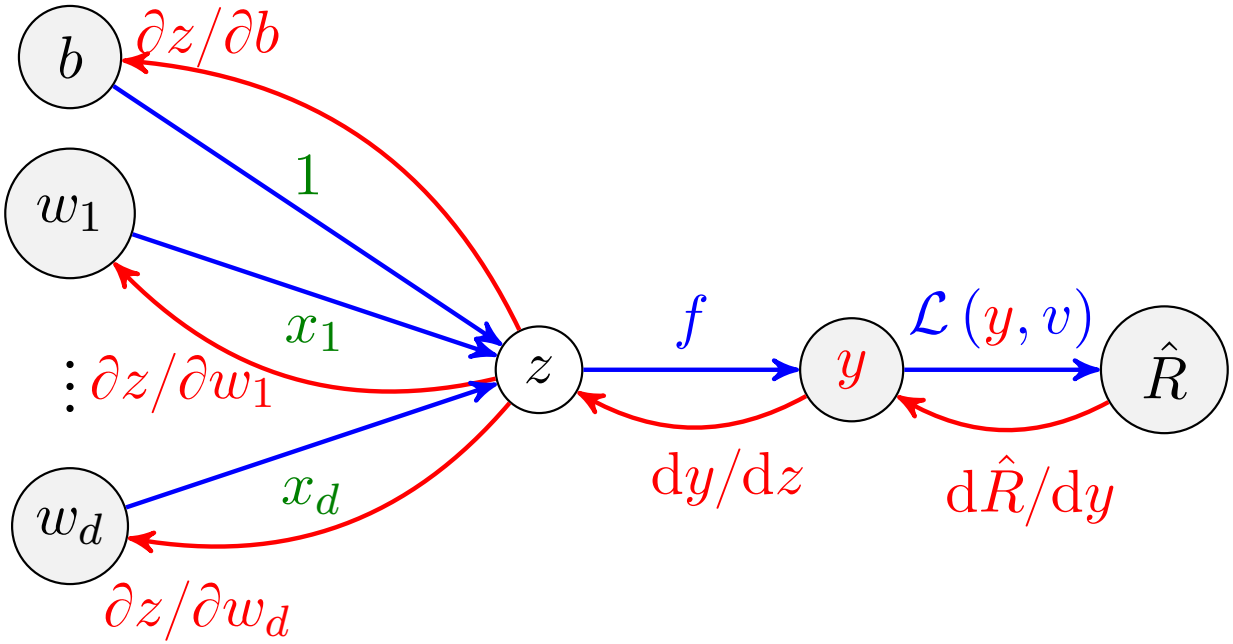
The sample loss is a *nested function* of model parameters

$$\nabla \hat{R}_n(\mathbf{w}) = \nabla \mathcal{L}\left(f_{\mathbf{w}}(\boldsymbol{x}_n), \boldsymbol{v}_n\right)$$

? *How can we compute the derivative w.r.t. weights?*

! *We represent the NN by a computation graph and use chain rule*

Example: *Single Neuron*

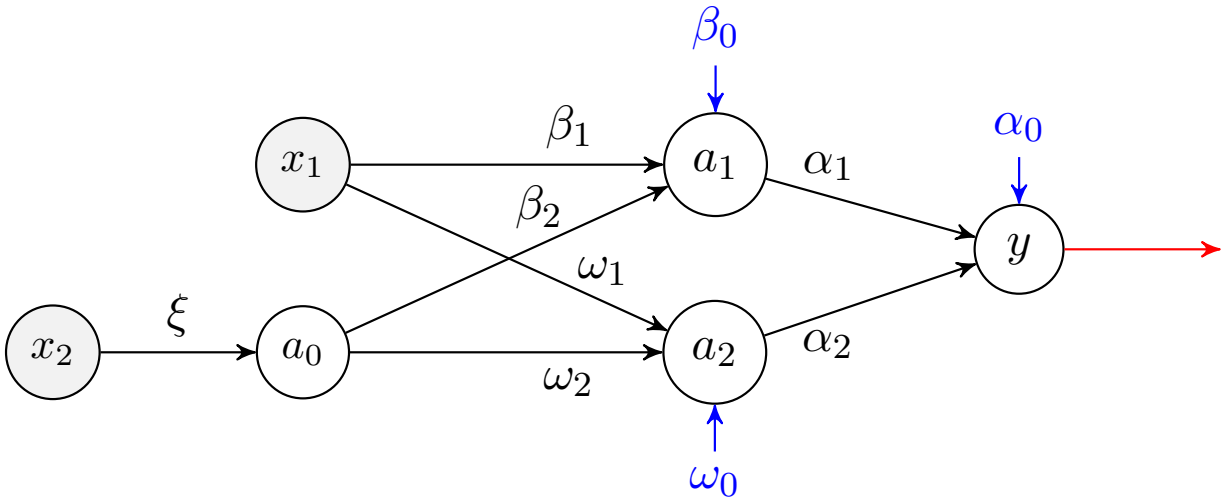


For weight w_n, we can write

$$\frac{\partial \hat{R}}{\partial w_n} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z}\frac{\partial z}{\partial w_n}$$

the same is valid for the bias

Example: *Simple Neural Network*



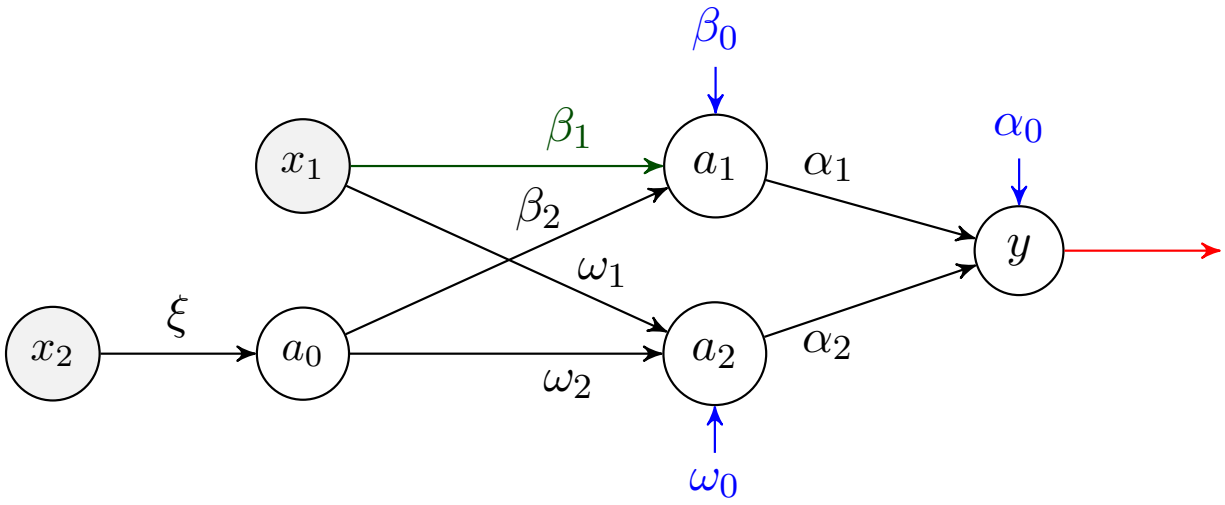
In first hidden node, we have $h_0 = \xi x_2$ *and* $a_0 = f(h_0)$*; then, we get*

$$h_1 = \beta_0 + \beta_1 x_1 + \beta_2 a_0 \rightsquigarrow a_1 = f(h_1)$$
$$h_2 = \omega_0 + \omega_1 x_1 + \omega_2 a_0 \rightsquigarrow a_2 = f(h_2)$$

and at the output we have

$$z = \alpha_0 + \alpha_1 a_1 + \alpha_2 a_2 \rightsquigarrow y = f(z)$$

Example: *Simple Neural Network*



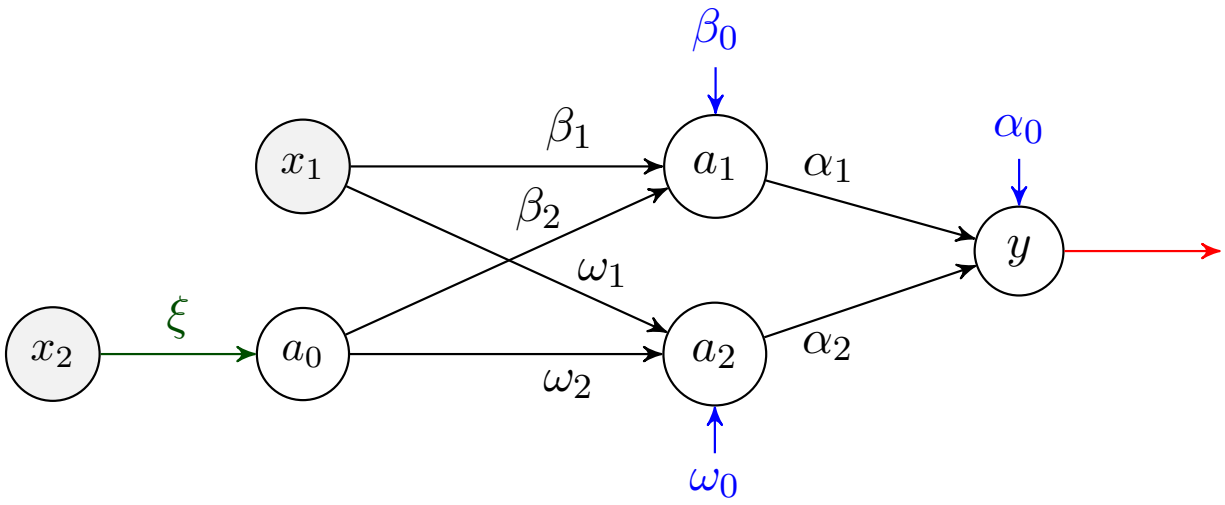
By chain rule, we have

$$\frac{\partial \hat{R}}{\partial \beta_1} = \frac{\partial a_1}{\partial \beta_1} \frac{\partial \hat{R}}{\partial a_1}$$

and we have

$$\frac{\partial \hat{R}}{\partial a_1} = \frac{\partial y}{\partial a_1} \frac{\partial \hat{R}}{\partial y}$$

Example: *Simple Neural Network*



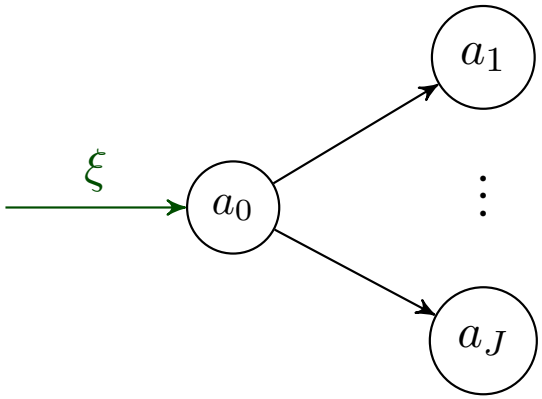
By chain rule, we have

$$\frac{\partial \hat{R}}{\partial \xi} = \frac{\partial a_0}{\partial \xi} \frac{\partial \hat{R}}{\partial a_0}$$

and we have

$$\frac{\partial \hat{R}}{\partial a_0} = \frac{\partial a_1}{\partial a_0} \frac{\partial \hat{R}}{\partial a_1} + \frac{\partial a_2}{\partial a_0} \frac{\partial \hat{R}}{\partial a_2}$$

Extension to General NN



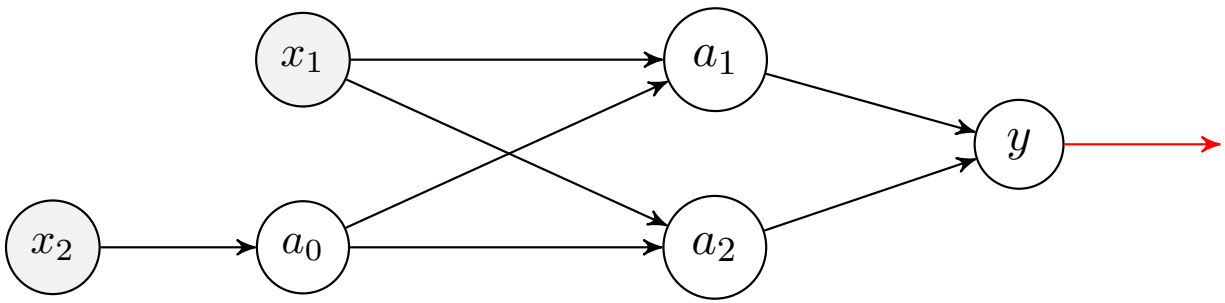
For a neuron with J children, we can write

$$\frac{\partial \hat{R}}{\partial \xi} = \frac{\partial h}{\partial \xi} \frac{\partial \hat{R}}{\partial a_0}$$

and we have

$$\frac{\partial \hat{R}}{\partial a_0} = \sum_{j=1}^{J} \frac{\partial a_j}{\partial a_0} \frac{\partial \hat{R}}{\partial a_j}$$

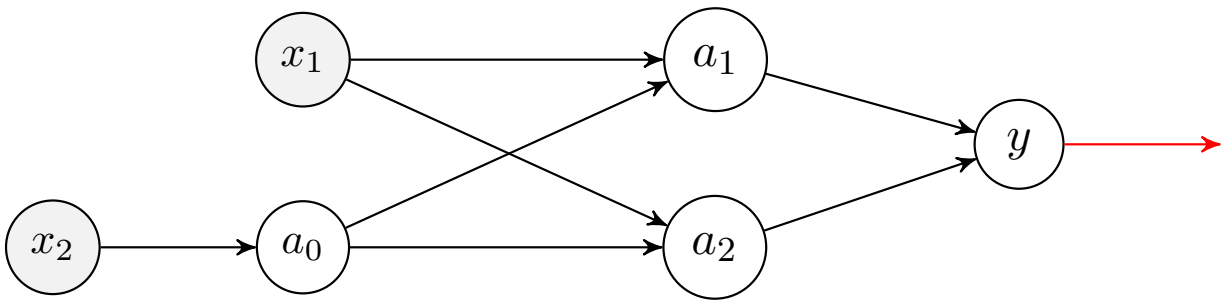
Backpropagation: *Forward Pass*



`Forward`$\left(\boldsymbol{x}_n \middle| \mathbf{w}^t\right)$:

1: Send $\boldsymbol{x}_n$ over the network
2: **for** *each neuron from input to output* **do**
3: Compute the linear output h and activated output a
4: **end for**
5: Return all linear and activated outputs

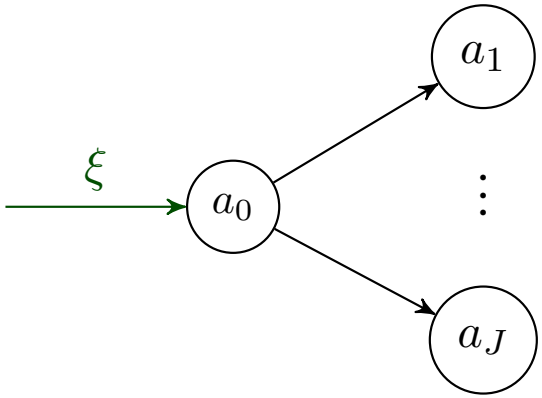
Backpropagation: *Backward Pass*



`Backward`$\left(\boldsymbol{v}_n | \mathbf{w}^t\right)$:

1: Set $\hat{R}_n = \mathcal{L}\left(\boldsymbol{y}_n, \boldsymbol{v}_n\right)$ and compute $\nabla_{\boldsymbol{y}} \hat{R}_n$
2: Send $\nabla_{\boldsymbol{y}} \hat{R}_n$ backward over the network
3: **for** *each neuron* a_i from output to input **do**
4: Compute $\partial \hat{R}_n / \partial a_i$ using its children
5: Compute derivative of $\hat{R}_n$ w.r.t. weights of a_i from $\partial \hat{R}_n / \partial a_i$
6: **end for**
7: Return $\nabla_{\mathbf{w}^t} \hat{R}_n$

Backpropagation: *Need for Gradient*

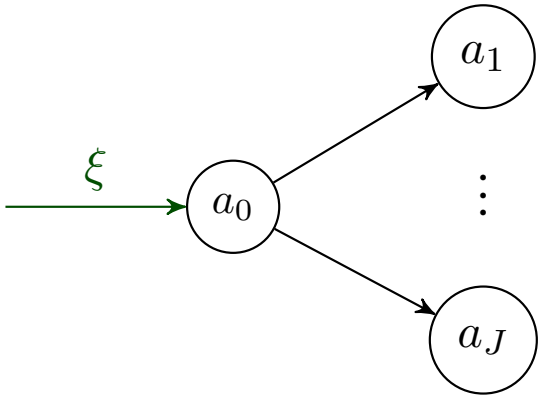


At any activated neuron, we have $a_j = f(h_j)$

$$\frac{\partial a_j}{\partial a_0} = \frac{\partial a_j}{\partial h_j}\frac{\partial h_i}{\partial a_0}$$
$$= f'(h_j)\frac{\partial h_i}{\partial a_0}$$

Activation should be differentiable!

Backpropagation: *Need for Gradient*



By backtracking, we will end up with

$$\frac{\partial \hat{R}}{\partial \xi} \rightsquigarrow \frac{\partial \hat{R}}{\partial a_0} \rightsquigarrow \frac{\partial \hat{R}}{\partial a_j} \rightsquigarrow \ldots \rightsquigarrow \nabla_{\boldsymbol{y}} \hat{R}$$

Loss should be differentiable!

Backpropagation in Practice: *PyTorch Example*

Most packages have backpropagation implemented

```
prediction = model(data) # forward pass
```

We just need to pass backward

```
loss = (prediction - labels).sum()
loss.backward() # backward pass
```

Further Read

- Goodfellow
  - ↳ Chapter 6: *Section 6.5* ***Backpropagation***
- Bishop
  - ↳ Chapter 4 ***Backpropagation***
- ESL
  - ↳ Chapter 7 ***Backpropagation***

Gradient Descent with Backpropagation

`GradientDescent()`:

1: Choose some *learning rate* η
2: Start with an arbitrary $\mathbf{w}^0$
3: **while** *not converged* **do**
4: Compute the gradient at $\mathbf{w}^t$
5: Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R}\left(\mathbf{w}^t\right)$
6: Update $t \leftarrow t + 1$
7: **end while**
8: Return final $\mathbf{w}^t$

We know that

$$\nabla \hat{R}\left(\mathbf{w}^t\right) = \frac{1}{N} \sum_{n=1}^{N} \nabla \hat{R}_n\left(\mathbf{w}^t\right)$$

We can compute $\nabla \hat{R}_n\left(\mathbf{w}^t\right)$ *for each* n *by backpropagation*

Gradient Descent with Backpropagation

`GradientDescent()`:

1: Choose some *learning rate* η
2: Start with an arbitrary $\mathbf{w}^0$
3: **while** *not converged* **do**
4: Compute the gradient at $\mathbf{w}^t$
5: **for** *all samples in* $\mathbb{D}$ **do**
6: Compute $\nabla \hat{R}_n \left(\mathbf{w}^t\right)$ with *backpropagation*
7: **end for**
8: $\nabla \hat{R} \left(\mathbf{w}^t\right) = \texttt{average} \left\{\nabla \hat{R}_1, \ldots, \nabla \hat{R}_N\right\}$
9: Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R} \left(\mathbf{w}^t\right)$
10: Update $t \leftarrow t + 1$
11: **end while**
12: Return final $\mathbf{w}^t$

Training with Complete Dataset

`FullBatchGD()`:

1: Choose some *learning rate* η
2: Start with an arbitrary $\mathbf{w}^0$
3: **while** *not converged* **do**
4: Compute the gradient at $\mathbf{w}^t$
5: **for** *all samples in* $\mathbb{D}$ **do**
6: Compute $\nabla \hat{R}_n \left(\mathbf{w}^t\right)$ with *backpropagation*
7: **end for**
8: $\nabla \hat{R} \left(\mathbf{w}^t\right) = \texttt{average} \left\{\nabla \hat{R}_1, \ldots, \nabla \hat{R}_N\right\}$
9: Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R} \left(\mathbf{w}^t\right)$
10: Update $t \leftarrow t + 1$
11: **end while**
12: Return final $\mathbf{w}^t$

Here, we use all samples for every single GD iteration

↳ *We need many GD iterations to converge!*

Training with Complete Dataset: *ImageNet*

In ImageNet we have 1,281,167 images belonging to 1000 classes!

- ↳ *Each iteration of GD requires 1,281,167 forward and backward passes!*
- ↳ *This does not sound practical!*

GD with Single Sample Gradient

`SampleGD():`

1: Choose some *learning rate* η
2: Start with an arbitrary $\mathbf{w}^0$
3: **while** *not converged* **do**
4: Compute the gradient at $\mathbf{w}^t$
5: Compute $\nabla \hat{R}_n\left(\mathbf{w}^t\right)$ with *backpropagation*
6: Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R}_n\left(\mathbf{w}^t\right)$
7: Update $t \leftarrow t+1$
8: **end while**
9: Return final $\mathbf{w}^t$

The idea is very simple

do not loop over samples: update after each backpropagation

GD with Single Sample Gradient: *Possibility of Repetition*

`SampleGD()`:

1: Choose some *learning rate* η
2: Start with an arbitrary $\mathbf{w}^0$
3: **while** *not converged* **do**
4: Compute the gradient at $\mathbf{w}^t$
5: Compute $\nabla \hat{R}_n\left(\mathbf{w}^t\right)$ with *backpropagation*
6: Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R}_n\left(\mathbf{w}^t\right)$
7: Update $t \leftarrow t+1$
8: **end while**
9: Return final $\mathbf{w}^t$

? *What if after last sample, we end up with initial weights?!*

! *Well, we could try shuffling the samples randomly*

Stochastic GD

`SGD()`:

1: Choose some *learning rate* η
2: Start with an arbitrary $\mathbf{w}^0$
3: **while** *not converged* **do**
4: Compute the gradient at $\mathbf{w}^t$
5: **if** *we finished with all samples in* $\mathbb{D}$ **then**
6: Shuffle $\mathbb{D}$ randomly and start over
7: **end if**
8: Compute $\nabla \hat{R}_n \left(\mathbf{w}^t\right)$ with *backpropagation*
9: Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R}_n \left(\mathbf{w}^t\right)$
10: Update $t \leftarrow t + 1$
11: **end while**
12: Return final $\mathbf{w}^t$

Full-Batch Training vs SGD

? *Do we pay any price for the speed boost in SGD?!*

! *Yes! Our gradient estimation have larger variance*

Gradient Estimate Variance

In fact, we could show that the gradients

$$\nabla \hat{R}\left(\mathbf{w}^t\right) \text{ and } \nabla \hat{R}_n\left(\mathbf{w}^t\right)$$

are both estimating $\nabla R\left(\mathbf{w}^t\right)$*, i.e., gradient of the true risk. In this case, the estimators used by SGD are noisier than those used by full batch GD!*

Moral of Story

SGD has lower quality of update in each GD iteration

SGD with Mini-Batches

`miniBatchSGD()`:

Choose some *learning rate* η
Split $\mathbb{D}$ to mini-batches, each of size B
Start with an arbitrary $\mathbf{w}^0$
while *not converged* **do**
    Compute the gradient at $\mathbf{w}^t$
    if *we finished with all mini-batches* **then**
        Shuffle them randomly and start over
    end if
    for *all samples in the mini-batch* **do**
        Compute $\nabla \hat{R}_n \left(\mathbf{w}^t\right)$ with *backpropagation*
    end for
    $\nabla \hat{R}_b \left(\mathbf{w}^t\right) = \texttt{average} \left\{\nabla \hat{R}_1, \ldots, \nabla \hat{R}_B\right\}$
    Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \nabla \hat{R}_b \left(\mathbf{w}^t\right)$
    Update $t \leftarrow t + 1$
end while
Return final $\mathbf{w}^t$

Few Definitions

Iteration

Iteration refers to one GD iteration, i.e., after each mini-batch

Epoch

Each time we go through all samples in $\mathbb{D}$, we are finished with one epoch

Batch-size

The size of each mini-batch B is called batch-size

Example

Consider MNIST with 60K samples used with batch-size $B = 100$

Classic Trade-off: *Variance vs Speed*

Mini-batch SGD is the most common form of GD we use

- *It provides a fair trade-off between speed and variance*
- *If we set mini-batch size B large*
  - ↳ *We have less variance* $\rightsquigarrow$ *better gradient quality in each iteration*
  - ↳ *We have lower speed* $\rightsquigarrow$ *longer training time*
- *If we set mini-batch size B small*
  - ↳ *We have higher variance* $\rightsquigarrow$ *lower gradient quality in each iteration*
  - ↳ *We have higher speed* $\rightsquigarrow$ *shorter training time*

Speed-Variance Trade-off

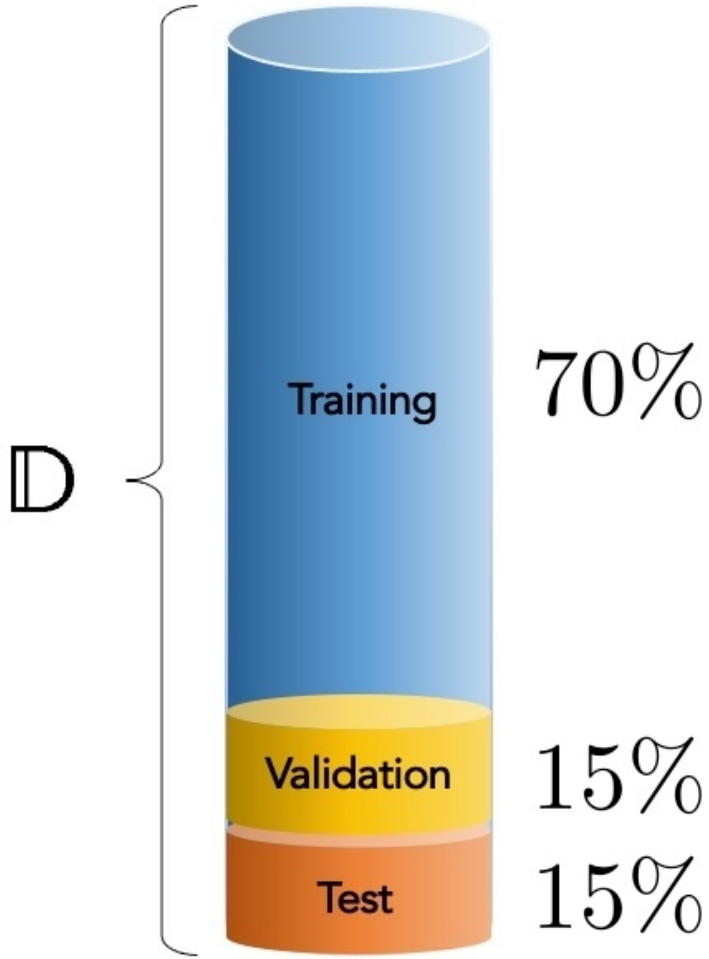
Choice of batch-size specifies the trade-off between variance and speed

Gradient-based Optimizer

Mini-batch SGD is probably not the one you use in your implementation

- *In practice we use other tricks to improve quality of gradient estimation*
  - ↳ *Momentum, learning rate scheduling, . . .*
- *The available optimizers all follow the mini-batch SGD approach*
  - ↳ *They only use tricks to improve quality of* $\nabla \hat{R}_b\left(\mathbf{w}^t\right)$ *before each iteration*
- *Some famous optimizers are*
  - ↳ *Rprop, RMSprop and Adam*
  - ↳ *Check PyTorch for more choices!*

Splitting Data



Building and Training NN

`TrainingLoop()`:

1: Build NN $\boldsymbol{y} = f_{\mathbf{w}}(\boldsymbol{x})$ with some hyperparameters and *initial weights*
2: Split training set to mini-batches
3: Specify the loss function $\mathcal{L}$
4: **for** *epochs* = $1, \ldots, E$ **do**
5: Keep applying mini-batch SGD
6: **end for**
7: Return final weights $\mathbf{w}^*$, average training loss, and accuracy on training set

Validation

`Validation():`

1: Realize NN $\boldsymbol{y} = f_{\mathbf{w}*}(\boldsymbol{x})$ with *trained weights* $\mathbf{w}^*$
2: **for** *sample* i *in validation set* **do**
3: Compute $\boldsymbol{y}_i = f_{\mathbf{w}*}(\boldsymbol{x}_i)$
4: Compute sample loss $\hat{R}_i = \mathcal{L}(\boldsymbol{y}_i, \boldsymbol{v}_i)$
5: Check whether inferred label $\hat{\boldsymbol{v}}_i$ is the same as true label $\boldsymbol{v}_i$
6: **end for**
7: Return average validation loss and accuracy

We repeat training with new hyperparameters till it's tuned

Evaluation

`Evaluation()`:

1: Realize NN $\boldsymbol{y} = f_{\mathbf{w}*}(\boldsymbol{x})$ with *trained weights* $\mathbf{w}^*$ *and tuned hyperparameters*
2: **for** *sample* i *in evaluation set* **do**
3: Compute $\boldsymbol{y}_i = f_{\mathbf{w}*}(\boldsymbol{x}_i)$
4: Compute sample loss $\hat{R}_i = \mathcal{L}(\boldsymbol{y}_i, \boldsymbol{v}_i)$
5: Check whether inferred label $\hat{\boldsymbol{v}}_i$ is the same as true label $\boldsymbol{v}_i$
6: **end for**
7: Return average evaluation loss and accuracy

Further Read

- Goodfellow
  - ↳ Chapter 6: *Section 6.2 – 6.3* ***Training***
- Bishop
  - ↳ Chapter 5 ***Optimizers***
- ESL
  - ↳ Chapter 10: *Section 10* ***Optimizers***