Lecture 6: Logistic Regression and Automatic Differentiation

Matt Ellis and Mike Smith

Announcements

Assignment

Part 1 released last Friday, available on Blackboard as a Jupyter notebook.

Part 2 will be released next week to give us time to cover some content.

Please submit **2 jupyter notebook** files. Deadline **23:59 6th December 2022**.

These are independent assignments, feel free to discuss the content but you should code it yourself.

Note: there was a typo in question 2, **8 columns** remaining not 9.

Session outline

Logistic regression

- Types of classifiers
- Bernoulli distribution
- Odds and Log Odds

Automatic differentiation

- Derivatives and ways to compute them
- Forward mode AD
- Reverse mode AD

Learning Objectives

By the end of this session you should be able to:

1. Explain the difference between **Generative** and **Discriminative** models
2. Explain the process of **logistic regression**
3. Understand what is meant by **automatic differentiation**
4. Construct a **computational graph** for a model
5. Apply **forward and reverse modes** of automatic differentiation

Logistic Regression

Probabilistic Classifiers

Determine a class from some features $\rightarrow P(y \mid \mathbf{x})$

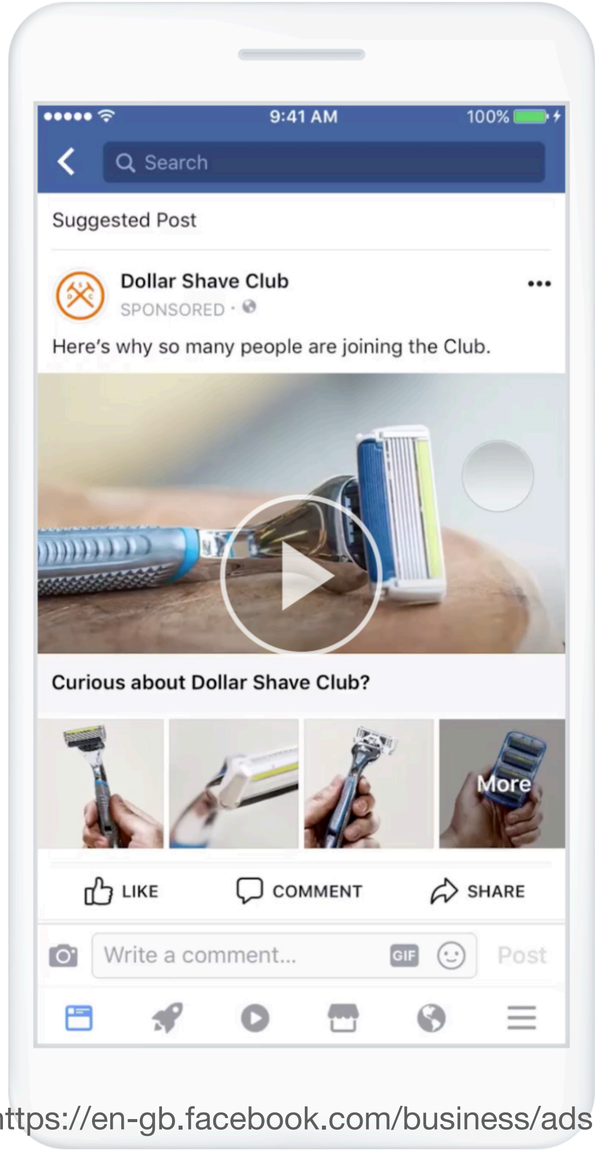
Discriminative

- Optimise an **explicitly defined** form of the decision boundary.
- Assume some functional form for $P(y \mid \mathbf{x})$.
- Estimate parameters for this based on training data.

Generative

- Define a **process** that could have generated the observations.
- Assume some functional form for $P(\mathbf{x} \mid y)$ and $P(\mathbf{x})$.
- Estimate parameters for these from training data.
- Use Bayes' rule to find $P(y \mid \mathbf{x})$.

Adverts - to click or not to click?



https://en-gb.facebook.com/business/ads

Click through rate (CTR) prediction

Estimating click probabilities is a useful business tool, e.g will user i click on ad j.

Not just ads:

- Optimise search results
- Suggest news articles
- Product recommendations

These are binary decisions - **logistic regression** is a model for predicting binary outcomes, used by many internet companies (e.g Meta).

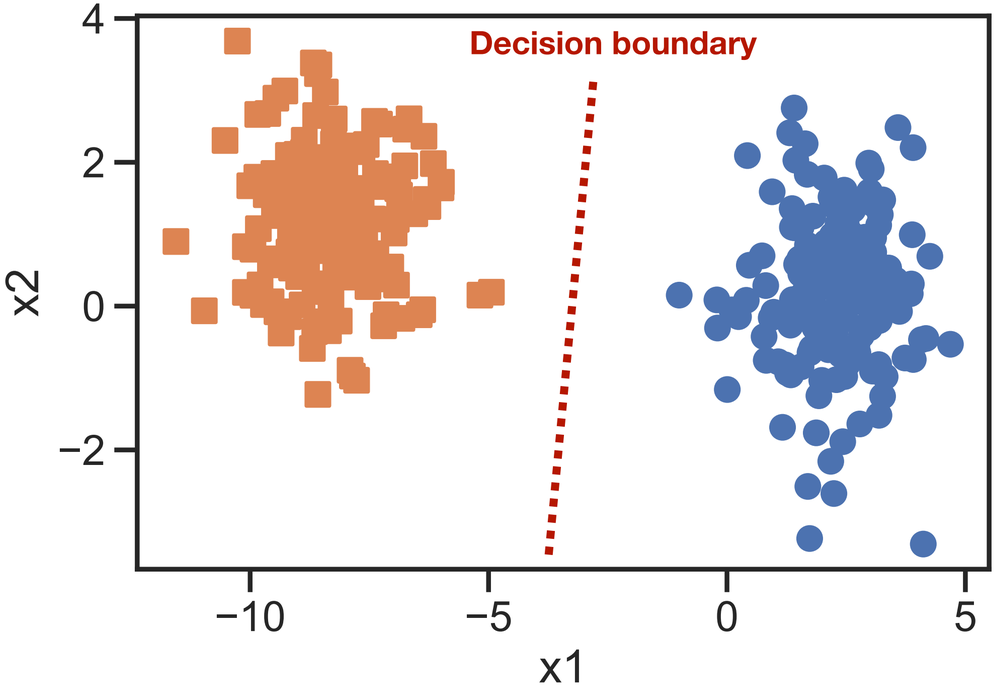
Binary classification

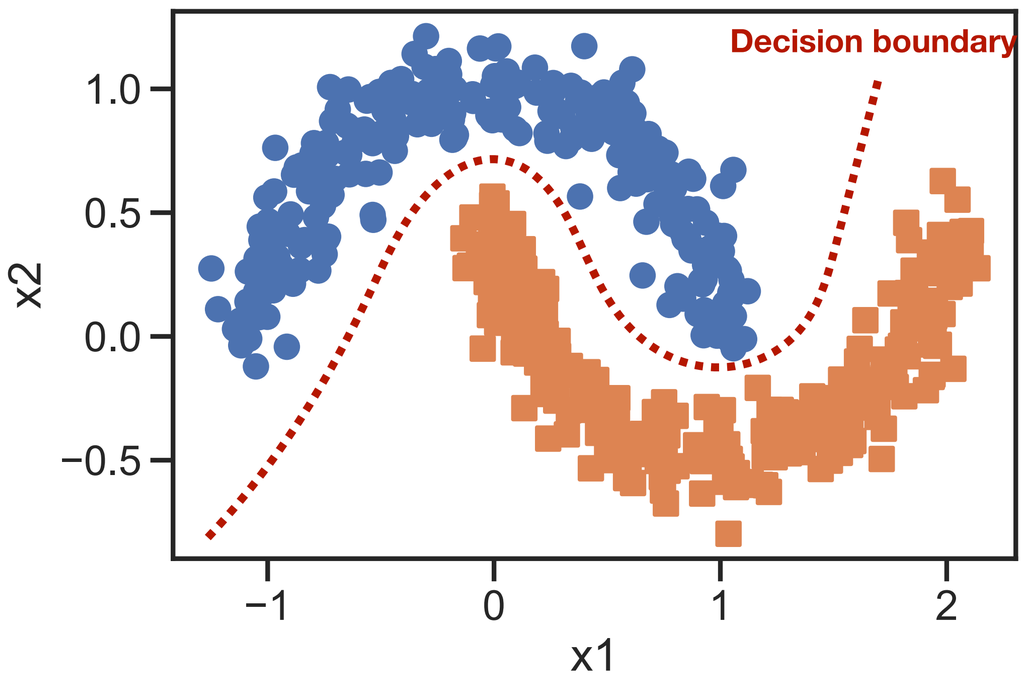
(One of the) Simplest prediction task → binary classification

Input (to predict from) : feature vectors

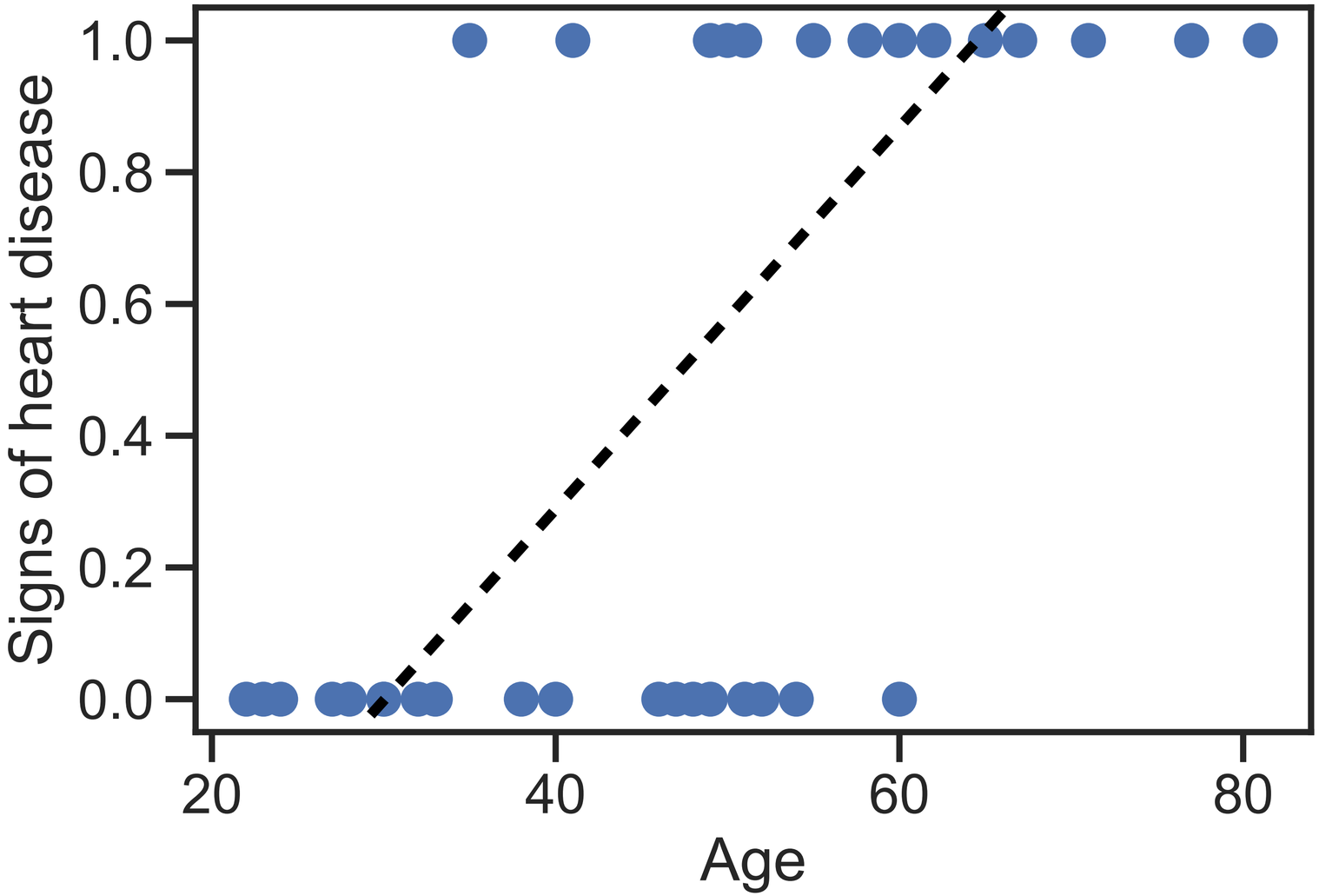
Output (to predict) : 0 or 1 (class 0 or class 1)

Difficulty determined by the distribution of the input





Example: Age and signs of coronary disease (CD)



Linear regession?

1. Not a good fit.
2. Out of range.

Binary outcomes - what distribution?

Bernoulli distribution $Y = \begin{cases} 1 & \text{with probability } \pi \\ 0 & \text{otherwise} \end{cases}$

Probabilities of each outcome:

$$P(Y = 1) = \pi$$
$$P(Y = 0) = 1 - \pi$$

$$\longrightarrow \quad P(Y = y) = \pi^{y}(1 - \pi)^{(1-y)}$$

What if we fit π as a function of our input features? $\pi = \mathbf{w}^T\mathbf{x}$

What is the range of π and $\mathbf{w}^T\mathbf{x}$? $\pi = [0,1]$ $\mathbf{w}^T\mathbf{x} = [-\infty, \infty]$

So what can we use?

Odds

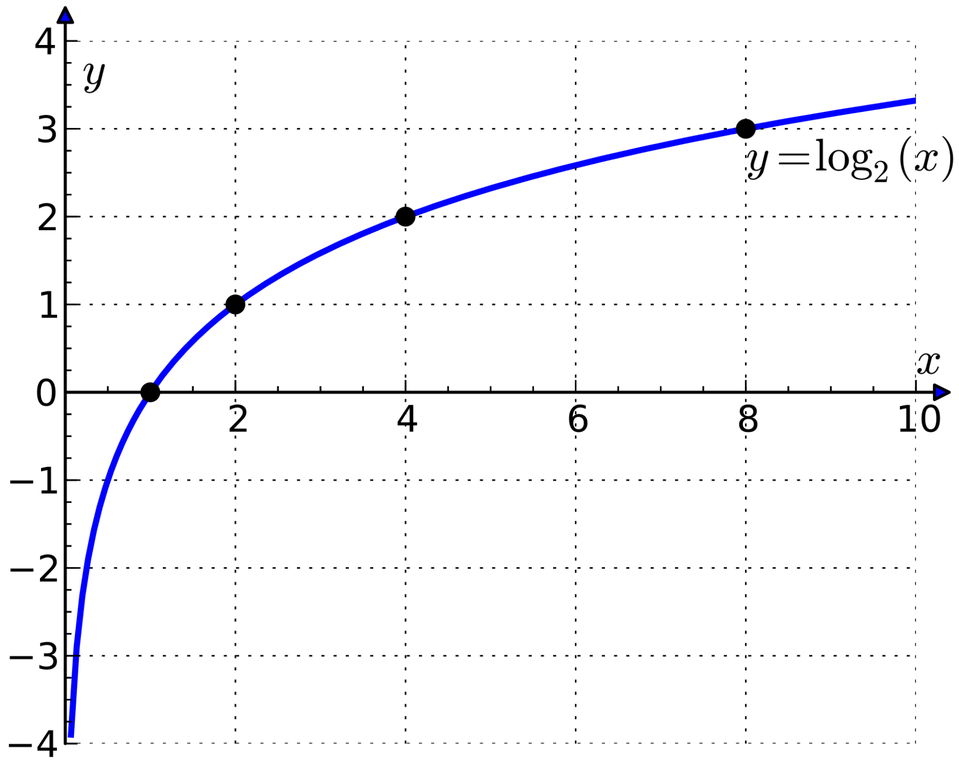
"Likelihood of something happening"
Ratio of number of events that produce that outcome against the number that does not.

$$\text{Odds} = \frac{\pi}{1-\pi} = [0,\infty]$$

If we take the logarithm then the numbers close to zero will be spread to $-\infty$.
Log Odds = Logit

$$\text{Logit}(\pi) = \log\left(\frac{\pi}{1-\pi}\right)$$

$$= [-\infty, \infty]$$



From Logit to Logistic

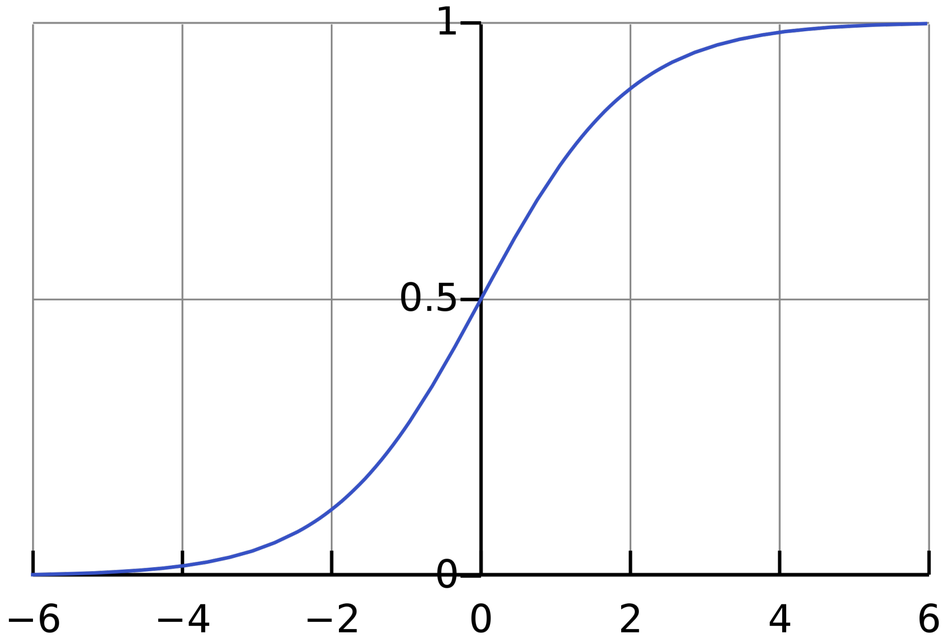
Linear **regression** on the **log odds** → Logistic Regression

$$\text{Logit}(\pi) = \log\left(\frac{\pi}{1-\pi}\right) = \mathbf{w}^T\mathbf{x}$$

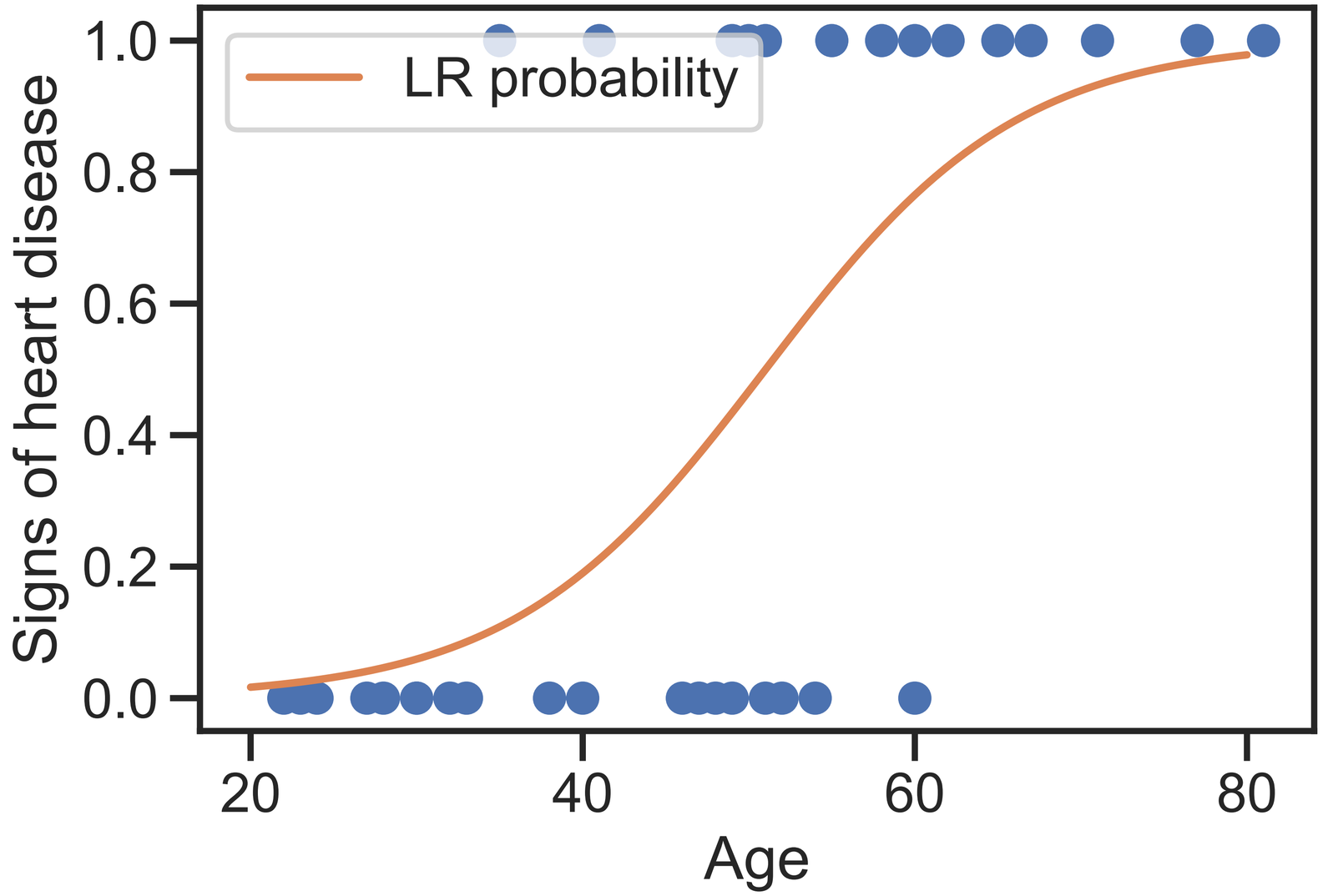
Inverse of Logit function = Logistic sigmoid

$$P(y = 1 \mid \mathbf{x}) = \pi(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x})}$$

Exercise: Verify the logistic sigmoid function from the logit function.



Example


LR probability
Signs of heart disease
1.0
0.8
0.6
0.4
0.2
0.0
20
40
60
80
Age

How do we estimate w?

What would like our model to do?

Correctly predict (or describe) our training data - maximise the likelihood of our output samples given our choice of weights

Assumption: conditional independence of the data

Likelihood of the data: $$L = P(\mathbf{y} \mid \mathbf{x}) = \prod_{i=1}^{N} P(y_i \mid \mathbf{x}_i)$$

Maximising the likelihood is the same as minimising the negative log likelihood (nll):

$$l = -\sum_{i=1}^{N} \log P(y_i \mid \mathbf{x}_i) = -\sum_{i=1}^{N} \left[y_i \log(\pi(\mathbf{x}_i)) + (1 - y_i) \log(1 - \pi(\mathbf{x}_i)) \right]$$

MLE: no closed form -> perform gradient descent on nll.

Iterative method to learn the weights

Minimise the negative log-likelihood w.r.t the weights $\mathbf{w}$:

$$l = -\sum_{i=1}^{N} \left[y_i \log(\pi(\mathbf{x}_i)) + (1 - y_i) \log(1 - \pi(\mathbf{x}_i)) \right]$$

Gradient descent $\quad \mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \frac{\partial l}{\partial \mathbf{w}}$

$$\frac{\partial l}{\partial \mathbf{w}} = -\sum_{i=1}^{N} \left[y_i \frac{\partial \log(\pi(\mathbf{x}_i))}{\partial \mathbf{w}} + (1 - y_i) \frac{\partial \log(1 - \pi(\mathbf{x}_i))}{\partial \mathbf{w}} \right]$$

$$= -\sum_{i=1}^{N} \left[\frac{y_i}{\pi(\mathbf{x}_i)} - \frac{1 - y_i}{1 - \pi(\mathbf{x}_i)} \right] \frac{\partial \pi(\mathbf{x}_i)}{\partial \mathbf{w}} \qquad = -\sum_{i=1}^{N} \left[y_i - \pi(\mathbf{x_i}) \right] \mathbf{x}_i$$

As long as $\pi(\cdot)$ is the logistic sigmoid.

Logistic regression ingredients

Input: Data + pre-processing

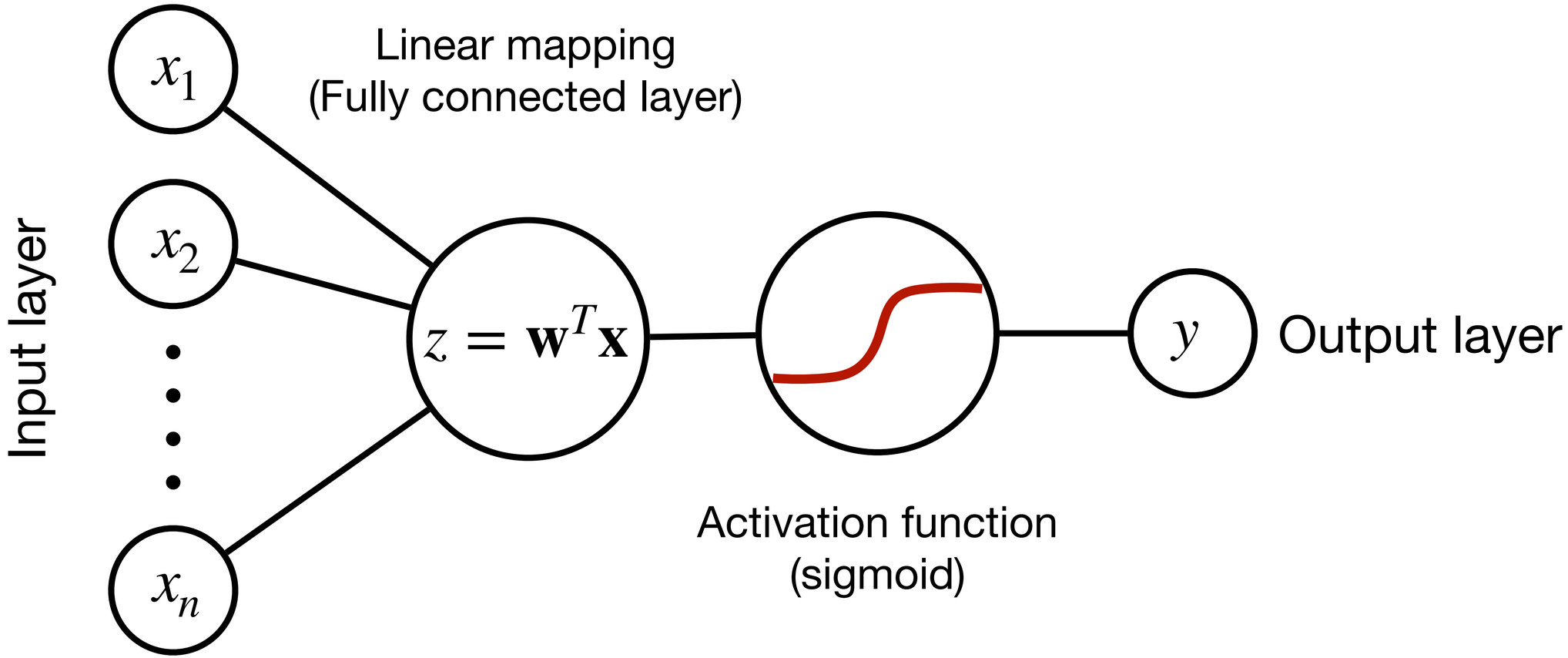
Model:

- **Structure/architecture**: linear with the log odds
- **Parameters:** Weights and biases
- **Hyper-parameters:** None (unless regularisation is applied)

Evaluation: max likelihood (min negative log likelihood)

Optimisation: Gradient descent, some Newton methods available

LR as a neural network - the perceptron



What about multiple classes?

A simple way: **one-vs-rest logistic regression**

- Run binary classification for all possible classes
- Pick the one with the highest output

More mathematical: **multinomial logistic regression**

- Generalise LR to multiple classes
  - Bernoulli distribution → categorical distribution
  - Sigmoid function → softmax function
- A linear classifier for multiple classes

Summary for logistic regression

Discriminative classifiers **directly model** the likelihood P(y | x)

A simple **linear classifier** that retains a **probabilistic** semantics

Parameters in LR are learned by **iterative optimisation** (e.g gradient descent), no closed form solutions like linear regression

Analogous to a simple neural network - a **perceptron**

Automatic Differentiation

Session outline

Logistic regression

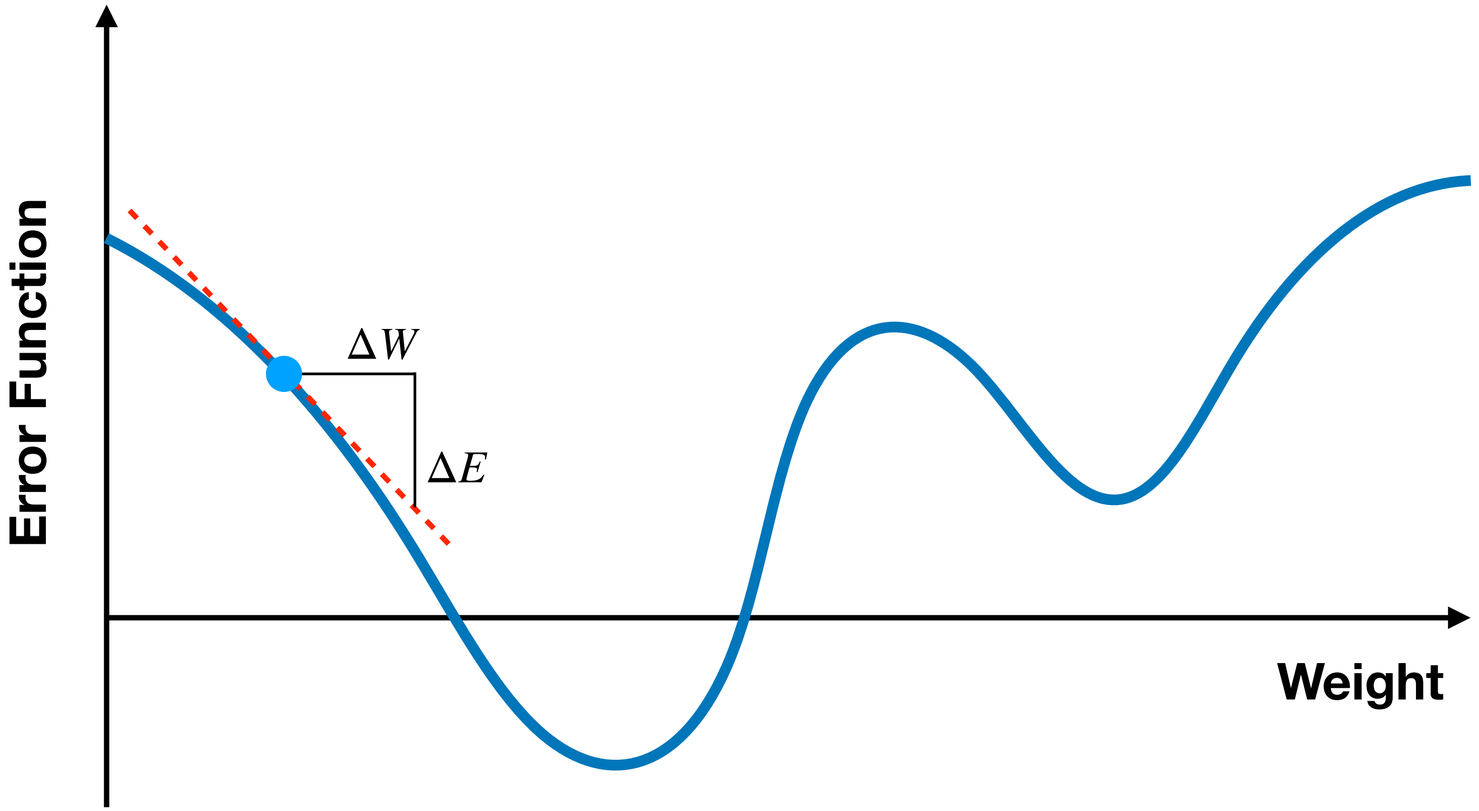
- Types of classifiers
- Bernoulli distribution
- Odds and Log Odds

Automatic differentiation

- Derivatives and ways to compute them
- Computational graph
- Forward and reverse mode AD

Error functions and derivatives

Usually our goal is to find the weights that will minimise the error or cost function.



<u>Gradient</u>
tells us how the error will change given a change in the weights

Derivatives are also necessary for computing Hessians (used for second-order optimisation methods).

How do we compute derivatives for a computer program?

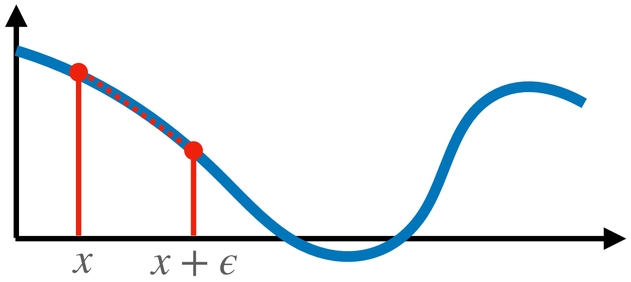
By hand, then code them

$$\frac{\partial}{\partial x}\left(x^2 y + y + 2\right) = 2xy$$

$$\frac{\partial}{\partial y}\left(x^2 y + y + 2\right) = x^2 + 1$$

```
dfdx_analytical  = 2*x_0*y_0
dfdy_analytical = x_0**2 + 1
```

Numerically with finite differences




```
dfdx_numerical = (f(x_0+epsilon, y_0) - f(x_0, y_0))/epsilon
dfdy_numerical = (f(x_0, y_0+epsilon) - f(x_0, y_0))/epsilon
```

Symbolic differentiation

WolframAlpha computational intelligence

differentiate x^2 y + y +2 wrt x

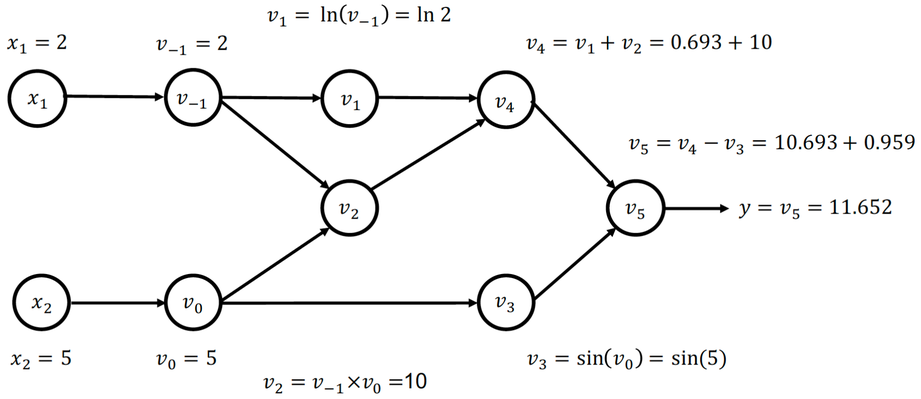
NATURAL LANGUAGE | MATH INPUT | EXTENDED KEYBOARD | EXAMPLES | UPLOAD | RANDOM

Derivative

Step-by-step solution

$$\frac{\partial}{\partial x}(x^2 y + y + 2) = 2xy$$

Automatic differentiation



Example

Suppose we have the following function

$$f(x, y) = x^2 y + y + 2$$

We need to compute the gradient of this function for gradient descent,

$$\frac{df}{d\mathbf{z}} = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \end{bmatrix}$$

Where $\mathbf{z} = [x, y]^T$

Manual Differentiation

We can use our knowledge of calculus to derive the proper equation.

We need to apply the following rules of calculus:

The derivative of a constant is 0

The derivative of ax w.r.t x is a (which is a constant)

The derivative of x^n is nx^{n-1}

The derivative is a linear operation, so the derivative of a sum is the sum of the derivatives.

The derivative of a constant times a function is equal to the constant times the derivative of that function.

Manual Differentiation

Partial derivative w.r.t x

$$\frac{\partial}{\partial x} f(x, y) = \frac{\partial}{\partial x} \left(x^2 y + y + 2 \right) = 2xy$$

Partial derivative w.r.t y:

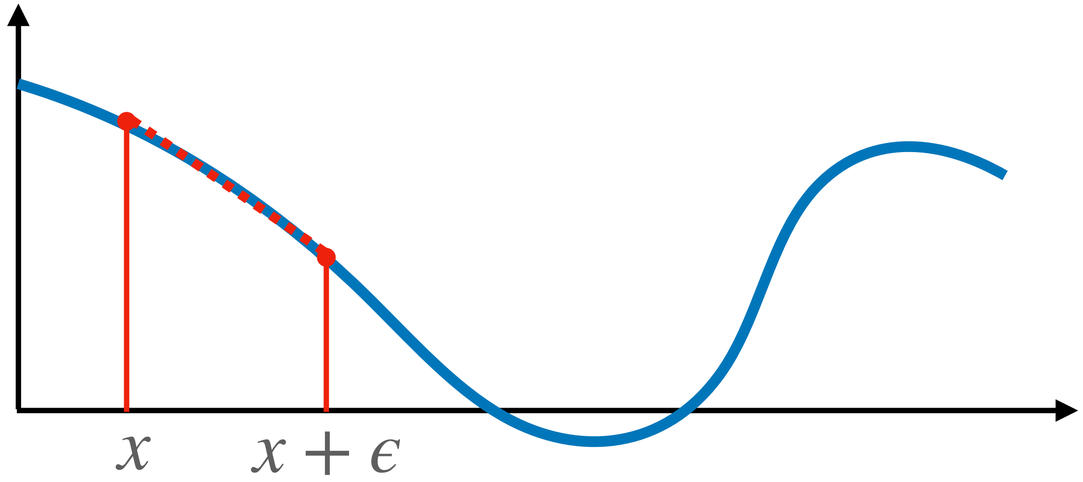
$$\frac{\partial}{\partial y} f(x, y) = \frac{\partial}{\partial y} \left(x^2 y + y + 2 \right) = x^2 + 1$$

Then we can write

$$\frac{df(x, y)}{d\mathbf{z}} = \begin{bmatrix} \frac{df(x, y)}{dx} & \frac{df(x, y)}{dy} \end{bmatrix} = \begin{bmatrix} 2xy & x^2 + 1 \end{bmatrix}$$

What are the drawbacks to this approach?

Finite difference approximations



Derivative of $f(x)$ at a point x_0 is

$$\frac{df(x_0)}{dx} = \lim_{\epsilon \to 0} \frac{f(x_0+\epsilon) - f(x)}{\epsilon}$$

Similarly the partial derivatives at (x_0, y_0) are:

$$\frac{\partial f(x_0, y_0)}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x_0+\epsilon, y_0) - f(x_0, y_0)}{\epsilon}$$

$$\frac{\partial f(x_0, y_0)}{\partial y} = \lim_{\epsilon \to 0} \frac{f(x_0, y_0+\epsilon) - f(x_0, y_0)}{\epsilon}$$

How many times do we need to call f()?

Finite difference approximations

```python
def f(x, y):
    return x**2*y + y + 2

x_0 = 3
y_0 = 2
epsilon = 1e-6
dfdx_numerical = (f(x_0+epsilon, y_0) - f(x_0, y_0))/epsilon
dfdy_numerical = (f(x_0, y_0+epsilon) - f(x_0, y_0))/epsilon

dfdx_analytical  = 2*x_0*y_0
dfdy_analytical = x_0**2 + 1
```

Script in python for the finite differences

```
In [22]: dfdx_numerical
Out[22]: 12.000002001855137

In [23]: dfdx_analytical
Out[23]: 12
```

```
In [24]: dfdy_numerical
Out[24]: 10.000000003174137

In [25]: dfdy_analytical
Out[25]: 10
```

Easy to implement
Can use it to test whether the manual implementation is correct.

Depends on ϵ
Results can be imprecise and is worse for more complicated functions.

Repeated calls to f()
Inefficient for large parametric models.

Symbolic Differentiation

Symbolic differentiation performs an **automatic manipulation** of expressions to obtain the corresponding derivative expressions.

The mathematical expression is represented using data structures (e.g trees, lists, etc).

It is then possible to follow a mechanistic process to obtain the derivatives.

Examples

Mathematica

Maxima

Maple

Example with Mathematica

```
In[34]:= D[4 x (1 - x), x]
Out[34]= 4 (1 - x) - 4 x

In[35]:= D[16 x (1 - x) ((1 - 2 x)^2), x]
Out[35]= 16 (1 - 2 x)^2 (1 - x) - 16 (1 - 2 x)^2 x - 64 (1 - 2 x) (1 - x) x

In[36]:= D[64 x (1 - x) ((1 - 2 x)^2) ((1 - 8 x + 8 x^2)^2), x]
Out[36]= 128 (1 - 2 x)^2 (1 - x) x (-8 + 16 x) (1 - 8 x + 8 x^2) +
    64 (1 - 2 x)^2 (1 - x) (1 - 8 x + 8 x^2)^2 - 64 (1 - 2 x)^2 x (1 - 8 x + 8 x^2)^2 -
    256 (1 - 2 x) (1 - x) x (1 - 8 x + 8 x^2)^2
```

Due to the mechanistic approach, there is usually a lot of redundancy in the generated expressions.

If not handled properly, it produces unnecessarily long expressions difficult to make sense of and to evaluate. This is known as **expression swell.**

Automatic differentiation (AD)

AD is concerned about exact numeric computation rather than their actual symbolic form.

It computes the derivative by only storing the values of intermediate sub-expressions.

It uses a combination of:

- **Symbolic differentiation** at the elementary operation level
- Storing **intermediate** numerical results

Derivatives with many inputs and outputs

Say that we have several functions that depend on several input variables:

$$
\begin{aligned}
y_1 &= f_1(x_1, \ldots, x_n) \\
y_2 &= f_2(x_1, \ldots, x_n) \\
&\vdots \\
y_m &= f_m(x_1, \ldots, x_n)
\end{aligned}
$$

Then the Jacobian, **J**, is a m by n matrix with entries:

$$
J_{ij} = \frac{\partial f_i}{\partial x_j}
$$

$$
\mathbf{J} = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\
\vdots & \vdots & \vdots & \vdots \\
\frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n}
\end{bmatrix}
$$

Also, the Hessian is the derivative of the Jacobian.

Evaluation trace

Evaluation trace: composition of elementary operations that lead to a full expression.

General notation:

Let $\mathbf{y} = \mathbf{f}(x_1, \ldots, x_n)$ with n inputs and m outputs ($\mathbb{R}^n \rightarrow \mathbb{R}^m$)

- Input variables: $v_{i-n} = x_i$ with $i = 1, \ldots, n$
- Intermediate variables: v_i with $i = 1, \ldots, l$
- Output variables: $y_i = v_{l+i}$ with $i = 1, \ldots, m$

Computational graph: network showing how all these variables are connected.

Example

Let's think about logistic regression with 1 input and 1 weight:

$$y = \text{Sigmoid}(w_1 x_1).$$

Inputs

$$v_{-1} = x_1$$

$$v_0 = w_1$$

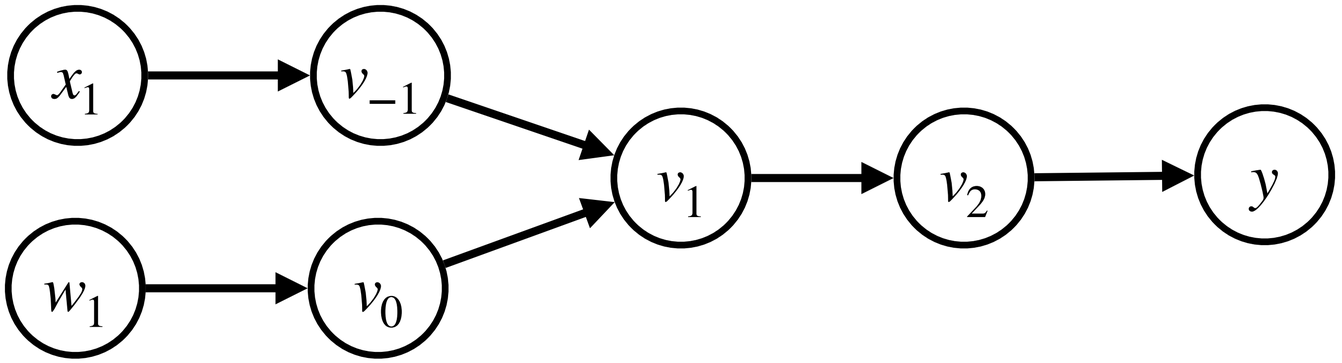
Intermediates

$$v_1 = v_{-1} v_0$$

$$v_2 = \text{Sigmoid}(v_1)$$

Outputs

$$y = v_2$$



Exercise

Consider the following example

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2).$$

The inputs are x_1 and x_2. What are the elementary operations?

$$v_{-1} = x_1$$

$$v_0 = x_2$$

$$v_1 = \ln(v_{-1})$$

$$v_2 = v_{-1} v_0$$

$$v_3 = \sin(v_0)$$

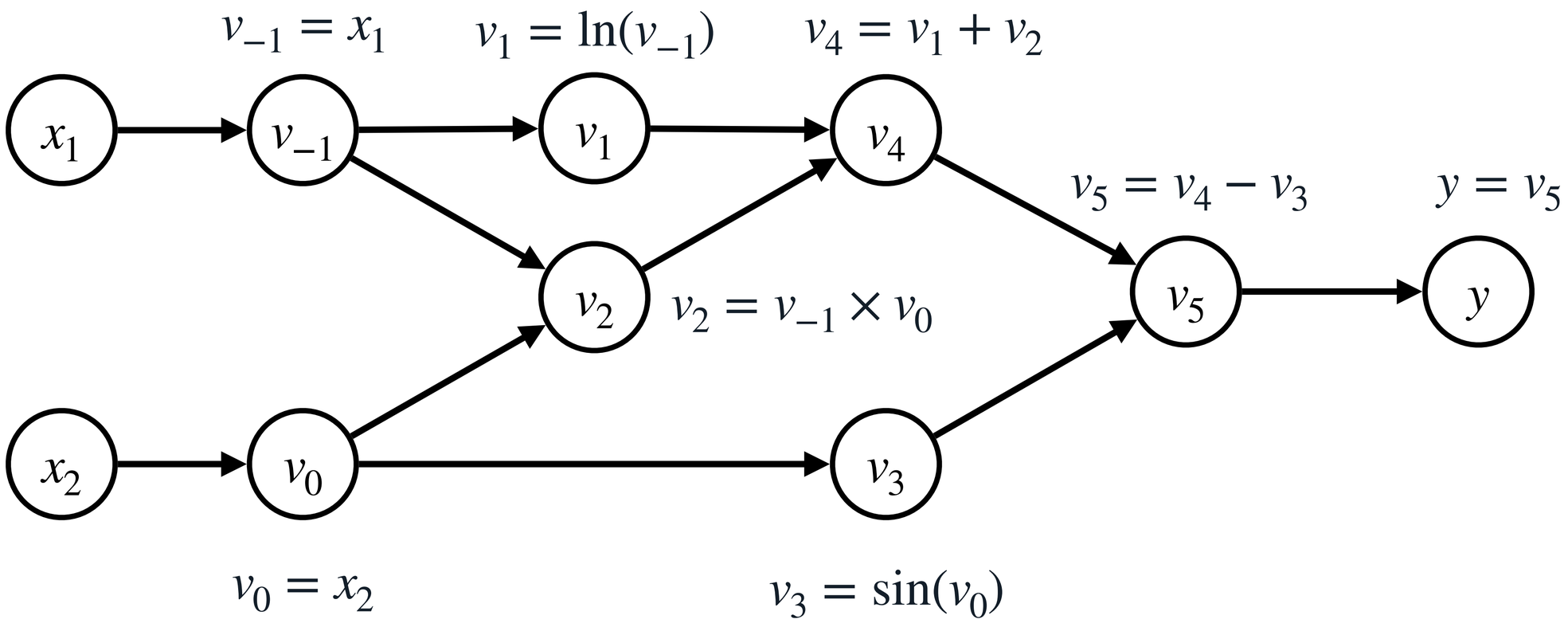
$$v_4 = v_1 + v_2$$

$$v_5 = v_4 - v_3$$

$$y = v_5$$

Let's draw a **computational graph** to show how they interact.

Computational graph



Forward accumulation mode

Also called tangent linear mode.

To compute the derivative of f w.r.t x_k, each intermediate variable v_i has a derivative

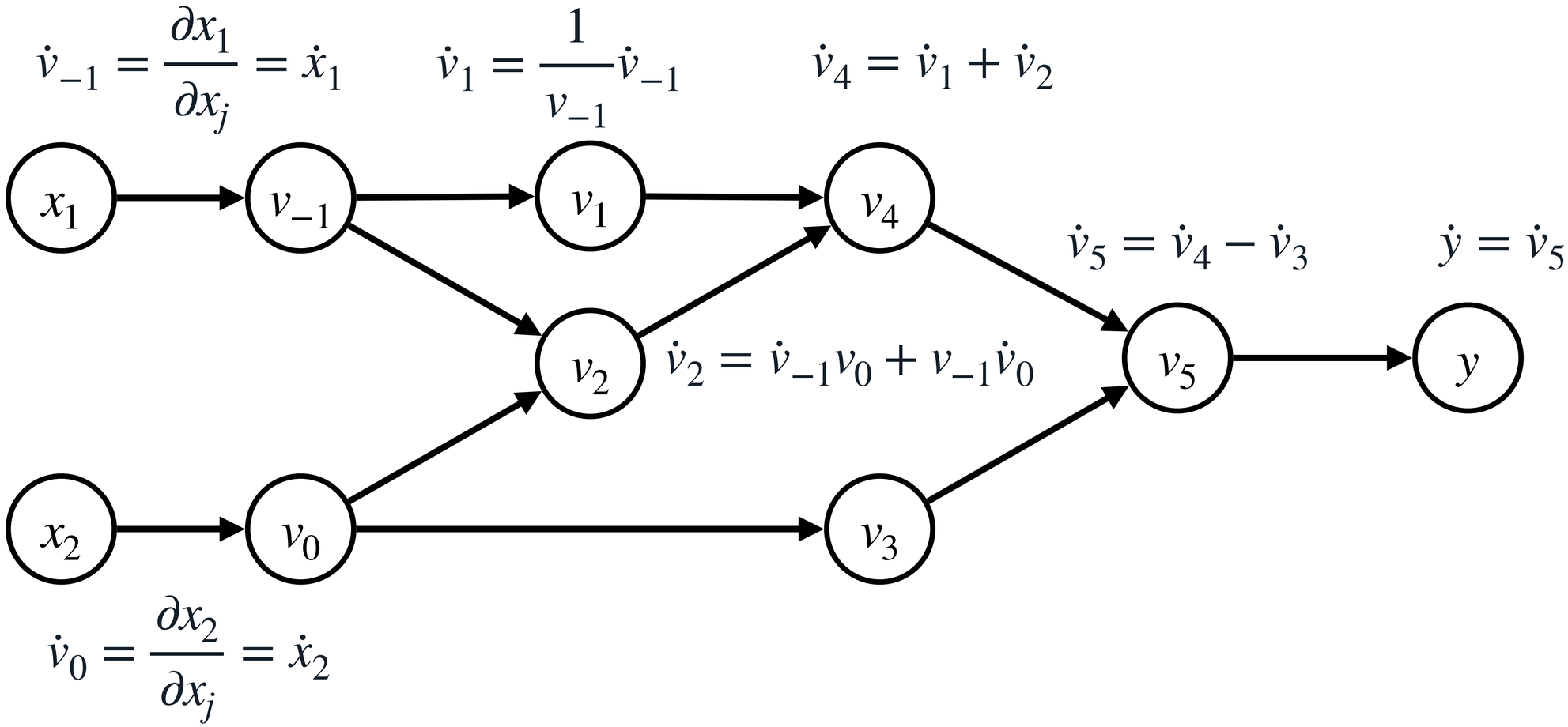
$$\dot{v}_i = \frac{\partial v_i}{\partial x_j}$$

For each evaluation (or forward primal) trace, it builds a forward derivative (or tangent) trace.

Essentially, this is just implementing the chain rule:

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

Forward tangent trace



Example

Compute the derivative $\partial y/\partial x_1$ at $x_1 = 2, x_2 = 5$.

Forward primal trace	Values
$v_{-1} = x_1$	$= 2$
$v_0 = x_2$	$= 5$
$v_1 = \ln(v_{-1})$	$= \ln(2)$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5$
$v_3 = \sin(v_0)$	$= \sin(5)$
$v_4 = v_1 + v_2$	$= 0.693 + 10$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$
$y = v_5$	$= 11.652$

Forward tangent trace	Values
$\dot{v}_{-1} = \dot{x}_1$	$= 1$
$\dot{v}_0 = \dot{x}_2$	$= 0$
$\dot{v}_1 = \frac{1}{v_{-1}}\dot{v}_{-1}$	$= \frac{1}{2}1$
$\dot{v}_2 = \dot{v}_{-1} \times v_0 + v_{-1} \times \dot{v}_0$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3 = \cos(v_0)\,\dot{v}_0$	$= \cos(5) \times 0$
$\dot{v}_4 = \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5 = \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y} = \dot{v}_5$	$= 5.5$

If we want to compute for $\partial y/\partial x_2$ then swap initial tangent trace values.

Generalisation to the Jacobian of a function

Remember $\mathbf{f}(x_1, \ldots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a function with n independent variables and m dependent variables.

The derivatives of the Jacobian, $\partial y_i / \partial x_j$, are computed by making $\dot{x}_j = 1$ initially and all the other derivatives $\dot{x}_k = 0$ for $k \neq i$.

The values of all the derivatives $\dot{y}_i = \left. \frac{\partial y_i}{\partial x_j} \right|_{x=a}$ are obtained by a forward pass.

Note that for a specific x_j we can compute all the derivatives for all outputs I, which corresponds to the column in the Jacobian.

To compute the whole Jacobian, we need n forward passes, one per input variable.

Complexity

AD forward mode is efficient for functions like $\mathbf{f} : \mathbb{R} \rightarrow \mathbb{R}^m$.

As be saw before, this is because we can compute all the output derivatives for a single input in one pass.

In the other extreme $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}$, it would need n forward passes which becomes computationally expensive when n is large.

In general, when $n \gg m$ the reverse mode of AD is preferred.

AD reverse mode - backpropagation

AD in reverse mode propagates the derivatives backwards from a given output.

It is done by computing the **adjoints** of the intermediate variables

$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$ representing the sensitivity of the output y_j to variable v_i.

AD in reverse mode uses two-phases:

- A **forward step** to compute the variables v_i and to **store** dependencies in the computational graph.
- A backward or **reverse step**, in which the **adjoints** are used to compute the derivatives, starting from the outputs and going back to the inputs.

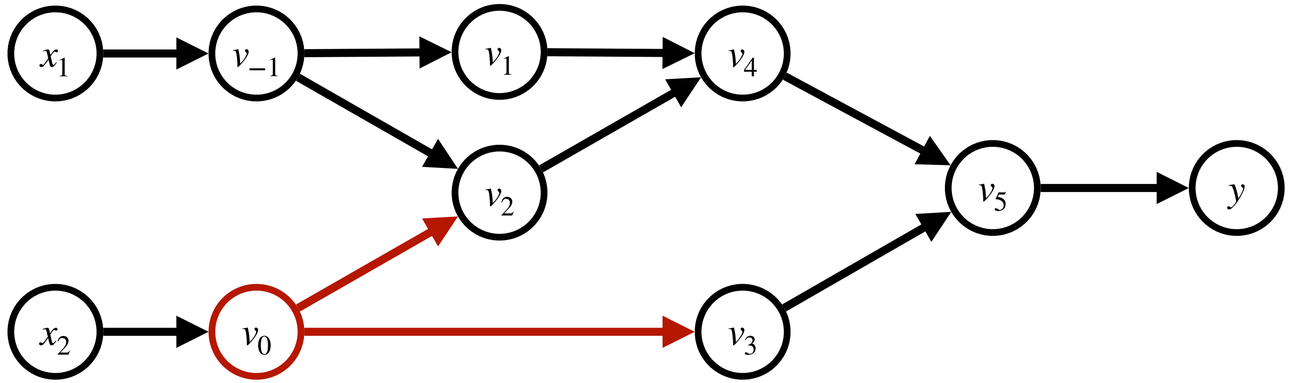
Example

Let's go back to the same example

$$f(x_1, x_2) = y = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

and focus on finding the derivative for x_2.

We need to compute the adjoint $\bar{v}_0 = \partial y / \partial v_0$, this is how a change in v_0 affects the output y. From the graph, we can see that v_0 affects y through v_2 and v_3:



Example

So the contributions of v_0 to y is given as

$$\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0}.$$

By definition $\frac{\partial y}{\partial v_2} = \bar{v}_2$ and $\frac{\partial y}{\partial v_3} = \bar{v}_3$ so we can simplify this to

$$\frac{\partial y}{\partial v_0} = \bar{v}_2\frac{\partial v_2}{\partial v_0} + \bar{v}_3\frac{\partial v_3}{\partial v_0}.$$

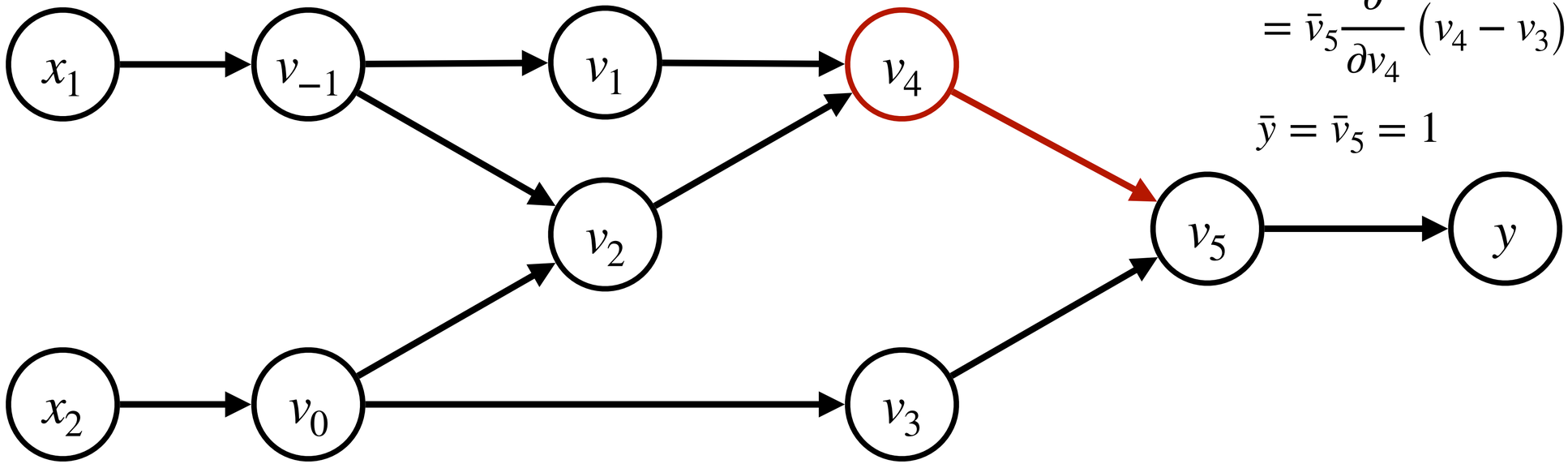
After the forward pass to compute v_i, the reverse pass computes the adjoints.
Starting with $\bar{y} = \bar{v}_5 = 1$ and computing the derivatives w.r.t the inputs at the end.

Reverse adjoint trace

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_5\frac{\partial v_5}{\partial v_4}$$

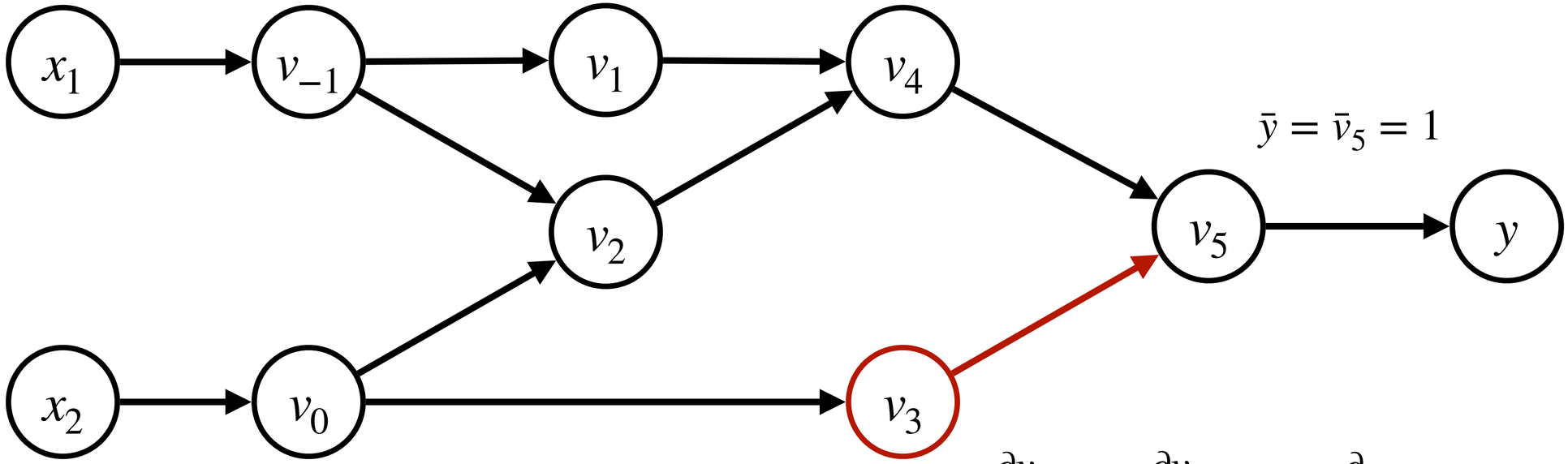
$$= \bar{v}_5\frac{\partial}{\partial v_4}\left(v_4 - v_3\right) = \bar{v}_5$$

$$\bar{y} = \bar{v}_5 = 1$$



Reverse adjoint trace

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \bar{v}_5$$

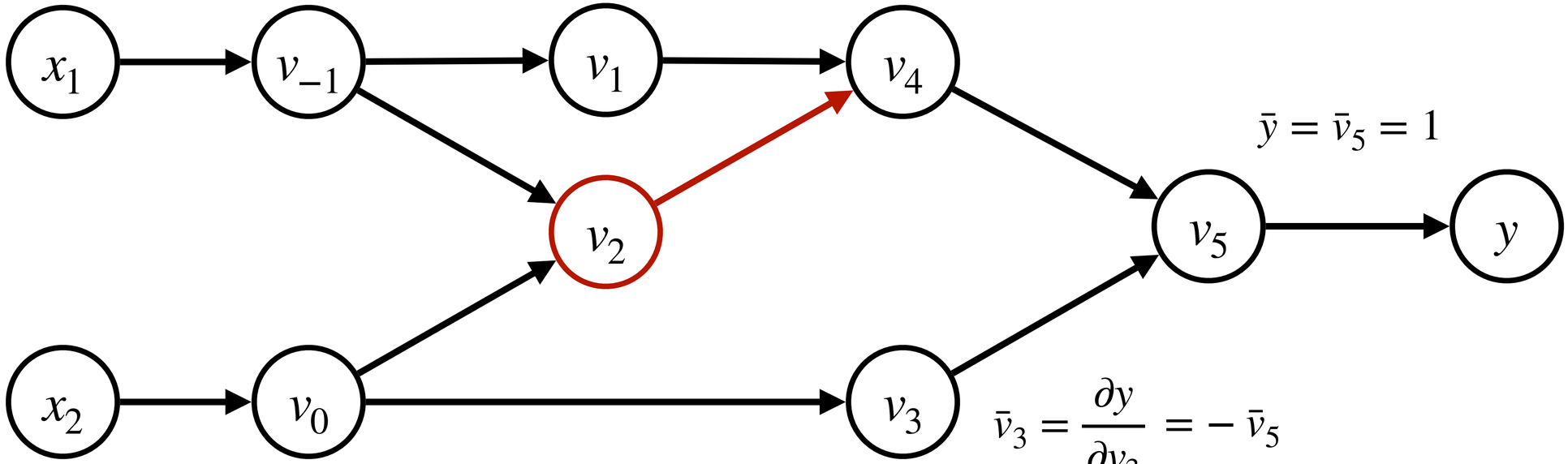


$\bar{y} = \bar{v}_5 = 1$

$$\bar{v}_3 = \frac{\partial y}{\partial v_3} = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \bar{v}_5 \frac{\partial}{\partial v_3}\left(v_4 - v_3\right) = -\bar{v}_5$$

Reverse adjoint trace

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \bar{v}_5$$

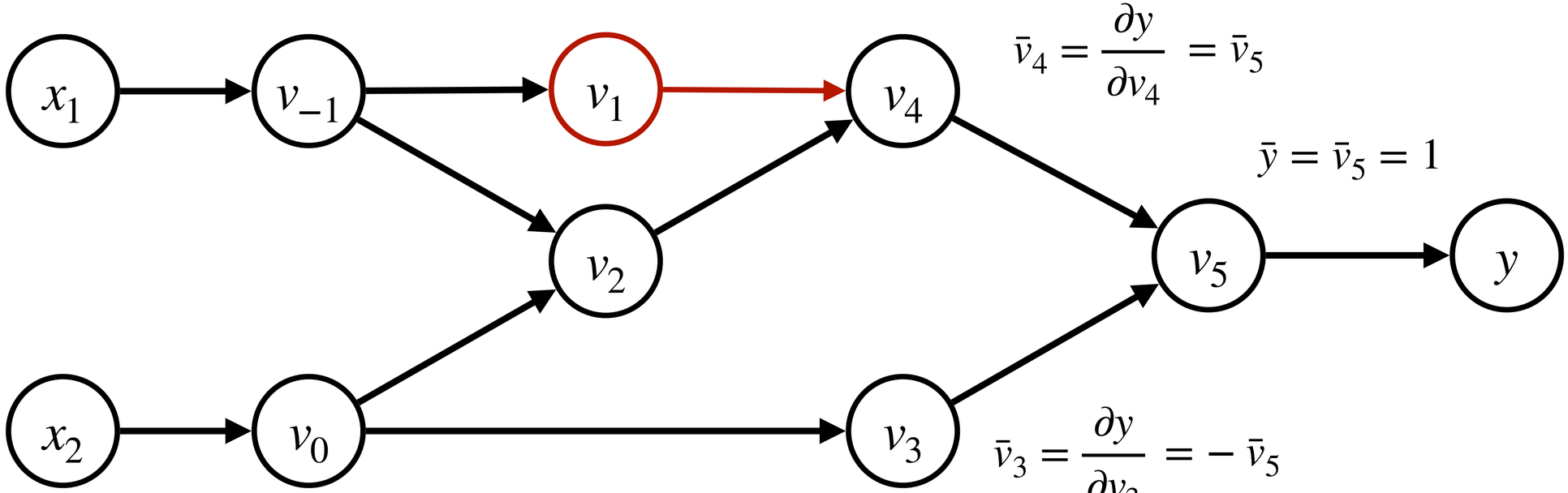


$$\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$$

$$= \bar{v}_4 \frac{\partial}{\partial v_2}\left(v_1 + v_2\right) = \bar{v}_4$$

Reverse adjoint trace

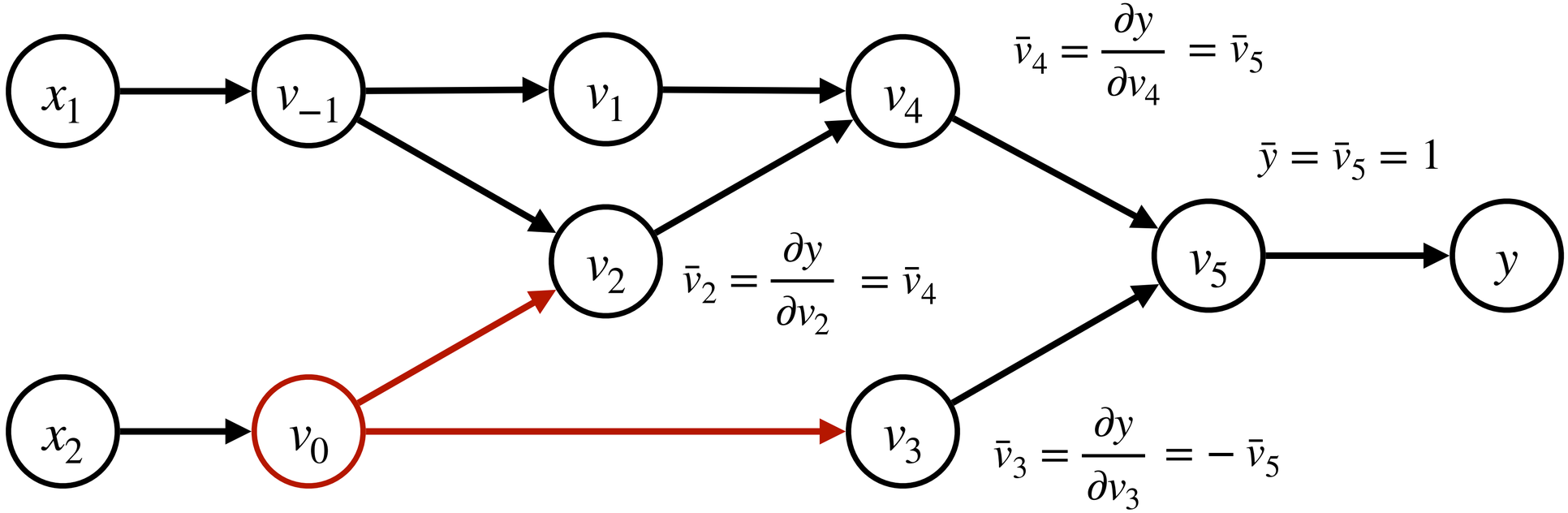
$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \frac{\partial}{\partial v_1}\left(v_1 + v_2\right) = \bar{v}_4$$



$$\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_4$$

Reverse adjoint trace

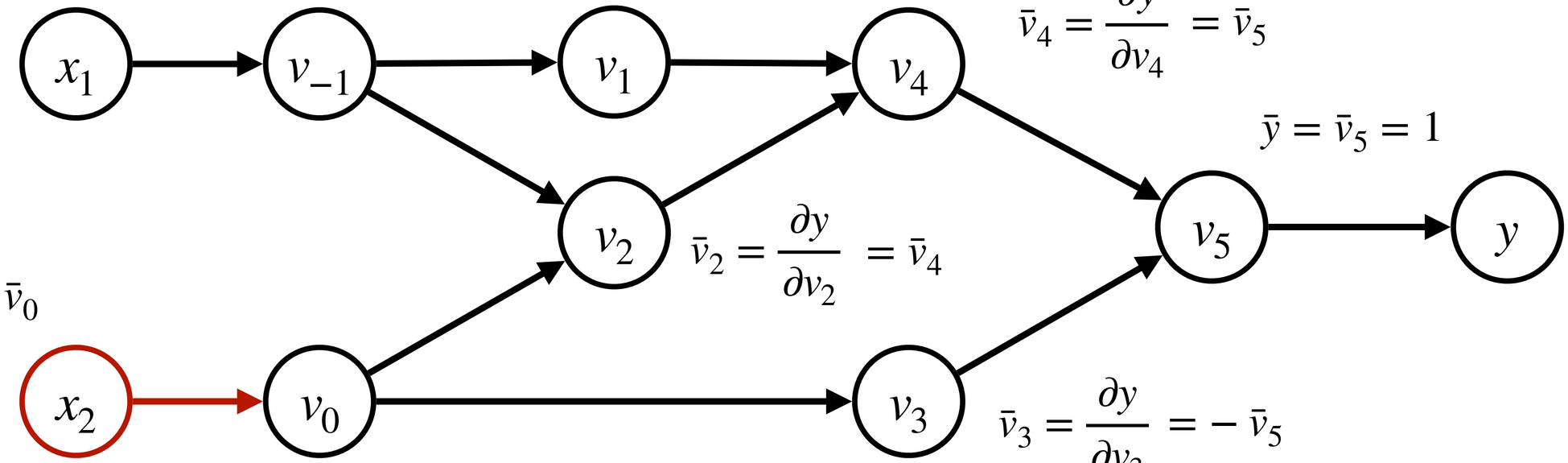
$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_4$$



$$\bar{v}_0 = \frac{\partial y}{\partial v_0} = \bar{v}_2 \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \frac{\partial v_3}{\partial v_0} = \bar{v}_2 \frac{\partial}{\partial v_0}\left(v_{-1} v_0\right) + \bar{v}_3 \frac{\partial}{\partial v_0}\left(\sin(v_0)\right) = \bar{v}_2 v_{-1} + \bar{v}_3 \cos(v_0)$$

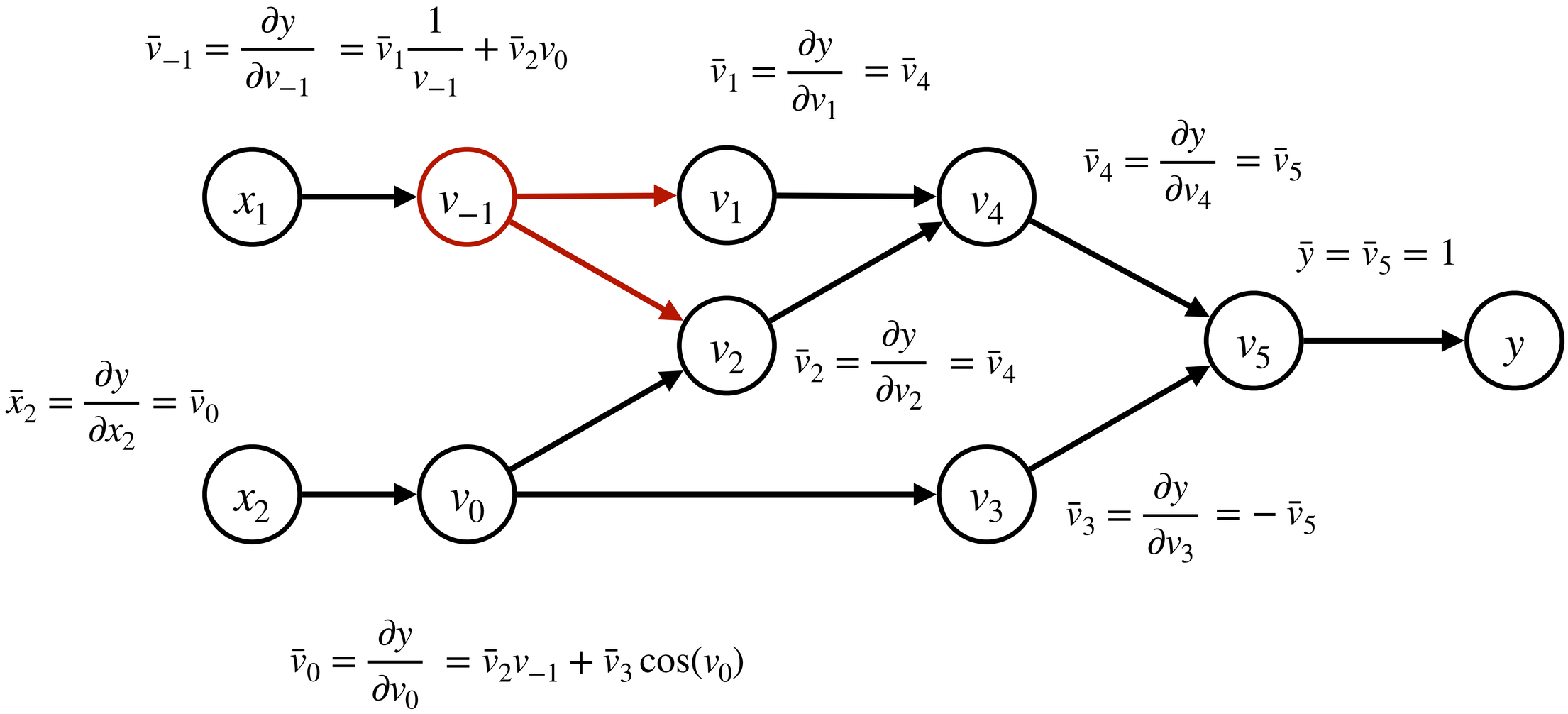
Reverse adjoint trace

$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_4$$



$$\bar{v}_0 = \frac{\partial y}{\partial v_0} = \bar{v}_2 v_{-1} + \bar{v}_3 \cos(v_0)$$

Reverse adjoint trace

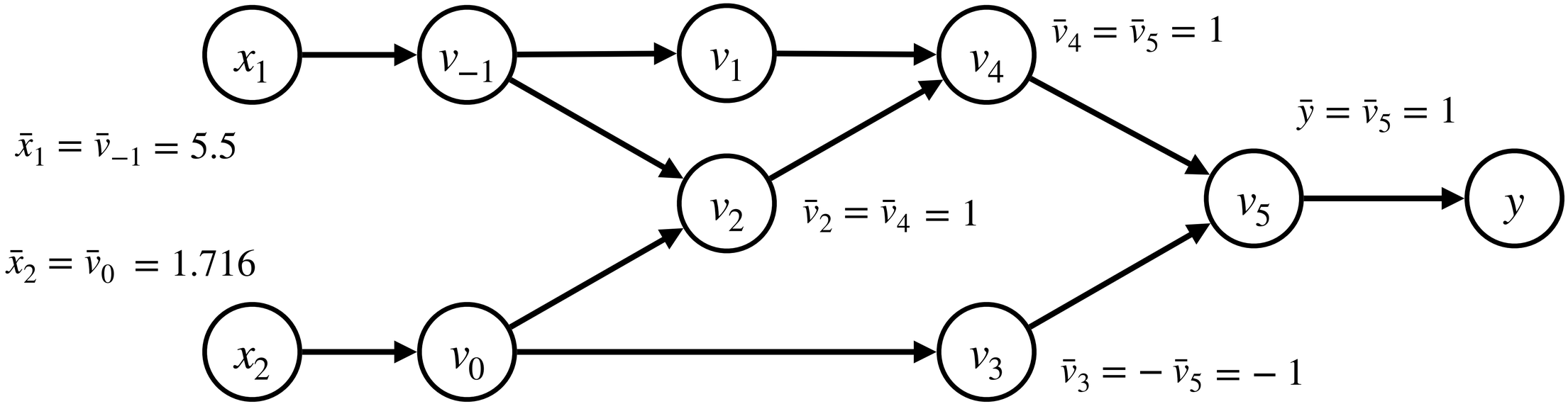


Numerical evaluation of the reverse adjoint trace

$$\bar{v}_{-1} = \bar{v}_1 \frac{1}{v_{-1}} + \bar{v}_2 v_0$$

$$= 1 \times \frac{1}{2} + 1 \times 5 = 5.5$$

$\bar{v}_1 = \bar{v}_4 = 1$



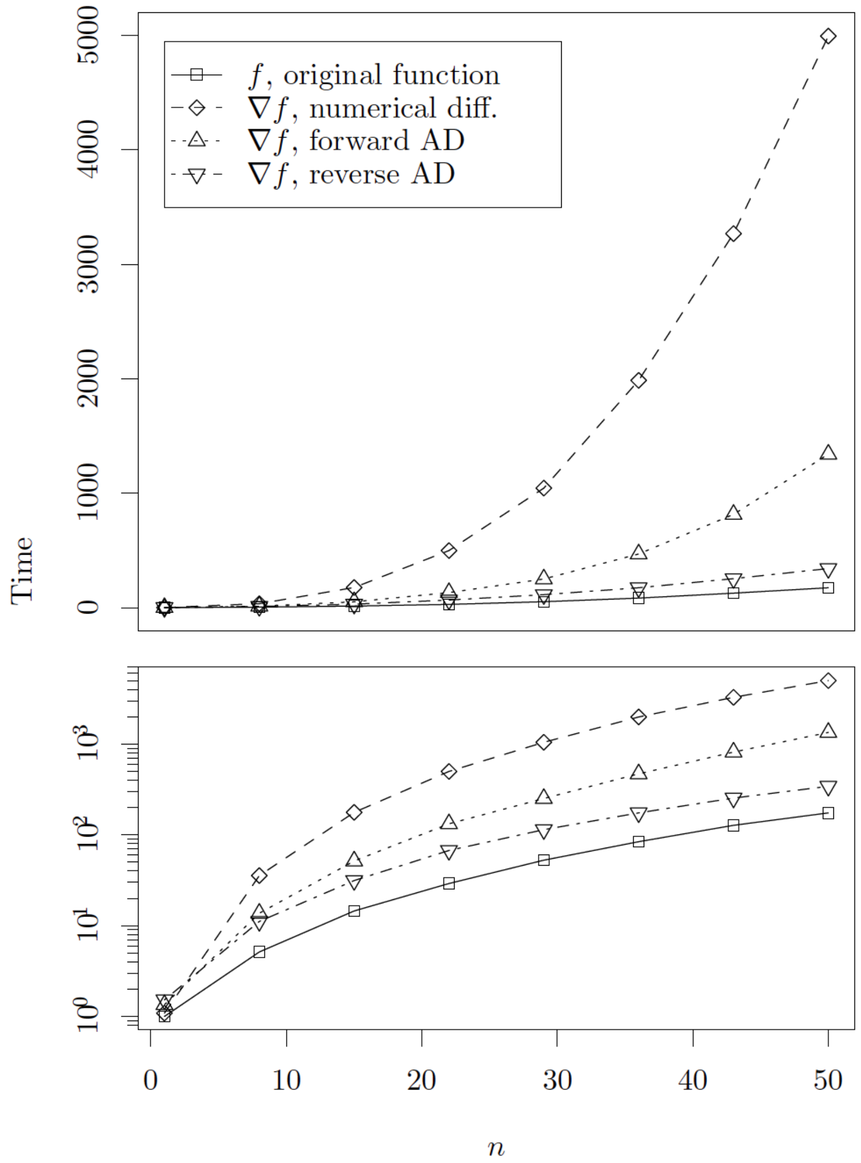
$$\bar{v}_0 = \bar{v}_2 v_{-1} + \bar{v}_3 \cos(v_0) \quad = 1 \times 2 - 1 \times \cos(5) = 1.716$$

Complexity

Reverse mode AD performs better when $n \gg m$.

As we see on the right, reverse mode is generally faster than other types.

The downside is the cost of increased storage, since we need to save the intermediate values for the adjoint evaluation trace.



From Automatic differentiation in machine learning: A survey (see readings)

Reverse mode AD and back propagation

Reverse mode AD is the algorithm used to train neural networks and deep learning models.

To train a neural network model, we optimise an objective function, $E(\mathbf{w})$ that usually depends on a high-dimensional input vector of parameters $\mathbf{w} \in \mathbb{R}^n$ (i.e many inputs but only 1 output (the error).

In the ML community, reverse mode AD goes by the name **backpropagation**, which you will see next week when we look at neural networks.

Common implementations include PyTorch, TensorFlow but many versions in different languages.

Take home messages

A complex function can be **decomposed into intermediate variables** where the derivatives are simple to perform.

Forward mode **accumulates the derivative** for a given input variable, **requires n passes**.

Reverse mode **stores the intermediate values** and **evaluates the adjoints** on a reverse trace. Better when many inputs, fewer outputs.

Further reading

Logistic regression:

Sections 4.1, 4.2, 4.3 (pages 139 to 150) and 5.2.2 (pages 179 to 182) from **A First Course in Machine Learning** by Rogers and Girolami

Automatic Differentiation:

Automatic differentiation in machine learning: A survey by Baydin et al.

https://arxiv.org/abs/1502.05767