COMP3242/6242: Deep Learning

Week 2 — Linear Classifiers and Multilayer Perceptrons

Stephen Gould
stephen.gould@anu.edu.au

Australian National University

Semester 1, 2026

Overview

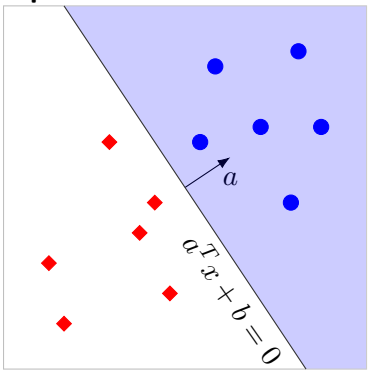
- binary classification
- logistic regression
- the XOR problem
- multilayer perceptron
- activation functions
- the universal approximation theorem
- multi-class classification
- loss functions
- gradient descent optimization
- calculating gradients

Binary (linear) Classification

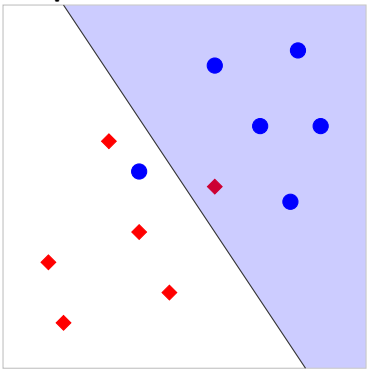
- **data**: training examples $\mathcal{D} = \left\{ \left(x^{(i)}, y^{(i)} \right) \right\}_{i=1}^{N}$ composed of
  - features $x^{(i)} \in \mathbb{R}^n$
  - target labels $y^{(i)} \in \{0, 1\}$
- **regression task**: learn a density function $P(y \mid x)$ from the training examples, which estimates the probability of 0 or 1 on an input x
- **classifier task**: learn a linear classifier $f(x) = a^T x + b$ from the training examples, which predicts 1 on an input x if $a^T x + b \geq 0$ and 0 otherwise

2D Illustration

separable case:



inseparable case:



- $a^T x + b$ measures (signed) distance from the hyperplane

Logistic Regression

Let $y \in \{0, 1\}$ be a random variable with distribution

$$P(y = 1 \mid x) = \frac{\exp(a^T x + b)}{1 + \exp(a^T x + b)}$$

where a, b are parameters and $x \in \mathbb{R}^n$ is observed features.

The **maximum likelihood principle** states that we should choose parameters a, b that maximizes the probability of the observed data under the model,

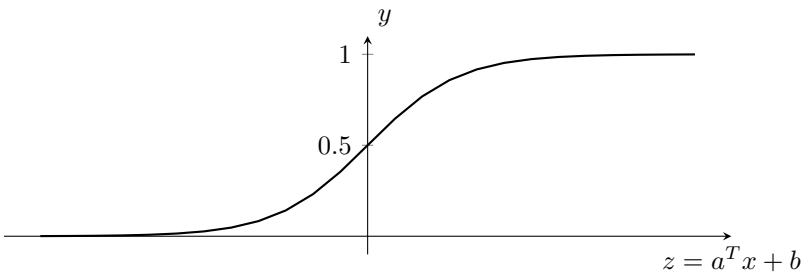
$$P(\mathcal{D}) = \prod_{i=1}^{N} P\left(y^{(i)} \mid x^{(i)}\right)$$

In practice we (equivalently) minimize the negative log-likelihood function instead,

$$\begin{aligned}
\ell(a, b; \mathcal{D}) &= -\log\left(\prod_{i:y^{(i)}=1} \frac{\exp(a^T x^{(i)} + b)}{1 + \exp(a^T x^{(i)} + b)} \prod_{i:y^{(i)}=0} \frac{1}{1 + \exp(a^T x^{(i)} + b)}\right) \\
&= -\sum (a^T x^{(i)} + b) + \sum^{N} \log\left(1 + \exp(a^T x^{(i)} + b)\right)
\end{aligned}$$

Logistic Function

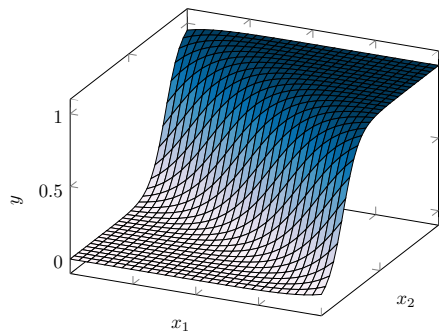
$$y = \sigma(z) = \frac{\exp(z)}{\exp(z)+1} = \frac{1}{1+\exp(-z)}$$



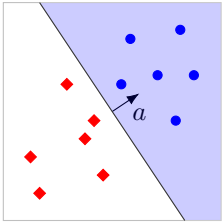
- observe that $\sigma : \mathbb{R} \to [0,1]$ and $\sigma(z) \geq \frac{1}{2} \iff z = a^T x + b \geq 0$

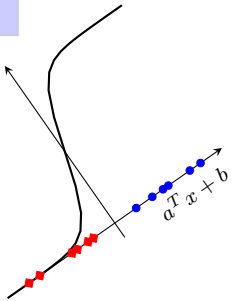
Logistic Function (2D Feature Space)

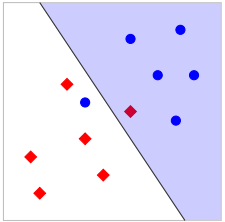
$$y = \frac{1}{1 + \exp(-a^T x - b)}$$

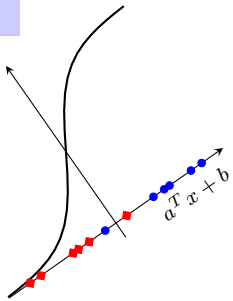


2D Illustration Revisited









Regularization

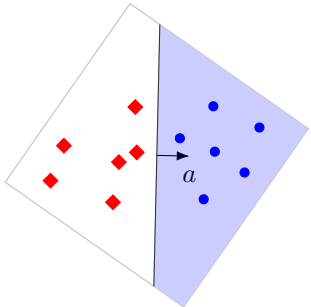
- we often don't want our classifiers to be too confident in the margin between positive and negative samples
- regularization is one way to reduce confidence in that region and generalize better to unseen data
- add a penalty on the magnitude of parameters

$$\text{minimize}_{a,b} \quad \ell(a, b; \mathcal{D}) + \frac{\lambda}{2}\|a\|^2$$

- also called weight decay

$$\nabla_a \left(\ell(a, b; \mathcal{D}) + \frac{\lambda}{2}\|a\|^2 \right) = \nabla_a \ell + \lambda a$$

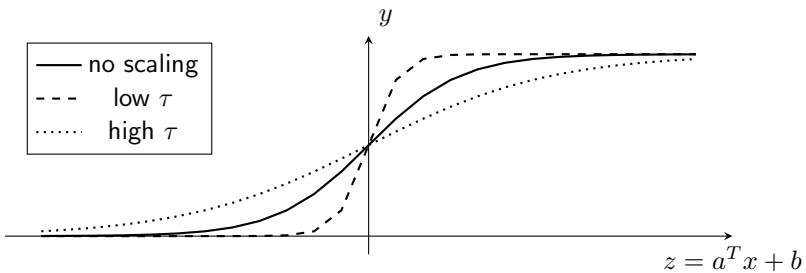
- we'll see other techniques for preventing over fitting later, e.g., data augmentation



Temperature Scaling

Introduce temperature parameter $\tau > 0$,

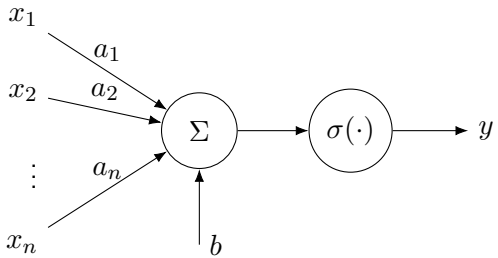
$$y = \sigma\left(\frac{z}{\tau}\right) = \frac{1}{1 + \exp\left(-z/\tau\right)}$$



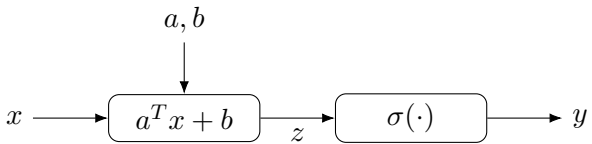
- high τ has similar affect to regularization (restricted to logistic and not trained)
- can be applied after training to sharpen/flatten distribution

Logistic Regression as a Single Neuron

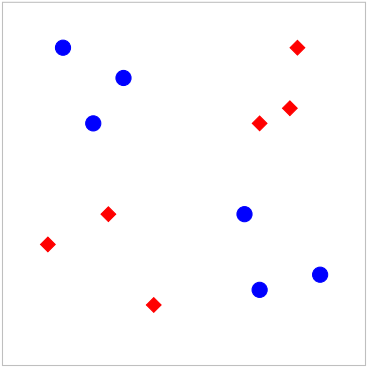
classic view, $y = \sigma\left(\sum_{i=1}^{n} a_i x_i + b\right)$



more modern view, $y = \sigma(a^T x + b)$

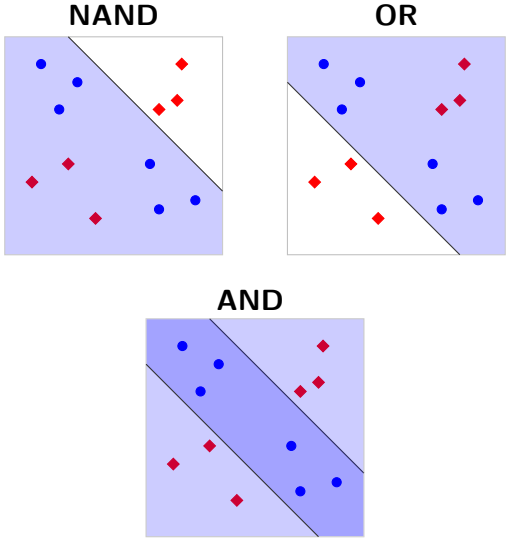


XOR Problem



- cannot be separated by a (single) linear decision boundary
- but can be separated by an additional layer of processing

XOR Problem: Two Layers



(the first layer defines a new features space; here we are visualizing in the original feature space)

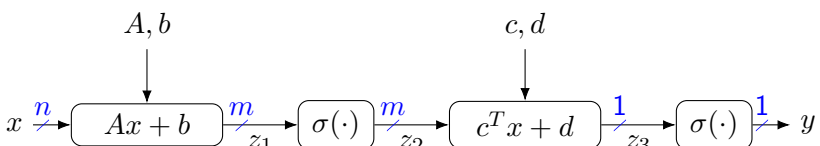
Multilayer Perceptron

(Rosenblatt 1961, Widrow & Hoff 1960)

Compose multiple logistic layers together. E.g., for two-layers,

$$y = \sigma(c^T \sigma(Ax + b) + d)$$

where σ is applied elementwise. Here $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, and $d \in \mathbb{R}$.



Activation Functions

- logistic function (sigmoid)

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

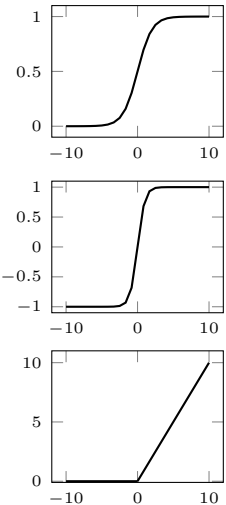
- hyperbolic tangent

$$\sigma(z) = \tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

- rectified linear unit (ReLU)

$$\sigma(z) = \max\{0, z\}$$

- and many others (e.g., Leaky ReLU, GLU, etc.)



Universal Approximation Theorem

(Cybenko 1989, Hornik 1991)

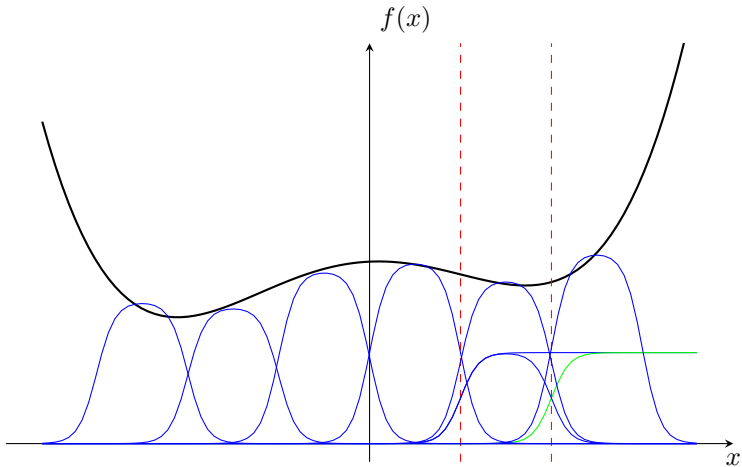
Theorem (roughly):

- let $\sigma : \mathbb{R} \to \mathbb{R}$ be a (well-behaved) activation function
- let $f : \mathbb{R}^n \to \mathbb{R}$ be any continuous function
- then there exists parameters $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, and $d \in \mathbb{R}$ such that the function

$$\hat{f}(x) = c^T \sigma(Ax + b) + d$$

  approximates $f(x)$ everywhere
- that is, $|\hat{f}(x) - f(x)| \leq \epsilon$ for all x

Proof Sketch (1D)

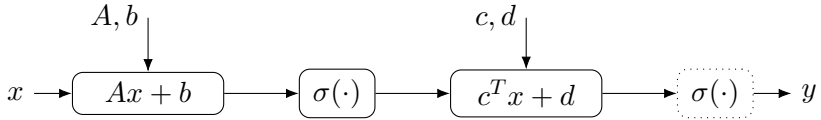


Why Deep Learning?

- so we can approximate any function with a two-layer network
- **problem 1:** doesn't tell us how many parameters
  - still active area of research, but indication is that we need $O(e^{1/\epsilon})$, in general
- **problem 2:** doesn't tell us how to find the parameters efficiently
- we can reduce the number of parameters by going deeper
- learning turns out to be "easier" in deeper networks
  - currently not well understood
- in practice we choose activation functions that violate the conditions of the theorem, e.g., ReLU

Advanced Topic: Identifiablity

- consider the two-layer perceptron (w/ or w/o second activation function),



- let $P \in \mathbb{R}^{m \times m}$ be a permutation matrix, then

$$
\begin{aligned}
y &= c^T \sigma(Ax + b) + d \\
&= c^T \sigma(P^{-1} P A x + P^{-1} P b) + d && \text{(since } P^{-1} P = I\text{)} \\
&= c^T P^{-1} \sigma(PAx + Pb) + d && \text{(since } \sigma \text{ is applied elementwise)} \\
&= \tilde{c}^T \sigma(\tilde{A} x + \tilde{b}) + d
\end{aligned}
$$

- so neural network parameters are never unique

Multi-class Classification

- **task**: K-way classification (e.g., ImageNet)
- **data**: training examples $\mathcal{D} = \left\{ \left(x^{(i)}, y^{(i)} \right) \right\}_{i=1}^{N}$ composed of
  - features $x^{(i)} \in \mathbb{R}^n$
  - target labels $y^{(i)} \in \{1, \ldots, K\}$
- we can extend the logistic function to the multi-class logistic

$$P(y = k \mid x) = \frac{\exp(a_k^T x + b_k)}{\sum_{j=1}^{K} \exp(a_j^T x + b_j)} \qquad \text{for } k = 1, \ldots, K$$

  parametrized by $\{(a_k, b_k) \mid k = 1, \ldots, K\}$
- the function that takes vector $z = (z_1, \ldots, z_K)$ and computes $\left(\frac{\exp z_1}{Z}, \ldots, \frac{\exp z_K}{Z} \right)$ with $Z = \sum_{k=1}^{K} \exp z_k$ is called **softmax**; the z_k are called **logits**
- can be computed in a numerically stable way (how?)

Multi-label Classification

- **task**: set of K binary classification problems (e.g., attribute classification)
- **data**: training examples $\mathcal{D} = \left\{ \left(x^{(i)}, y^{(i)} \right) \right\}_{i=1}^{N}$ composed of
  - features $x^{(i)} \in \mathbb{R}^n$
  - target labels $y^{(i)} \in \{0, 1\}^K$
- model with a set of K sigmoid functions

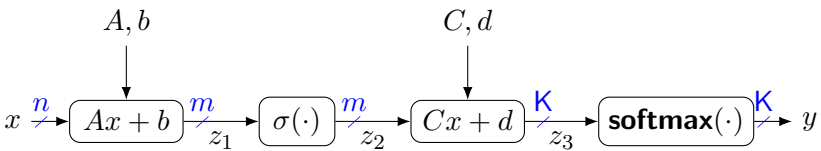
$$P(y_k = 1 \mid x) = \frac{\exp(a_k^T x + b_k)}{1 + \exp(a_k^T x + b_k)} \qquad \text{for } k = 1, \ldots, K$$

parametrized by $\{(a_k, b_k) \mid k = 1, \ldots, K\}$

As Multilayer Perceptrons

with parameters $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $C \in \mathbb{R}^{K \times m}$, and $d \in \mathbb{R}^K$,

$$y^{\text{multi-class}} = \textbf{softmax}(C\sigma(Ax + b) + d)$$
$$y^{\text{multi-label}} = \textbf{sigmoid}(C\sigma(Ax + b) + d)$$



Common Loss Functions

- loss functions tell the optimizer what to do
- a surrogate for what we really care about
- typically decompose over training examples, $(x, y) \sim \mathcal{D}$

Mean Square Error. standard loss for regression, $\frac{1}{2}\|f_\theta(x) - y\|^2$

0-1 Loss. what we care about for classification, 0 if $[\![f_\theta(x) \geq 0]\!] = y$ and 1 otherwise but 0-1 loss is non-differentiable so difficult to optimize

Negative Log-likelihood. standard loss for classification, $-\log p_\theta(y \mid x)$

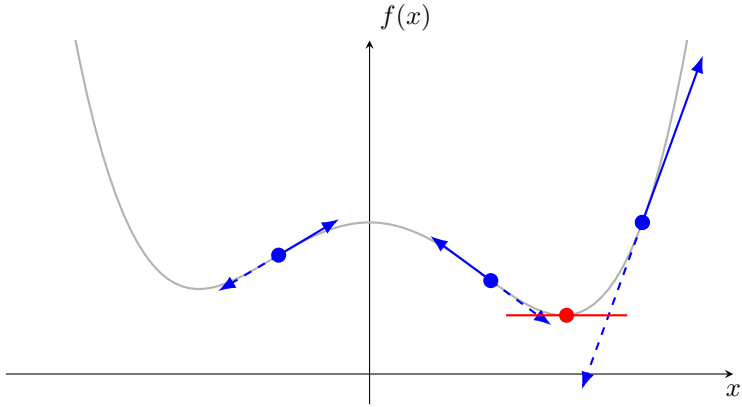
Cross Entropy. negative log-likelihood loss where p_θ is the multi-class logistic,

$$p_\theta \propto \exp(f_\theta(x))$$

Binary Cross Entropy. negative log-likelihood loss for K independent binary variables,

$$-\sum_{k=1}^{K} y_k \log p_{\theta,k}(1 \mid x) + (1 - y_k) \log p_{\theta,k}(0 \mid x)$$

Gradient Descent Optimization



- function decreases in the negative gradient direction
- but if we move too far the function may increase again

Multi-variate Calculus Review: Gradients

Let $f : \mathbb{R}^n \to \mathbb{R}$ be some function, fix some $x \in \mathbb{R}^n$, and consider the expression

$$\lim_{\alpha \to 0} \frac{f(x + \alpha e_i) - f(x)}{\alpha}$$

where $e_i = (0, \ldots, 1, 0, \ldots)$ is the i-th canonical vector.

If the limit exists, it's called the i-th **partial derivative** of f at x and denoted by $\frac{\partial f(x)}{\partial x_i}$.

The **gradient** of $f(x)$ with respect to $x \in \mathbb{R}^n$ is the vector of partial derivatives

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

Gradient Descent Algorithm

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla \ell(\theta^{(t)})$$

- η is the (iteration dependent) step size or step length or learning rate
  - ideally chosen by line search (expensive)
  - instead, in deep learning, chosen by fixed schedule

```
def gradient_descent(loss, theta0):
    """Generic gradient descent method."""

    theta = theta0
    for t in range(max_iters):
        dtheta = -1.0 * gradient(loss, theta)   # descent direction is negative of gradient at theta
        eta = line_search(loss, theta, dtheta)  # choose step size by line search
        theta = theta + eta * dtheta            # update

        if (converged()):                       # check stopping criteria
            break

    return theta
```

- stopping criterion usually of the form $\|\nabla \ell(\theta)\|_2 \leq \epsilon$
- very simple—just need gradient of ℓ with respect to θ—but can be very slow

(Vanilla) Gradient Descent Optimization in PyTorch

```
dataloader = ...     # way of loading batches of input-label pairs for training
model = ...          # definition of our prediction function

criterion = torch.nn.CrossEntropyLoss(reduction='mean')
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(max_epochs):
    for iteration, batch in enumerate(dataloader, 0):
        inputs, labels = batch

        # zero gradient buffers
        optimizer.zero_grad()

        # compute model outputs for given inputs
        outputs = model(inputs)

        # compute and print the loss
        loss = criterion(outputs, labels)
        print(loss.item())

        # compute gradient of the loss wrt model parameters
        loss.backward()

        # take a gradient step
        optimizer.step()
```

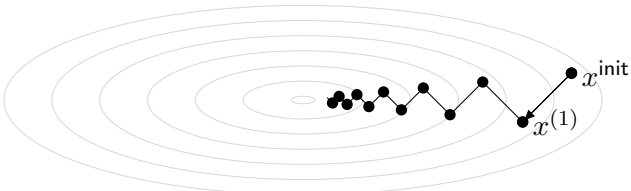
Example Quadratic Problem in $\mathbb{R}^2$

(Boyd and Vandenberghe, 2024)

$$\ell(\theta) = \frac{1}{2}\left(\theta_1^2 + \gamma\theta_2^2\right) \qquad (\gamma > 0)$$

with exact line search, starting at $\theta^{(0)} = (\gamma, 1)$:

$$\theta_1^{(t)} = \gamma\left(\frac{\gamma-1}{\gamma+1}\right)^t, \quad \theta_2^{(t)} = \left(-\frac{\gamma-1}{\gamma+1}\right)^t$$



- very slow if $\gamma \gg 1$ or $\gamma \ll 1$
- example above is for $\gamma = 10$

Calculating Gradients: Worked Example

- two-layer perceptron with sigmoid activation

$$z = \sigma(Ax + b)$$
$$y = \sigma(c^T z + d)$$

  where $\sigma(\xi) = (1 + e^{-\xi})^{-1}$ is applied elementwise
- arbitrary loss function to minimize ℓ
- for gradient descent we need $\frac{\mathrm{d}\ell}{\mathrm{d}A}$, $\frac{\mathrm{d}\ell}{\mathrm{d}b}$, etc.
- observe that

$$\begin{aligned}\frac{\mathrm{d}\sigma}{\mathrm{d}\xi} &= (-e^{-\xi})(-1)(1 + e^{-\xi})^{-2} \\ &= \left(\frac{1}{1 + e^{-\xi}}\right)\left(\frac{e^{-\xi}}{1 + e^{-\xi}}\right) \\ &= \sigma(\xi)(1 - \sigma(\xi))\end{aligned}$$

Calculating Gradients: Worked Example (cont.)

- let $\xi = c^T z + d$
- layer 2 parameters:

$$\begin{aligned}\frac{\partial \ell}{\partial d} &= \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\partial y}{\partial d} \\ &= \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\frac{\partial \xi}{\partial d} \\ &= \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1-y)\end{aligned}$$

$$\begin{aligned}\nabla_c \ell &= \frac{\mathrm{d}\ell}{\mathrm{d}y}\nabla_c y \\ &= \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\nabla_c \xi \\ &= \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1-y) z\end{aligned}$$

- where the last line is since $y = \sigma(\xi)$

Calculating Gradients: Worked Example (cont.)

- layer 2 input / layer 1 output:

$$\nabla_z \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1-y)c$$

- layer 1 parameters:

$$\nabla_b \ell = \nabla_z \ell \circ z \circ (1-z)$$
$$\nabla_A \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1-y)\left(c \circ z \circ (1-z)\right) x^T$$

  where $\circ$ denotes elementwise product

- note that we have reused calculations from evaluation of ℓ; we could have written y and z in terms of x in the gradient expressions (memory-vs-compute trade-off)