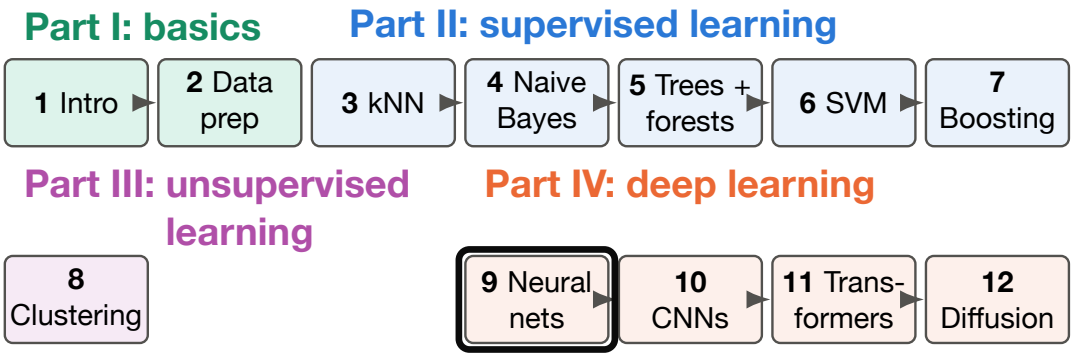
Introduction to deep learning

229352 Lecture 9: introduction to deep learning

Course map


Part I: basics
Part II: supervised learning
1 Intro
2 Data prep
3 kNN
4 Naive Bayes
5 Trees + forests
6 SVM
7 Boosting
Part III: unsupervised learning
Part IV: deep learning
8 Clustering
9 Neural nets
10 CNNs
11 Trans-formers
12 Diffusion

Textbook

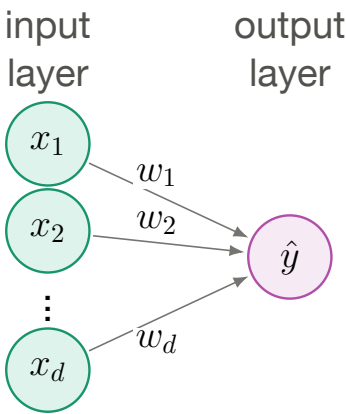
Dive into Deep Learning

A. Zhang, Z. C. Lipton, M. Li and A. J. Smola

Cambridge University Press, 2023

Free online: `https://d2l.ai`

Linear regression as a network



Input: $x = (x_1, \ldots, x_d)$, the d features of one data point.

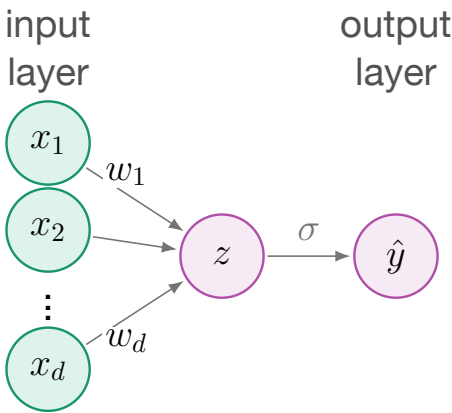
x_j: feature j.

Prediction:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$$

Each edge carries a **weight** w_j; the output node adds the **bias** b.

Logistic regression as a network



$$z = w_1 x_1 + \cdots + w_d x_d + b$$

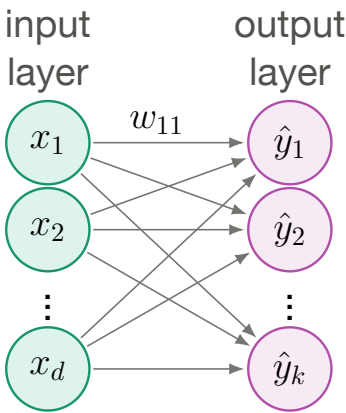
Sigmoid function:

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

Prediction: $\hat{y} = \sigma(z)$

$\hat{y}$ lies in $(0, 1)$: the predicted probability of class 1.

Regression with k outputs



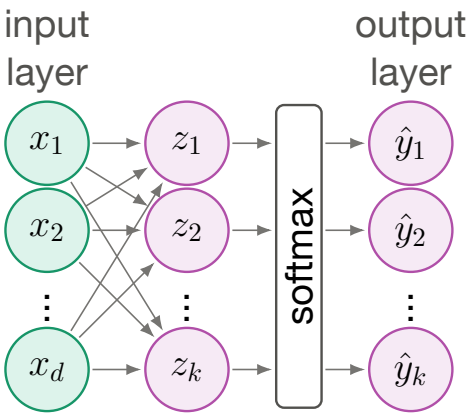
Output: $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_k)$.

$$\hat{y} = Wx + b$$

$$W = \begin{pmatrix} w_{11} & \cdots & w_{1d} \\ \vdots & & \vdots \\ w_{k1} & \cdots & w_{kd} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_k \end{pmatrix}$$

W is $k \times d$. w_{12}: the weight from x_2 into $\hat{y}_1$ (row: output; column: input).

Classification with k classes: softmax



$z = Wx + b$: one number z_c for each class $c = 1, \ldots, k$.

Softmax: $\hat{y}_c = \dfrac{e^{z_c}}{e^{z_1} + \cdots + e^{z_k}}$

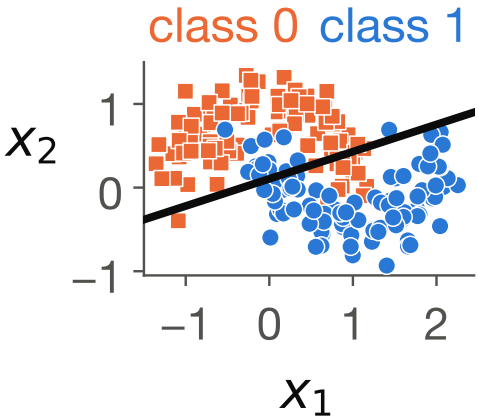
Each $\hat{y}_c$ lies in $(0, 1)$, and $\hat{y}_1 + \cdots + \hat{y}_k = 1$.

Example: $z = (2, 1, 0)$ gives $\hat{y} = (0.67,\ 0.24,\ 0.09)$.

Summary: models as networks

- Linear regression: d input nodes, one output node, $\hat{y} = w_1 x_1 + \cdots + w_d x_d + b$.
- Logistic regression: the same sum z, then the sigmoid: $\hat{y} = \sigma(z)$.
- k outputs: $\hat{y} = Wx + b$, with a $k \times d$ weight matrix W.
- k classes: softmax turns $z = Wx + b$ into k probabilities that sum to 1.

Data that a straight line cannot split

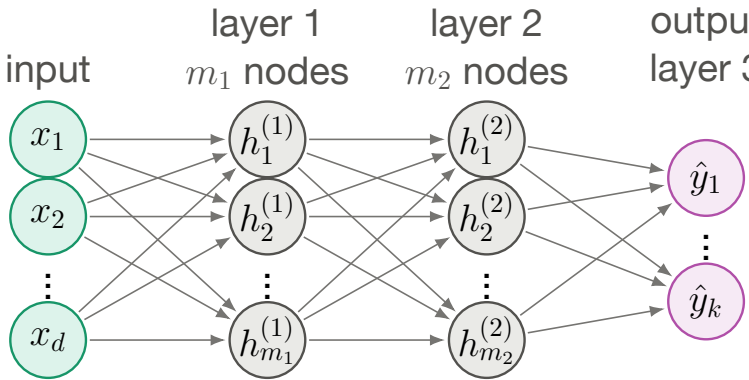


Training set: 200 points, two features x_1, x_2, two classes (simulated: the “moons”). Test set: 1,000 new points.

Logistic regression gives one straight decision boundary.

It classifies 86.4% of the test set correctly: its **test accuracy**.

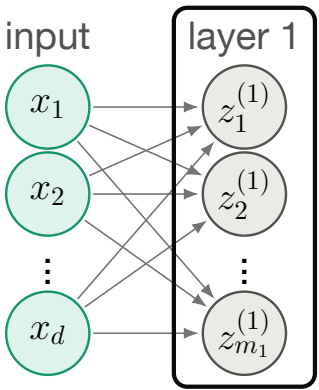
A neural network with two hidden layers



Hidden layers: layers 1 and 2. Superscript (l): layer l.

Nodes after the input add a bias to a weighted sum of the layer before and apply the **activation function** σ_l (not drawn).

Forward pass: layer 1, weighted sums



The **forward pass** computes $\hat{y}$ from x, layer by layer.

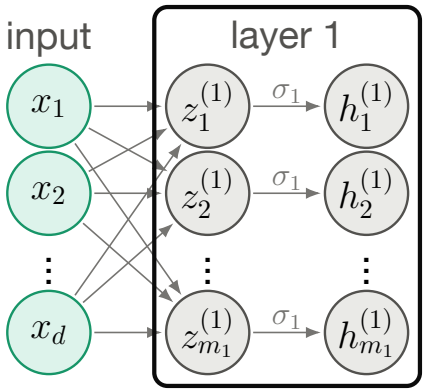
$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & \cdots & w_{1d}^{(1)} \\ \vdots & & \vdots \\ w_{m_1 1}^{(1)} & \cdots & w_{m_1 d}^{(1)} \end{pmatrix}$$

$$z^{(1)} = W^{(1)} x + b^{(1)}$$

$W^{(1)}$: $m_1 \times d$; $b^{(1)}$: m_1 entries.

$w_{21}^{(1)}$: weight from x_1 into node 2.

Forward pass: layer 1, activation



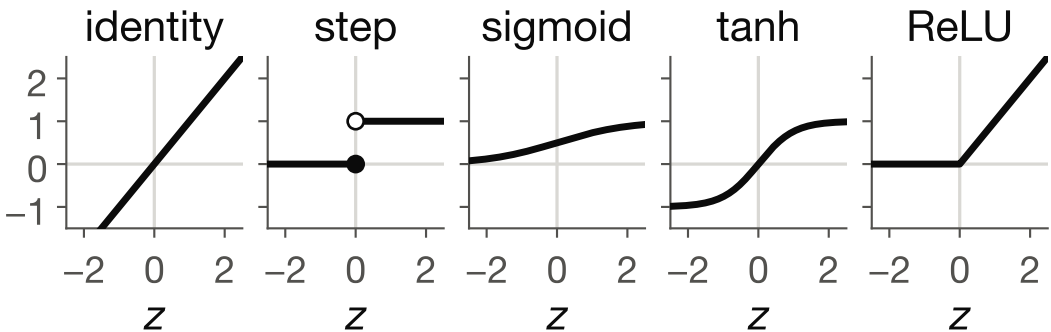
σ_1: the activation function of layer 1 (for example the sigmoid σ), applied to each entry:

$$h^{(1)} = \sigma_1\left(z^{(1)}\right)$$

$$\begin{pmatrix} h_1^{(1)} \\ \vdots \\ h_{m_1}^{(1)} \end{pmatrix} = \begin{pmatrix} \sigma_1\left(z_1^{(1)}\right) \\ \vdots \\ \sigma_1\left(z_{m_1}^{(1)}\right) \end{pmatrix}$$

$h^{(1)}$: the output of layer 1.

Examples of activation functions



each takes one number z and returns one number

Activation functions: formulas

name	value at z	range
identity	z	any number
step	1 if $z > 0$, else 0	0 or 1
sigmoid	$\sigma(z) = \dfrac{1}{1 + e^{-z}}$	between 0 and 1
tanh	$\tanh(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	between -1 and 1
ReLU	$\max(0, z)$	0 or larger

tanh: hyperbolic tangent. ReLU: rectified linear unit.

Nonlinear hidden-layer activations

Take a network with one hidden layer, so its output layer is layer 2, and let every activation be the identity:

$$h^{(1)} = W^{(1)}x + b^{(1)}, \qquad \hat{y} = W^{(2)}h^{(1)} + b^{(2)}$$

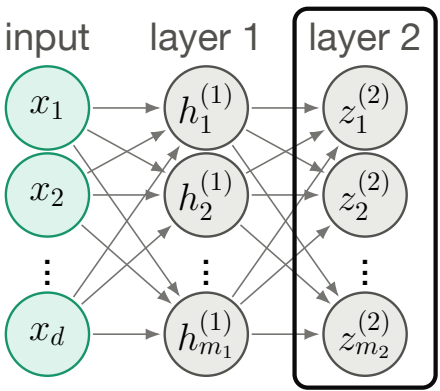
Substitute $h^{(1)}$:

$$\hat{y} = \underbrace{W^{(2)}W^{(1)}}_{\text{a matrix}}x + \underbrace{W^{(2)}b^{(1)} + b^{(2)}}_{\text{a vector}}$$

This is again $\hat{y} = Wx + b$: a linear model.

So at least one hidden-layer activation must be nonlinear.

Forward pass: layer 2, weighted sums



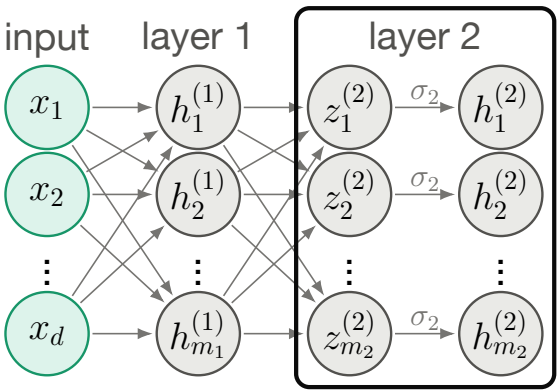
The input of layer 2 is $h^{(1)}$.

$$W^{(2)} = \begin{pmatrix} w_{11}^{(2)} & \cdots & w_{1m_1}^{(2)} \\ \vdots & & \vdots \\ w_{m_2 1}^{(2)} & \cdots & w_{m_2 m_1}^{(2)} \end{pmatrix}$$

$$z^{(2)} = W^{(2)} h^{(1)} + b^{(2)}$$

$W^{(2)}$: $m_2 \times m_1$, one row for each node of layer 2; $b^{(2)}$: m_2 entries.

Forward pass: layer 2, activation

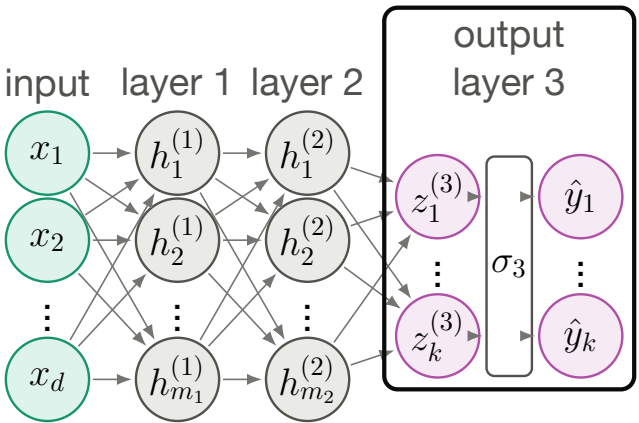


σ_2: the activation function of layer 2

$$h^{(2)} = \sigma_2\big(z^{(2)}\big)$$

Here $z^{(2)} = \big(z_1^{(2)}, \ldots, z_{m_2}^{(2)}\big)$ and $h^{(2)} = \big(h_1^{(2)}, \ldots, h_{m_2}^{(2)}\big)$, each with m_2 entries.

Forward pass: output layer



$W^{(3)}$: $k \times m_2$

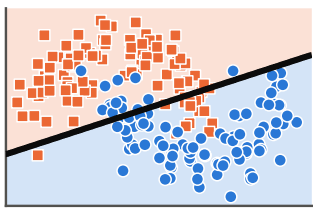
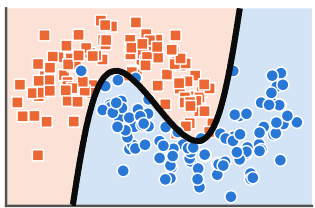
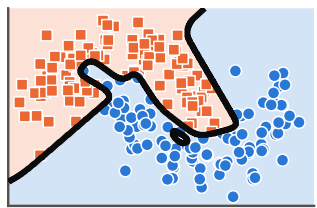
$b^{(3)}$: k entries

$$z^{(3)} = W^{(3)} h^{(2)} + b^{(3)}$$

$$\hat{y} = \sigma_3\left(z^{(3)}\right)$$

σ_3: identity (regression); sigmoid (two classes, $k = 1$); softmax (k classes).

Boundaries on the moons

logistic regression	network with 2 hidden nodes	network with 8 hidden nodes
		
test: 86.4%	test: 96.2%	test: 93.3%

under each panel: accuracy on the 1,000 test points

Accuracy on the moons

- Logistic regression: training 84.0%, test 86.4%.
- 2 hidden nodes (tanh): a curved boundary; test 96.2%.
- 8 hidden nodes (tanh): training 100.0% (all 200 points), but test 93.3%; the boundary bends around single training points.
- This is **overfitting** (Lecture 5): the fit follows the training set too closely, so the error on new data grows.
- Training these networks, and the choice among 10 **random starts** (random initial values of w), come later.

Summary: the forward pass

- Layer 1: $z^{(1)} = W^{(1)}x + b^{(1)}$, $h^{(1)} = \sigma_1\big(z^{(1)}\big)$; the next layer uses $h^{(1)}$ in place of x.
- $W^{(l)}$: one row per node of layer l, one column per node of the layer before.
- At least one hidden-layer activation must be nonlinear: with identity activations the network is a linear model.
- The forward pass computes $\hat{y}$ from x, layer by layer.

Checkpoint 1

Moons network: $d = 2$ inputs,
$m_1 = 8$ hidden nodes, $k = 1$ output.

What are the dimensions of $W^{(1)}$ and $W^{(2)}$?
How many weights and biases in total?

Checkpoint 1

Moons network: $d = 2$ inputs,
$m_1 = 8$ hidden nodes, $k = 1$ output.

What are the dimensions of $W^{(1)}$ and $W^{(2)}$?
How many weights and biases in total?

$W^{(1)}$: 8×2, $b^{(1)}$: 8; $W^{(2)}$: 1×8, $b^{(2)}$: 1

in total $16 + 8 + 8 + 1 = 33$

Training a neural network

Data: $\big(x^{(1)}, y^{(1)}\big), \ldots, \big(x^{(n)}, y^{(n)}\big)$; (i): data point i, not a layer.

Parameters: $w = (w_1, \ldots, w_K)$, the list of all K weights and biases (here w_j is not the weight of linear regression).

Prediction: $\hat{y}^{(i)} = \mathrm{NeuralNet}\big(x^{(i)}; w\big)$

$$\textbf{Loss}: \quad L(w) = \frac{1}{n} \sum_{i=1}^{n} \ell\big(y^{(i)}, \hat{y}^{(i)}\big)$$

the average of ℓ, the loss of one data point.

Goal: find the w with the smallest $L(w)$.

Loss for regression

For y and $\hat{y}$ with k entries, the loss is the **squared error**:

$$\ell(y, \hat{y}) = \|y - \hat{y}\|^2 = (y_1 - \hat{y}_1)^2 + \cdots + (y_k - \hat{y}_k)^2$$

The loss L averages ℓ over the n data points.

With one output ($k = 1$), L is the **mean squared error** (MSE).

Loss for two classes

Label $y \in \{0, 1\}$; $\hat{y} \in (0, 1)$: the probability of class 1.

$$\ell(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

the **binary cross-entropy** ($\log$: natural log).

$y = 1$: $\hat{y} = 0.9$ gives $\ell = 0.11$; $\hat{y} = 0.1$ gives $\ell = 2.30$.

A confident wrong prediction has a much larger loss.

Loss for k classes

One-hot label y: all entries are 0, except $y_c = 1$, where c is the true class of the data point.

Prediction: the softmax output $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_k)$.

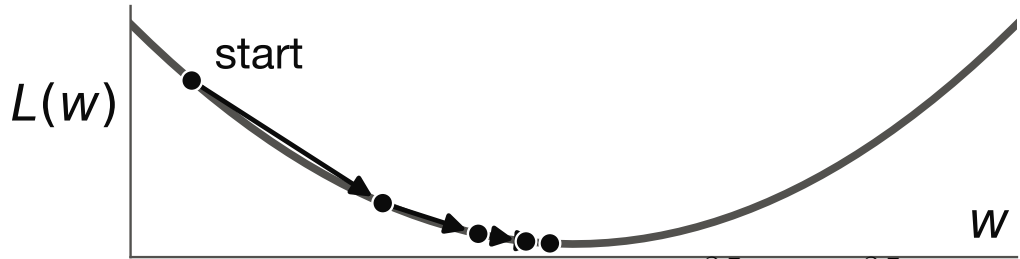
$$\ell(y, \hat{y}) = -y_1 \log \hat{y}_1 - y_2 \log \hat{y}_2 - \cdots - y_k \log \hat{y}_k$$

the **categorical cross-entropy**.

Only one y_c is 1, so $\ell = -\log \hat{y}_c$ for the true class c.

Gradient descent

Recap of Lecture 7 (θ replaced by w, p replaced by K); K: number of parameters.

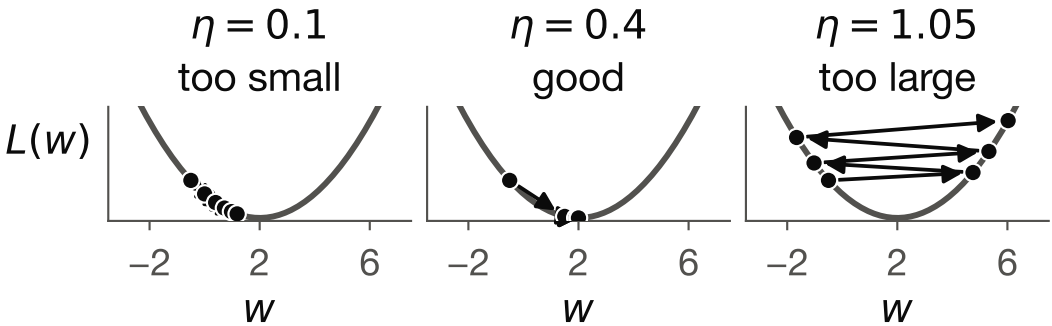


$$w \leftarrow w - \eta \nabla L(w), \qquad \nabla L(w) = \left(\frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_K}\right)$$

$\nabla L(w)$ points uphill, so a small enough step goes downhill.

$\eta > 0$: the **learning rate**.

The learning rate



$L(w) = (w-2)^2 + 0.5$, start at $w = -0.5$, 5 steps

η too small: slow; η too large: each step overshoots

Gradient descent for a neural network

1. Choose $\eta > 0$; start with random values $w_1, \ldots, w_K$.
2. Repeat a fixed number of times (or until $L(w)$ stops decreasing): $w \leftarrow w - \eta \nabla L(w)$.

$\nabla L(w)$: the average of the gradients of ℓ over the data.

Can we compute $\partial L/\partial w_1, \ldots, \partial L/\partial w_K$ efficiently?

Summary: loss and gradient descent

- Training finds the weights and biases w with the smallest loss $L(w) = \frac{1}{n}\sum_i \ell\big(y^{(i)}, \hat{y}^{(i)}\big)$.
- Squared error for regression; cross-entropy for classification (labels $\{0, 1\}$ or one-hot).
- Gradient descent: start at a random w, then repeat $w \leftarrow w - \eta \nabla L(w)$; $\nabla L(w)$ is the average of the gradients of ℓ.
- Learning rate η: if it is too small, training is slow; if it is too large, the steps overshoot.

Checkpoint 2

$$L(w) = w^2, \text{ start at } w = 1.$$

What is w after one gradient-descent step with $\eta = 0.5$? With $\eta = 1.5$?

Checkpoint 2

$$L(w) = w^2, \text{ start at } w = 1.$$

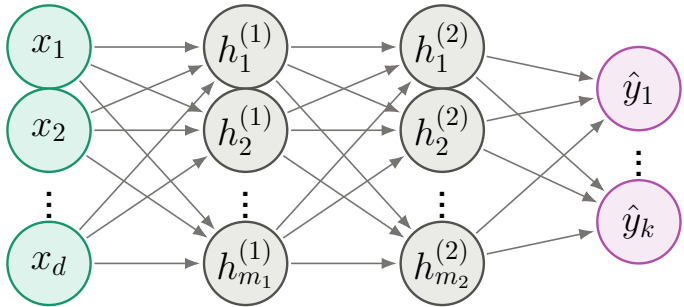
What is w after one gradient-descent step with $\eta = 0.5$? With $\eta = 1.5$?

$\nabla L(w) = 2w$, so $\nabla L(1) = 2$.

$\eta = 0.5$: $w = 1 - 0.5 \cdot 2 = 0$, the minimum.

$\eta = 1.5$: $w = 1 - 1.5 \cdot 2 = -2$, farther from 0 than the start.

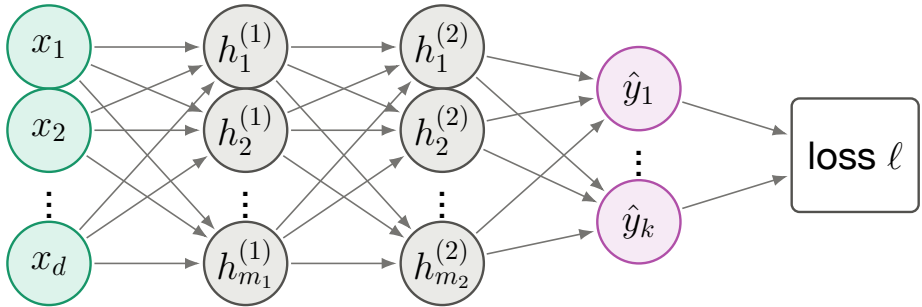
Backpropagation: the forward pass



Backpropagation computes all partial derivatives of the loss with one forward pass and one backward pass.

Forward pass: with the current w, pass the input x of one data point from **left to right**, and store the value of every node.

Backpropagation: the backward pass



Compute the loss ℓ of this data point. The **chain rule** gives the partial derivatives of ℓ with respect to every weight and bias, working from **right to left**.

D. E. Rumelhart, G. E. Hinton and R. J. Williams, Nature 323, 533–536, 1986

The chain rule

One path: if ℓ depends on w only through z, then

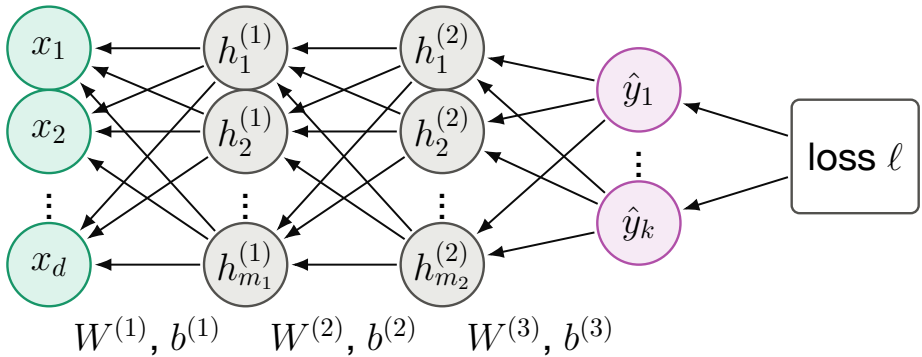
$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial z} \frac{\partial z}{\partial w}$$

Several paths: if ℓ depends on h only through $z_1, z_2, \ldots$, then

$$\frac{\partial \ell}{\partial h} = \frac{\partial \ell}{\partial z_1} \frac{\partial z_1}{\partial h} + \frac{\partial \ell}{\partial z_2} \frac{\partial z_2}{\partial h} + \cdots$$

In the network, a hidden node feeds every node of the next layer: one term for each edge.

The backward pass, layer by layer

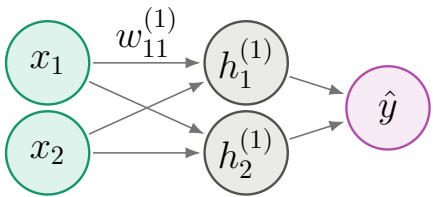


Each step computes the partial derivatives of the weights and biases shown below its two layers.

Each step uses the previous result and the stored values.

A tiny network: the hidden layer

hidden layer: sigmoid; output layer 2: $\hat{y} = z^{(2)}$; $\ell = (y - \hat{y})^2$



Given: $\sigma(0.5) = 0.6225$

$z^{(1)} =$

$x = (1,\ 2),\ y = 1$

$W^{(1)} = \begin{pmatrix} 1 & -0.5 \\ 0.5 & 0.5 \end{pmatrix}$

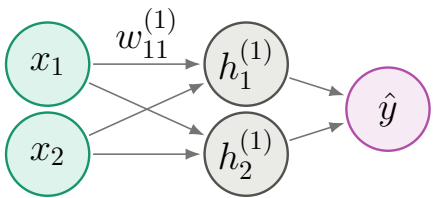
$h^{(1)} =$

$b^{(1)} = (0,\ -1)$

$W^{(2)} = (2\ \ 1),\ b^{(2)} = 0.5$

A tiny network: output and loss

hidden layer: sigmoid; output layer 2: $\hat{y} = z^{(2)}$; $\ell = (y - \hat{y})^2$



$\hat{y} =$

$x = (1,\ 2),\ \ y = 1$

$W^{(1)} = \begin{pmatrix} 1 & -0.5 \\ 0.5 & 0.5 \end{pmatrix}$

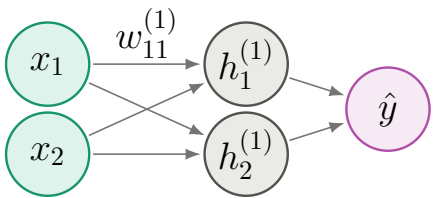
$b^{(1)} = (0,\ -1)$

$W^{(2)} = (2\ \ 1),\ \ b^{(2)} = 0.5$

$\ell =$

A tiny network: one partial derivative

hidden layer: sigmoid; output layer 2: $\hat{y} = z^{(2)}$; $\ell = (y - \hat{y})^2$



Sigmoid: $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$

$$\frac{\partial \ell}{\partial w_{11}^{(1)}} =$$

$x = (1,\ 2),\ y = 1$

$$W^{(1)} = \begin{pmatrix} 1 & -0.5 \\ 0.5 & 0.5 \end{pmatrix}$$

$b^{(1)} = (0,\ -1)$

$W^{(2)} = (2\ \ 1),\ b^{(2)} = 0.5$

Backpropagation, animated

A video by 3Blue1Brown (YouTube channel):

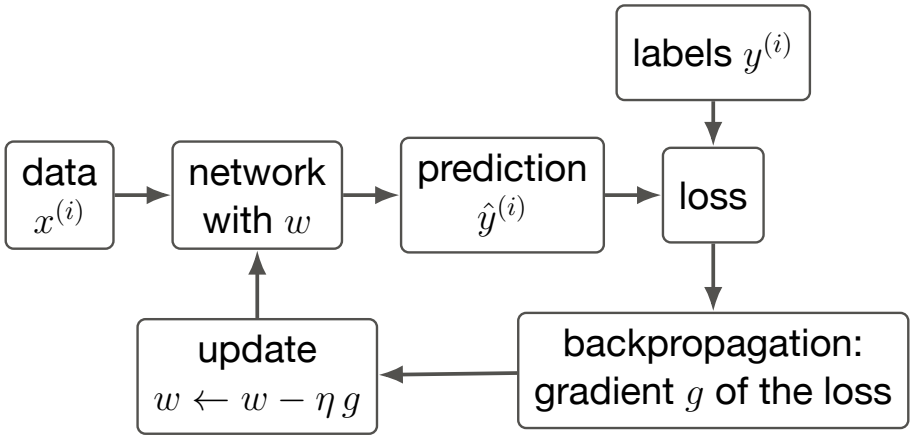
“Backpropagation, intuitively | Deep Learning Chapter 3”

`https://www.youtube.com/watch?v=Ilg3gGewQ5U`

Summary: backpropagation

- Forward pass: for one data point, compute and store the value of every node, from left to right.
- Backward pass: the chain rule gives the partial derivatives of ℓ with respect to every weight and bias, from right to left, with the stored values.
- One forward and one backward pass per data point give all K partial derivatives of ℓ. ∇L is the average of these gradients over the data points (in practice over a small random part of the data, defined next).

The training cycle



Hyperparameters of a neural network

Hyperparameters are settings chosen before training.
(Batch size and epochs: next slide.)

- number of layers
- number of nodes in each layer
- activation function of each layer
- batch size: training points per update
- number of epochs: passes over the training set
- learning rate η: not too small, not too large

Minibatches and epochs

Minibatch B: a random subset of the training set.

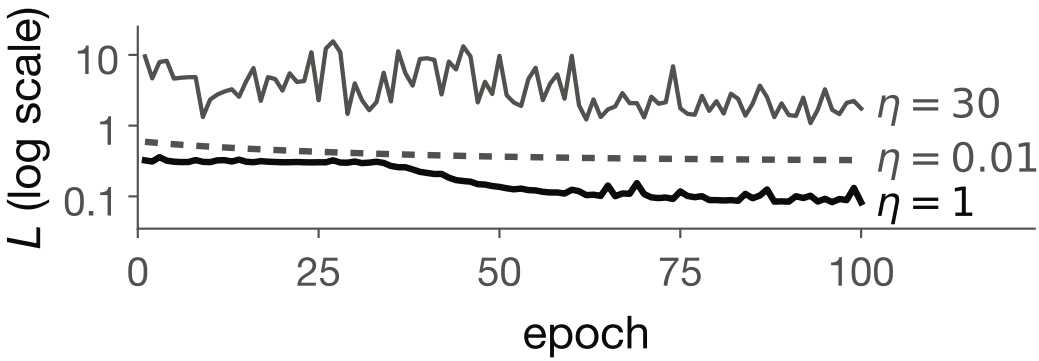
$$w \leftarrow w - \eta\, g, \qquad g = \nabla\Big(\frac{1}{|B|}\sum_{i\in B} \ell\big(y^{(i)}, \hat{y}^{(i)}\big)\Big)$$

g: gradient of the average loss over B (not all n points).

$|B|$: the **batch size**. **Epoch**: one pass over the training set.

Moons, batch size 20: 10 updates per epoch;
100 epochs: 1,000 updates.

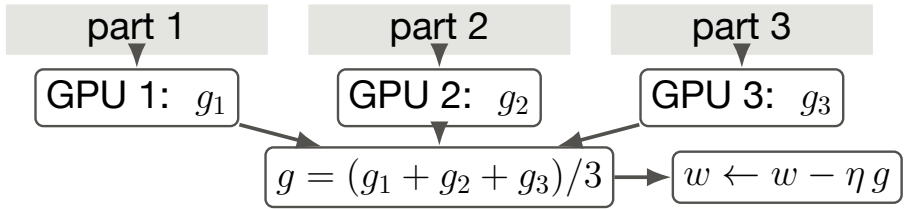
Learning rates on the moons



minibatch updates of 20 points

$\eta = 1$: L falls most; $\eta = 0.01$: slow; $\eta = 30$: L jumps

Training with GPUs



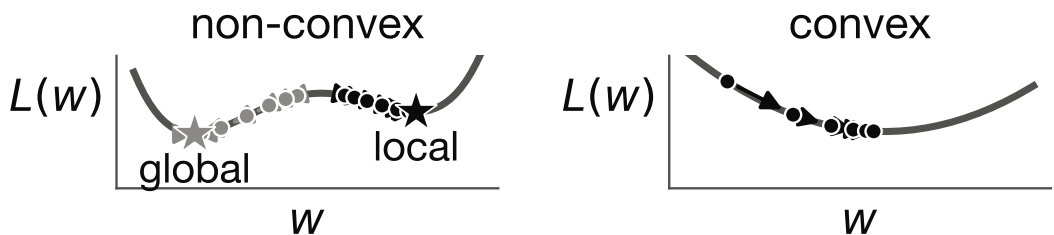
A **GPU** (graphics processing unit) computes in parallel.

Each GPU holds a copy of w and computes the gradient of the average loss on its part of the minibatch (equal parts).

The average g goes to every GPU, which updates its w.

Diagram after A. Zhang et al., *Dive into Deep Learning*, Section 13.5 (CC BY-SA 4.0)

Non-convex losses



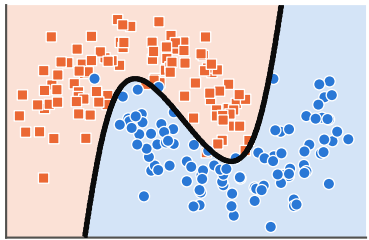
left: two starts; $\star$: where each run ends.

Local minimum: the lowest L among nearby w; it can be larger than the **global minimum**, the lowest L over all w.

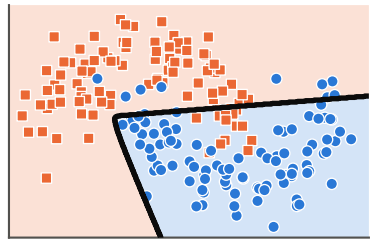
Convex: bowl-shaped (right); any local minimum is global.

Two starts on the moons

training loss 0.07



training loss 0.26



2 hidden nodes: smallest and largest training loss of 10 random starts.

8 of the 10 starts end near the larger loss (right).

Dealing with local minima

The loss of a neural network is usually non-convex.

- Several random starts: keep the one with the smallest training loss.
- A large learning rate at first and a smaller one later (a **learning-rate schedule**) can help.
- Optimizers that adapt the learning rate of each weight, such as **Adam** and **RMSProp**, often train faster.

D. P. Kingma and J. Ba, *Adam: a method for stochastic optimization*, ICLR 2015.
G. Hinton with N. Srivastava and K. Swersky, *Neural Networks for Machine Learning*, Coursera, 2012, lecture 6e: rmsprop.

Summary: training a neural network

- Each update, on one minibatch: forward and backward passes give the minibatch gradient g; then $w \leftarrow w - \eta\, g$.
- Hyperparameters: layers, nodes, activations, batch size, epochs, η.
- An epoch: one pass over the training set.
- Non-convex loss: several random starts, or a decreasing learning rate, can help; Adam adapts the learning rate of each weight.

A neural network in scikit-learn

```
from sklearn.datasets import make_moons
from sklearn.neural_network import MLPClassifier
X, y = make_moons(200, noise=0.2, random_state=0)
X_test, y_test = make_moons(1000, noise=0.2, random_state=1)
net = MLPClassifier(hidden_layer_sizes=(8,), activation="tanh",
                    solver="lbfgs", max_iter=5000,
                    random_state=0)
net.fit(X, y)
print([W.shape for W in net.coefs_])  # [(2, 8), (8, 1)]
print(net.score(X_test, y_test))  # test accuracy: 0.945
```