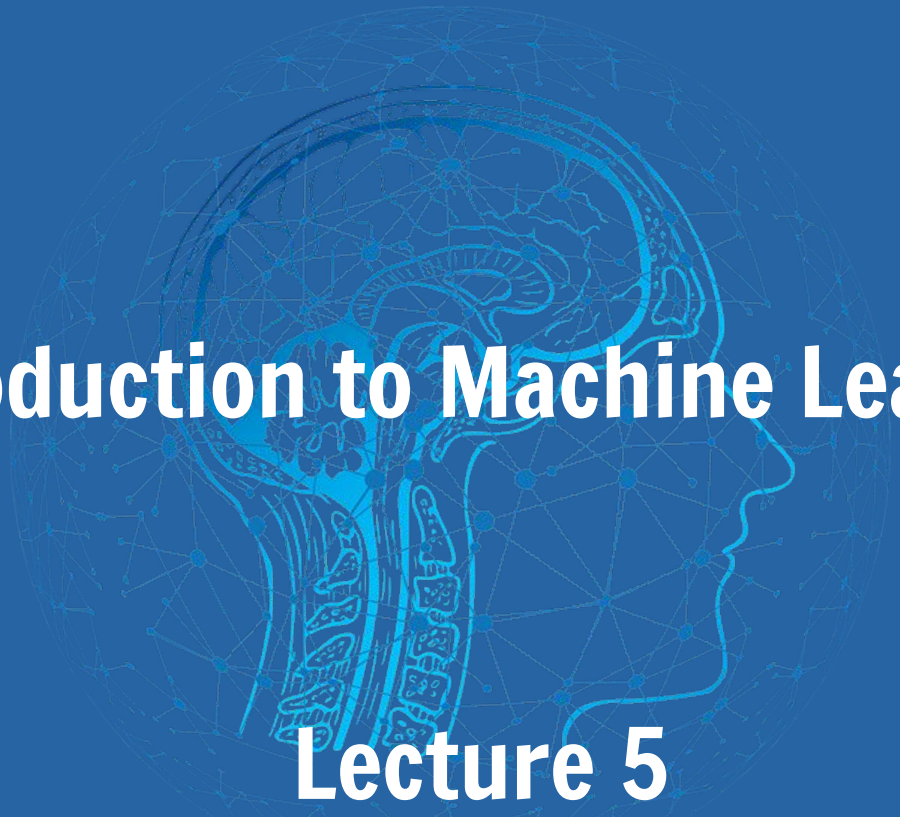
Imperial College London



Introduction to Machine Learning

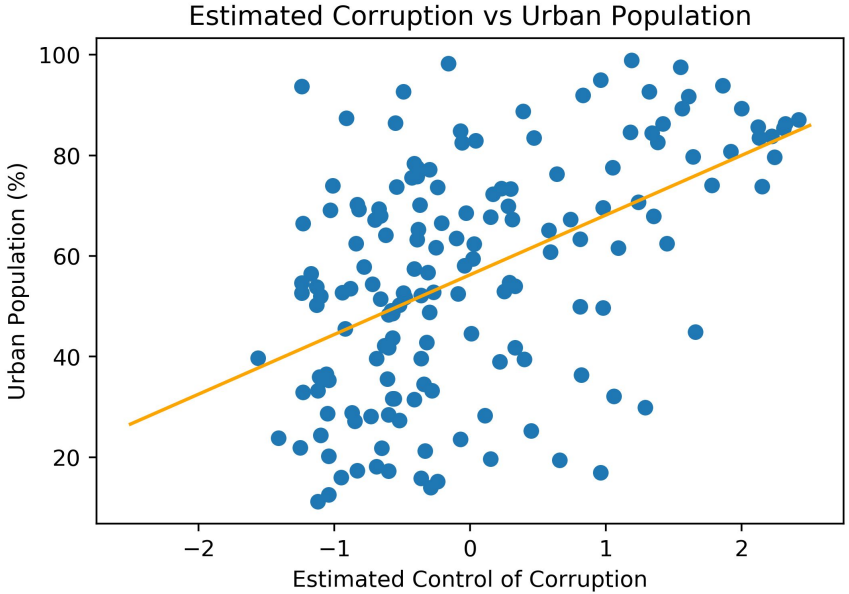
Lecture 5

Antoine Cully & Marek Rei & Josiah Wang

In the previous lecture...

Linear regression and gradient descent

Iteratively updating the weights for finding a good fit to data.

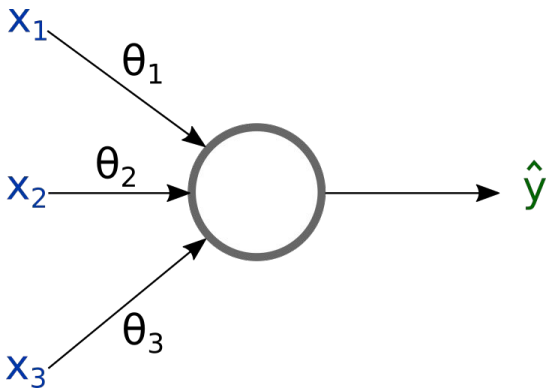


$$y = ax + b$$

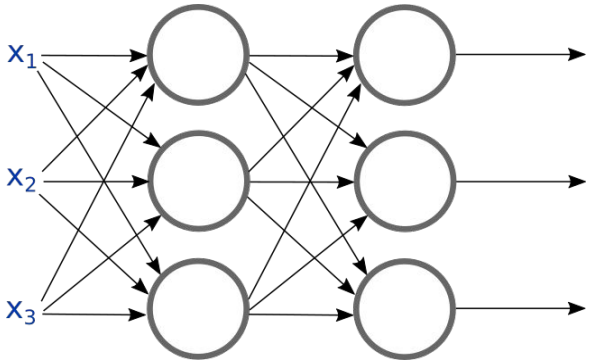
$$E = \frac{1}{2}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

$$a := a - \alpha\frac{\partial E}{\partial a} \qquad b := b - \alpha\frac{\partial E}{\partial b}$$

Artificial neuron and neural networks



$$\hat{y} = g(\sum_k \theta_k x_k + b)$$

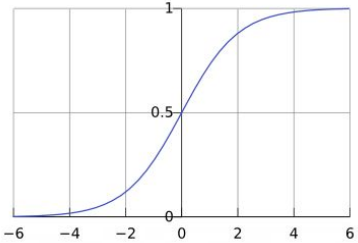


$$h = g_h(W_h^T x + b_h)$$

$$\hat{y} = g_{\hat{y}}(W_{\hat{y}}^T h + b_{\hat{y}})$$

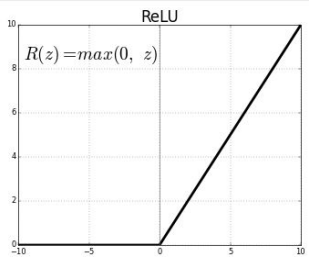
Activation Functions

Sigmoid/logistic activation



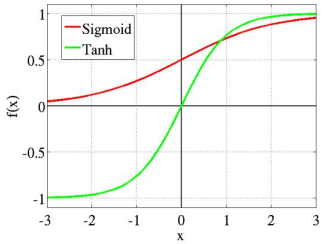
$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

ReLU activation



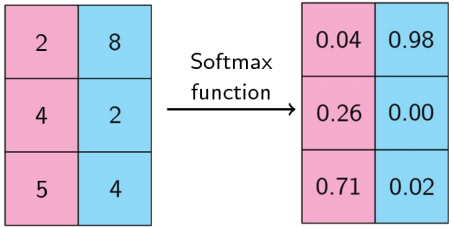
$$f(x) = ReLU(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

Tanh activation



$$f(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

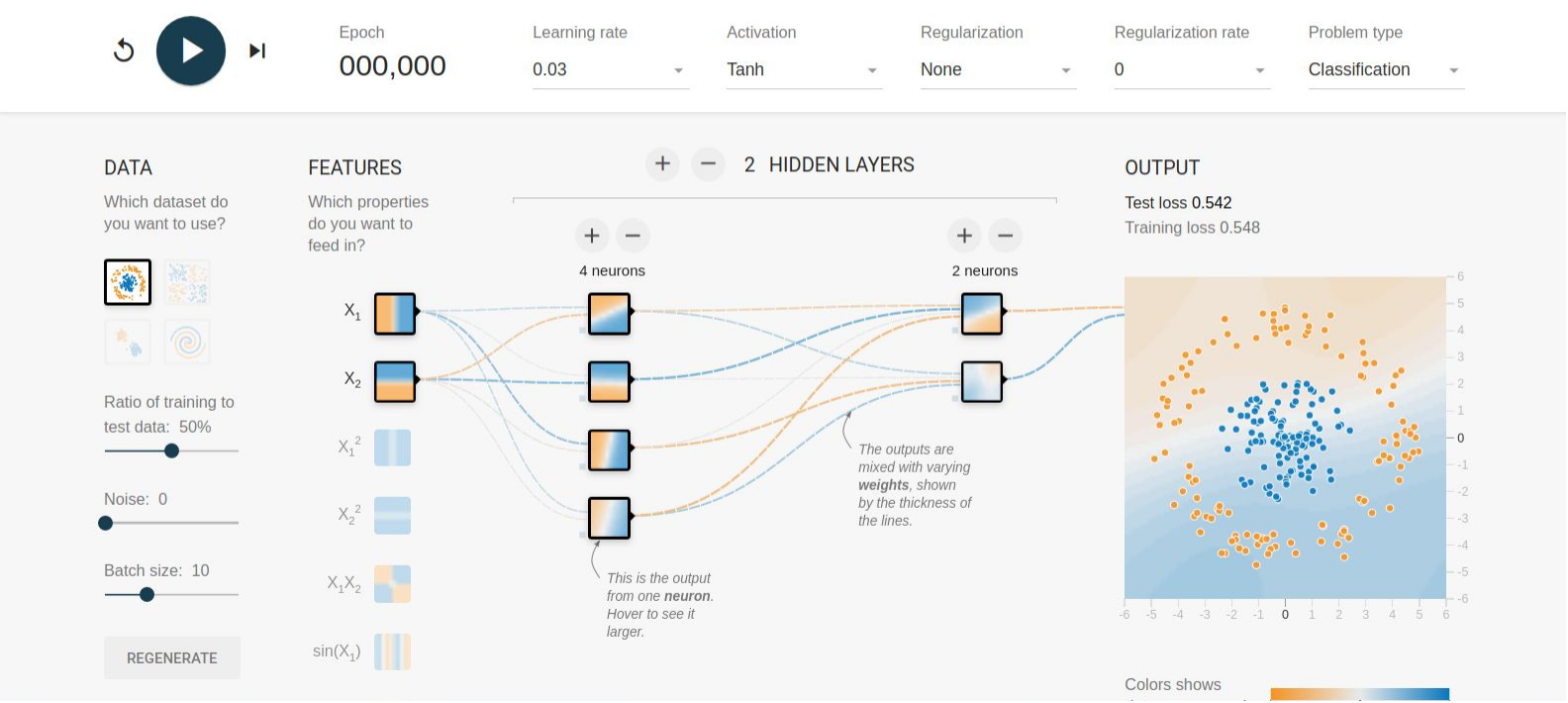
Softmax activation



$$softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

Today...

Tensorflow Playground



https://playground.tensorflow.org/

GPU options

Neural networks really benefit from training with GPUs.

1. The lab machines have GPUs. You can log in remotely.
   https://www.doc.ic.ac.uk/csg/facilities/lab/workstations

2. Colab notebooks run on GPUs.
   https://colab.research.google.com/

3. Kaggle also provides free GPUs for taking part in their tasks.
   https://www.kaggle.com/

Course plan

	Lecture
Week 2	Introduction to ML + Classification
Week 3	Instance-based Learning + Decision Trees
Week 4	Machine Learning Evaluation
Week 5	Artificial Neural Networks I
Week 6	Artificial Neural Networks II
Week 7	Unsupervised Learning
Week 8	Genetic Algorithms

Today's lecture

- Loss functions
  - Mean squared error and cross-entropy
- Backpropagation
  - Calculating derivatives for the weights
- Gradient descent
  - Mini-batching, learning rates, weight initialisation
- Overfitting in neural networks
  - Regularisation methods

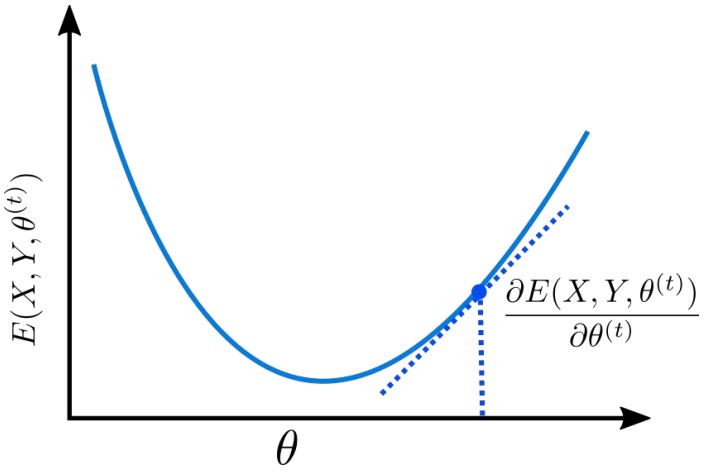
Loss functions

Optimising Neural Networks

Define a **loss function** that we want to minimize - having a lower loss means we are doing better on our task.

Update the parameters using **gradient descent**, taking small steps in the negative direction of the gradient (going downhill on the slope).

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \alpha \frac{\partial E}{\partial \theta_i^{(t)}}$$



Loss function for regression

We have a **regression task** when we need to predict a continuous variable. For example:

- Velocity of a car
- Price of a house

We would often use **linear activation** in the output layer and optimise the network with **mean squared error** (also known as *quadratic loss* or *L2 loss*).

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

MSE will be 0 when our predictions are equal to the true answers.

Classification

In classification, the model needs to choose between categorical (discrete) options.

- **Binary classification**. Only two possible classes. For example, does the patient have a disease or not.
- **Multi-class classification**. More than one possible class to choose from, but every input belongs to exactly one class. For example, detecting digits.
- **Multi-label classification**. Each input can belong to more than one class. For example, detecting all the objects in an image.

Cross-entropy

We want to **maximise the likelihood** of the network assigning the correct labels to all inputs in our dataset:

$$\prod_{i=1}^{N} p(y^{(i)}|x^{(i)};\theta)$$

Predict correct value given x(i) value and model parameter

Assuming that the examples are independent and identically distributed (i.i.d.) such that:

$$p(A \wedge B) = p(A)p(B)$$

For binary classification, we can regard the output of our network as a parameter of a **Bernoulli distribution**, the probability of success given an input:

$$\prod_{i=1}^{N} (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{(1-y^{(i)})}$$

The probability of one class is 1 minus the other. If we assign all probability to the correct label for every datapoint, the total is 1; if we always output the opposite, we get 0.

Cross-entropy

Maximising the logarithm is the same as maximising the original product, so let's take the log:

$$\sum_{i=1}^{N} y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)})$$

Turning this into a loss we can minimise gives us the **binary cross-entropy**:

$$L = -\frac{1}{N} \sum_{i=1}^{N} (y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)}))$$

This can be generalised to **categorical cross-entropy** for multi-class classification:

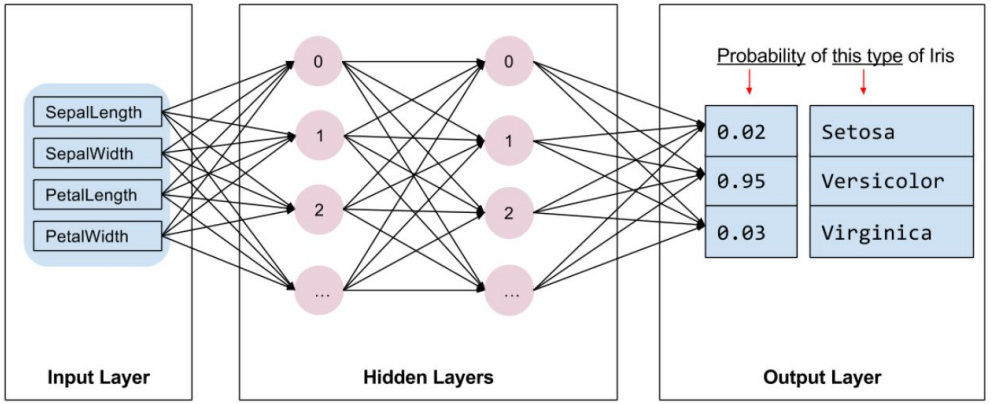
$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_c^{(i)} log(\hat{y}_c^{(i)})$$

yi = 1 if predict correctly, = 0 if predict incorrectly

Where C is the set of possible classes and $\hat{y}_c^{(i)}$ is the predicted probability of class c for datapoint i.

Loss function for classification

For **multi-class classification**, it is common to use softmax activation together with categorical cross-entropy loss.



Problem	Type	Output Activation	Loss
future stock prices	regression	linear	mse
stock prices up or down	binary	sigmoid	binary crossentropy
speech recognition	multi-class	softmax	categorical crossentropy
chemical properties	multi-label	sigmoid	binary crossentropy

Backpropagation

Batching

Batching: combining the vectors of several datapoints into one matrix to improve speed and reduce noise

$$\begin{bmatrix} x_{1,1} & x_{1,2} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} =$$

$$\begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} x_{2,1} & x_{2,2} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} =$$

$$\begin{bmatrix} x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} =$$

$$\begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{bmatrix}$$

Batching

Batching: combining the vectors of several datapoints into one matrix to improve speed and reduce noise

GPUs are more efficient in computing matrix multiplication

$$\begin{bmatrix} x_{1,1} & x_{1,2} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} =$$

$$\begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} x_{2,1} & x_{2,2} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} =$$

$$\begin{bmatrix} x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} =$$

$$\begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{bmatrix}$$

Forward and backward pass

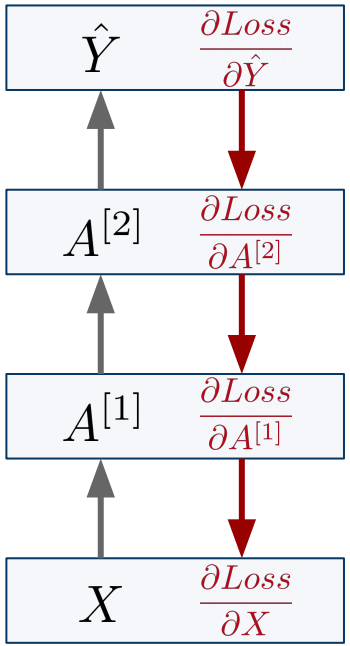
$$Loss = L(Y, \hat{Y})$$

$$\hat{Y} = g_o(Z^{[3]})$$
$$Z^{[3]} = A^{[2]}W^{[3]} + B^{[3]}$$

$$A^{[2]} = g_h(Z^{[2]})$$
$$Z^{[2]} = A^{[1]}W^{[2]} + B^{[2]}$$

$$A^{[1]} = g_h(Z^{[1]})$$
$$Z^{[1]} = XW^{[1]} + B^{[1]}$$

$$X \in \mathbb{R}^{N \times K}$$



All weights and bias should be updated

Backpropagation iteratively calculates all the necessary partial derivatives.

N

Chain rule

Chain rule for calculating the derivative of a composite function:

$$z = f(x) \qquad y = g(z) \qquad \frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$$

That means we can break our derivative down:

$$\frac{\partial Loss}{\partial W^{[1]}} = \frac{\partial Loss}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

$$\frac{\partial Loss}{\partial Z^{[1]}} = \frac{\partial Loss}{\partial A^{[1]}} \cdot \frac{\partial A^{[1]}}{\partial Z^{[1]}}$$

$$\frac{\partial Loss}{\partial W^{[1]}} = \frac{\partial Loss}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z^{[3]}} \cdot \frac{\partial Z^{[3]}}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial A^{[1]}} \cdot \frac{\partial A^{[1]}}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

Chain rule

Chain rule for calculating the derivative of a composite function:

$$z = f(x) \qquad y = g(z) \qquad \frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$$

That means we can break our derivative down:

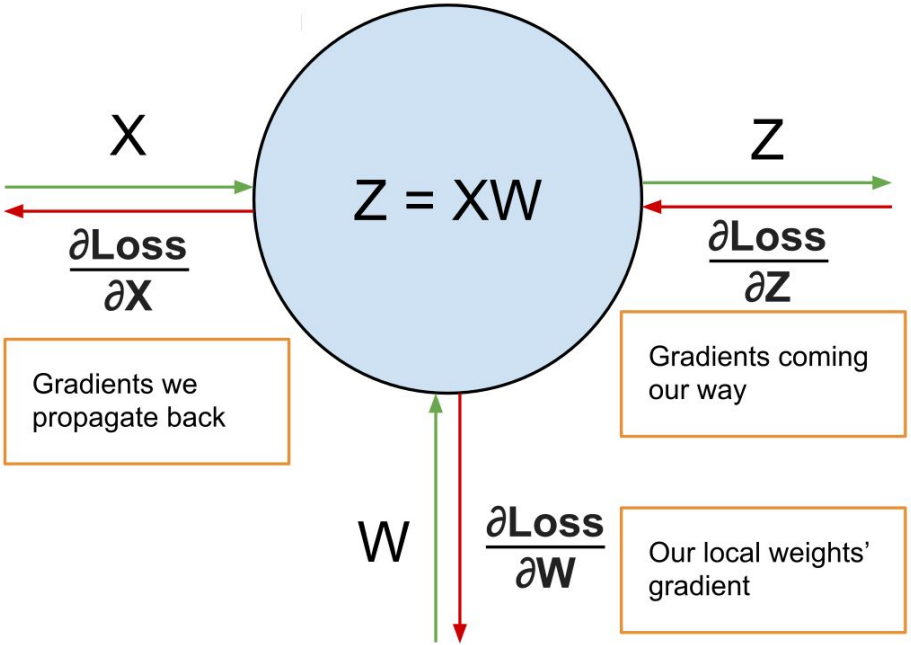
$$\frac{\partial Loss}{\partial W^{[1]}} = \frac{\partial Loss}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

$$\frac{\partial Loss}{\partial Z^{[1]}} = \frac{\partial Loss}{\partial A^{[1]}} \cdot \frac{\partial A^{[1]}}{\partial Z^{[1]}}$$

$$\frac{\partial Loss}{\partial W^{[1]}} = \frac{\partial Loss}{\partial \hat{Y}} \cdot \boxed{\frac{\partial \hat{Y}}{\partial Z^{[3]}}} \cdot \boxed{\frac{\partial Z^{[3]}}{\partial A^{[2]}}} \cdot \boxed{\frac{\partial A^{[2]}}{\partial Z^{[2]}}} \cdot \boxed{\frac{\partial Z^{[2]}}{\partial A^{[1]}}} \cdot \boxed{\frac{\partial A^{[1]}}{\partial Z^{[1]}}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

Don't try to apply this on matrices directly. These are 4D tensors!

Backpropagation



Backpropagation: Propagating the gradients backwards through the network layers

Note that we changed from $z = W^T x$ to $Z = XW$ for the convenience of working with batches.

$X \in \mathbb{R}^{N \times D}$ $W \in \mathbb{R}^{D \times M}$ $B \in \mathbb{R}^{N \times M}$ $Z \in \mathbb{R}^{N \times M}$

Backpropagation: linear layer

$$Z = XW + B$$

$$X \in \mathbb{R}^{N \times D} \quad W \in \mathbb{R}^{D \times M}$$

$$Z \in \mathbb{R}^{N \times M} \quad B \in \mathbb{R}^{N \times M}$$

Consider a particular case:

$$N = 2 \quad D = 2 \quad M = 3$$

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} \quad W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} \quad B = \begin{bmatrix} \mathbf{b} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

$$Z = \begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} + b_1 & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} + b_2 & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} + b_3 \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + b_1 & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} + b_2 & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} + b_3 \end{bmatrix}$$

Backpropagation: linear layer inputs

Let's assume that through backpropagation we already receive

$$\frac{\partial Loss}{\partial Z} = \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix}$$

Tip: The derivative of a scalar with respect to a matrix has the same shape as the matrix

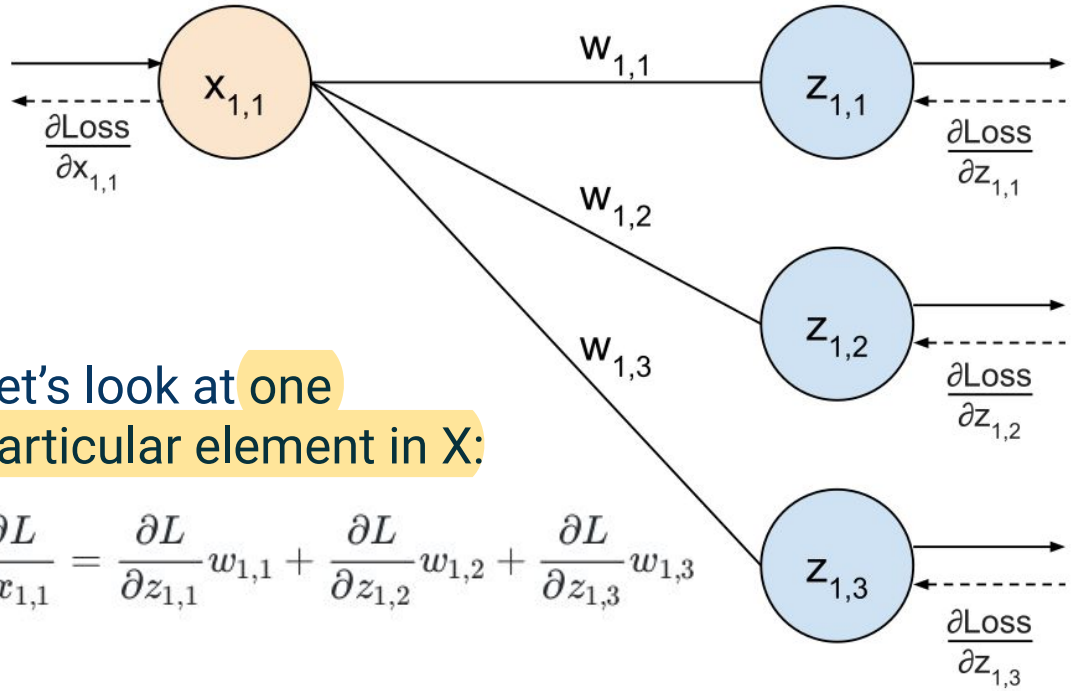
To update the weights we need to calculate

$$\frac{\partial Loss}{\partial W} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial W} \qquad \frac{\partial Loss}{\partial b} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial b}$$

And to pass the gradient on to the lower layer we need to calculate

$$\frac{\partial Loss}{\partial X} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial X}$$

Backpropagation: linear layer inputs



Let's look at one particular element in X:

$$\frac{\partial L}{\partial x_{1,1}} = \frac{\partial L}{\partial z_{1,1}} w_{1,1} + \frac{\partial L}{\partial z_{1,2}} w_{1,2} + \frac{\partial L}{\partial z_{1,3}} w_{1,3}$$

Intuitively, how $x_{1,1}$ affects the loss is determined by the weights it multiplies with and then whatever is using that output upstream.

Backpropagation: linear layer inputs

For one element of X:

$$\frac{\partial L}{\partial x_{1,1}} = \frac{\partial L}{\partial z_{1,1}} w_{1,1} + \frac{\partial L}{\partial z_{1,2}} w_{1,2} + \frac{\partial L}{\partial z_{1,3}} w_{1,3}$$

For all of X:

$$\frac{\partial Loss}{\partial X} = \begin{bmatrix} \frac{\partial L}{\partial z_{1,1}} w_{1,1} + \frac{\partial L}{\partial z_{1,2}} w_{1,2} + \frac{\partial L}{\partial z_{1,3}} w_{1,3} & \frac{\partial L}{\partial z_{1,1}} w_{2,1} + \frac{\partial L}{\partial z_{1,2}} w_{2,2} + \frac{\partial L}{\partial z_{1,3}} w_{2,3} \\ \frac{\partial L}{\partial z_{2,1}} w_{1,1} + \frac{\partial L}{\partial z_{2,2}} w_{1,2} + \frac{\partial L}{\partial z_{2,3}} w_{1,3} & \frac{\partial L}{\partial z_{2,1}} w_{2,1} + \frac{\partial L}{\partial z_{2,2}} w_{2,2} + \frac{\partial L}{\partial z_{2,3}} w_{2,3} \end{bmatrix}$$

This can also be written as a dot product:

$$\frac{\partial Loss}{\partial X} = \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \\ w_{1,3} & w_{2,3} \end{bmatrix}$$

Both of these matrices look familiar from before:

$$\frac{\partial Loss}{\partial X} = \frac{\partial Loss}{\partial Z} W^T$$

In some networks we can actually use this to update input X, but in most cases we just backprop to a lower level

Backpropagation: linear layer weights

We can do the same process to find the derivatives with respect to the weights:

$$\frac{\partial L}{\partial w_{1,1}} = \frac{\partial L}{\partial z_{1,1}} x_{1,1} + \frac{\partial L}{\partial z_{2,1}} x_{2,1}$$

$$Z = XW + B$$

For the full weight matrix:

$$\frac{\partial Loss}{\partial W} = \begin{bmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{bmatrix} \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix}$$

Affect 1 neuron, but with batching, two data points are involved

And substituting in familiar values:

$$\frac{\partial Loss}{\partial W} = X^T \frac{\partial Loss}{\partial Z}$$

Backpropagation: linear layer biases

And finally, the bias:

$$Z = XW + B \qquad B = \begin{bmatrix} \mathbf{b} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

We obtain:

$$\frac{\partial Loss}{\partial b} = \mathbf{1}^T \frac{\partial Loss}{\partial Z}$$

where $\mathbf{1}$ is a column vector of ones.

But the individual steps of finding that are left for you as an exercise.

Partial derivatives for vectors and matrices

Scalar-by-vector $$\frac{\partial y}{\partial \mathbf{x}} = \left[\frac{\partial y}{\partial x_1} \quad \frac{\partial y}{\partial x_2} \cdots \frac{\partial y}{\partial x_n}\right]$$

Vector-by-vector $$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

$z = Wx \quad \frac{\partial z}{\partial x} = W$	$z = Wx \quad \delta = \frac{\partial J}{\partial z}$ $\frac{\partial J}{\partial W} = \delta^\top x$
$z = x \quad \frac{\partial z}{\partial x} = I$	
$z = xW \quad \frac{\partial z}{\partial x} = W^\top$	$z = xW \quad \delta = \frac{\partial J}{\partial z}$ $\frac{\partial J}{\partial W} = x^\top \delta$

Backpropagation: activation functions

What about the activation functions?

$$A = g(Z) = \begin{bmatrix} g(z_{1,1}) & g(z_{1,2}) \\ g(z_{2,1}) & g(z_{2,2}) \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$$

This time we only need to find the derivative with respect to Z, to pass the gradient through:

$$\frac{\partial Loss}{\partial Z} = \frac{\partial Loss}{\partial A} \cdot \frac{\partial A}{\partial Z}$$

For a single element in Z:

$$\frac{\partial Loss}{\partial z_{1,1}} = \frac{\partial Loss}{\partial a_{1,1}} \cdot \frac{\partial a_{1,1}}{\partial z_{1,1}} = \frac{\partial Loss}{\partial a_{1,1}} g'(z_{1,1})$$

Where g’ is the derivative of our activation function.

Backpropagation: activation functions

Generalising to matrix form for the whole Z:

$$\frac{\partial Loss}{\partial Z} = \begin{bmatrix} \frac{\partial Loss}{\partial a_{1,1}} g'(z_{1,1}) & \frac{\partial Loss}{\partial a_{1,2}} g'(z_{1,2}) \\ \frac{\partial Loss}{\partial a_{2,1}} g'(z_{2,1}) & \frac{\partial Loss}{\partial a_{2,2}} g'(z_{2,2}) \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial Loss}{\partial a_{1,1}} & \frac{\partial Loss}{\partial a_{1,2}} \\ \frac{\partial Loss}{\partial a_{2,1}} & \frac{\partial Loss}{\partial a_{2,2}} \end{bmatrix} \circ \begin{bmatrix} g'(z_{1,1}) & g'(z_{1,2}) \\ g'(z_{2,1}) & g'(z_{2,2}) \end{bmatrix}$$

$$= \frac{\partial Loss}{\partial A} \circ g'(Z)$$

Where $\circ$ is element-wise multiplication (also known as Hadamard product).
This equation applies to all (element-wise) activation functions.

Derivatives of activation functions

Linear: $g(z) = z$ $\quad g'(z) = 1$

Sigmoid: $g(z) = \dfrac{1}{1 + e^{-z}}$ $\quad g'(z) = g(z)(1 - g(z))$

Tanh: $g(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ $\quad g'(z) = 1 - g(z)^2$

ReLU: $g(z) = \begin{cases} z & \text{for } z > 0 \\ 0 & \text{for } z \leq 0 \end{cases}$ $\quad g'(z) = \begin{cases} 1 & \text{for } z > 0 \\ 0 & \text{for } z \leq 0 \end{cases}$

Derivatives of activation functions

Softmax is normally used together with cross-entropy loss in the output layer and their joint derivative has a very nice form:

$$\hat{y}_i = softmax(z_i) \qquad softmax(z_{i,c}) = \frac{e^{z_{i,c}}}{\sum_k e^{z_{i,k}}}$$

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} log(\hat{y}_{i,c})$$

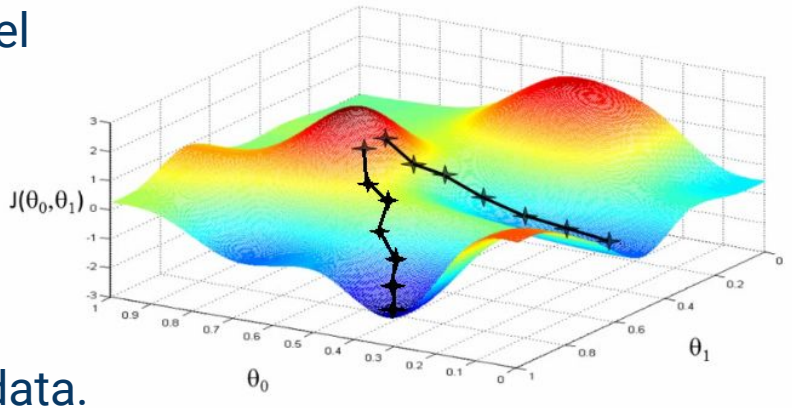
$$\frac{\partial L}{\partial z} = \frac{1}{N}(\hat{y} - y)$$

Yi is a vector, Y is a matrix (batch of vectors)

Gradient Descent

Gradient Descent

Training: the process of adjusting the model parameters to fit the given data.



Gradient descent: Repeatedly update parameters by taking small steps in the negative direction of the partial derivative, so the model gets better at predicting our data.

$$W = W - \alpha \frac{\partial L}{\partial W}$$

where α is the learning rate / step size.

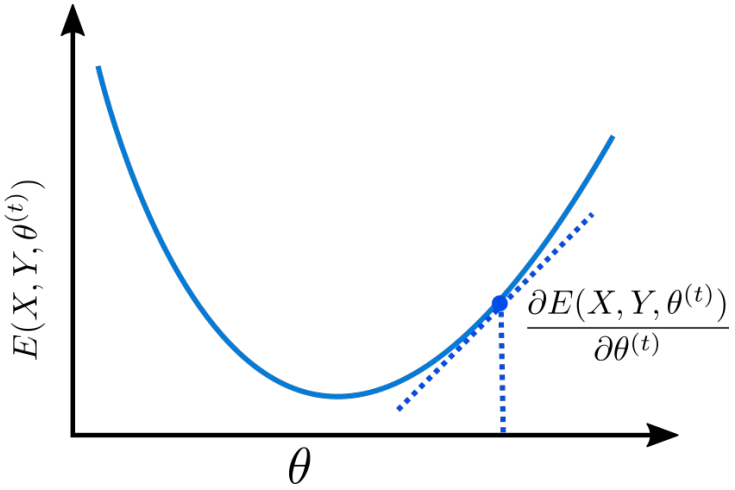
This assumes that the gradients can be computed - all our network functions and the loss need to be differentiable.

Except Perceptron

Gradient Descent

Algorithm

1. Initialize weights randomly
2. Loop until convergence:
3. Compute gradient based on the **whole dataset**:
4. Update weights
5. Finish



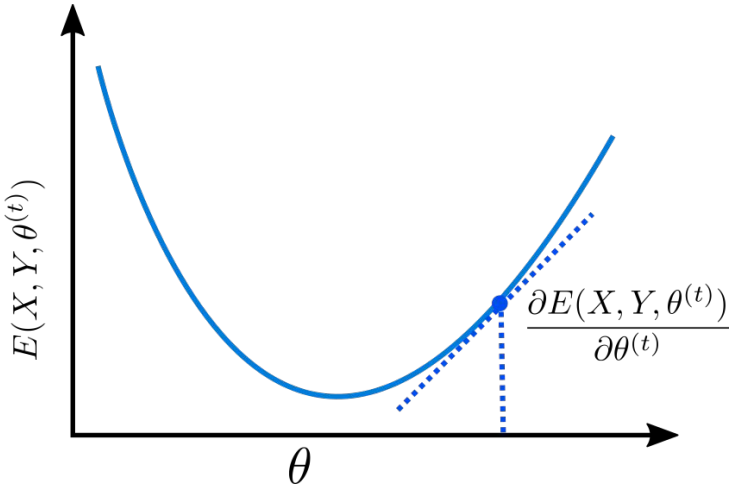
Tip: Compute all gradients before updating the weights!

In practice, datasets are often too big for this.

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly
2. Loop until convergence:
3. Loop over **each datapoint**:
4. Compute gradient based on the datapoint
5. Update weights
6. Finish

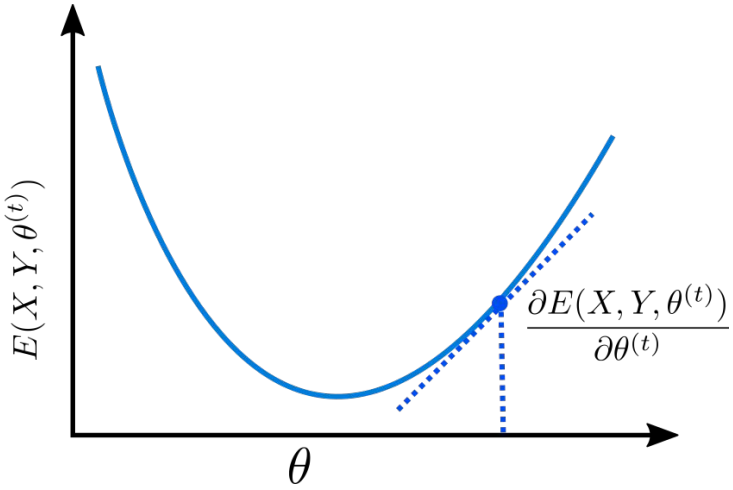


Very noisy to take steps based only on a single datapoint

Mini-batched Gradient Descent

Algorithm

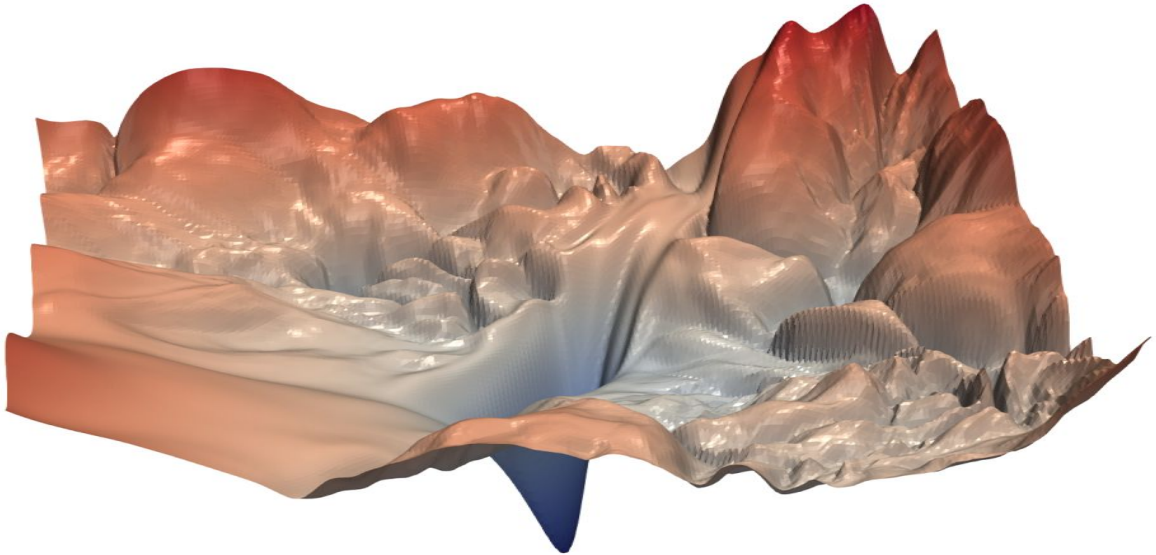
1. Initialize weights randomly
2. Loop until convergence:
3. Loop over **batches of datapoints**:
4. Compute gradient based on the batch
5. Update weights
6. Finish



This is what we mostly use in practice

Optimising Neural Networks

Neural networks have very complex loss surfaces and finding the optimum is difficult.

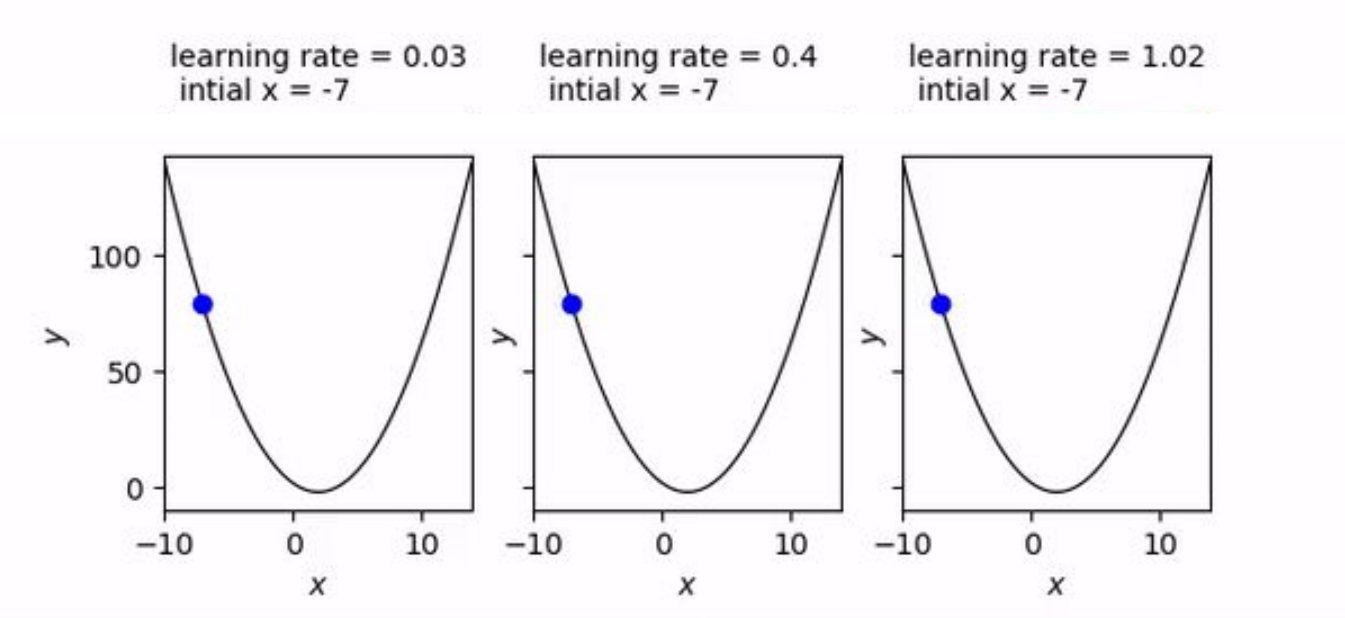


Li et al., 2018. “Visualizing the Loss Landscape of Neural Nets”

The Importance of the Learning Rate

If the learning rate is too low, the model will take forever to converge.

If the learning rate is too high, we will just keep stepping over the optimal values.



https://jed-ai.github.io/opt2_gradient_descent_1/

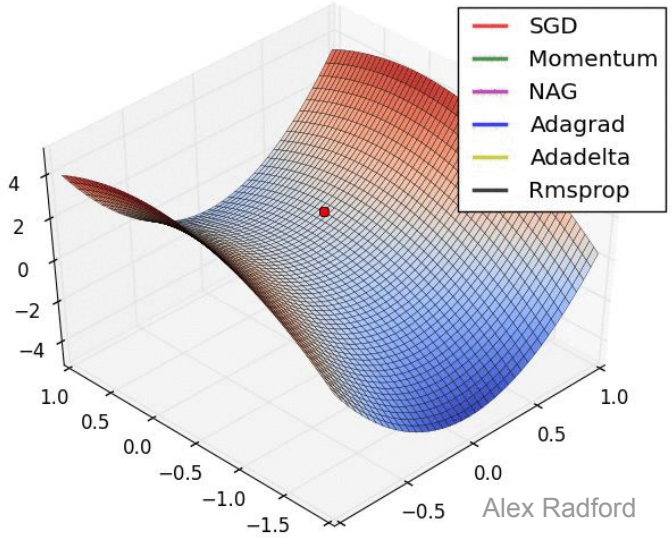
Adaptive Learning Rates

Intuition:

Have a different learning rate for each parameter.

Take bigger steps if a parameter has not been updated much recently.

Take smaller steps if a parameter has been getting many big updates.



Tip: "Adam" and "AdaDelta" work quite well.

Learning rate decay

Reducing the learning rate by a factor:

$$\alpha \leftarrow \alpha d \qquad d \in [0, 1]$$

The intuition is to take smaller steps as we get closer to the minimum, so we don't overshoot it.

Strategies for performing the update:

- Every epoch
- After a certain number of epochs
- When performance on the validation set hasn't improved for several epochs.

Weight initialisation

What values do we set the weights to before training?

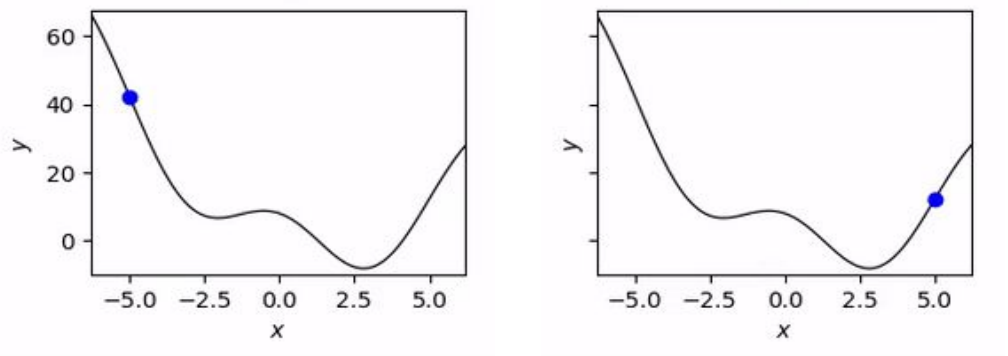
- **Zeros**: It is common to set the bias weights to zero, because we don't want the neurons to start out with a bias.
- **Normal**: Draw randomly from a normal distribution. Mean 0 and variance 1 or 0.1 are common choices.
  We want the weights to be different, or they all get the same updates!
- **Xavier Glorot** (named after one of the authors):

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}]$$

  where U is the uniform distribution, n_j the number of neurons in the previous layer, n_{j+1} the number of neurons in the next layer.

Randomness in the Network

Different **random initializations** lead to different results.



Solution: Explicitly set the random seed. All the random seeds!

BUT!

GPU threads finish in a random order, also leading to randomness! Small rounding errors really add up! Doesn't affect all operations.

Solution: Embrace randomness, run with different random seeds and report the average.

Data normalisation

Min-max normalisation: Scaling the smallest value to *a* and largest value to *b*. For example [0, 1] or [-1, 1].

$$X' = a + \frac{(X - X_{min})(b - a)}{X_{max} - X_{min}}$$

Standardization (z-normalization): Scaling the data to have mean 0 and standard deviation 1.

$$X' = \frac{X - \mu}{\sigma}$$

Normalisation helps because weight updates are **proportional to the input**.

$$\frac{\partial Loss}{\partial W} = X^T \frac{\partial Loss}{\partial Z}$$

The scaling values need to be calculated based on the **training set only**!

Gradient checking

A way to check that your gradient descent is implemented correctly. Two alternative ways to isolate the gradient.

Method 1: from the weight difference before and after gradient descent

$$w^{(t)} = w^{(t-1)} - \alpha \cdot \frac{\partial L(w)}{\partial w} \qquad \frac{\partial L(w)}{\partial w} = \frac{w^{(t-1)} - w^{(t)}}{\alpha}$$

Method 2: actually changing the weight a bit and measuring the change in loss

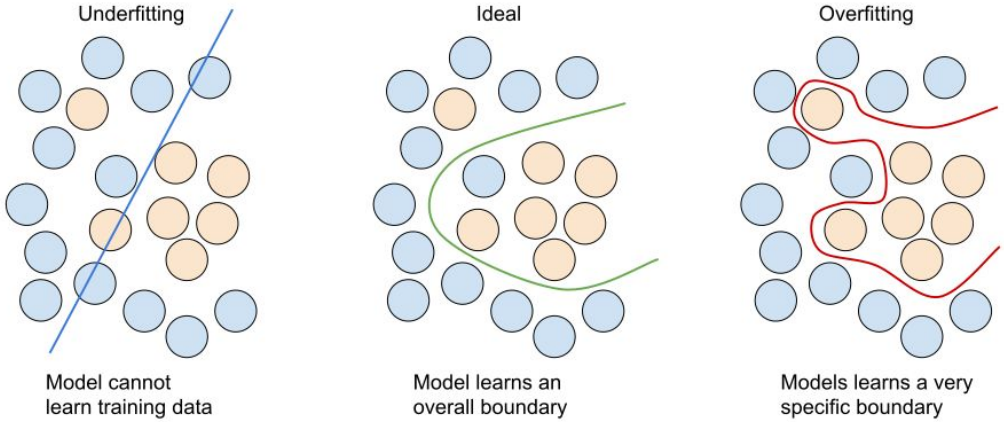
$$\frac{\partial L(w)}{\partial w} = \lim_{\epsilon \to 0} \frac{L(w+\epsilon) - L(w-\epsilon)}{2\epsilon} \quad \frac{\partial L(w)}{\partial w} \approx \frac{L(w+\epsilon) - L(w-\epsilon)}{2\epsilon}$$

Both methods should give very similar values of $\frac{\partial L(w)}{\partial w}$

Overfitting in Neural Networks

Overfitting

Neural networks with enough capacity will **overfit** to the training set quite easily.



Imagine we have 100 data points and a network with 100 neurons. Each neuron can just memorise the answer to one datapoint.

Very important to use **held-out validation and test sets**!

Network capacity

There is a correlation between the **capacity** of a neural network and its ability to overfit.

Many trainable parameters -> the model can **memorise** the data instead of learning informative patterns.

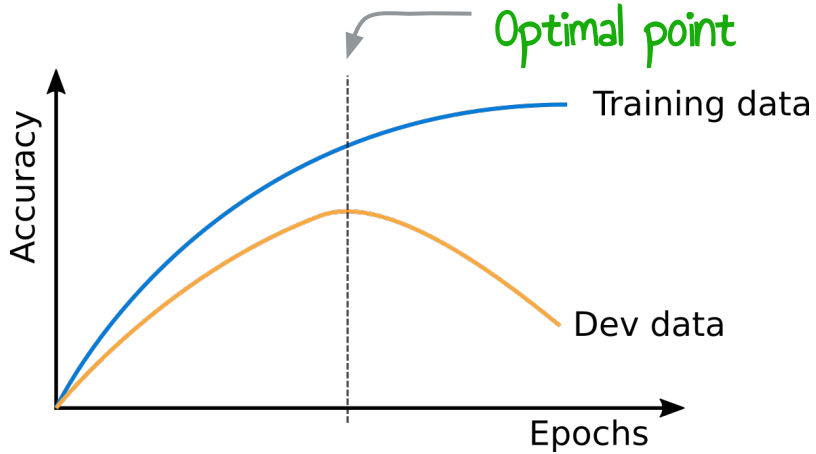
- If the network is **underfitting** on the training data, we can try increasing the number of neurons/layers.

- If the network is **overfitting** too fast, we can try reducing its capacity by lowering the number of neurons/layers.

The best solution to overfitting is to get **more data**, but that is not always possible.

Early stopping

A sufficiently powerful model will keep improving on the training data until it **overfits**.
We can use the **validation** data to choose when to stop.

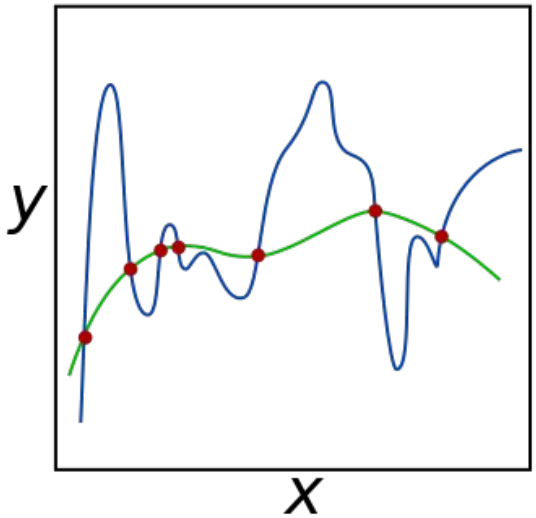


In practice: Evaluate on validation data at every epoch, always store the best model so far, stop training when performance hasn't improved for a number of epochs.

Regularisation

Regularisation: adding some information or constraints to stop the model from overfitting.

For example, penalising how large the model weights can be effectively reduces the model capacity.



L2 and L1 regularisation

L2 regularisation: adding the squared weights to the loss function. Pushing the weights towards zero, larger weights get penalised more. Encourages sharing between features, as small weights are not penalised much.

More commonly used

$$J(\theta) = Loss(y, \hat{y}) + \lambda \sum_{w} w^2 \qquad w \leftarrow w - \alpha(\frac{\partial Loss}{\partial w} + 2\lambda w)$$

Reducing the weight at each iteration

L1 regularisation: adding the absolute weights to the loss function. Encourages feature sparsity, the model will keep only the most important features.

$$J(\theta) = Loss(y, \hat{y}) + \lambda \sum_{w} |w| \qquad w \leftarrow w - \alpha(\frac{\partial Loss}{\partial w} + \lambda \text{sign}(w))$$

Try to keep as much as weights at 0

Dropout

During training, randomly set some neural activations to zero.

Typically drop 50% of activations in a layer.

During testing, use all the neurons (but scale the activations).

Form of regularisation - prevents the network from relying on any one node.