Deep learning practical 2

Glossary, notation, jupyter & colab setup, concepts

Practical overview

Welcome to the second deep learning practical. This practical has the following purpose:

- Ensure a suitable deep learning training environment (espcially for those who did not manage to get their environment setup last week)
- Review and recap the main mathematical notation & concepts used in the module
- Introduce a few foundational deep learning concepts
- Chat with the staff about any questions about the module

Environment setup

If you already have your environment set up, feel free to skip this section and move on to the next one. There are generally four ways to run the code and train the large deep learning models required for this module:

- Use Google Colab—easy to get started, but the usage gets throttled and the kernel is not persistent
- Use the NCC Jupyter server—recommended for most students. The I/O can be a little slow
- SSH to NCC and use SLURM
- Use the lab machines or your home machines—not recommended if the GPUs are inferior

Using Google Colab

- You need a Google account for this
- Simply open an existing .ipynb in Colab and run the cells—here are two to get started with:
  - Train a classifier: download
  - Train a convnet: download
- Check that your 'Runtime > Change runtime type' is a GPU

NCC usage via Jupyter (recommended)

- Connect to a machine from the university network, otherwise use the VPN to connect from outside
- Login to `ncc1.clients.dur.ac.uk/COMP0000` with your CIS username and password (if you get an authentication failure, you might not have an account on NCC and need to email Rob Powell)
- Select appropriate preferences for the job, e.g. `ug-gpu-small` with `1 GPU`
- Once it takes you to your home directory click "Assignments" tab and "Fetch" the environment creation folder
- Click the title of the folder "Create Jupyter Environment Script" and click the "CreateJupyterPythonEnvironment" script that is inside it
- In the second code cell you need to enter a Python version, and two names (these can be the same if you like). The virtualEnvName becomes the directory name of the virtualenv in your NCC home directory. The jupKernelName is what appears in the Jupyter menu later. If you're only using PyTorch you shouldn't need modules as CUDA is bundled
- Run the whole notebook (Kernel > Restart and Run All). Once it has finished (about 4-5 mins) you can close this tab and return to the previous tab that should still be on the Assignments page.
- Click "Files" to see home directory again and then refresh the page otherwise the new kernel won't appear in the list!

- Click "New" and select your kernel name from the dropdown list
- Run your `!pip install` commands (only needs to be done the first time you ever use the kernel, unlike Colab, as our storage is persistent)
- Download, upload to NCC and run the classifier.ipynb and convnet.ipynb examples

NCC usage via SSH

Only do this if you know what you're doing, e.g. have used SLURM before:

- SSH to the head node `CIS-username@ncc1.clients.dur.ac.uk` (either via VPN or Mira)
- Do not run code on this head node. Doing so will cause problems for all other users, can interrupt active postgraduate research and may get you banned
- Before running any code, read the NCC documentation carefully: `ncc1.clients.dur.ac.uk` and do your compute with SLURM
- I recommend using `tmux` for a persistent shell along with `zsh`
- Please respect other users in the queue

Mathematical notation

While notation differs slightly between papers, there has been an attempt to standardize mathematical notation in some top conferences. When writing your coursework, please try to follow the standardized ICLR conference notation guidelines here:

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
$\mathbf{A}$	A tensor
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	The parents of x_i in $\mathcal{G}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor $\mathbf{A}$
$\mathbf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Indefinite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x}\sim P}[f(x)]$ or $\mathbb{E}f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P\|Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\zeta(x)$	Softplus, $\log(1+\exp(x))$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$
$\mathbf{1}_{\mathrm{condition}}$	is 1 if the condition is true, 0 otherwise

Deep neural networks

When training deep neural networks, we mostly use an algorithm called stochastic mini batch gradient descent. The word 'stochastic' refers to the fact we randomly sample small batches of data $\mathbf{x} \sim p_{\mathrm{data}}$ from our dataset (data distribution) p_{data}. For each mini batch of data, we perform 'gradient descent', which means we iteratively make tiny changes to the parameters θ—moving them a little bit α in the negative direction of the gradient of the loss (error or cost) function that we wish to optimise. So we iteratively tweak the parameters up or down such as to minimise the loss $\mathcal{L}$ where

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} \mathcal{L}(\theta).$$

Well, that was a lie; in practice, we mostly don't use stochastic gradient descent but an algorithm called 'Adam' that adds a few extra tricks to the above (cleverly adapting α separately for each parameter), but this is the core idea.

The deep neural network itself is a differentiable function $f_\theta : \mathbb{R}^m \to \mathbb{R}^n$ with lots of parameters θ. Typical sizes are:

- 10k-100k parameters (tiny-small network size)
- 1 million parameters (medium network)

- 100 million parameters (large network)
- 1000,000,000+ (1 billion) parameters (very large network)
- 100 billion+ (corporate or crowd-funded GPT-3 sized network. GPT-3 requires 700GB of GPU memory to store its parameters)

In the past, we had to differentiate by hand—which was extremely time-consuming and error-prone for large deep neural network architectures. However, modern deep learning libraries, such as PyTorch or JAX, support automatic differentiation (usually via the backpropagation algorithm) to make the process extremely simple. Also, these libraries are optimised for parallel computation on GPUs and TPUs.

However, to effectively read the mathematical notation in deep learning papers and to understand what's happening inside these libraries, it is helpful to have a recap of the foundational calculus that is used.

Recap of foundational calculus

Note: *There's a lot of maths covered in the next few sections and it is completely natural for some to feel rusty or overwhelmed by it. For those of you, don't panic! The mathematics covered is certainly useful but mastery of it is not required to design, train and test deep neural networks.*

Derivatives

This section starts by recapping the definition of derivative for a simple real-valued function and then we extend this definition to vector-valued functions that are typically used in neural networks.

Differentiation is the operation of finding the derivative of a function.

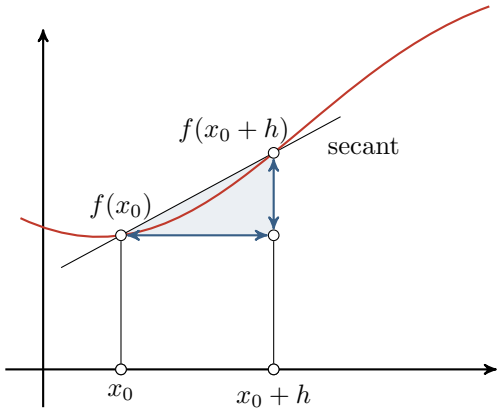
For $h > 0$ the derivative of a function $f : \mathbb{R} \to \mathbb{R}$ at x is defined as the limit

$$f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

- The derivative of a function measures the sensitivity of a functions outputs with respect to changes in inputs.
  - In deep learning, we wish to modify a neural networks parameters such as to minimise error of the network outputs.
  - Therefore, we use and calculate derivatives.
- The formal definition of the derivative is defined as a limit value as the ratio of the differences tends to 0.
  - Basically, as we take h to zero, we end up with a tangent line whose slope is the derivative.

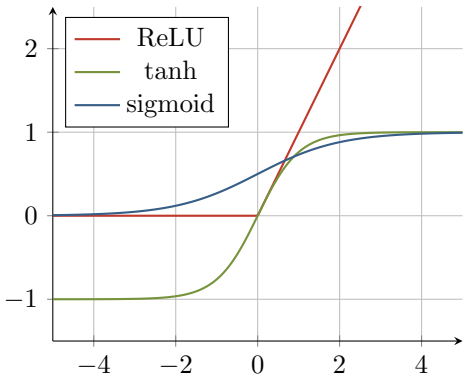
Exercise: visit: `https://www.desmos.com/calculator`

- Type $f(x) = \tanh(x)$, type $f'(x)$ for the first derivative, $f''(x)$ for the second...



Common derivatives in deep learning

In deep learning, we often chain together linear transformations with nonlinear 'activation' functions. The following are common choices for non-linearities with their associated derivatives:



1. $\text{ReLU}'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$
2. $\tanh'(x) = 1 - \tanh^2(x)$
3. $\text{sigmoid}'(x) = \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x))$
4. $\sin'(x) = \cos(x)$

Rules of differentiation

The following are the main rules of differentiation used to differentiate real-valued functions.

The **sum rule** is defined as:

$$(f(x) + g(x))' = f'(x) + g'(x)$$

The **product rule** is defined as:

$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x)$$

The **quotient rule** is defined as:

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x) - f(x)g'(x)}{(g(x))^2}$$

The **chain rule** is defined as:

$$(g(f(x)))' = (g \circ f)'(x) = g'(f(x))f'(x)$$

The **power rule** is defined as:

$$(x^n)' = nx^{n-1}$$

Exercise:

What is the derivative of $h(x) = \sin(x^2)$?

$$\begin{aligned} g(x) &= \sin(x) \\ g'(x) &= \cos(x) \\ f(x) &= x^2 \\ f'(x) &= 2x && \triangleright \text{ power rule} \\ h'(x) &= g'(f(x))f'(x) && \triangleright \text{ chain rule} \\ &= \cos(x^2)2x \end{aligned}$$

Explanation:

- We want to differentiate the function $h(x) = \sin(x^2)$
- First, we can break this down into two functions:
  1. $g(x) = \sin(x)$

2. $f(x) = x^2$

- We can calculate their derivatives respectively.
  - The derivative of $g(x) = \sin(x)$ is $g'(x) = \cos(x)$
  - The derivative of $f(x) = x^2$ is, by use of the power rule, $f'(x) = 2x^1 = 2x$
- We can then substitute these into the definition of the chain rule.
- Which gives the answer $\cos\left(x^2\right)2x$ —you may want to check these steps in desmos.

Partial derivatives

Now that we can differentiate any univariate function $f : \mathbb{R} \to \mathbb{R}$, we need to learn to differentiate multivariate functions (functions of more than one variable).

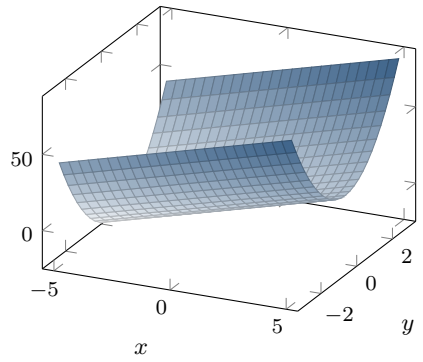
For a function $f : \mathbb{R}^n \to \mathbb{R}$ of n variables $x_1, ..., x_n$, the partial derivatives are defined as

$$\frac{\partial f}{\partial x_1} = \lim_{h \to 0} \frac{f(x_1 + h, x_2, ..., x_n) - f(\mathbf{x})}{h}$$
$$\vdots$$
$$\frac{\partial f}{\partial x_n} = \lim_{h \to 0} \frac{f(x_1, ..., x_{n-1}, x_n + h) - f(\mathbf{x})}{h}$$

which get collected into a row vector known simply as the gradient of f with respect to $\mathbf{x}$

$$\nabla_{\mathbf{x}} f = \frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}} = \left[\frac{\partial f(\mathbf{x})}{\partial x_1} \cdots \frac{\partial f(\mathbf{x})}{\partial x_n}\right] \in \mathbb{R}^{1 \times n}$$

Example function: $f(x, y) = 4x + 7y^2$



- For example, the function above has two inputs $f(x, y)$ and if you solve it, you get one output so it maps from $\mathbb{R}^n \to \mathbb{R}$
- Therefore the function admits two partial derivatives:
  1. $\frac{\partial f}{\partial x}$ read: partial derivative of f with respect to x
  2. $\frac{\partial f}{\partial y}$ read: partial derivative of f with respect to y

Calculating the partial derivatives:

- For a given function $f(x, y) = 4x + 7y^2$ go through each input (in our case x and y)
  - Write down $\frac{\partial f}{\partial x} =$
  - Write down $\frac{\partial f}{\partial y} =$
- Look at the variable in the denominator
- See if it exists in any of the terms, if it doesn't that's '0'
- If it does, differentiate it, then keep the other variables which are present

This gives

- $\frac{\partial f}{\partial x} = 4 + 0$
- $\frac{\partial f}{\partial y} = 0 + 14y$

Exercise:

Calculate $\nabla_{\mathbf{x}} f$ for $f(x, y, z) = \cos(x) + z^3 - x^2 y$:

- Write down $\frac{\partial f}{\partial x} = ... \frac{\partial f}{\partial y} = ... \frac{\partial f}{\partial z} = ...$
- Go through each variable in the denominator
- If it exists, differentiate it, if it doesn't that's a 0
- This gives:
  - $\frac{\partial f}{\partial x} = -\sin(x) + 0 - 2xy$
  - $\frac{\partial f}{\partial y} = 0 + 0 - x^2$
  - $\frac{\partial f}{\partial z} = 0 + 3z^2 - 0$

For functions with lots of inputs (like a deep neural network operating on images), we collect all these partial derivatives into a row vector.

- This vector is called the gradient of f with respect to its multiple inputs $\mathbf{x}$

Further practice and checking:

Visit: `https://www.wolframalpha.com` and type in f(x,y,z) = ... solve and check answers, especially for any functions that you're not sure about.

Rules of partial differentiation

These rules of differentiation still apply, replacing derivatives with partial derivatives

The **sum rule** is defined as:

$$\frac{\partial}{\partial \mathbf{x}}(f(\mathbf{x}) + g(\mathbf{x})) = \frac{\partial f}{\partial \mathbf{x}} + \frac{\partial g}{\partial \mathbf{x}}$$

The **product rule** is defined as:

$$\frac{\partial f}{\partial \mathbf{x}}(f(\mathbf{x})g(\mathbf{x})) = \frac{\partial f}{\partial \mathbf{x}} g(\mathbf{x}) + f(\mathbf{x}) \frac{\partial g}{\partial \mathbf{x}}$$

The **chain rule** is defined as:

$$\frac{\partial}{\partial \mathbf{x}}(g \circ f)(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}}(g(f(\mathbf{x}))) = \frac{\partial g}{\partial f} \frac{\partial f}{\partial \mathbf{x}}$$

Exercise:

Calculate the partial derivative of $z^4 - \sin\left(y^2 + x\right)$ w.r.t. y

By use of the chain rule

$$\frac{\partial}{\partial y}\left(z^4 - \sin\left(y^2 + x\right)\right) = -2y\cos\left(y^2 + x\right)$$

Also we can calculate for x and z

$$\frac{\partial}{\partial x}\left(z^4 - \sin\left(y^2 + x\right)\right) = -\cos\left(y^2 + x\right)$$
$$\frac{\partial}{\partial z}\left(z^4 - \sin\left(y^2 + x\right)\right) = 4z^3$$

Try your own and test your answers on `https://www.wolframalpha.com` e.g. type $d/dy - sin(y^2 + x)$
Rules of partial differentiation

- These rules are actually the same as the rules of differentiation, just instead of $f'(x)$ we write $\frac{\partial f}{\partial \mathbf{x}}$

- Note that the $\mathbf{x}$ is bold—which we're using to denote gradients involving vectors and matrices.
- Matrix multiplication as in the chain rule is non-commutative, and therefore the order matters!
- A minor comment - you may see the notation $\frac{\partial}{\partial \mathbf{x}}(...)$. This just means the partial of whatever function is in the brackets with respect to $\mathbf{x}$
- Another shorter example:
  - You may see $\frac{\partial}{\partial y} 3yx^2$
  - which means the partial derivative of $3yx^2$ with respect to y
  - $\frac{\partial}{\partial y} 3yx^2 = 3x^2$

The Jacobian matrix

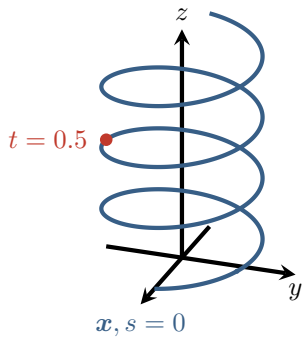
- In the previous example, we looked at differentiating functions $f : \mathbb{R}^n \to \mathbb{R}$
- Now, we want to extend our definition to differentiating vector-valued functions of domain $\mathbb{R}^n$ to the co-domain $\mathbb{R}^m$
- This is an easy extension, basically you calculate the partial derivatives for each of the functions individually with respect to each of the inputs
  - which gives an $m \times n$ matrix of partial derivatives

Jacobian matrix definition: The collection of all first-order partial derivatives of a vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$

$$\boldsymbol{J}_f = \nabla_{\mathbf{x}} f = \frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}} = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_m(\mathbf{x})}{\partial x_n} \end{bmatrix},$$

$$\boldsymbol{J}_f(i,j) = \frac{\partial f_i}{\partial x_j}$$

Example function $f : \mathbb{R}^2 \to \mathbb{R}^3$
$f(t,s) = \langle \sin(t) + s, \cos(t), \frac{6t}{\pi} \rangle$



Example:

- Consider this example function which parameterises a 3D helix curve.
- This function has two input variables, t and s, and 3 outputs corresponding to the Cartesian x, y, z coordinates.
  - So it maps from $\mathbb{R}^2 \to \mathbb{R}^3$
  - as you increase t, the x, y, z coordinates move in a 3D helical curve
  - as you change s, you can see this just translates the helix along the x-axis

Calculate the Jacobian for $f(t,s) = \langle \sin(t) + s, \cos(t), \frac{6t}{\pi} \rangle$ w.r.t. (t,s)

- The Jacobian is an $\mathbb{R}^{m\times n}$ matrix. In this case $\mathbb{R}^{3\times 2}$ where

$$\boldsymbol{J}_f = \begin{bmatrix} \frac{\partial \sin(t)+s}{\partial t} & \frac{\partial \sin(t)+s}{\partial s} \\ \frac{\partial \cos(t)}{\partial t} & \frac{\partial \cos(t)}{\partial s} \\ \frac{\partial}{\partial t}\left(\frac{6t}{\pi}\right) & \frac{\partial}{\partial s}\left(\frac{6t}{\pi}\right) \end{bmatrix} = \begin{bmatrix} \cos(t) & 1 \\ -\sin(t) & 0 \\ 6/\pi & 0 \end{bmatrix}$$

- As an exercise, cover the right side of the above and complete it yourself

Mathematics for neural networks

Now that we can calculate Jacobians to differentiate any vector-valued function, we can differentiate deep neural networks.

- These are just compositions of many layers of functions, which need to be broken down to their foundations, so we can then just use the chain rule one at a time.
- By doing this, we will be able to compute the gradient of the network objective function with respect to each of its parameters.
  - We can then update each parameter in the negative direction of the gradient, such as to minimise the error of whole network.

Multilinear maps & vector summation

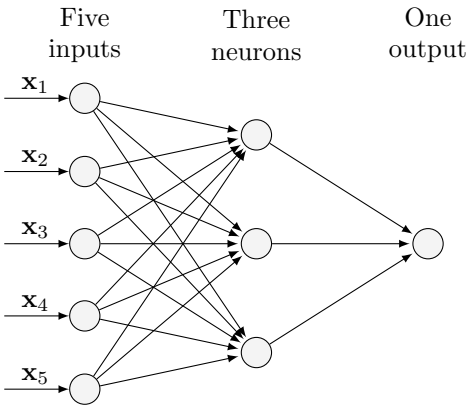
The most simple neural network function is the linear layer, consisting of a learnable weights $\boldsymbol{W}$ and biases $\mathbf{b}$.

More formally, a multilinear map is a function $f : \mathbb{R}^n \to \mathbb{R}^m$

$$f(\mathbf{x}) = \boldsymbol{W}\mathbf{x} + \mathbf{b}$$
$$\frac{\partial}{\partial \mathbf{x}}(\boldsymbol{W}\mathbf{x} + \mathbf{b}) = \boldsymbol{W}$$

Example neural network

- The diagram underneath represents a very simple example feed-forward neural network.
  - It has 5 inputs.
  - Then a hidden layer with 3 neurons.
    - Zooming in on these 3 neurons, if you remember the function of an artificial neuron was a weighted summation of each of the dendrite weights (edges) then we add a bias vector to each of the outputs;
    - ignoring the nonlinearity for now, we can express this mathematically as a multilinear map.



- So there are $5 \times 3 = 15$ edges which we represent as a 3×5 weight matrix $\boldsymbol{W}$
- the operation is to then multiply this by the $\mathbb{R}^5$ input vector $\mathbf{x}$ which gives an $\mathbb{R}^3$ output vector, we then add the $\mathbb{R}^3$ bias vector to this

- if you differentiate that w.r.t $\mathbf{x}$, you will see it's just $\boldsymbol{W}$

A simple loss function:

- Lets say those three neurons are the output prediction of the neural network
- Ultimately, we need some cost function which somehow measures the quality of the outputs' prediction and spits out a single value, which is the network error
  - for example we could take the Euclidean distance between those three outputs and some target vector, which gives a single scalar
- The cost functions usually at one point requires a vector summation, and we can see the Jacobian of the summation function is just a row vector of 1's.

More formally, vector summation is defined $f : \mathbb{R}^n \to \mathbb{R}$

$$f(\mathbf{x}) = \sum_{i=1} x_i$$

$$\boldsymbol{J}_f = \left[\frac{\partial x_1}{\partial x_1}, \frac{\partial x_2}{\partial x_2}, \cdots, \frac{\partial x_n}{\partial x_n}\right] = [1, 1, ..., 1]$$

Extension to an L_2 norm

$$\frac{\partial}{\partial x_j} f(x) = \frac{\partial}{\partial x_j} \sum_{k=1}^{n} x_k^2 = \sum_{k=1}^{n} \underbrace{\frac{\partial}{\partial x_j} x_k^2}_{\substack{=0, \text{ if } j \neq k, \\ =2x_j, \text{ else}}} = [2x_1, 2x_2, \cdots, 2x_n] = 2x_k$$

Computational graphs

Deep neural networks get very large and complicated very quickly, where we need an automatic way to differentiate them efficiently.

- The most common algorithm we use is called backpropagation, which essentially implements the reverse mode of differentiation.

Consider a neural network with one linear layer:

$$f(\mathbf{x}) = \boldsymbol{W}\mathbf{x} + \mathbf{b},$$

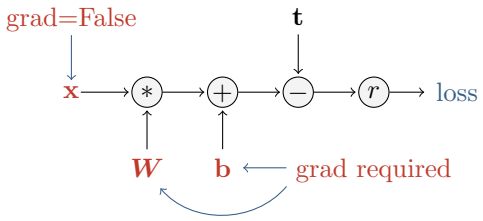
and r as the squared L_2 (Euclidean) norm:

$$r(\mathbf{x}) = ||\mathbf{x}||_2^2 = \sum_{i=1} x_i^2,$$

where the network loss function $f : \mathbb{R}^n \to \mathbb{R}$ is the cost from ground truth labels $\mathbf{t}$

$$\text{loss} = ||f(\mathbf{x}) - \mathbf{t}||_2^2 = ||(\boldsymbol{W}\mathbf{x} + \mathbf{b}) - \mathbf{t}||_2^2.$$

This is implemented as a computational graph



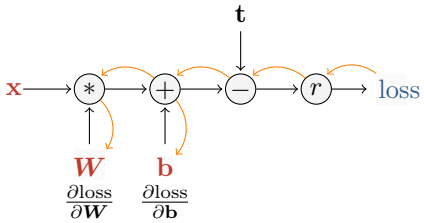
- We start by representing our function as a computational graph
  - this is a directed graph
  - nodes are operations
  - variables are fed into the nodes

  - the output is the loss, which is a single value that we wish to minimise
- The example graph you see is just an extension to what we've already differentiated manually.
  - Instead of summing the outputs, we have an actual loss function.
  - This measure outputs squared Euclidean distance between the predictions and the target labels $\mathbf{t}$
  - What we want to do is compute the gradients of the loss w.r.t the network parameters, which are leaf variables $\boldsymbol{W}$ and $\mathbf{b}$.

Backpropagation: reverse accumulation

Backpropagation is a reverse accumulation method suited for $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \ll n$ (usually $m = 1$). Despite all the maths we have gone through, implementing it using a deep library like PyTorch is quite simple. To do this in practice with a deep learning library, we:

1. set `requires_grad=True` for any parameters we want to optimise ($\boldsymbol{W}$ and $\mathbf{b}$)
2. calculate the forward pass (feed the network $\mathbf{x}$ and see the prediction) then calculate the loss (error function)
   - when doing this, retain the intermediate values from earlier layers (PyTorch will do this internally for all subsequent operations on variables with `requires_grad=True`)
3. from the loss, traverse the graph in reverse to accumulate the derivatives of the loss at the leaf nodes (in PyTorch, we simply call `loss.backward()` making all this internal math effortless.



Manual PyTorch implementation of this with iterative stochastic gradient descent $\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} \mathcal{L}(\theta)$:

Listing 1: Python

```python
class Network:
    def __init__(self): # initialise the learnable parameters
        self.W = torch.randn(64,32) * 0.01
        self.b = torch.zeros(32)
        self.W.requires_grad = True
        self.b.requires_grad = True

    def forward(self, x):
        return self.W*x + self.b

f = Network()

# train on a synthetic dataset
for i in range(100):
    x = torch.randn(32) # sample data from our dataset (has requires_grad=False)
    t = torch.ones(32)  # target label we learn to predict (has requires_grad=False)

    loss = ((f.forward(x) - t)**2).mean()

    # backpropgate gradients of the loss w.r.t to parameters W and b
    loss.backward()

    # now we have grads for W and b, update our params via a gradient descent step
    f.W.data = f.W.data - 0.1*f.W.grad.data
    f.b.data = f.b.data - 0.1*f.b.grad.data

    # zero grads for next gradient descent step
    f.W.grad.zero_()
    f.b.grad.zero_()

    print(loss.item())
```

A note on forward accumulation:

- If you've been following everything so far, to differentiate the loss w.r.t. $\boldsymbol{W}$ and $\mathbf{b}$
  - the obvious approach would be to start at the inputs and work your way down the graph. So why don't we?
  - **hang on!** if $\mathbf{x}$ has say a million values (e.g. a medium-sized image), and let's assume the next layer is the same size, then the Jacobian will be a $1,000,000 \times 1,000,000$ matrix, which will require 1TB of memory!
  - this 'forward accumulation' strategy would, however, be very efficient if the network had 1 input
    - or where $f : \mathbb{R}^n \to \mathbb{R}^m$ with $n \ll m$

    as only n sweeps are necessary
  - it's good to remember this, as there are situations where forward accumulation is more efficient, and it is supported in PyTorch (torch.autograd) and JAX
- Instead, the vast majority of deep neural networks minimise an objective loss function, which conveniently only output a single error value, so we use reverse accumulation.

Reverse accumulation:

- We use backpropagation, which is an example of dynamic programming to do the least amount of computation in cases where $m \ll n$.
- As we've already seen, after backpropagation, we can adjust the parameters $\theta = \{\boldsymbol{W}, \mathbf{b}\}$ a small amount α in the direction of the negative gradient (gradient descent).
- It turns out that this strategy, if the network only has one output, is (surprisingly) of similar complexity to the forward pass $f(x)$ itself
  - this is counter-intuitive considering how complex the expression of the partial derivatives is
  - ...and that we need to compute so many of them (for many layers with millions of parameters)
  - ...and considering how easy it is to implement $f(x)$
- Ian Goodfellow once tweeted that he can't imagine that in cases where backpropagation can be used there will ever be a more efficient algorithm, so it's probably 'here to stay'.
