Chapter 7, Part 4: Backpropagation and Automatic Differentiation

Advanced Topics in Statistical Machine Learning

Tom Rainforth

Hilary 2024

rainforth@stats.ox.ac.uk

Backpropagation and Automatic Differentiation

Being able to differentiate the empirical risk is key to deep learning as it allows us to optimize the parameters using gradient methods

In this lecture we will look at how we can actually calculate these derivatives in practice

In particular, we will look at

- **Backpropagation**: a particular way of applying the chain rule that minimizes the cost of calculating the derivatives
- **Automatic differentiation**: a programming languages tool that will allow us to perform this backpropagation automatically and forms the basis for deep learning packages like PyTorch and Tensorflow

Empirical Risk

For network f_θ with parameters θ, loss function L, regularizer r, our regularized empirical risk is[1]

$$\hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f_\theta(x_i)) + \lambda r(\theta)$$

Using the shorthand $L_i = L(y_i, f_\theta(x_i))$, the derivative is thus

$$\nabla_\theta \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L_i + \lambda \nabla_\theta r(\theta)$$

Presuming suitable choices for r, $\nabla_\theta r(\theta)$ can always be calculated straightforwardly, so the key term we need to calculate is $\nabla_\theta L_i$

[1] The loss might also depend directly on x_i but we are omitting this for brevity. We can also have unsupervised settings where the loss is simply of the form $L(f(x_i))$ or $L(x_i, f(x_i))$, for which all the ideas will equally apply.

The Chain Rule for Feed–Forward Neural Networks

An arbitrary network without loops, skip connections, or parameter sharing between layers (e.g. MLP, basic CNN) can be expressed as

$$h^0 = x, \quad h^\ell = f^\ell_{\theta^\ell}\left(h^{\ell-1}\right) \ \forall \ell = \{1, \ldots, m\}, \quad f_\theta(x) = h^m$$

We will further introduce the notation $h_i^\ell \in \mathbb{R}^{d_\ell}$ to denote the vector of hidden units values h^ℓ when the network is given input x_i

Noting the Markovian dependencies in the layers and applying the chain rule yields the series of vector and matrix products

$$\frac{\partial L_i}{\partial \theta^\ell} = \frac{\partial L_i}{\partial h_i^m} \frac{\partial h_i^m}{\partial h_i^{m-1}} \cdots \frac{\partial h_i^{\ell+1}}{\partial h_i^\ell} \frac{\partial h_i^\ell}{\partial \theta^\ell}$$

and thus

$$\frac{\partial \hat{R}(\theta)}{\partial \theta^\ell} = \frac{1}{n} \sum_{i=1}^n \frac{\partial L_i}{\partial h_i^m} \frac{\partial h_i^m}{\partial h_i^{m-1}} \cdots \frac{\partial h_i^{\ell+1}}{\partial h_i^\ell} \frac{\partial h_i^\ell}{\partial \theta^\ell} + \lambda \frac{\partial r(\theta)}{\partial \theta^\ell} \qquad (1)$$

Individual Terms

Breaking down the terms in $\frac{\partial L_i}{\partial \theta^\ell}$, we have that, given input x_i,

- $\frac{\partial L_i}{\partial h_i^m}$ is a **row** vector with d_m elements representing the derivative of the loss with respect to our set of output units $h_i^m = f_\theta(x_i)$ (note that even if our output is multi-dimensional, our loss is still a scalar)
- Each $\frac{\partial h_i^k}{\partial h_i^{k-1}}$ is a $d_k \times d_{k-1}$ matrix representing the Jacobian of $f_{\theta^k}^k$ with respect to the input h_i^{k-1}
- $\frac{\partial h_i^\ell}{\partial \theta^\ell}$ is a $d_\ell \times p_\ell$ matrix (where p_ℓ is the number of parameters in θ^ℓ) representing the Jacobian of $f_{\theta^\ell}^\ell$ with respect to its parameters θ^ℓ

Computation Order

Presuming that the loss, regularizer, and each layer in our network is differentiable (with respect to both its inputs and parameters), we can directly calculate the overall derivative by calculating each such term individual and then combining them as per (1)

However, the order of the computations will massively change the cost: we can either use the breakdown

$$\frac{\partial L_i}{\partial \theta^\ell} = \left(\left(\left(\frac{\partial L_i}{\partial h_i^m}\frac{\partial h_i^m}{\partial h_i^{m-1}}\right)\cdots\frac{\partial h_i^{\ell+1}}{\partial h_i^\ell}\right)\frac{\partial h_i^\ell}{\partial \theta^\ell}\right) \tag{2}$$

or

$$\frac{\partial L_i}{\partial \theta^\ell} = \left(\frac{\partial L_i}{\partial h_i^m}\left(\frac{\partial h_i^m}{\partial h_i^{m-1}}\cdots\left(\frac{\partial h_i^{\ell+1}}{\partial h_i^\ell}\frac{\partial h_i^\ell}{\partial \theta^\ell}\right)\right)\right) \tag{3}$$

Computational Cost

For (2) and (3) we now respectively have the following costs[2]

$$
\begin{aligned}
(2): \quad & O\left(d_m d_{m-1} + d_{m-1} d_{m-2} + \cdots + d_{\ell+1} d_\ell + d_\ell p_\ell\right) \\
&\qquad = O\left(d_\ell p_\ell + \sum\nolimits_{k=\ell+1}^{m} d_k d_{k-1}\right) \\
(3): \quad & O\left(d_{\ell+1} d_\ell p_\ell + d_{\ell+2} d_{\ell+1} p_\ell + \cdots + d_m d_{m-1} p_\ell + d_m p_\ell\right) \\
&\qquad = O\left(d_m p_\ell + \sum\nolimits_{k=\ell+1}^{m} d_k d_{k-1} p_\ell\right)
\end{aligned}
$$

The latter is roughly p_ℓ times more costly; as p_ℓ could easily be a million or more parameters, this is a massive difference!

Calculating our derivatives by applying the chain rule in the former order is known as **backpropagation** in the deep learning literature

[2] In many common cases, e.g. a fully connected layer, $\partial h_i^\ell / \partial \theta^\ell$ is sparse and so the cost for (2) can be further reduced to $O\left(p_\ell + \sum_{k=\ell+1}^{m} d_k d_{k-1}\right)$

Backpropagation

The name backpropagation derives from the fact that we calculate our derivatives in a **backwards** fashion for the network:

- We first calculate the gradient of loss derivatives with respect to the last layer $\frac{\partial L_i}{\partial h_i^m} (= \frac{\partial L_i}{\partial f(x_i)})$
- We then go backwards through the network and **recursively** calculate the **vector–matrix products**

$$\frac{\partial L_i}{\partial h_i^\ell} = \frac{\partial L_i}{\partial h_i^{\ell+1}} \frac{\partial h_i^{\ell+1}}{\partial h_i^\ell} \quad \forall \ell = m-1, m-2, \dots, 1$$

- From these we can calculate the empirical risk derivatives for each set of parameters as

$$\frac{\partial \hat{R}(\theta)}{\partial \theta^\ell} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L_i}{\partial h_i^\ell} \frac{\partial h_i^\ell}{\partial \theta^\ell} + \lambda \frac{\partial r(\theta)}{\partial \theta^\ell}$$

Backpropagation in Sum Notation

When performing backpropagation manually, it can sometimes be helpful to express these rules as summations rather than vector matrix products. Namely, using the shorthand $h_{ij}^{\ell} = \left\{h_i^{\ell}\right\}_j$, we have

$$\frac{\partial L_i}{\partial h_{ij}^{\ell}} = \sum_{k=1}^{d_{\ell+1}} \frac{\partial L_i}{\partial h_{ik}^{\ell+1}} \frac{\partial h_{ik}^{\ell+1}}{\partial h_{ij}^{\ell}} \tag{4}$$

$$\frac{\partial \hat{R}(\theta)}{\partial \theta^{\ell}} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{d_{\ell}} \frac{\partial L_i}{\partial h_{ij}^{\ell}} \frac{\partial h_{ij}^{\ell}}{\partial \theta^{\ell}} + \lambda \frac{\partial r(\theta)}{\partial \theta^{\ell}} \tag{5}$$

Note that backpropagation is a **recursive algorithm**: we do **not** algebraically rollout the equation, but instead step backwards and separately calculate the **values** of the gradient at each layer from those at the next for each $\{x_i, y_i\}$

Backpropagation Example

Consider an MLP with squared loss $L(y_i, f(x_i)) = (y_i - f_\theta(x_i))^2$ and $\tanh$ activations (note $\frac{d\tanh(a)}{da} = 1 - \tanh^2 a$). Here we have

$$\frac{\partial L_i}{\partial h_i^m} = 2(h_i^m - y_i) \quad \text{(can be a scalar or a row vector)}$$

$$\frac{\partial L_i}{\partial h_{ij}^\ell} = \sum_{k=1}^{d_{\ell+1}} \frac{\partial L_i}{\partial h_{ik}^{\ell+1}} \frac{\partial h_{ik}^{\ell+1}}{\partial h_{ij}^\ell} \qquad \forall \ell \in \{1, \ldots, m-1\}$$

$$\begin{aligned} \frac{\partial h_{ik}^{\ell+1}}{\partial h_{ij}^\ell} &= \frac{\partial}{\partial h_{ij}^\ell} \tanh\left(\sum\nolimits_{t=1}^{d_\ell} W_{kt}^{\ell+1} h_{it}^\ell + b_k^{\ell+1}\right) \\ &= W_{kj}^{\ell+1}\left(1 - \left(h_{ik}^{\ell+1}\right)^2\right) \end{aligned}$$

$$\implies \frac{\partial L_i}{\partial h_{ij}^\ell} = \sum_{k=1}^{d_{\ell+1}} \frac{\partial L_i}{\partial h_{ik}^{\ell+1}} W_{kj}^{\ell+1}\left(1 - \left(h_{ik}^{\ell+1}\right)^2\right)$$

Backpropagation Example (2)

These can be **recursively** calculated and we further have

$$\frac{\partial \hat{R}(\theta)}{\partial \theta^\ell} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d_\ell}\frac{\partial L_i}{\partial h_{ij}^\ell}\frac{\partial h_{ij}^\ell}{\partial \theta^\ell} + \lambda\frac{\partial r(\theta)}{\partial \theta^\ell}$$

$$\frac{\partial h_{ij}^\ell}{\partial W_{ts}^\ell} = \mathbb{I}(t=j)h_{is}^{\ell-1}\left(1-\left(h_{ij}^\ell\right)^2\right)$$

$$\implies \frac{\partial \hat{R}(\theta)}{\partial W_{ts}^\ell} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial h_{it}^\ell}h_{is}^{\ell-1}\left(1-\left(h_{it}^\ell\right)^2\right) + \lambda\frac{\partial r(\theta)}{\partial W_{ts}^\ell}$$

$$\frac{\partial h_{ij}^\ell}{\partial b_t^\ell} = \mathbb{I}(t=j)\left(1-\left(h_{ij}^\ell\right)^2\right)$$

$$\implies \frac{\partial \hat{R}(\theta)}{\partial b_t^\ell} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial h_{it}^\ell}\left(1-\left(h_{it}^\ell\right)^2\right) + \lambda\frac{\partial r(\theta)}{\partial b_t^\ell}$$

Automatic Differentiation (AutoDiff)

- Many modern systems can calculate derivatives for you automatically, even for large complex programs, using a method called **automatic differentiation** (AutoDiff)
  - Most popular frameworks: **PyTorch** and **Tensorflow**
- In practice you never need to manually calculate network derivatives (except potentially in exams)!
- AutoDiff is exact (i.e. it is not a numerical approximation) but it is not symbolic either
- It has two forms: **forward mode** and **reverse mode**
- The latter is cheap and scalable for deep learning: calculating derivatives only adds a small **constant** factor to the runtime compared to simply evaluating the program itself
- It still works for network that the do not satisfy our earlier feed–forward assumptions (e.g. RNNs, ResNets)

Reverse Mode AutoDiff Performs Backpropagation

Consider an arbitrary node u in a computation graph and assume that its child nodes are $v_{1:n_c}$

By the chain rule, the gradient of some arbitrary downstream node in the computation graph, z, with respect to u is given by

$$\frac{\partial z}{\partial u} = \sum_{j=1}^{n_c} \frac{\partial z}{\partial v_j} \frac{\partial v_j}{\partial u}$$

Here $\frac{\partial v_j}{\partial u}$ is a local computations and so if all the $\frac{\partial z}{\partial v_j}$ are known, the above allows us to calculate $\frac{\partial z}{\partial u}$ using only the current value of u and its relationship with its children.

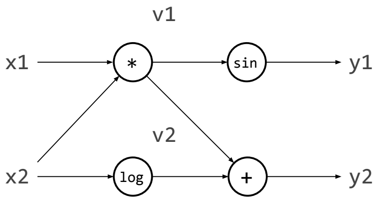
If all parent–child derivatives are known, we can calculate the partial derivatives of z with respect to **all** nodes by **recursively** calculating the derivatives of parent nodes from child their nodes

Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```



Slide Credit: Gunes Baydin

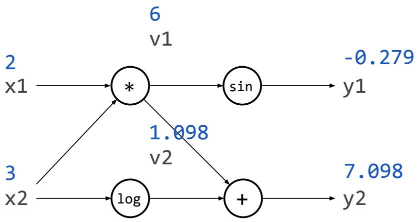
Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



Slide Credit: Gunes Baydin

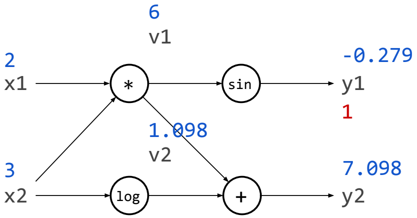
Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial y_1} = 1$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

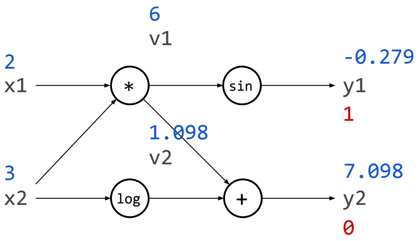
Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial y_2} = 0$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

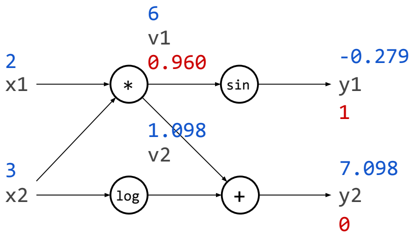
Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial v_1} = \frac{\partial y_1}{\partial v_1}\frac{\partial y_1}{\partial y_1} + \frac{\partial y_2}{\partial v_1}\frac{\partial y_1}{\partial y_2} = \cos(v_1)\frac{\partial y_1}{\partial y_1}$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

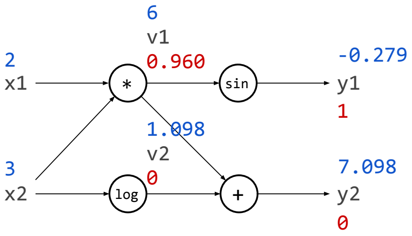
Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial v_2} = \frac{\partial y_2}{\partial v_2}\frac{\partial y_1}{\partial y_2} = 0$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

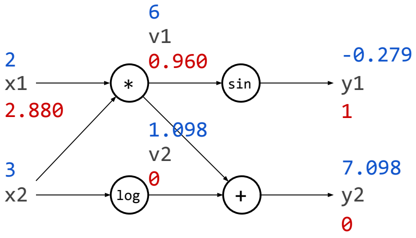
Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial x_1} = \frac{\partial v_1}{\partial x_1}\frac{\partial y_1}{\partial v_1} = x_2\frac{\partial y_1}{\partial v_1}$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

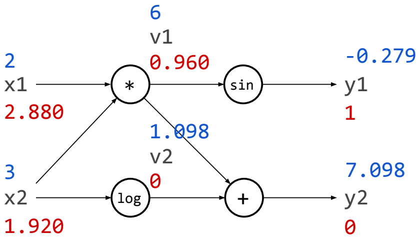
Reverse Mode AutoDiff Example

Presume we want to calculate derivatives for y1

```
f(x1, x2):
  v1 = x1 * x2
  v2 = log(x2)
  y1 = sin(v1)
  y2 = v1 + v2
  return (y1, y2)

f(2, 3)
```

Forward function calculations in blue



$$\frac{\partial y_1}{\partial x_2} = \frac{\partial v_1}{\partial x_2}\frac{\partial y_1}{\partial v_1} + \frac{\partial v_2}{\partial x_2}\frac{\partial y_2}{\partial v_2} = x_1\frac{\partial y_1}{\partial v_1}$$

Reverse derivative calculations in red

Slide Credit: Gunes Baydin

Creating an AutoDiff System

Consider an arbitrary primitive operation `pr` that takes in inputs $u_1, \ldots, u_{n_u}$ and returns outputs $v_1, \ldots, v_{n_v}$. Assume for simplicity that each $u_j, v_j \in \mathbb{R}$.

To construct a reverse mode AutoDiff system, each such `pr` must have two associated methods:

- The forward calculation $(v_1, \ldots, v_{n_v}) = \texttt{pr}(u_1, \ldots, u_{n_u})$
- A **vector–Jacobian product** calculator, `vjp`, that takes an additional input Δ of the same size as v, such that

$$\texttt{pr.vjp}(u, \Delta) = \Delta^T \frac{\partial v}{\partial u} = \left[\sum_{j=1}^{n_v} \Delta_j \frac{\partial v_j}{\partial u_1}, \ldots, \sum_{j=1}^{n_v} \Delta_j \frac{\partial v_j}{\partial u_{n_u}} \right]$$

  where $\frac{\partial v}{\partial u}$ is a Jacobian of `pr` whose i, j^{th} entry is $\frac{\partial v_i}{\partial u_j}$ and we are implicitly evaluating these at the provided input values

Creating an AutoDiff System (2)

Example: product operator primitive (with $u \in \mathbb{R}^2$, $v \in \mathbb{R}$)

$$\texttt{prod}(u_1, u_2) = u_1 u_2 \qquad \texttt{prod.vjp}\,(u, \Delta) = [u_2\Delta, u_1\Delta]$$

If our language only allows such primitives, we can perform reverse mode AutoDiff on arbitrary computation graphs: if Δ is the set of required derivatives for the output nodes, $\texttt{pr.vjp}\,(u, \Delta)$ produces the set of required derivatives for the input nodes

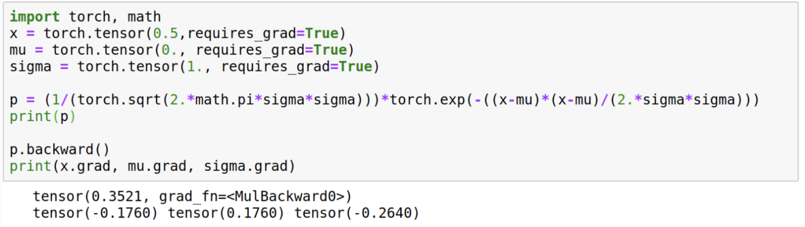
We can also implicitly define a `vjp` operator for compound operations by running AutoDiff on the low–level graph, e.g.

$$h_i^m = f_{\theta^m}^m(h_i^{m-1}) \qquad f_{\theta^m}^m.\texttt{vjp}\left(h_i^{m-1}, \frac{\partial L_i}{\partial h_i^m}^T\right) = \frac{\partial L_i}{\partial h_i^m}\frac{\partial h_i^m}{\partial h_i^{m-1}}$$

Most systems further allow arbitrary tensor sizes for u and v, e.g. so that we can batch computation across multiple different i.

AutoDiff in PyTorch

In PyTorch, AutoDiff allows us to calculate gradients by calling `.backward()` on the term we wish to calculate gradients for



```python
import torch, math
x = torch.tensor(0.5,requires_grad=True)
mu = torch.tensor(0., requires_grad=True)
sigma = torch.tensor(1., requires_grad=True)

p = (1/(torch.sqrt(2.*math.pi*sigma*sigma)))*torch.exp(-((x-mu)*(x-mu)/(2.*sigma*sigma)))
print(p)

p.backward()
print(x.grad, mu.grad, sigma.grad)
```

```
tensor(0.3521, grad_fn=<MulBackward0>)
tensor(-0.1760) tensor(0.1760) tensor(-0.2640)
```

Credit: Güneș Baydin

In the context of deep learning, given a forward model, we can introduce a variable `loss` corresponding to $L(y_i, f_\theta(x_i))$ (typically as vector over multiple i) and the simply call `loss.backward()` to calculate the gradients of all our network parameters

Recap

- The **order** we do computations in is critically important when using the chain rule
- **Backpropagation** allows us to efficiently calculate all the required gradients for a neural network using the **recursive** equations (for feed–forward networks)

$$\frac{\partial L_i}{\partial h_i^\ell} = \frac{\partial L_i}{\partial h_i^{\ell+1}} \frac{\partial h_i^{\ell+1}}{\partial h_i^\ell}$$

$$\frac{\partial \hat{R}(\theta)}{\partial \theta^\ell} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L_i}{\partial h_i^\ell} \frac{\partial h_i^\ell}{\partial \theta^\ell} + \lambda \frac{\partial r(\theta)}{\partial \theta^\ell}$$

- In practice, we do not need to calculate these manually, as **AutoDiff** systems allow the gradient calculations to be performed automatically

Further Reading

- Helpful videos by 3Blue1Brown https://youtu.be/Ilg3gGewQ5U and https://youtu.be/tIeHLnjs5U8
- Lecture 4 from Stanford course (https://youtu.be/d14TUNcbn1k)
- Güneș Baydin's slides on automatic differentiation https://www.cs.ox.ac.uk/teaching/courses/2019-2020/advml/ and tutorial paper https://www.jmlr.org/papers/volume18/17-468/17-468.pdf
- Have a play with PyTorch and/or Tensorflow!