# Deep Learning and NLP for Diagnosis

Lecture Notes

Should be considered as a cheat sheet alongside the lecture slides rather than a self-sufficient document. Most of the intuition (arguably the most important thing you should focus on) behind things comes from attending the lectures. There might be typos, send me an email at romain.lhotte@centralesupelec.fr if there are any.

Romain Lhotte

2025-2026

# Contents

**1 Python and Mathematical Prerequisites** **1**
1.1 Python classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Dunder methods (magic methods) . . . . . . . . . . . . . . . . . . . . . . . 1
1.3 Inheritance and `nn.Module` . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.4 Derivatives in one dimension . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.5 Gradients and partial derivatives . . . . . . . . . . . . . . . . . . . . . . . 3
1.6 Rank of a matrix and the special case $aa^\top$ . . . . . . . . . . . . . . . . . . . . 4

**2 PyTorch Essentials** **7**
2.1 Tensors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Autograd: automatic differentiation . . . . . . . . . . . . . . . . . . . . . 7
2.3 `nn.Module` and model definition . . . . . . . . . . . . . . . . . . . . . . . 10
2.4 Optimisers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.5 Datasets and DataLoaders . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

**3 From Neurons to Deep Networks** **13**
3.1 Biological vs artificial neurons . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Width, depth, and layers . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.3 Forward, backward, and parameter updates . . . . . . . . . . . . . . . . . . 14
    3.3.1 Backward without PyTorch . . . . . . . . . . . . . . . . . . . . . . 14
    3.3.2 Backward with PyTorch . . . . . . . . . . . . . . . . . . . . . . . . 16
3.4 Train / validation / test sets . . . . . . . . . . . . . . . . . . . . . . . . 16

**4 Optimisation and Training Tricks** **17**
4.1 Stochastic gradient descent . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.2 Momentum and Adam . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.3 Learning-rate schedules . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.4 Regularization: gradient clipping, batch norm, dropout . . . . . . . . . . . . 18
4.5 Loss functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.6 Softmax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**5 Linear Layers and Universal Approximation** **21**
5.1 Universal approximation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.2 Wide vs deep networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**6 Convolutions, Pooling, and Softmax** **25**
6.1 Why convolution? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
6.2 Padding, stride, dilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
6.3 Pooling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
6.4 Altogether . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
6.5 Data augmentation and imbalance . . . . . . . . . . . . . . . . . . . . . . . 28
    6.5.1 Data augmentation . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

6.5.2 Imbalanced data . . . . . . 29

**7 Architectures and Transfer Learning** **31**
7.1 Typical classification architecture . . . . . . 31
7.2 Segmentation and U-Net . . . . . . 32
7.3 Variable-length sequences and RNNs . . . . . . 33
7.4 Going from a string to numbers . . . . . . 34
7.5 Sequence-to-sequence with RNNs . . . . . . 36
7.6 Transfer learning with `torchvision` . . . . . . 37

**8 Attention, Transformers** **39**
8.1 Why attention? How attention? . . . . . . 39
8.2 Transformers (Focus on the scaled dot-product attention) . . . . . . 40
8.3 Transformers (Focus on the Positional Encoding) . . . . . . 41
8.3.1 Sinusoidal positional encoding . . . . . . 41
8.3.2 A concrete numerical example . . . . . . 42

**9 GPT and Low-Rank Adaptation (LoRA)** **43**
9.1 Finetuning GPT-style language models is hard . . . . . . 43
9.2 Easy Tricks . . . . . . 43
9.3 LoRA: low-rank adaptation . . . . . . 43

## Chapter 1

# Python and Mathematical Prerequisites

## 1.1 Python classes

Python is an object-oriented language: you group data (attributes) and behavior (methods) into *classes*. Then you can make *instances* of that class, and use the attributes/methods you defined.

**A minimal Python class**

```
class LinearModel:
    def __init__(self, w, b):
        # parameters of the model y = w * x + b
        self.w = w
        self.b = b

    def predict(self, x):
        return self.w * x + self.b

# Instantiate an object from that class
model = LinearModel(w=2.0, b=0.5)
# Use a method
print(model.predict(3.0))  # 6.5
# Retrieve some data from that class
print(model.w)  # 2.0
```

Key ideas:

- `__init__` is the constructor: basically it runs when you create the object.
- `self` is the object itself (so `self.w` is how we get we get `w` in the method `predict` even though there is no `w` in the argument list of `predict`).
- Methods are just functions whose first argument is `self`.

## 1.2 Dunder methods (magic methods)

**Dunder** = “double underscore”. Python uses special method names to overload behavior. Common ones you’ll see in deep learning code:

- `__len__`: it’s a method that will be called when you do `len(obj)`.

- `__getitem__`: it's a method that will be called when you do `obj[i]`.

PyTorch `Dataset` makes heavy use of these:

**A tiny Dataset using dunder methods**

```
import torch
from torch.utils.data import Dataset

class SimpleDataset(Dataset):
    def __init__(self, data, targets):
        self.data = torch.tensor(data, dtype=torch.float32)
        self.targets = torch.tensor(
            targets,
            dtype=torch.float32
        )

    def __len__(self):
        # number of samples
        return len(self.data)

    def __getitem__(self, idx):
        # one sample (x, y)
        return self.data[idx], self.targets[idx]
```

## 1.3 Inheritance and `nn.Module`

Inheritance lets you build a new class from an existing one. The stuff that was written in the existing class will be transferred to your new class without you having to write anything. In PyTorch, all neural networks should be subclasses of `nn.Module` (the way to see it is that they did loads of things for you already, you may as well inherit from it).

**A minimal PyTorch module**

```
import torch
import torch.nn as nn

class TinyMLP(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        # Call the __init__ of the parent class
        super().__init__()

        # Do your __init__ stuff
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(hidden_features, out_features)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.fc2(x)
        return x

model = TinyMLP(10, 32, 1)
```

```
x = torch.randn(4, 10)
y = model(x)
print(y.shape)  # torch.Size([4, 1])
```

Takeaways:

- Always call `super().__init__()` in your constructor.
- Put layers as attributes (e.g. `self.fc1`).
- Implement `forward(self, x)` for the computation.

## 1.4 Derivatives in one dimension

**Definition 1.1: Derivative in one dimension**

Let $f : \mathbb{R} \to \mathbb{R}$. The derivative of $f$ at point $x$ is

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

when this limit exists. Intuitively, $f'(x)$ is the slope of the curve $y = f(x)$ at the point $x$. That's important because if we have the slope, we can use the slope to know where to go to make our function smaller. This is Machine Learning 101. If the slope is positive, we should decrease $x$ to minimise our function. The functions we'll consider here are the losses (basically how much our model is wrong) and we want to change $x$ to eventually have fewer classification errors for example[a].

---

[a] Technically, you could also use evolutionary algorithms (`https://en.wikipedia.org/wiki/Neuroevolution`) to find the minimum without needing to ever use a gradient. You could also use Newton's method to find $x$ such that $f'(x) = 0$ (assuming that that's enough to find the minimum which it isn't). The reason why we used to do it a little bit (when I was in primary school and you in kindergarten?), and we don't anymore is detailed in `NewtonMachineLearning.pdf`.

Example: $f(x) = x^2$. Then $f'(x) = 2x$.

PyTorch can compute derivatives for you with `autograd`:

**Scalar autograd example**

```
import torch

x = torch.tensor(1.0, requires_grad=True)
y = x**2  # y = f(x)

y.backward()      # computes dy/dx at x = 1.0
print(x.grad)     # tensor(2.) which is 2 * 1
```

## 1.5 Gradients and partial derivatives

For a function of many variables $f : \mathbb{R}^d \to \mathbb{R}$, the *gradient* at $\boldsymbol{x} = (x_1, \ldots, x_d)$ is the vector of partial derivatives.

**Definition 1.2: Gradient and partial derivatives**

Let $f : \mathbb{R}^d \to \mathbb{R}$. The partial derivative w.r.t. $x_i$ is

$$\frac{\partial f}{\partial x_i}(\boldsymbol{x}) = \lim_{h \to 0} \frac{f(x_1, \dots, x_i + h, \dots, x_d) - f(x_1, \dots, x_i, \dots, x_d)}{h}.$$

The gradient is

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1}(\boldsymbol{x}) \\ \vdots \\ \dfrac{\partial f}{\partial x_d}(\boldsymbol{x}) \end{bmatrix}.$$

PyTorch handles vector gradients too:

**Vector autograd example**

```
import torch

x = torch.tensor([1.0, 2.0], requires_grad=True)

# f(x1, x2) = x1**2 + 3 * x2
f = x[0]**2 + 3 * x[1]

f.backward()  # computes grad f with regards to x
print(x.grad)
# Output: tensor([2.0, 3.0]),
# make sure you find the same result manually.
```

Here, `x.grad` contains $\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$.

## 1.6 Rank of a matrix and the special case $aa^\top$

Informal idea: the *rank* of a matrix is the number of independent directions it can generate.

**Definition 1.3: Rank of a matrix**

Let $A \in \mathbb{R}^{m \times n}$. The rank of $A$, denoted $\operatorname{rank}(A)$, is the dimension of the vector space spanned by its columns (equivalently, by its rows).

Now consider a column vector $a \in \mathbb{R}^n$. Form the matrix

$$A = aa^\top \in \mathbb{R}^{n \times n}.$$

**Theorem 1.1: Rank of $aa^\top$**

Let $a \in \mathbb{R}^n$ be a non-zero column vector and $A = aa^\top$. Then

$$\operatorname{rank}(A) = 1.$$

If $a = 0$, then $\operatorname{rank}(A) = 0$.

**Proof.** First, look at the columns of $A$. The $j$-th column of $A$ is

$$A_{\cdot j} = a\, a_j,$$

i.e. scalar $a_j$ times the vector $a$. So every column of $A$ is a scalar multiple of the same vector $a$. Therefore the column space of $A$ is

$$\mathcal{C}(A) = \{\lambda a : \lambda \in \mathbb{R}\} = \operatorname{span}\{a\}.$$

If $a \neq 0$, the set $\{a\}$ is linearly independent and its span has dimension 1. Hence $\operatorname{rank}(A) = 1$.
If $a = 0$, then $A$ is the zero matrix and its column space is just $\{0\}$, whose dimension is 0. So $\operatorname{rank}(A) = 0$.

This "outer product" pattern (vector times vector transposed) is exactly what is exploited in low-rank adaptation methods such as LoRA as we'll see later in our class. For now this probably means nothing to you, but when you read the whole thing again at the end of the class, this should make sense.

We can check this numerically with PyTorch:

**Checking that $aa^\top$ is rank 1 in PyTorch**

```
import torch

a = torch.tensor([1.0, 2.0, 3.0])
A = torch.outer(a, a)  # A = a a^T
print(A)

rank = torch.linalg.matrix_rank(A)
print(rank)  # tensor(1)
```

# Chapter 2

# PyTorch Essentials

## 2.1 Tensors

Tensors are the core data structure: multi-dimensional arrays, similar to NumPy arrays but living on CPU or GPU.

**Creating tensors**

```
import torch

x = torch.tensor([1.0, 2.0, 3.0])  # 1D tensor (vector)
M = torch.zeros(2, 3)  # 2*3 matrix
T = torch.randn(4, 3, 32, 32)  # 4 images (each image is 3*32*32)
print(T.device)  # cpu (because we didn't ask it to be moved)
```

**Moving tensors to the GPU**

```
import torch

# 1. Check if a GPU is available
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

# 2. Create a tensor and move it to device
#    (obviously useless if you don't have a GPU)
x = torch.tensor([1.0, 2.0, 3.0])
print(f"Original device: {x.device}")
x = x.to(device)
print(f"New device: {x.device}")
```

## 2.2 Autograd: automatic differentiation

PyTorch builds a computation graph and applies the chain rule. A computation graph is a directed acyclic graph where nodes represent tensors and edges represent the mathematical operations that produce them. When you define $z = (x/y) + \cos(x) \cdot \sin(y)$, PyTorch remembers the order of operations so it can "walk backward" through them later. It notes that $x$ went through a cos() function. It notes that the result of $x/y$ was added to something else. When you later call `z.backward()`, PyTorch starts at the end ($z$) and traverses the graph in the exact opposite direction. The first thing it sees is the addition, which will lead to two branches, etc ...

**Autograd with multiple inputs**

```
import torch

x = torch.tensor(1.0, requires_grad=True)
y = torch.tensor(5.0, requires_grad=True)

z = (x / y) + torch.cos(x) * torch.sin(y)
z.backward()

print(x.grad, y.grad)
```

x.grad will contain $\frac{\partial z}{\partial x}(x, y)$. Similarly, y.grad will contain $\frac{\partial z}{\partial y}(x, y)$.
The underlying graph PyTorch builds is shown in Figure 2.1.

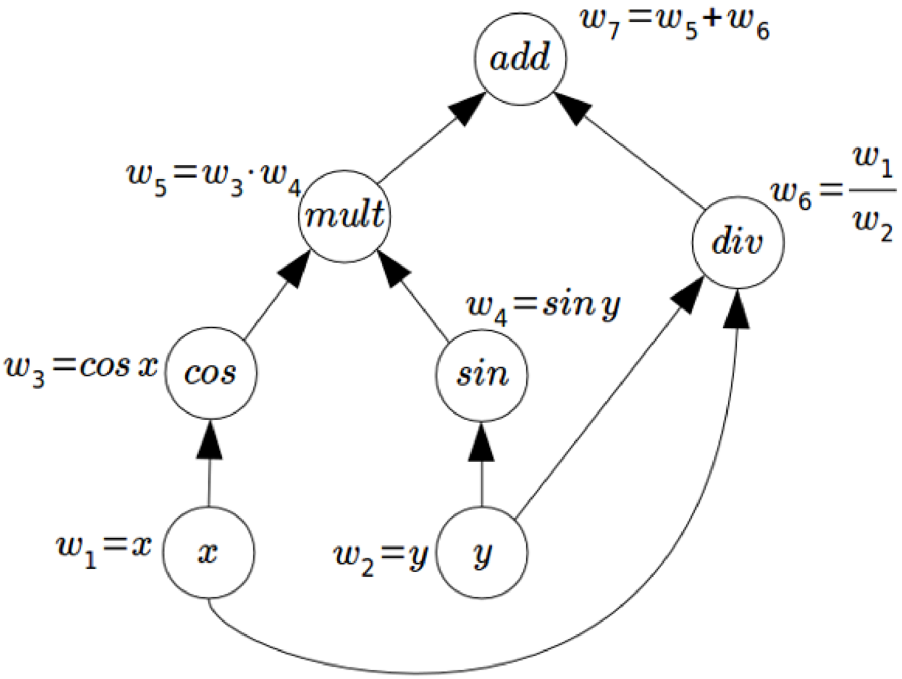


Figure 2.1: Computation graph built by PyTorch for the example above. Made pretty and annotations added for clarity. `.backward` will start from `w7` and traverse the graph "backwards" like we did when we computed the gradients by hand in class.

**Control flow (How do you take the derivative of `if` statements?).** When Python control flow (e.g., an `if` statement) is used, PyTorch builds the computation graph *dynamically*. Only the operations that are actually executed at runtime are recorded in the graph. Branches that are not taken simply do not exist in the computation graph and therefore contribute nothing to backpropagation.

**Autograd with control flow**

```
import torch

x = torch.tensor(1.0, requires_grad=True)
```

```
if x.item() > 0:
    z = x * x          # executed branch
else:
    z = x * x * x      # NOT executed, not in the graph
```

In the generated graph, only the multiplication corresponding to the executed branch appears; the unused branch is completely absent, as shown in the computation graph in Figure 2.2.

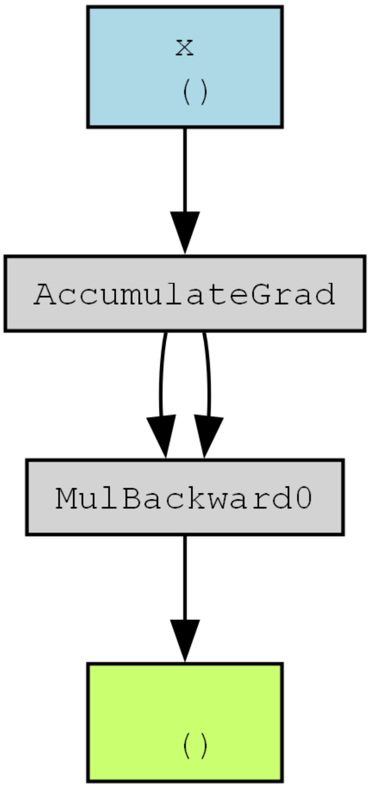


Figure 2.2: Computation graph built by PyTorch for the example above. Contrary to Figure 2.1, this is the raw output of `torchviz`, so slightly harder to read, but it is actually what happens under the hood. In the `torchviz` visuals, output tensors are shown in green, and leaf tensors in blue[2]. First, notice how there are two arrows to `MulBackward`, this is because `MulBackward` has one incoming edge from each of its input and here, both the inputs are `x`, therefore we get two edges. The top two boxes represent the tensor `x` we created. And the bottom box is the result of our operation `z` and is where `.backward()` starts. since `x * x * x` is not executed, we don't have two `MulBackward`.

In Figure 2.3, we show what graph we would've had if `x` was a `torch.tensor(-1.0, requires_grad=True)`, and so what graph we would've had if we had `z = x * x * x` executed.

---

[2] See `https://deepwiki.com/szagoruyko/pytorchviz` for more information, paragraph "Node Types and Color Coding" (Last Accessed 03/02/2026).

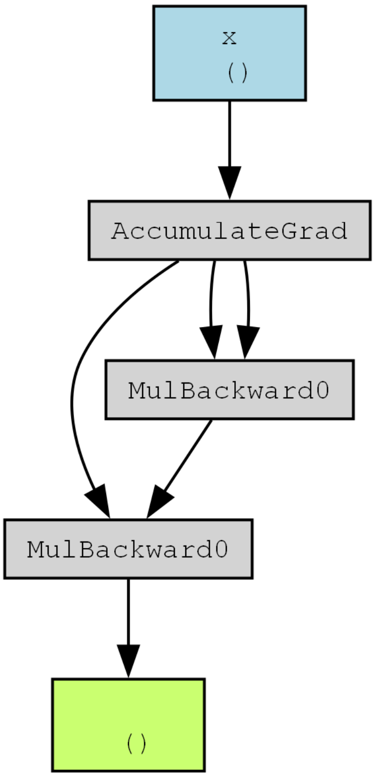


Figure 2.3: Computation graph built by PyTorch. Since `z = x * x * x` is executed, we have two `MulBackward`. The first one represents `hidden = x * x` and the second one represents `z = hidden * x`.

## 2.3 `nn.Module` and model definition

**A small regression network**

```
import torch.nn as nn

class RegressionNet(nn.Module):
    def __init__(self):
        super().__init__()
            self.net = nn.Sequential(
                nn.Linear(1, 10),
                nn.ReLU(),
                nn.Linear(10, 1)
            )

def forward(self, x):
    return self.net(x)
```

## 2.4 Optimisers

All optimisers we saw in our class are variants of gradient descent: SGD, Adam, RMSProp, etc.

**Switching optimisers**

```
import torch.optim as optim

model = RegressionNet()

opt_sgd = optim.SGD(model.parameters(), lr=1e-2)
opt_adam = optim.Adam(model.parameters(), lr=1e-3)
opt_rms = optim.RMSprop(model.parameters(), lr=1e-3)
```

Note: `torch.optim.SGD` doesn't have any randomness (stochasticity) in it, so it does seem misleading to call it stochastic gradient descent. Calling it gradient descent (or GD) would've be more accurate. But, in practice, we never go through the *whole dataset* at each pass, we only go through *batches* of the dataset. And those *batches* are sampled at random from the *whole dataset*, so technically, everytime you'll use SGD, you'll have some stochasticity even though PyTorch SGD doesn't have any stochasticity embedded in it.
Here is a Minimal Reproducible Example that proves that PyTorch SGD doesn't have any stochasticity embedded in it:

**torch.optim.SGD has no stochasticity embedded in it**

```
import torch

def run():
    # "inputs" (fixed, not learnable)
    x = torch.tensor([[0.78], [0.44], [0.80]])  # shape (3,1)
    # "targets" (fixed, not learnabble)
    y = torch.tensor([1.0, 1.0, 1.0])

    # single-neuron parameter (learnable)
    # our neural network is
    # - one input neuron -> one output neuron, nothing in-between
    # - to go from the input to the output, we multiply by w
    w = torch.tensor(0.5, requires_grad=True) # scalar

    opt = torch.optim.SGD([w], lr=0.1)

    # neuron output: scalar
    y_hat = w * x  # shape (3,)
    loss = ((y_hat - y) ** 2).mean()

    opt.zero_grad()
    loss.backward()
    opt.step()

    return w.detach()

w1 = run()
w2 = run()
w3 = run()
print(w1, w2, w3)  # we get the same result three times
```

## 2.5 Datasets and DataLoaders

**Using Dataset and DataLoader**

```python
import torch
from torch.utils.data import DataLoader
from torch.utils.data import Dataset

class SimpleDataset(Dataset):
    def __init__(self, data, targets):
        self.data = torch.tensor(
            data,
            dtype=torch.float32
        )
        self.targets = torch.tensor(
            targets,
            dtype=torch.float32
        )

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx], self.targets[idx]

dataset = SimpleDataset(
    data=[[0.0], [1.0], [2.0]],
    targets=[[0.0], [1.0], [4.0]]
)
loader = DataLoader(dataset, batch_size=2, shuffle=True)

for batch_x, batch_y in loader:
    print(batch_x.shape, batch_y.shape)
```

Note that if you execute that code, you'll see:

```
torch.Size([2, 1]) torch.Size([2, 1])
```

then

```
torch.Size([1, 1]) torch.Size([1, 1])
```

That's because, the first batch has two inputs, and two targets. The second batch would love to also have two inputs, and two targets, but it can't, we've gone through most of our dataset, and the length of our dataset is not a multiple of the batch size, so we have a smaller-than-usual batch at the end.

If you don't want that behaviour, you can drop the last batch if it's not a multiple of the batch size by adding this argument `drop_last=True` to the `DataLoader` object.

# Chapter 3

# From Neurons to Deep Networks

## 3.1 Biological vs artificial neurons

Biological neurons receive signal**s** through dendrites, combine them in the cell body, and produce **an** output along the axon if the total signal crosses a threshold. An artificial neuron abstracts this as:

$$z = w_1 x_1 + \cdots + w_d x_d + b, \quad y = \sigma(z),$$

where $\sigma$ is an activation function (ReLU, sigmoid, . . . ).
There's not much more to the comparison between biological and artifical neurons[1].

**A single neuron in PyTorch**

```
import torch
import torch.nn as nn

neuron = nn.Linear(3, 1)   # 3 inputs -> 1 output

x = torch.randn(1, 3)      # batch of 1 sample
y = neuron(x)
print(y.shape)             # torch.Size([1, 1])
```

## 3.2 Width, depth, and layers

Stacking many neurons into a layer gives a **dense layer** (fully connected). The more neurons there are on one layer, the **wider** we say the layer is. Stacking layers gives a **deep** network. Here is an example of a tiny neural network that is neither wide (16 neurons at most on a layer) nor deep (4 layers: 2 → 16 → 16 → 1).

**A tiny multi-layer perceptron**

```
import torch.nn as nn

class TinyNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(2, 16),
```

---

[1] Shockingly, biological neurons don't run gradient descents.

```
            nn.ReLU(),
            nn.Linear(16, 16),
            nn.ReLU(),
            nn.Linear(16, 1)
        )

    def forward(self, x):
        return self.layers(x)
```

## 3.3 Forward, backward, and parameter updates

Training loop ingredients:

1. Forward pass: compute predictions.
2. Compute loss.
3. Backward pass: use `loss.backward()` to compute gradients.
4. Update parameters with an optimiser.

### 3.3.1 Backward without PyTorch

It's simply computing derivatives! Cf. Exercise 1. The way we did it in class is by always differentiating **with respect to scalars**. But you can differentiate **with respect to vectors** (each component is the scalar derivative, we say differentiation is performed component-wise) or **matrices** (same, each component is the scalar derivative). You need to learn some definitions and theorems to do so, but once it's done, it'll be less tedious than what we did in class (should you ever want to do more manual differentiation ... a reasonable follow-up question would be "why would I ever want to").

**Definition 3.1: Differentiating "with respect to vectors and matrices"**

Formally (but I'm just repeating what was in Section 1.5), let $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$. If $f : \mathbb{R}^n \to \mathbb{R}$ (note that here I only talk about functions that end up in $\mathbb{R}$ because we typically want to minimise a loss that is a real value anyway), the derivative of $f$ with respect to the vector $\mathbf{x}$ is simply the gradient

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

And you can do the same with matrices: let $\mathbf{X} \in \mathbb{R}^{n \times m}$ be a matrix with entries

$$\mathbf{X} = (x_{ij})_{1 \le i \le n,\, 1 \le j \le m}.$$

If

$$f : \mathbb{R}^{n \times m} \to \mathbb{R},$$

the derivative of $f$ with respect to the matrix $\mathbf{X}$ is defined component-wise as the collection of partial derivatives with respect to each entry of $\mathbf{X}$, namely

$$\nabla_{\mathbf{X}} f(\mathbf{X}) = \begin{pmatrix} \frac{\partial f}{\partial x_{11}} & \cdots & \frac{\partial f}{\partial x_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_{n1}} & \cdots & \frac{\partial f}{\partial x_{nm}} \end{pmatrix}.$$

Thus, the gradient of $f$ with respect to $\mathbf{X}$ has the same shape as $\mathbf{X}$, and each of its entries is the scalar partial derivative of $f$ with respect to the corresponding entry of the input matrix.

Now, let

$$z = Wu + b \in \mathbb{R}^m \qquad J = J(z) \in \mathbb{R}$$

$$b \in \mathbb{R}^m \qquad W \in \mathbb{R}^{m \times d} \qquad u \in \mathbb{R}^d$$

In layman terms, $z$ is the last layer's values. $u$ is the penultimate layer's values. J is the criteria (usually the loss) we're trying to optimise (usually minimise). We have beautiful properties such as

$$\frac{\partial J(z)}{\partial W} = \frac{\partial J(z)}{\partial z} \cdot u^T$$

$$\in \mathbb{R}^{m \times d} \qquad \in \mathbb{R}^{m \times 1} \qquad \in \mathbb{R}^{1 \times d}$$

These beautiful properties save us a lot of effort when manually computing the derivatives (again, why would you ever want to? That's a good question, I honestly don't know, maybe if you want to develop a library rival to PyTorch?).

**Proof**

*Proof.* Let $i \in [1; m]$ and $j \in [1; d]$

$$\frac{\partial J(z)}{\partial w_{i,j}} = \frac{\partial\big(J(Wu + b)\big)}{\partial w_{i,j}}$$

$$= \frac{\partial\left(J\left(\left(\begin{pmatrix} w_{1,1}u_1 + w_{1,2}u_2 ... \\ \cdots \\ w_{m,1}u_1 + w_{m,2}u_2 ... \end{pmatrix} + \begin{pmatrix} b_1 \\ \cdots \\ b_m \end{pmatrix}\right)\right)\right)}{\partial w_{i,j}}$$

only the $i$th row $z_i = w_{i,1}u_1 + \ldots + b_i$ is of interest ; the rest are constants with regards to $w_{i,j}$. So we can simplify to:

$$= \frac{\partial\left(J\left(\left(w_{i,1}u_i + \ldots\right) + \left(b_i\right)\right)\right)}{\partial w_{i,j}}$$

Since $(f \circ g)(x) = g'(x)\, f'(g(x))$[a], (the $x$ being $w_{i,j}$, f being J, and $g$ being $z_i$), that's just:

$$= u_j \frac{\partial J}{\partial z_i}$$

☐

---

[a] If the notation $f'(g(x))$ feels ambiguous, the Leibniz notation can make it clearer. Use whatever suits you. The Chain Rule rewritten states $\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{i,j}}$.

### 3.3.2 Backward with PyTorch

**A minimal training step**

```
import torch
import torch.nn as nn
import torch.optim as optim

model = MyModelThatIDefined()
optimiser = optim.SGD(model.parameters(), lr=1e-2)
loss_fn = nn.MSELoss()

x = torch.randn(32, 2)
y_true = torch.randn(32, 1)

# forward
y_pred = model(x)
loss = loss_fn(y_pred, y_true)

# backward + update
optimiser.zero_grad()
loss.backward()
optimiser.step()
```

That's very minimal. Note that our `x` should come from a dataset, and we should do more than one iteration.

## 3.4 Train / validation / test sets

**Definition 3.2: Train / validation / test**

- **Training set**: data used to fit the model parameters.
- **Validation set**: data used to choose hyperparameters and architectures.
- **Test set**: data used only once at the very end for an unbiased estimate of performance.

Splitting properly is crucial to avoid overly optimistic results. Technically, for the Kaggle problems I give you, the test set I hold private is split in two. Because otherwise, you could submit a lot of models, and select the model that performs the best on the leaderboard but that would be *bad/biased*: you *learned* something by submitting a lot of models you thought were equivalent and checking which model performs the best on the leaderboard. Thus, I have a real private test set which is totally unbiased and that's why the score you see on the leaderboard won't be the actual end score. When the kaggle closes, you'll see both the "public test set" score and the "private test" score.

# Chapter 4

# Optimisation and Training Tricks

## 4.1 Stochastic gradient descent

Using batches makes optimisation stochastic (cf. 2.4) and often much faster. After going through the whole dataset once (through the use of many batches), we typically go through it again many a time. We usually encapsulate that in a `for epoch in range(max_epoch)` loop.

**Training with epochs mini-batches**

```
train_loader = DataLoader(dataset, batch_size=32, shuffle=True)
model = RegressionNet()
optimiser = optim.SGD(model.parameters(), lr=1e-2)
loss_fn = nn.MSELoss()

for epoch in range(10):
    for inputs, targets in train_loader:
        optimiser.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, targets)
        loss.backward()
        optimiser.step()
```

## 4.2 Momentum and Adam

Momentum remembers past gradients (think of it as “Don’t react to every twitch; instead move in the direction that keeps showing up”); Adam also adapts learning rates per-parameter (think of it as “If a parameter has been changing wildly, be cautious. If a parameter rarely moves, allow bigger steps.”).

**Adam optimiser**

```
optimiser = optim.Adam(
    model.parameters(),
    lr=1e-3,
    betas=(0.9, 0.999)
)
```

## 4.3 Learning-rate schedules

No one forces you to use the same learning-rate across all epochs. There are many ways to decrease the learning-rate across epochs. One way is to use the StepLR function which decays the learning rate of each parameter group by `gamma`[1] every `step_size` epochs.

**Step LR schedule**

```
optimiser = optim.SGD(model.parameters(), lr=0.1)
scheduler = optim.lr_scheduler.StepLR(
    optimiser,
    step_size=100,
    gamma=0.1
)

for epoch in range(50):
    train_one_epoch(...)
    scheduler.step()
```

## 4.4 Regularization: gradient clipping, batch norm, dropout

Gradient clipping caps the update size and might help keep training stable.

**Gradient clipping**

```
for inputs, targets in train_loader:
    optimiser.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(
        model.parameters(),
        max_norm=1.0
    )
    optimiser.step()
```

Batch Normalization stabilizes and accelerates training by normalising layer activations[2], while Dropout improves generalization by randomly deactivating neurons during training to reduce overfitting.

**BatchNorm and Dropout in a network**

```
class RegularizedNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(100, 100),
            nn.BatchNorm1d(100),
```

---

[1] Multiplicative factor

[2] Normalising layer activations reduces internal covariate shift. Internal covariate shift refers to the problem where the distribution of activations in a layer changes during training because the parameters of preceding layers are constantly being updated, making optimisation potentially slower and less stable.

```
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(100, 10)
        )
        # You don't need to use both at the same time
        # I just there both in the same example
        # to not duplicate blocks of code

        def forward(self, x):
            return self.net(x)
```

Note that for dropouts, the kept activations are scaled during training so that the expected activation magnitude remains unchanged at inference time[3]. Formally, for an activation $x$, dropout defines the output $y$ as (when we are in `model.train()` mode):

$$y = \begin{cases} 0, & \text{with probability } p, \\ \dfrac{x}{1-p}, & \text{with probability } 1-p. \end{cases}$$

The expected value of $y$ is:

$$\mathbb{E}[y] = 0 \cdot p + \frac{x}{1-p} \cdot (1-p) = x.$$

i.e., the expected activation magnitude remains unchanged at inference time (when we are in `model.eval()` mode, and $p$ is set to 0).

## 4.5 Loss functions

**Definition 4.1: Common losses**

- Regression: L1, L2 (MSE).
- Binary classification: binary cross-entropy.
- Multi-class classification: cross-entropy.

**Cross entropy in PyTorch**

```
# logits: [batch, num_classes]
logits = torch.randn(8, 3)
targets = torch.randint(0, 3, (8,))

loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(logits, targets)
```

---

[3] https://docs.pytorch.org/docs/stable/generated/torch.nn.Dropout.html#:~:text=Furthermore%2C%20the%20outputs,an%20identity%20function.

## 4.6 Softmax

**Definition 4.2: Softmax**

Given logits $z \in \mathbb{R}^K$, the softmax outputs probabilities

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \quad k = 1, \dots, K.$$

# Chapter 5

# Linear Layers and Universal Approximation

## 5.1 Universal approximation

Stacking loads of neurons in **one** hidden layer already gives a strong (theoretical) approximation power.

**Theorem 5.1: Universal approximation (informal)**

A feedforward neural network with a single hidden layer and a non-polynomial activation function (such as ReLU) can approximate any continuous function as closely as we like, provided we use enough hidden units.

**1D function approximation**

```
class OneHiddenNet(nn.Module):
    def __init__(self, width=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, width),
            nn.ReLU(),
            nn.Linear(width, 1)
        )

    def forward(self, x):
        return self.net(x)

model = OneHiddenNet(width=64)
```

You can find a proof of a weaker form of the theorem in `proof_networks_approximation.pdf`. Note: It is important to note that this is an **existence** theorem. It guarantees that a network with the right structure exists, but it does not provide a method for finding the network's parameters (i.e., training it), nor does it specify exactly how large the network must be for a given function.

## 5.2 Wide vs deep networks

For now, the only thing we learned to play with are dense networks. We can play on their width and their depth.

**Note 5.1: Width vs depth**

- Wide networks (many neurons in a single layer)
- Deep networks (many layers)

Larger layers mean the network can "remember" more features, more layers mean it can "remember" more complex features, or also more features if there aren't any complex ones in the dataset. For a given amount of weights, there's always a balance to be had between layer size and count. Typically, you'd figure the optimal ratio out through automated hyperparameter tuning.

**Vanishing gradient problem:** In a deep network, the gradient of the loss $L$ with respect to the weights $W_k$ at layer $k$ is given by the Chain Rule:

$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial a_n} \cdot \left( \prod_{i=k+1}^{n} \frac{\partial a_i}{\partial a_{i-1}} \right) \cdot \frac{\partial a_k}{\partial W_k}$$

If you are using the sigmoid activation function, $\sigma(x)$, the term $\frac{\partial a_i}{\partial a_{i-1}}$ includes the derivative $\sigma'(z)$. Because the maximum value of $\sigma'(z)$ is only 0.25, multiplying many of these fractions together causes the gradient to "vanish" toward zero, preventing the weights in early layers from updating effectively.

**Proof**

*Proof.* To prove that the derivative of the sigmoid function is bounded by 0.25, we start with the definition of the sigmoid function:

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

1. Find the Derivative
First, we express the derivative $\sigma'(x)$ in terms of $\sigma(x)$:

$$\sigma'(x) = \frac{d}{dx}(1 + e^{-x})^{-1} = -(1 + e^{-x})^{-2} \cdot (-e^{-x}) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

We can rewrite this as:

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

2. Maximize the Function
Let $f(s) = s(1 - s)$, where $s = \sigma(x)$. Since the range of the sigmoid function is $(0, 1)$, we look for the maximum of $f(s)$ on the interval $s \in (0, 1)$.This is a downward-opening parabola:

$$f(s) = s - s^2$$

To find the critical point (which is the maximum since downward-opening parabolas are concave), take the derivative with respect to $s$ and set it to zero:

$$f'(s) = 1 - 2s$$

$$1 - 2s = 0 \implies s = 0.5$$

3. Calculate the Maximum Value
Substitute $s = 0.5$ back into the derivative formula:

$$\sigma'(x)_{max} = 0.5(1 - 0.5) = 0.5 \cdot 0.5 = 0.25$$

As always, it's a good habit to double check what we prove with some simulations; if the following code does not raise an `AssertionError`, that means for all `x` tested, $\sigma'(x)$ is indeed below 0.25.

**Little sanity check**

```python
import numpy as np

def sigmoid_prime(x: float) -> float:
    return np.exp(-x)/((1+np.exp(-x))**2)

for x in range(-100, 100):
    assert sigmoid_prime(x) <= 0.25, sigmoid_prime(x)
```

□

# Chapter 6

# Convolutions, Pooling, and Softmax

## 6.1 Why convolution?

Although fully connected feedforward neural networks can be used to learn features and classify data, this architecture is generally impractical for larger inputs (e.g., high-resolution images), which would require massive numbers of neurons because each pixel is a relevant input feature. A fully connected layer for an image of size 100 × 100 has 10,000 weights for each neuron in the second layer.

Besides, dense layers ignore spatial structure; convolutional layers share weights across space and exploit local patterns. The idea behind convolutional layers is to treat the first part of neural networks analysing images as a feature extractor. To extract those features we slide through the entire image with a learnable filter, computing local responses that detect patterns such as edges, textures, or shapes at different spatial locations. The whole idea is that if the image is shifted one pixel to the right, or one pixel to the left, we don't expect much to have changed, i.e., we expect the same features after the first part of our neural network, therefore the parameters of each of our filter should be the same wherever we're using it (whether a bird is at the top right of the input picture or bottom left of the input picture, if one filter is in charge of detecting the texture of a bird, it should do it the same way wherever it's being applied).

**Each output channel** has its own filter spanning **all input channels**. For an input of shape 3 × 16 × 16 (batch size ignored) and an output shape 16 × 16 × 16, a kernel size of 3, and a padding of 1, we expect 3 × 3 × 3 parameters per filter, 16 filters total, therefore 16 × 3 × 3 × 3 = 432 weights. Moreover, there is also a bias, each output channel has one scalar bias, and that same scalar is added to every spatial position in that channel. So that's 432+16=448 weights. We can verify that by running this PyTorch code:

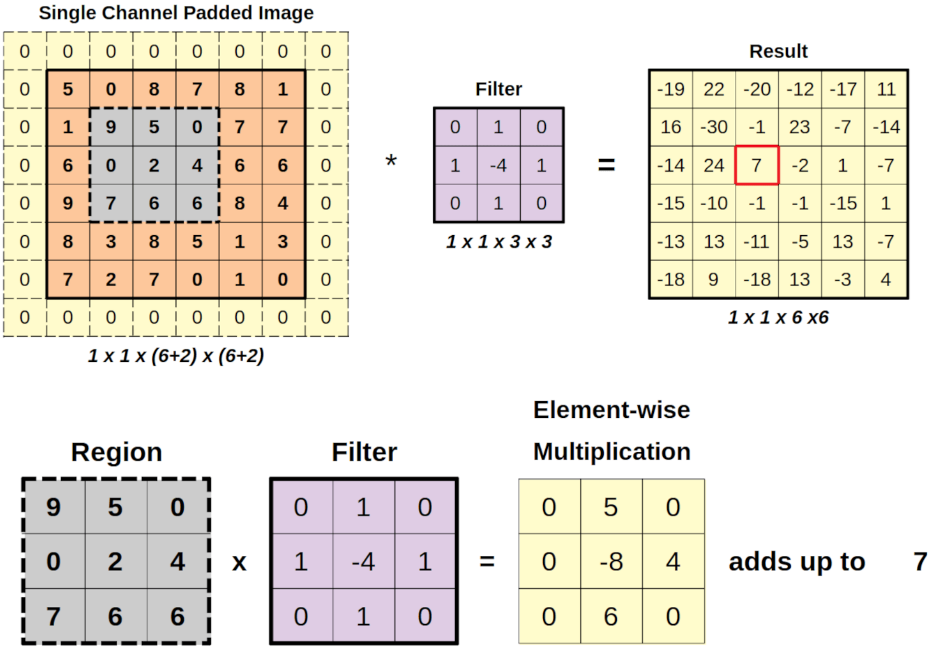
**Get number of parameters of a Conv2d layer**

```python
import torch
import torch.nn as nn

conv = nn.Conv2d(
    in_channels=3,
    out_channels=16,
    kernel_size=3,
    padding=1,
    bias=True
)

total_params = sum(p.numel() for p in conv.parameters())
print("Total parameters:", total_params)  # 448
```

Figure 6.1 illustrates how convolutional layers work:


Single Channel Padded Image
1 x 1 x (6+2) x (6+2)
*
Filter
1 x 1 x 3 x 3
=
Result
1 x 1 x 6 x6
Region
x
Filter
=
Element-wise Multiplication
adds up to
7

Single Channel Padded Image

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 5 | 0 | 8 | 7 | 8 | 1 | 0 |
| 0 | 1 | 9 | 5 | 0 | 7 | 7 | 0 |
| 0 | 6 | 0 | 2 | 4 | 6 | 6 | 0 |
| 0 | 9 | 7 | 6 | 6 | 8 | 4 | 0 |
| 0 | 8 | 3 | 8 | 5 | 1 | 3 | 0 |
| 0 | 7 | 2 | 7 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Filter

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 1 | -4 | 1 |
| 0 | 1 | 0 |

Result

| | | | | | |
|---|---|---|---|---|---|
| -19 | 22 | -20 | -12 | -17 | 11 |
| 16 | -30 | -1 | 23 | -7 | -14 |
| -14 | 24 | 7 | -2 | 1 | -7 |
| -15 | -10 | -1 | -1 | -15 | 1 |
| -13 | 13 | -11 | -5 | 13 | -7 |
| -18 | 9 | -18 | 13 | -3 | 4 |

Region

| | | |
|---|---|---|
| 9 | 5 | 0 |
| 0 | 2 | 4 |
| 7 | 6 | 6 |

Filter

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 1 | -4 | 1 |
| 0 | 1 | 0 |

Element-wise Multiplication

| | | |
|---|---|---|
| 0 | 5 | 0 |
| 0 | -8 | 4 |
| 0 | 6 | 0 |

Figure 6.1: A worked example of performing a convolution. The convolution has stride 1, padding of 1, with kernel size 3-by-3. Source: Daniel Voigt Godoy - `https://github.com/dvgodoy/dl-visuals/`.

## 6.2 Padding, stride, dilation

- **Padding**: add zeros around the border to control output size.
- **Stride**: step size of the convolution kernel.
- **Dilation**: spacing inside the kernel (for larger receptive field). See Figure 6.2.

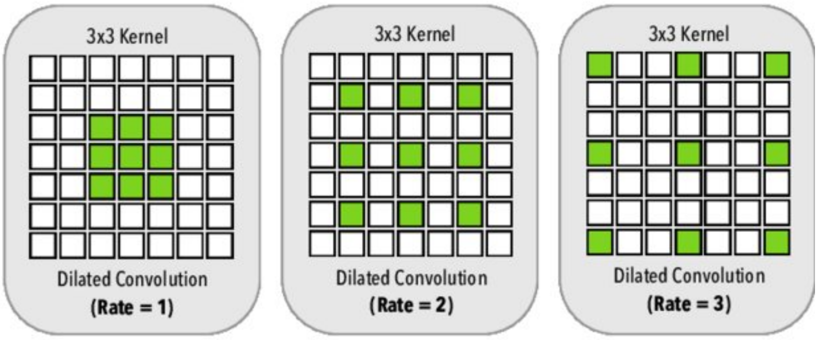

3x3 Kernel
Dilated Convolution
(Rate = 1)
3x3 Kernel
Dilated Convolution
(Rate = 2)
3x3 Kernel
Dilated Convolution
(Rate = 3)

Figure 6.2: Dilated convolution. On the left, we have the dilated convolution with dilation rate r = 1, equivalent to the standard convolution. In the middle with have a dilation r = 2 and in the right a dilation rate of r = 3. All dilated convolutions have a 3x3 kernel size and the same number of parameters. Source: Christian Perone, Evan Calabrese, and Julien Cohen-Adad. 10.1038/s41598-018-24304-3.

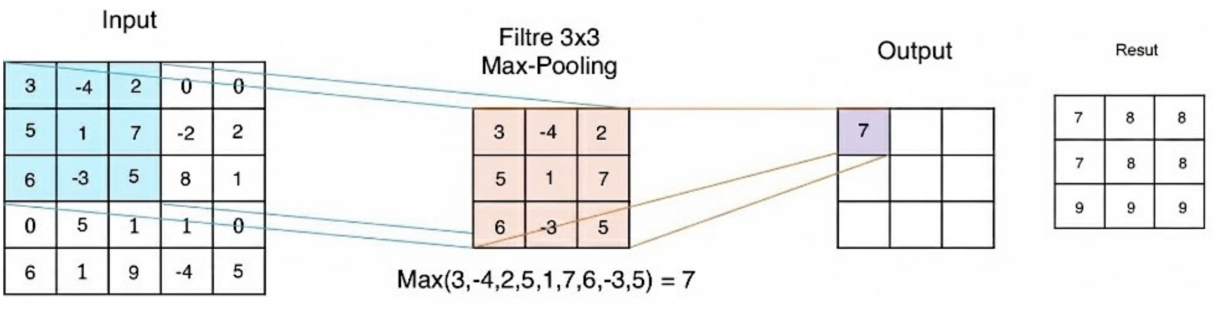
## 6.3 Pooling

Pooling layers reduce spatial size and add some[1] translation invariance.

**Max pooling**

```
pool = nn.MaxPool2d(kernel_size=2, stride=2)
x = torch.randn(1, 3, 32, 32)
y = pool(x)  # 3*32*32 -> 3*16*16
```

Figure 6.3 illustrates how pooling layers work:



|  |  |  |  |  |
|---|---|---|---|---|
| 3 | -4 | 2 | 0 | 0 |
| 5 | 1 | 7 | -2 | 2 |
| 6 | -3 | 5 | 8 | 1 |
| 0 | 5 | 1 | 1 | 0 |
| 6 | 1 | 9 | -4 | 5 |

|  |  |  |
|---|---|---|
| 3 | -4 | 2 |
| 5 | 1 | 7 |
| 6 | -3 | 5 |

|  |  |  |
|---|---|---|
| 7 |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
|---|---|---|
| 7 | 8 | 8 |
| 7 | 8 | 8 |
| 9 | 9 | 9 |

Figure 6.3: Process of applying the max-pooling filter. Kernel size 3x3. Stride 1. Source: Adapted from Alexis f. 1703.

When you do global average pooling (H x W x C → 1 x 1 x C), you are explicitly discarding where features occurred. At that point wherever was the bird detected with your convolutions, we end up with the same representation. Location is gone by design. It's up to you to decide if that matters.

**Global Average pooling**

```
pool = nn.MaxPool2d(kernel_size=32)
x = torch.randn(1, 3, 32, 32)
y = pool(x)  # 3*32*32 -> 3*1*1
```

A question that may come to mind is: how do we take the derivative of a `max` operation? The answer is similar to paragraph **Control flow (How do you take the derivative of `if` statements?)**[2] in Section 2.2.

---

[1] read the next bit above global average pooling to get the intuition of why

[2] *Technically* if there are two values that are exactly the same, it's a bit tricky/it's not exactly *like* an `if` statement. But I don't feel like digressing even more.

## 6.4 Altogether

**A simple ConvNet for images**

```
import torch.nn.functional as F

class TinyConvNet(nn.Module):
    # for 3*32*32 input
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(
            in_channels=3,
            out_channels=16,
            kernel_size=3,
            padding=1
        )
        self.pool  = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(
            16,
            32,
            kernel_size=3,
            padding=1
        )
        self.fc    = nn.Linear(32 * 8 * 8, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        # 3*32*32 -> 16*16*16

        x = self.pool(F.relu(self.conv2(x)))
        # 16*16*16 -> 32*8*8

        x = x.view(x.size(0), -1)
        # flatten

        x = self.fc(x)

        return x
```

## 6.5 Data augmentation and imbalance

### 6.5.1 Data augmentation

In practice:

- **Images**: flips, rotations, crops, noise, color jitter, etc.
- **Audio**: time shift, pitch shift, noise, filtering.
- **Text**: synonym replacement, random insertion/deletion.

**Image augmentation with torchvision**

```
import torchvision.transforms as T
```

```
transform = T.Compose([
    T.RandomHorizontalFlip(p=0.5),
    T.RandomResizedCrop(size=32, scale=(0.8, 1.0)),
    T.ColorJitter(brightness=0.2, contrast=0.2),
    T.ToTensor(),
])
```

You would then adapt your `Dataset` class to include the transformations. For example:

**Dataset with data augmentation**

```
class MyDataset(Dataset):
    def __init__(self, image_dir, labels, transform=None):
        self.image_dir = image_dir
        self.labels = labels
        self.transform = transform
        self.images = sorted(os.listdir(image_dir))

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        img_path = os.path.join(
            self.image_dir,
            self.images[idx]
        )
        image = Image.open(img_path).convert("RGB")
        label = self.labels[idx]

        if self.transform is not None:
            image = self.transform(image)

        return image, label
```

### 6.5.2 Imbalanced data

Assume “yes” is the positive class, and the true class distribution is: 99% yes, 1% no. If your model always predicts “yes”, you’ll get a pretty good accuracy (99%). That’s why accuracy is not always a good metric. For example, the recall[3] of your model will be 1 for yes and 0 for no. Furthermore, the precision[4] of your model will be 0.99 for yes and undefined (0/0) for no. Metrics like Macro F1 summarize all of that.

In practice, you can re-weight classes or over/under-sampling to avoid the model to converge to saying yes all the time.

---

[3] Of all the samples that truly belong to class C, how many did the model correctly identify?

[4] Of all the samples the model predicted as class C, how many are actually class C?

# Chapter 7

# Architectures and Transfer Learning

## 7.1 Typical classification architecture

A very standard CNN classifier:

- Several blocks of Conv + Activation + Pool that typically sequentially increase the number of Channels and decrease the Height and Width.
- One or more fully connected layers.
- Softmax output.

**A simple image classifier**

```
class SimpleClassifier(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(32 * 8 * 8, 128),
            nn.ReLU(),
            nn.Linear(128, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

In this example, make sure you understand[1] why `nn.Conv2d(3, 16, 3, padding=1)` transforms the dimension from Nb × 3 × H × W (with `Nb` the batch size) to Nb × 16 × H × W. Similarly, make sure you understand[2] why `nn.MaxPool2d(2)` brings the dimension from Nb × C × H × W to Nb × C × H//2 × W//2.

---

[1] You can use the formulas in the PyTorch docs: `https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html` but I'd suggest you convince yourself with Figure 6.1

## 7.2 Segmentation and U-Net

Segmentation predicts a label for each pixel, so we need to have an output that has the same shape as the input. U-Net combines:

- Downsampling path (encoder) with convolutions + pooling (cf. 7.1).
- Upsampling path (decoder) with transposed convolutions / upsampling.
- Skip connections that copy features from encoder to decoder.

**Very small U-Net skeleton**

```
class DoubleConv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.net(x)

class TinyUNet(nn.Module):
    def __init__(self, num_classes=1):
        super().__init__()
        self.down1  = DoubleConv(1, 16)
        self.pool1  = nn.MaxPool2d(2)
        self.down2  = DoubleConv(16, 32)
        self.up     = nn.ConvTranspose2d(32, 16, 2, stride=2)
        self.upconv = DoubleConv(16 + 16, 16)
        self.out    = nn.Conv2d(16, num_classes, 1)

    def forward(self, x):
        x1 = self.down1(x)
        x2 = self.pool1(x1)
        x3 = self.down2(x2)
        x4 = self.up(x3)
        x5 = torch.cat([x4, x1], dim=1)
        x6 = self.upconv(x5)
        logits = self.out(x6)
        return logits
```

---

[2] Either use the formulas in the PyTorch docs: https://docs.pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html or Figure 6.3. Make sure you notice that, according to the docs, if we only specify the `kernel_size` of the layer, the stride is also defined to be that number.

## 7.3 Variable-length sequences and RNNs

It should be easy to realize that when the input length is variable, we have a problem with the tools we've had at our disposal till now. Recurrent neural networks process sequences of arbitrary length by maintaining a hidden state.

**Basic RNN layer (but you should be able to write your own like we did)**

```
batch_size, seq_len, input_size, hidden_size = 4, 5, 10, 20

rnn = nn.RNN(input_size, hidden_size, batch_first=True)
x = torch.randn(batch_size, seq_len, input_size)
output, h_n = rnn(x)
print(output.shape)  # [4, 5, 20]
print(h_n.shape)  # [1, 4, 20]
```

Notice that in `nn.RNN` (and `nn.LSTM` if you ever use it), the module returns the hidden state at every time step (`output`), in addition to a variable with just the final one (`h_n`). This is why `output.shape` is equal to batch_size × seq_len × hidden_size. Similarly, `h_n` is of shape 1 × batch_size × hidden_size (see [https://docs.pytorch.org/docs/stable/generated/torch.nn.RNN.html](https://docs.pytorch.org/docs/stable/generated/torch.nn.RNN.html) for more details about that 1 and that order change, I don't make the rules).
If you want to do classification, note that the RNN itself does not perform classification. It outputs **hidden states**; you decide how to turn those into class scores. For sequence-level classification (sentiment analysis, intent detection, etc.), the standard pattern is:

**Basic RNN classifier (but you should be able to write your own like we did)**

```
rnn = nn.RNN(input_size, hidden_size, batch_first=True)
classifier = nn.Linear(hidden_size, num_classes)

output, h_n = rnn(x)
logits = classifier(h_n[-1])
```

Figure 7.1 is the diagram (from the slides) illustrating this for our classification task.

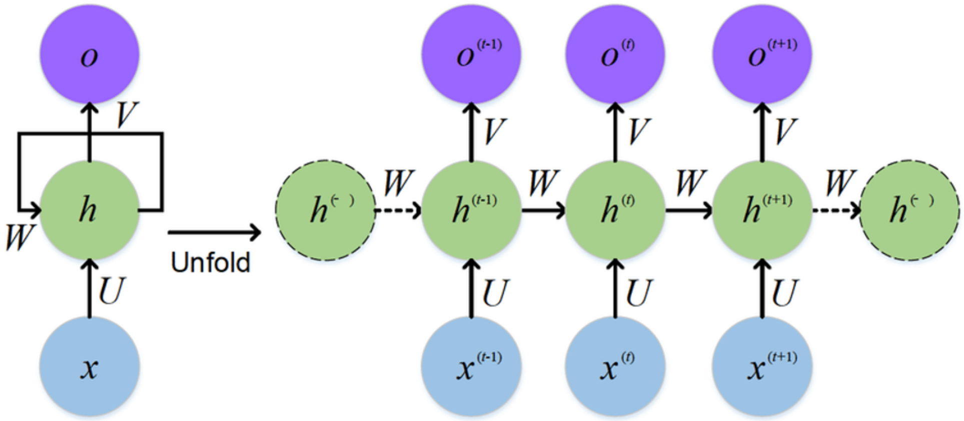


Figure 7.1: At each time step $t$, the hidden state $h^{(t)}$ is computed from the current input $x^{(t)}$ and the previous hidden state $h^{(t-1)}$ using shared weight matrices $U$ and $W$. A linear projection $V$ may be applied to the hidden state at any time step to produce an output $o^{(t)}$. Image source: P. Dubois.

Although the diagram shows an output $o^{(t)} = Vh^{(t)}$ at every time step, this reflects the most general formulation of an RNN. In sequence-level classification tasks, only the final hidden state $h^{(T)}$ is typically used, as it summarizes the entire input sequence. The classification head is therefore conceptually available/usable at every time step but is (logically) applied only to the final hidden state during training and inference. In contrast, tasks such as sequence labeling[3] may use the same output projection at every time step. Thus, the RNN architecture remains unchanged; only the manner in which its hidden states are consumed differs across tasks. Note that $V$ in our figure 7.1 corresponds to `classifier = nn.Linear(hidden_size, num_classes)` in our code above.

The first hidden state cannot be computed from the zeroth hidden state, by default we just set it to a vector of zeros.

## 7.4 Going from a string to numbers

Again, it should be easy to realize that multiplying "book" by $W$ is not defined mathematically. Because "book" is a word. And $W$ is a tensor. And we only multiply tensors between themselves (granted the dimensions match). So we need a way to transform "book" in a tensor (typically a vector). The naive way to do it is one-hot encoding, but then "teddy bear" and "soft" are equally as far (in cosine similarity[4] for example) as "soft" and "book". cf. Figure 7.2.

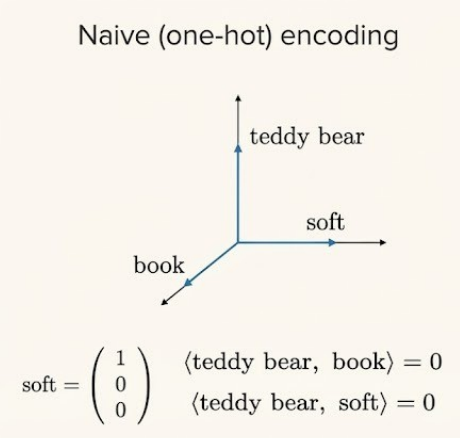


Figure 7.2: Source: Afshine Amidi, Shervine Amidi. Stanford University.

What we want is similar vectors to be pointing in similar directions, so that our neural network has an easier time learning. What a learned embedding could give is shown in Figure 7.3. There are million of ways to learn embeddings, in class we mentioned and played around with one of the two Word2Vec[5]'s idea: continuous bag of words.

---

[3] maybe we want to apply a label for each word in the input. Common tasks include: is it a noun, a verb, or an adjective?

[4] if you don't know what that is, don't look up the maths, it's a bit overwhelming. You just need to understand that it's how much they're pointing in the same direction. When two vectors are orthogonal, it's 0, when they point at the exact same direction, it's 1.

[5] Tomáš Mikolov, Kai Chen, Greg Corrado, Ilya Sutskever and Jeff Dean (2013) for more information.

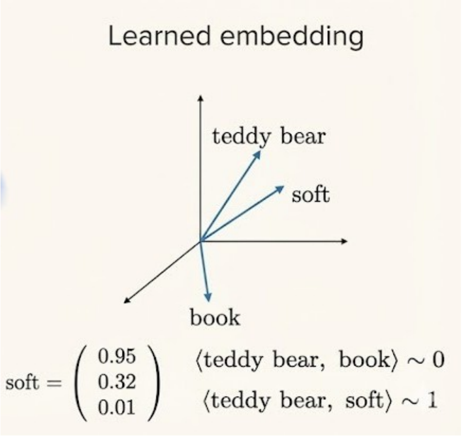
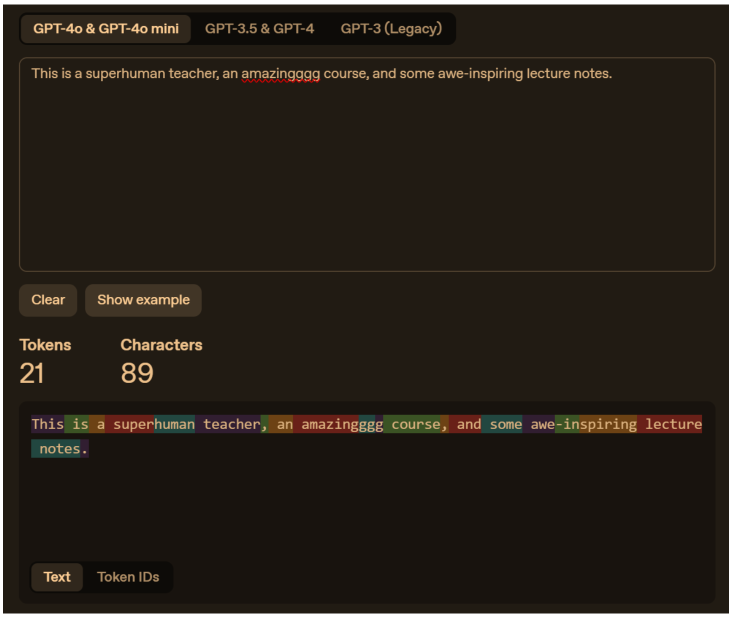

Learned embedding
teddy bear
soft
book
$\text{soft} = \begin{pmatrix} 0.95 \\ 0.32 \\ 0.01 \end{pmatrix}$
$\langle \text{teddy bear}, \text{book} \rangle \sim 0$
$\langle \text{teddy bear}, \text{soft} \rangle \sim 1$

Figure 7.3: Source: Afshine Amidi, Shervine Amidi. Stanford University.

This doesn't help when languages have the bright idea to have several meanings for a given word, e.g., set which holds the Guinness World Record for the most dictionary entries in the Oxford English Dictionary[6]. The embedding of set in the sentence "You set an example of generosity." is the same as the embedding of set in the sentence "Let $S$ be the set of primes whose index in the ordered primes, digit sum, and gap to the next prime are all themselves prime.". We need context-aware embeddings to solve that issue, and we will see this in the Chapter 8.

Our vocabulary depends on how we "tokenise" our input sentence. We can do it at the letter level (then our vocabulary size is small, but I doubt you can learn any meaningful embeddings/encodings at this level), at the subword level (vocabulary size is bigger), or at the word level (vocabulary size is even bigger). Modern NLP usually does it at the subword level. This allows us to be very rarely out of vocabulary, i.e., encountering a token we had never encountered before. For instance, GPT-4o tokenises "This is a superhuman teacher, an amazingggg course, and some awe-inspiring lecture notes." that way:


GPT-4o & GPT-4o mini
GPT-3.5 & GPT-4
GPT-3 (Legacy)
Clear
Show example
Text
Token IDs

This is a superhuman teacher, an amazingggg course, and some awe-inspiring lecture notes.

Tokens 21

Characters 89

This is a superhuman teacher, an amazingggg course, and some awe-inspiring lecture notes.

Figure 7.4: Source: https://platform.openai.com/tokenizer.

---

[6] Source: https://www.guinnessworldrecords.com/world-records/english-word-with-the-most-meanings. At least if you don't understand any of the maths or the computer science we're talking about here, you can go to sleep at peace tonight.

## 7.5 Sequence-to-sequence with RNNs

Text translation is a typical "seq2seq task": an *encoder RNN* reads the input sentence, a *decoder RNN* generates the output sentence.

**Encoder RNN skeleton**

```
class EncoderRNN(nn.Module):
    def __init__(self, vocab_size, emb_size, hidden_size):
        super().__init__()
        self.emb = nn.Embedding(
            vocab_size,
            emb_size
        )
        self.rnn = nn.RNN(
            emb_size,
            hidden_size,
            batch_first=True
        )

    def forward(self, x):
        x = self.emb(x)  # [batch, seq_len, emb_size]
        _, hidden = self.rnn(x)
        return hidden
```

**Decoder RNN skeleton**

```
class DecoderRNN(nn.Module):
    def __init__(self, vocab_size, emb_size, hidden_size):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, emb_size)
        self.rnn = nn.RNN(
            emb_size,
            hidden_size,
            batch_first=True
        )
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden):
        x = self.emb(x)  # [batch, seq_len, emb_size]
        outputs, hidden = self.rnn(x, hidden)
        logits = self.fc(outputs)
        # logits is of size: [batch, seq_len, vocab_size]
        return logits
```

Here, we use `nn.Embedding` which could come from Word2Vec, i.e. trained through a **proxy task**. But what's inside `nn.Embedding` can also be initialized from N(0,1) and trained along the way[7].

---

[7] Embeddings are rarely not taken from something pre-trained. That's why I only mentioned Word2Vec and didn't mention that we could train it along the way in class, even if we can.

## 7.6 Transfer learning with `torchvision`

Instead of training from scratch, we can start from a model trained on ImageNet for example. You will probably need to replace the last layer of ImageNet though since the number of classes for your task is likely to be different than what the base model was trained on.

**Transfer learning with ResNet-18**

```python
import torchvision.models as models
from torchvision.models import ResNet18_Weights

weights = ResNet18_Weights.DEFAULT
base_model = models.resnet18(weights=weights)

num_features = base_model.fc.in_features
base_model.fc = nn.Linear(num_features, 2)  # e.g., 2 classes

# Now fine-tune base_model on your dataset
```

# Chapter 8

# Attention, Transformers

## 8.1 Why attention? How attention?

**Why** Building on the Seq2seq task, attention mechanisms allow the decoder to "look back" on some parts of the input or the output, without regard to the distance with the current position and the attended part.

**How** The simplest way to model attention is a second neural network that tells us how much, at position $t$, we should pay attention of the earlier hidden states (from 1 to t - 1). A common question, is how to build that second network? It will output $t-1$ values. So there should be $t-1$ neurons on the last layer, but that's not constant! The answer is that we just make that second network[1] take as input **our** current state $h_t$, **one** hidden state $h_k$ s.t. $1 \leq k \leq t-1$ and the output is set to be **one** real number $a_{t,k}$. This captures how relevant the hidden state at position $k$ is for computing the output at time $t$. These scores are then normalized, usually using a softmax function, so that they form a probability distribution over past positions:

$$\alpha_{t,k} = \frac{\exp(a_k)}{\sum_{j=1}^{t-1} \exp(a_j)}.$$

The resulting coefficients $\alpha_{t,k}$ are called *attention weights*. They indicate how much each previous hidden state contributes to the current computation. Finally, a *context vector* is obtained as a weighted sum of the past hidden states:

$$c_t = \sum_{k=1}^{t-1} \alpha_{t,k} h_k.$$

This context vector summarizes the relevant information from the history in a single fixed-size representation, regardless of sequence length. Note that although the number of coefficients $\alpha_{t,k}$ depends on the input length, the resultant $c_t$ vector we have is of the same shape as all the hidden states $h_k$, which dimensions do not depend on the sequence length.
All in all, without attention, the decoder has to compress the whole sentence into a single hidden state. Attention allows the decoder to look back at *all* encoder states with a learned weighting.

It turns out, this is not good enough. The RNN has to wait for $h_{t-1}$ to be computed before computing $h_t$. This inherently sequential nature precludes parallelization and makes it hard to scale things out. The Transformer[2] is a model architecture that eschews recurrence and instead relies entirely on an attention mechanism to draw global dependencies between input and output.

---

[1] It can be as simple as a few fully connected layers!

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin (2017)

## 8.2 Transformers (Focus on the scaled dot-product attention)

Transformers rely mainly on self-attention, sometimes called intra-attention. It is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. It produces context-aware embeddings.

**Definition 8.1: Scaled dot-product attention**

Given query matrix $Q$, key matrix $K$, and value matrix $V$,

$$\mathrm{Attention}(Q,K,V)=\mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where $d_k$ is the key dimension.

Let the input sequence be represented by a matrix

$$X\in\mathbb{R}^{n\times d_{\mathrm{model}}},$$

where $n$ is the sequence length[3] and $d_{\mathrm{model}}$ the embedding dimension[4]. The query, key, and value matrices are obtained via learned linear projections:

$$Q=XW_Q,\quad K=XW_K,\quad V=XW_V,$$

with

$$W_Q\in\mathbb{R}^{d_{\mathrm{model}}\times d_k},\quad W_K\in\mathbb{R}^{d_{\mathrm{model}}\times d_k},\quad W_V\in\mathbb{R}^{d_{\mathrm{model}}\times d_v}.$$

Those three matrices are parameters of the model (and importantly do not depend on $n$. If they did, we wouldn't be able to consider them as parameters of the model since their dimension would depend on the length of the input). Thus,

$$Q\in\mathbb{R}^{n\times d_k},\quad K\in\mathbb{R}^{n\times d_k},\quad V\in\mathbb{R}^{n\times d_v}.$$

The attention weight matrix

$$\mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)\in\mathbb{R}^{n\times n}$$

encodes pairwise interactions between all positions in the sequence[5]. Multiplying by $V$ yields the self-attention output

$$\mathrm{Attention}(Q,K,V)\in\mathbb{R}^{n\times d_v}.$$

Notice how we are in $\mathbb{R}^{n\times d_v}$, so we still have one encoding per input token. But it's a **context-aware** embedding, rather than an embedding that just comes from Word2Vec, for example, and that could not tell the difference between *set* in "You *set* an example of generosity." and *set* in "Let $S$ be the *set* of primes whose index in the ordered primes, digit sum, and gap to the next prime are all themselves prime.".

---

[3] We used $t$ above but that's because we meant $t\in\{1,\dots,n\}$.

[4] If we encode input using Word2Vec, this corresponds to the dimension of the (purposely pretty small) hidden embedding layer.

[5] Given a matrix $A\in\mathbb{R}^{n\times m}$, the softmax operation is defined row-wise as

$$\mathrm{softmax}(A)_{ij}=\frac{\exp(A_{ij})}{\sum_{k=1}^{m}\exp(A_{ik})}.$$

Each row of $\mathrm{softmax}(A)$ is therefore a probability distribution, with non-negative entries that sum to one. In the case of self-attention, where $A=QK^\top/\sqrt{d_k}\in\mathbb{R}^{n\times n}$, the $i$-th row represents the attention weights assigned by position $i$ to all positions in the sequence.

In self-attention, queries, keys, and values are computed from the same input sequence. In cross-attention (that lives in the **decoder** part of the transformer), the query, key, and value matrices are no longer derived from the same sequence. Instead, queries typically come from one sequence (e.g. the **decoder**), while keys and values come from another (e.g. the **encoder**):

$$Q' = X_{\text{out}} W'_Q, \qquad K' = X_{\text{in}} W'_K, \qquad V' = X_{\text{in}} W'_V.$$

Here,

$$X_{\text{out}} \in \mathbb{R}^{n_{\text{out}} \times d_{\text{model}}}, \qquad X_{\text{in}} \in \mathbb{R}^{n_{\text{in}} \times d_{\text{model}}},$$

where $n_{\text{out}}$ and $n_{\text{in}}$ denote the output and input sequence lengths, respectively. Consequently,

$$Q' \in \mathbb{R}^{n_{\text{out}} \times d_k}, \qquad K' \in \mathbb{R}^{n_{\text{in}} \times d_k}, \qquad V' \in \mathbb{R}^{n_{\text{in}} \times d_v}.$$

The resulting attention weight matrix

$$\operatorname{softmax}\left(\frac{Q' K'^{\top}}{\sqrt{d_k}}\right) \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}$$

encodes how each output position attends to all input positions. Multiplication by $V'$ yields the cross-attention output

$$\text{Attention}(Q', K', V') \in \mathbb{R}^{n_{\text{out}} \times d_v}.$$

We end up in $\mathbb{R}^{n_{\text{out}} \times d_v}$ which is consistent with the fact that we are in the **decoder** part of the Transformer.

## 8.3 Transformers (Focus on the Positional Encoding)

Self-attention, as defined above, is *permutation-invariant*: if we shuffle the rows of $X$, the output of $\text{Attention}(Q, K, V)$ is shuffled in the same way. In other words, the attention mechanism itself has no notion of token order. In RNNs, sequential structure was built into the computation, so we lost something.

To inject information about token positions into the model, Transformers add a *positional encoding* to the input embeddings before computing $Q$, $K$, and $V$. Formally, instead of using $X$ (the encoded input), the Transformer uses

$$\tilde{X} = X + P,$$

where $P \in \mathbb{R}^{n \times d_{\text{model}}}$ is a matrix whose rows encode the position of each token in the sequence.

### 8.3.1 Sinusoidal positional encoding

The original Transformer uses a deterministic, **non-learned**[6] positional encoding defined as follows. For a position $\text{pos} \in \{0, \dots, n-1\}$ and dimension index $i \in \{0, \dots, d_{\text{model}} - 1\}$,

$$P_{\text{pos},\,2i} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right), \qquad P_{\text{pos},\,2i+1} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right).$$

Even dimensions use sine functions, odd dimensions use cosine functions. The distribution between cosines and sines is not really important. What's important is that frequencies change as the column index increases (it could be increasing or decreasing as the column index increases, and shouldn't matter much[7]). This construction has two key properties: (i) Positions that are close have similar encodings. (ii) All coefficients are bounded between -1 and 1, so that the positional information remains on a comparable scale and does not overwhelm the embedding values.

---

[6] By this, we mean no gradient descent is done on this, this is not considered a "parameter" but rather a "hyperparameter".

[7] This was once revealed to me in a dream.

### 8.3.2 A concrete numerical example

Consider a sequence of length $n = 40$ and a small embedding dimension $d_{\text{model}} = 8$. The positional encoding matrix $P \in \mathbb{R}^{40\times 8}$ is given by something that approximately looks like this:

$$P \approx \begin{pmatrix}
0.000 & 1.000 & 0.000 & 1.000 & 0.000 & 1.000 & 0.000 & 1.000 \\
0.841 & 0.540 & 0.100 & 0.995 & 0.010 & 1.000 & 0.001 & 1.000 \\
0.909 & -0.416 & 0.199 & 0.980 & 0.020 & 1.000 & 0.002 & 1.000 \\
0.141 & -0.990 & 0.296 & 0.955 & 0.030 & 1.000 & 0.003 & 1.000 \\
-0.757 & -0.654 & 0.389 & 0.921 & 0.040 & 0.999 & 0.004 & 1.000 \\
-0.959 & 0.284 & 0.479 & 0.878 & 0.050 & 0.999 & 0.005 & 1.000 \\
-0.279 & 0.960 & 0.565 & 0.825 & 0.060 & 0.998 & 0.006 & 1.000 \\
0.657 & 0.754 & 0.644 & 0.765 & 0.070 & 0.998 & 0.007 & 1.000 \\
0.989 & -0.146 & 0.717 & 0.697 & 0.080 & 0.997 & 0.008 & 1.000 \\
0.412 & -0.911 & 0.783 & 0.622 & 0.090 & 0.996 & 0.009 & 1.000 \\
-0.544 & -0.839 & 0.841 & 0.540 & 0.100 & 0.995 & 0.010 & 1.000 \\
-1.000 & 0.004 & 0.891 & 0.454 & 0.110 & 0.994 & 0.011 & 1.000 \\
-0.537 & 0.844 & 0.932 & 0.362 & 0.120 & 0.993 & 0.012 & 1.000 \\
0.420 & 0.907 & 0.964 & 0.267 & 0.130 & 0.992 & 0.013 & 1.000 \\
0.991 & 0.137 & 0.985 & 0.170 & 0.140 & 0.990 & 0.014 & 1.000 \\
0.650 & -0.760 & 0.997 & 0.071 & 0.149 & 0.989 & 0.015 & 1.000 \\
-0.288 & -0.958 & 1.000 & -0.029 & 0.159 & 0.987 & 0.016 & 1.000 \\
-0.961 & -0.275 & 0.992 & -0.129 & 0.169 & 0.986 & 0.017 & 1.000 \\
-0.751 & 0.660 & 0.974 & -0.227 & 0.179 & 0.984 & 0.018 & 1.000 \\
0.150 & 0.989 & 0.946 & -0.323 & 0.189 & 0.982 & 0.019 & 1.000 \\
0.913 & 0.408 & 0.909 & -0.416 & 0.199 & 0.980 & 0.020 & 1.000 \\
0.837 & -0.548 & 0.863 & -0.505 & 0.208 & 0.978 & 0.021 & 1.000 \\
-0.009 & -1.000 & 0.808 & -0.589 & 0.218 & 0.976 & 0.022 & 1.000 \\
-0.846 & -0.533 & 0.746 & -0.666 & 0.228 & 0.974 & 0.023 & 1.000 \\
-0.906 & 0.424 & 0.675 & -0.737 & 0.238 & 0.971 & 0.024 & 1.000 \\
-0.132 & 0.991 & 0.598 & -0.801 & 0.247 & 0.969 & 0.025 & 1.000 \\
0.763 & 0.647 & 0.516 & -0.857 & 0.257 & 0.966 & 0.026 & 1.000 \\
0.956 & -0.292 & 0.427 & -0.904 & 0.267 & 0.964 & 0.027 & 1.000 \\
0.271 & -0.963 & 0.335 & -0.942 & 0.276 & 0.961 & 0.028 & 1.000 \\
-0.664 & -0.748 & 0.239 & -0.971 & 0.286 & 0.958 & 0.029 & 1.000 \\
-0.988 & 0.154 & 0.141 & -0.990 & 0.296 & 0.955 & 0.030 & 1.000 \\
-0.404 & 0.915 & 0.042 & -0.999 & 0.305 & 0.952 & 0.031 & 1.000 \\
0.551 & 0.834 & -0.058 & -0.998 & 0.315 & 0.949 & 0.032 & 0.999 \\
1.000 & -0.013 & -0.158 & -0.987 & 0.324 & 0.946 & 0.033 & 0.999 \\
0.529 & -0.849 & -0.256 & -0.967 & 0.333 & 0.943 & 0.034 & 0.999 \\
-0.428 & -0.904 & -0.351 & -0.936 & 0.343 & 0.939 & 0.035 & 0.999 \\
-0.992 & -0.128 & -0.443 & -0.897 & 0.352 & 0.936 & 0.036 & 0.999 \\
-0.644 & 0.765 & -0.530 & -0.848 & 0.362 & 0.932 & 0.037 & 0.999 \\
0.296 & 0.955 & -0.612 & -0.791 & 0.371 & 0.929 & 0.038 & 0.999 \\
0.964 & 0.267 & -0.688 & -0.726 & 0.380 & 0.925 & 0.039 & 0.999
\end{pmatrix}$$

The actual input to the Transformer is therefore $\tilde{X} = X + P$, that is, each token embedding $\mathbf{x}_t$ is shifted by a position-dependent vector $P_t$. The components of that vector $P_t$ (the rows of the matrix above) are just cosines and sines that have different periods and all sampled at "time $t$".

Crucially, the dimensionality of $\tilde{X}$ is unchanged, so nothing we said above in Section 8.2 changes. But now, the Transformer can reason about both *content* (from $X$) and *order* (from $P$) while remaining fully parallelisable.

## Chapter 9

# GPT and Low-Rank Adaptation (LoRA)

### 9.1 Finetuning GPT-style language models is hard

GPT models are large Transformers trained to predict the next token given previous tokens. Collecting data is trivial, any text will do. We can then finetune on our custom dataset by using a simple Cross-Entropy loss for example[1]. GPT-2 has about 1.5B parameters; GPT-3/4/5 have many more. Fine-tuning such models naively is expensive. How can we, mere mortals, do that?

### 9.2 Easy Tricks

- **Quantization**: store weights in fewer bits (e.g. 8-bit) to reduce memory and speed up inference.
- **Freezing some layers**: instead of computing gradients for the entire neural network, we do it only a subset of its layers, in which case the layers that are not being fine-tuned are "frozen" (i.e., not changed during backpropagation).

### 9.3 LoRA: low-rank adaptation

LoRA updates a weight matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ by adding a low-rank matrix $\Delta W = BA$, where $B \in \mathbb{R}^{d_{\text{out}} \times r}$ and $A \in \mathbb{R}^{r \times d_{\text{in}}}$ with small $r$.

**Definition 9.1: LoRA update**

Given a base weight $W_0$, LoRA uses

$$W = W_0 + \alpha \frac{1}{r} BA,$$

where $A, B$ are trainable, low-rank factors and $\alpha$ is a scaling factor.

The intuition is that when we're doing fine-tuning, we have fewer examples so we don't expect $\Delta W$ to need as many degrees of freedom as $W$ initially needed during pre-training. By writing $\Delta W = BA$, we're restricting the number of degrees of freedom the model has when it changes $W$: it's easier for the model, and more importantly for your poor little GPU since there are less gradients to compute.

---

[1] [Advanced] Technically, label smoothing (probably defined in your Machine Learning class) is beneficial, cf. the paper Attention is all you need that reported improved accuracy in translation tasks.

Note the similarity with the outer-product structure $aa^\top$ from the prerequisites chapter: $BA$ is a sum of $r$ rank-1 matrices, so $\operatorname{rank}(BA) \leq r$.