10907 Pattern Recognition

Lecturers
Prof. Dr. Ivan Dokmanić ⟨ivan.dokmanic@unibas.ch⟩

Tutors
Felicitas Haag ⟨felicitas.haag@unibas.ch⟩
Alexandra Spitzer ⟨alexandra.spitzer@unibas.ch⟩
Cheng Shi ⟨cheng.shi@unibas.ch⟩
Vinith Kishore ⟨vinith.kishore@unibas.ch⟩

Problem set 5

Math

Exercise 1 (Why "transposed convolutions"; adapted from Bishops' deep learning book - ⋆⋆)**.** We have seen in class that transposed convolution is a particular way to upsample an image. In this exercise we'll see where the term "transposed convolution" from deep learning newspeak comes from. Consider a one-dimensional strided convolutional layer with an input having four units with activations $\boldsymbol{x} = \left[x[0], x[1], x[2], x[3]\right]^T$, which is padded with zeros to give $\tilde{\boldsymbol{x}} = \left[0, x[0], x[1], x[2], x[3], 0\right]^T$, and a filter with parameters $\boldsymbol{w} = \left[w[0], w[1], w[2]\right]$.

(i) Write down the one-dimensional activation vector of the output layer assuming a stride of 2 (equivalently, of a stride-1 convolution followed by a downsampling by a factor of 2).

(ii) Express this output in the form of a matrix–vector multiplication $\boldsymbol{A}\tilde{\boldsymbol{x}}$ with some matrix $\boldsymbol{A}$.

(iii) Now consider a layer that takes as input a vector of length 2 $\boldsymbol{z} = \left[z[0], z[1]\right]^T$ and multiplies it by the *transposed* matrix $\boldsymbol{A}^T$. Compute its output.

(iv) Compare this with the output of a 1D transposed convolution layer in `pytorch` with the filter $\boldsymbol{w}$ and stride 2. Consult the documentation for `ConvTranspose1d` for details. In particular, note that since the filter is longer (= 3) than the stride (= 2), the overlapping values are simply summed.

Part (i): The output of the "convolutional" layer with stride 2 is (recall that the convolution layers in deep learning frameworks typically don't flip the filter and thus implement cross-correlation, hence the '+' in the argument of $\tilde{x}$ below):

$$y[n] = \sum_{m=0}^{2} w[m]\tilde{x}[2n+m],$$

for $n = 0, 1$. Thus,

$$\begin{aligned} y[0] &= w[0]\tilde{x}[0] + w[1]\tilde{x}[1] + w[2]\tilde{x}[2], \\ y[1] &= w[0]\tilde{x}[2] + w[1]\tilde{x}[3] + w[2]\tilde{x}[4] \end{aligned}$$

Part (ii): This can be written as

$$\boldsymbol{y} = \begin{bmatrix} y[0] \\ y[1] \end{bmatrix} = \underbrace{\begin{bmatrix} w[0] & w[1] & w[2] & 0 & 0 & 0 \\ 0 & 0 & w[0] & w[1] & w[2] & 0 \end{bmatrix}}_{=:\boldsymbol{A}} \underbrace{\begin{bmatrix} 0 \\ x[0] \\ x[1] \\ x[2] \\ x[3] \\ 0 \end{bmatrix}}_{=:\tilde{\boldsymbol{x}}} = \boldsymbol{A}\tilde{\boldsymbol{x}}.$$

Part (iii): We compute

$$\boldsymbol{A}^T\boldsymbol{z} = \begin{bmatrix} w[0] & 0 \\ w[1] & 0 \\ w[2] & w[0] \\ 0 & w[1] \\ 0 & w[2] \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z[0] \\ z[1] \end{bmatrix} = \begin{bmatrix} w[0]z[0] \\ w[1]z[0] \\ w[2]z[0] + w[0]z[1] \\ w[1]z[1] \\ w[2]z[1] \\ 0 \end{bmatrix}.$$

Part (iv): In `pytorch`, we can implement the transposed convolution as follows:

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

conv_transpose = nn.ConvTranspose1d(in_channels=1, out_channels=1, kernel_size=3,
↪  stride=2, bias=False)
conv_transpose.weight = torch.nn.Parameter(torch.tensor([[[w0, w1, w2]]]))
z = torch.tensor([[[z0, z1]]], dtype=torch.float32)
output = conv_transpose(z)
```

The output will be

$$\begin{bmatrix} w[0]z[0] \\ w[1]z[0] \\ w[2]z[0] + w[0]z[1] \\ w[1]z[1] \\ w[2]z[1] \\ 0 \end{bmatrix},$$

which matches the result from part (iii). To understand how this works, note that the transposed convolution with stride 2 can be seen as inserting zeros between the elements of $\boldsymbol{z}$ (upsampling by a factor of 2) followed by a standard convolution with the filter $\boldsymbol{w}$. If you do the math you see that the overlapping contributions from the convolution are summed, which is why we see terms like $w[2]z[0] + w[0]z[1]$ in the output.

For intuition: the "forward" (not transposed) convolution with stride 2 can be seen as a cascade of a usual convolution with stride 1 and a downsampling by a factor of 2. Let $\boldsymbol{A}_1$ be the matrix corresponding to the stride-1 convolution with $\boldsymbol{w}$,

$$\boldsymbol{A}_1 = \begin{bmatrix} w[0] & w[1] & w[2] & 0 & 0 & 0 \\ 0 & w[0] & w[1] & w[2] & 0 & 0 \\ 0 & 0 & w[0] & w[1] & w[2] & 0 \\ 0 & 0 & 0 & w[0] & w[1] & w[2] \end{bmatrix},$$

and let $\boldsymbol{D}$ be the downsampling matrix by a factor of 2,

$$\boldsymbol{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Then you can check that the overall operation is "convolve-then-downsample",

$$\boldsymbol{A} = \boldsymbol{D}\boldsymbol{A}_1.$$

The transposed convolution is the transpose (adjoint) of this process. We have

$$\boldsymbol{A}^T = (\boldsymbol{D}\boldsymbol{A}_1)^T = \boldsymbol{A}_1^T\boldsymbol{D}^T = \begin{bmatrix} w[0] & 0 \\ w[1] & 0 \\ w[2] & w[0] \\ 0 & w[1] \\ 0 & w[2] \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix},$$

which corresponds to first upsampling by a factor of 2 (the transpose of downsampling, inserting zeros between samples) followed by a standard convolution with the filter $\boldsymbol{w}$ (the transpose of the stride-1 convolution matrix $\boldsymbol{A}_1$). This is exactly what the transposed convolution layer in deep learning frameworks does.

Do read the docstring here: `https://docs.pytorch.org/docs/stable/generated/torch.nn.ConvTranspose1d.html` and check out the visualizations here: `https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md`

Exercise 2 (Transformer equivariance - ⋆⋆)**.** Show formally that transformers which use self-attention layers without positional encodings are equivariant to permutations of the input tokens.

Let the input sequence consist of n tokens with embeddings collected in a matrix

$$X \in \mathbb{R}^{n\times d},$$

where each row corresponds to one token. A permutation of the input tokens is represented by a permutation matrix

$$P \in \mathbb{R}^{n\times n}.$$

Permutation equivariance of a map T means

$$T(PX) = P\,T(X) \quad \text{for all permutation matrices } P.$$

Self-attention: Consider a single-head self-attention layer without positional encodings:

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

where $W_Q, W_K, W_V \in \mathbb{R}^{d\times d_k}$ are shared across all tokens. The attention scores are

$$S(X) = \frac{1}{\sqrt{d_k}} QK^\top = \frac{1}{\sqrt{d_k}} XW_Q W_K^\top X^\top,$$

and the attention weights

$$A(X) = \mathrm{softmax}_{\mathrm{row}}(S(X)).$$

The attention output is

$$\mathrm{Att}(X) = A(X)\,V.$$

Let $\tilde{X} = PX$. Then

$$\tilde{Q} = PQ, \quad \tilde{K} = PK, \quad \tilde{V} = PV,$$

and the new score matrix satisfies

$$S(\tilde{X}) = \frac{1}{\sqrt{d_k}} \tilde{Q}\tilde{K}^\top = PS(X)P^\top.$$

Since the row-wise softmax is equivariant to permutations,

$$\mathrm{softmax}_{\mathrm{row}}(PSP^\top) = P\,\mathrm{softmax}_{\mathrm{row}}(S)P^\top,$$

we obtain

$$A(\tilde{X}) = PA(X)P^\top.$$

Therefore,

$$\mathrm{Att}(\tilde{X}) = A(\tilde{X})\tilde{V} = (PA(X)P^\top)(PV) = PA(X)V = P\,\mathrm{Att}(X).$$

Thus self-attention is permutation equivariant.

The same argument applies independently to each head in multi-head attention; concatenation and output projections are row-wise linear maps and therefore commute with permutations. Hence,

$$\mathrm{MHA}(PX) = P\,\mathrm{MHA}(X).$$

Feedforward layers and LayerNorm: The position-wise feedforward network applies the same function independently to each token:

$$\mathrm{FFN}(X)_i = f(W_2\phi(W_1x_i)).$$

Since all rows are processed identically, it follows that

$$\mathrm{FFN}(PX) = P\,\mathrm{FFN}(X).$$

LayerNorm (as used in transformers) is also applied independently per token and is therefore permutation equivariant.

Full transformer layer: A transformer layer without positional encodings has the form

$$X' = X + \mathrm{MHA}(X), \qquad Y = X' + \mathrm{FFN}(X').$$

Using equivariance of the individual components,

$$X'_{\text{perm}} = PX + \mathrm{MHA}(PX) = P(X + \mathrm{MHA}(X)) = PX',$$

and

$$Y_{\text{perm}} = X'_{\text{perm}} + \mathrm{FFN}(X'_{\text{perm}}) = P(X' + \mathrm{FFN}(X')) = PY.$$

Hence,

$$T(PX) = P\,T(X).$$

Since the composition of permutation-equivariant maps is again permutation equivariant, any transformer built from self-attention, feedforward layers, residual connections, and LayerNorm—without positional encodings—is equivariant to permutations of its input tokens.

Exercise 3 (Temperature in softmax - $\star$)**.** In all modern generative language models there is a tunable temperature parameter which affects the softmax which "predict the next word" (or: gives a distribution that we sample from to get the next token). Suppose that the logits (pre-softmax activations) produced by the model are given by the vector $\boldsymbol{z} = [z_1, z_2, \dots, z_n]$. The temperature-scaled softmax is defined as:

$$[\mathrm{softmax}_T(\boldsymbol{z})]_i = \frac{\exp(z_i/T)}{\sum_{j=1}^n \exp(z_j/T)} \quad \text{for } i = 1, 2, \dots, n.$$

Argue mathematically how varying the temperature T affects the output probabilities. In particular, show that as the temperature gets low ($T \to 0$), we will sample fewer and fewer distinct outputs, ultimately simply selecting the most probable token, while as the temperature gets high ($T \to \infty$), we will tend to sample all tokens with approximately the same probability.

Let $z = (z_1, \dots, z_n)$ be the logits and

$$\big[\mathrm{softmax}_T(z)\big]_i = p_i(T) := \frac{\exp(z_i/T)}{\sum_{j=1}^n \exp(z_j/T)}, \qquad i = 1, \dots, n,$$

for temperature $T > 0$. For any i, k we have

$$\frac{p_i(T)}{p_k(T)} = \frac{\exp(z_i/T)}{\exp(z_k/T)} = \exp\left(\frac{z_i - z_k}{T}\right).$$

If $z_i > z_k$, then $z_i - z_k > 0$ and this ratio *increases* as $T \to 0$ and tends to $+\infty$. Conversely, as $T \to \infty$, we have $\frac{z_i - z_k}{T} \to 0$ and hence $\frac{p_i(T)}{p_k(T)} \to 1$. Thus, decreasing T makes the distribution more peaked (concentrated on the largest logits), while increasing T flattens it.

Limit $T \to 0$: Let $M = \max_j z_j$ and define the index set of maximizers

$$S := \{i : z_i = M\}.$$

Pick any $k \in S$. For $T > 0$ we can write

$$p_i(T) = \frac{\exp(z_i/T)}{\sum_{j=1}^n \exp(z_j/T)} = \frac{\exp\big((z_i - M)/T\big)}{\sum_{j=1}^n \exp\big((z_j - M)/T\big)}.$$

If $i \notin S$, then $z_i - M < 0$, so $\exp((z_i - M)/T) \to 0$ as $T \to 0$. If $i \in S$, then $z_i = M$ and hence $\exp((z_i - M)/T) = 1$ for all T. Therefore,

$$\lim_{T\to 0} p_i(T) = \begin{cases} \frac{1}{|S|}, & i \in S, \\ 0, & i \notin S. \end{cases}$$

In particular, if the maximum is unique ($|S| = 1$), the limit distribution is a point mass on the argmax of z. This means that for very low temperature we effectively "always pick" the most probable token (or split mass equally among several ties for the maximum), so we sample very few distinct outputs.

Limit $T \to \infty$: For each fixed i we have $\frac{z_i}{T} \to 0$ as $T \to \infty$, hence by continuity of the exponential,

$$\exp(z_i/T) \xrightarrow{T\to\infty} \exp(0) = 1.$$

Since the denominator is a finite sum of n terms, we may take limits termwise:

$$\sum_{j=1}^n \exp(z_j/T) \xrightarrow{T\to\infty}; \sum_{j=1}^n 1 \;=\; n.$$

Therefore,

$$\boxed{\lim_{T\to\infty} \frac{\exp(z_i/T)}{\sum_{j=1}^n \exp(z_j/T)} \;=\; \frac{1}{n} \quad \text{for all } i = 1, \dots, n.}$$

In words: as the temperature grows, the softmax distribution converges to the uniform distribution over the n indices.

Exercise 4 (Positional encodings and relative distances - ⋆⋆)**.** Consider a simplified sinusoidal positional encoding with a single frequency ω (a simplified version of the one used in the original Transformer paper [1]). The positional encoding of a position $p \in \mathbb{R}$ is defined as,

$$\mathrm{pe}_\omega(p) = \begin{bmatrix} \sin(\omega p) \\ \cos(\omega p) \end{bmatrix},$$

and the full encoding with frequencies $\omega_1, \dots, \omega_{d/2}$:

$$\mathrm{PE}(p) = \big(\sin(\omega_1 p), \cos(\omega_1 p), \dots, \sin(\omega_{d/2} p), \cos(\omega_{d/2} p)\big).$$

(i) Show that for any p,

$$\|\mathrm{pe}_\omega(p)\|_2^2 = 1.$$

Interpret geometrically what happens to $\mathrm{pe}_\omega(p)$ as p varies.

Consider the unit circle in $\mathbb{R}^2$:

$$\mathcal{C} = \{(u, v) \in \mathbb{R}^2 : u^2 + v^2 = 1\}.$$

By definition of sine and cosine, for any real angle x the point on the unit circle at angle x (measured from the positive u-axis) has coordinates

$$(\cos x,\ \sin x).$$

Since this point lies on $\mathcal{C}$, it satisfies the circle equation:

$$(\cos x)^2 + (\sin x)^2 = 1.$$

For a single frequency ω, the positional encoding is

$$\mathrm{pe}_\omega(p) = \begin{pmatrix} \sin(\omega p) \\ \cos(\omega p) \end{pmatrix}.$$

Its squared ℓ_2-norm is

$$\|\mathrm{pe}_\omega(p)\|_2^2 = \sin^2(\omega p) + \cos^2(\omega p) = 1,$$

using the identity $\sin^2 x + \cos^2 x = 1$. Hence $\|\mathrm{pe}_\omega(p)\|_2^2 = 1$ for all p.

Geometric interpretation: The vector $\mathrm{pe}_\omega(p)$ always lies on the *unit circle* in $\mathbb{R}^2$. As p varies, the angle $\theta = \omega p$ changes linearly, so $\mathrm{pe}_\omega(p)$ moves along the unit circle at constant angular speed ω. In particular, the mapping is periodic in p with period $2\pi/\omega$.

(ii) Compute the dot product of encodings of two positions for a single frequency,

$$\mathrm{pe}_\omega(p)^T \mathrm{pe}_\omega(q) = \sin(\omega p)\sin(\omega q) + \cos(\omega p)\cos(\omega q),$$

and use the identity

$$\cos(A - B) = \cos A \cos B + \sin A \sin B$$

to show that the dot product only depends on the distance between p and q, but not on the absolute positions.

Let

$$\mathrm{pe}_\omega(p) = \begin{pmatrix} \sin(\omega p) \\ \cos(\omega p) \end{pmatrix}, \qquad \mathrm{pe}_\omega(q) = \begin{pmatrix} \sin(\omega q) \\ \cos(\omega q) \end{pmatrix}.$$

Then their dot product is

$$\mathrm{pe}_\omega(p)^\top \mathrm{pe}_\omega(q) = \sin(\omega p)\sin(\omega q) + \cos(\omega p)\cos(\omega q).$$

Using the trigonometric identity

$$\cos(A - B) = \cos A \cos B + \sin A \sin B,$$

with $A = \omega p$ and $B = \omega q$, we obtain

$$\begin{aligned} \mathrm{pe}_\omega(p)^\top \mathrm{pe}_\omega(q) &= \cos(\omega p - \omega q) \\ &= \cos\big(\omega(p - q)\big). \end{aligned}$$

Hence the dot product depends on p and q only through their difference $p - q$, i.e. only on the (signed) relative distance between the two positions, and not on their absolute locations.

(iii) Show that the dot product of the full encodings can be written as

$$\langle \mathrm{PE}(p), \mathrm{PE}(q) \rangle = \sum_{k=1}^{d/2} \cos\big(\omega_k (p - q)\big).$$

Explain why this means that sinusoidal positional encodings implicitly encode *relative* position information (the offset $p - q$), even though they are defined using absolute positions p and q. Think about how the attention mechanism can learn to prefer certain relative distance patterns (e.g. attend more to tokens $1 - 3$ steps back, or attend to tokens that a multiple of 4 away).

Recall the full sinusoidal positional encoding (with d even)

$$\mathrm{PE}(p) = \big(\sin(\omega_1 p), \cos(\omega_1 p), \ldots, \sin(\omega_{d/2} p), \cos(\omega_{d/2} p)\big) \in \mathbb{R}^d.$$

Its dot product with $\mathrm{PE}(q)$ is the sum over the $d/2$ frequency-pairs:

$$\langle \mathrm{PE}(p), \mathrm{PE}(q)\rangle = \sum_{k=1}^{d/2} \Big(\sin(\omega_k p)\sin(\omega_k q) + \cos(\omega_k p)\cos(\omega_k q)\Big).$$

Using the identity $\cos(A-B) = \cos A\cos B + \sin A \sin B$ with $A = \omega_k p$ and $B = \omega_k q$, each summand becomes

$$\sin(\omega_k p)\sin(\omega_k q) + \cos(\omega_k p)\cos(\omega_k q) = \cos\big(\omega_k (p-q)\big).$$

Therefore

$$\langle \mathrm{PE}(p), \mathrm{PE}(q)\rangle = \sum_{k=1}^{d/2} \cos\big(\omega_k (p-q)\big).$$

Why this implies relative-position information is encoded: The expression depends on p and q *only through the offset* $\Delta = p - q$. So, although $\mathrm{PE}(p)$ and $\mathrm{PE}(q)$ are computed from absolute positions, their similarity (dot product) is a function of relative distance:

$$\langle \mathrm{PE}(p), \mathrm{PE}(q)\rangle = f(p-q), \quad \text{where} \quad f(\Delta) = \sum_{k=1}^{d/2} \cos(\omega_k \Delta).$$

In self-attention, scores typically involve dot products between transformed token representations, e.g. $(x_p + \mathrm{PE}(p))W_Q$ and $(x_q + \mathrm{PE}(q))W_K$. Because dot products of positional components produce functions of $(p-q)$, the attention mechanism can learn weights (via W_Q, W_K) that make certain offsets yield larger scores.

(iv) Show that this dependence on the relative distance also "enters" the attention weights. Assume for simplicity that tokens are equal to the positional encodings (the part that depends on the input is zero) and compute the attention weight using the usual formula, for some query and key matrices $\boldsymbol{W}_q, \boldsymbol{W}_k \in \mathbb{R}^{d\times d}$.

Assume the token at position p equals its positional encoding:

$$x_p = \mathrm{PE}(p) \in \mathbb{R}^d, \qquad X = \begin{pmatrix} x_1^\top \\ \vdots \\ x_n^\top \end{pmatrix} \in \mathbb{R}^{n\times d}.$$

With (single-head) attention projections $W_q, W_k \in \mathbb{R}^{d\times d_k}$,

$$Q = XW_q, \qquad K = XW_k,$$

and the (pre-softmax) attention score from position p to q is

$$s_{pq} = \frac{1}{\sqrt{d_k}} Q_p^\top K_q = \frac{1}{\sqrt{d_k}} (x_p W_q)^\top (x_q W_k) = \frac{1}{\sqrt{d_k}} x_p^\top \underbrace{(W_q W_k^\top)}_{=:M} x_q.$$

So the score is a bilinear form in the positional encodings.

To see the role of relative distance concretely, consider first the single-frequency case ($d = 2$), so $x_p = \mathrm{pe}_\omega(p)$ and $M \in \mathbb{R}^{2\times 2}$. Writing

$$\mathrm{pe}_\omega(p) = \begin{pmatrix} s_p \\ c_p \end{pmatrix} \quad \text{with} \quad s_p = \sin(\omega p),\ c_p = \cos(\omega p),$$

we obtain

$$s_{pq} = \frac{1}{\sqrt{d_k}} \begin{pmatrix} s_p & c_p \end{pmatrix} \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} s_q \\ c_q \end{pmatrix}.$$

Expanding and regrouping with the identities

$$s_p s_q + c_p c_q = \cos(\omega(p-q)), \qquad c_p s_q - s_p c_q = \sin(\omega(p-q)),$$

shows that (in general) s_{pq} contains terms that are functions of $\omega(p-q)$; i.e. the relative offset *appears inside* the attention score, and therefore also inside the attention weight

$$\alpha_{pq} = \frac{\exp(s_{pq})}{\sum_{j=1}^{n} \exp(s_{pj})}.$$

In particular, if M is (for this frequency pair) of the rotation/scaled-rotation form

$$M = \begin{pmatrix} a & -b \\ b & a \end{pmatrix},$$

then one gets the clean dependence

$$s_{pq} = \frac{1}{\sqrt{d_k}} \big(a \cos(\omega(p-q)) + b \sin(\omega(p-q))\big),$$

which depends *only* on the relative distance $p - q$.

For the full encoding, the same reasoning applies frequency-wise (each $(\sin(\omega_k p), \cos(\omega_k p))$ pair can contribute a term depending on $\omega_k(p-q)$), so relative-distance information can directly shape the attention weights.

Exercise 5 (Debugging Transformers $\star\star\star$)**.** Below is a PyTorch implementation of a single Transformer encoder layer. It is supposed to implement multi-head self-attention, a feed-forward MLP, residual (skip) connections, and LayerNorm.

1. The code snippet below contains five conceptual and implementation mistakes (shape issues, wrong dimensions, missing scaling, etc.). Identify them and explain why each is wrong.
2. Propose a corrected version of the layer. Specify
   - the correct shapes of q, k, v and the attention output,
   - where the scaling by $\sqrt{d_{\text{head}}}$ is applied,
   - how the mask is applied to the attention scores,
   - the correct order of residual connections and LayerNorms,
   - the correct dimensions of the feed-forward layers and LayerNorms.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class BuggyTransformerLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_head = d_model // n_heads

        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)
        self.W_o = nn.Linear(d_model, d_model)

        self.ff1 = nn.Linear(d_model, d_ff)
        self.ff2 = nn.Linear(d_model, d_ff)

        self.ln1 = nn.LayerNorm(d_ff)
        self.ln2 = nn.LayerNorm(d_ff)

    def forward(self, x, mask=None):
        B, T, D = x.shape

        q = self.W_q(x)
        k = self.W_k(x)
        v = self.W_v(x)

        q = q.view(B, T, self.n_heads, self.d_head).permute(0, 2, 1, 3)
        k = k.view(B, T, self.n_heads, self.d_head).permute(0, 2, 1, 3)
        v = v.view(B, T, self.n_heads, self.d_head).permute(0, 2, 1, 3)

        attn_scores = torch.matmul(q, k.transpose(-1, -2))


        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, 0.0)

        attn_weights = F.softmax(attn_scores, dim=-1)
        attn = torch.matmul(attn_weights, v)

        attn = attn.permute(0, 2, 1, 3)
        attn = attn.view(B, T, D)

        x = x + self.W_o(attn)
        x = self.ln1(x)

        ff = self.ff1(x)
        ff = F.relu(ff)
        ff = self.ff2(ff)

        x = x + ff
        x = self.ln2(x)
        return x
```

1. **Missing scaling by $\sqrt{d_{\text{head}}}$.** Scaled dot-product attention uses

$$S = \frac{QK^\top}{\sqrt{d_{\text{head}}}}$$

to keep score magnitudes roughly independent of dimension; without scaling, softmax saturates and gradients degrade.

2. **Mask is applied incorrectly (values).** The code uses `masked_fill(..., 0.0)` on the scores. This does *not* suppress masked positions after softmax (they can still receive nonzero probability). Correct is to set masked scores to $-\infty$ (or a very large negative number) *before* softmax so that their softmax probability is (numerically) zero.

3. **Mask is applied incorrectly (shape/broadcasting).** Attention scores have shape $(B, n_{\text{heads}}, T, T)$. A typical padding mask has shape (B, T) (keys that exist). It must be

broadcast to $(B, 1, 1, T)$ (or equivalent) to mask *key* positions for every head and every query.

4. **Feed-forward network second linear layer has wrong dimensions.** The code defines `ff2 = Linear(d_model, d_ff)` but it should map *back* to d_{model}:

$$\text{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2, \qquad W_1 : \ d_{\text{model}} \to d_{\text{ff}}, \quad W_2 : \ d_{\text{ff}} \to d_{\text{model}}.$$

   Otherwise the residual addition $x + \text{FFN}(x)$ is impossible (shape mismatch).

5. **LayerNorm dimensions are wrong.** The buggy code uses `LayerNorm(d_ff)` but applies it to x which has last dimension $D = d_{\text{model}}$. LayerNorm must match the last dimension of the tensor it normalizes; here it should be `LayerNorm(d_model)`.

Exercise 6 (Counting parameters in a Transformer layer - $\star$)**.** Give a formula for the number of parameters in the transformer layer from the previous question. Evaluate the formula for $d_{\text{model}} = 128$ and $d_{\text{ff}} = 512$.

Let $D := d_{\text{model}}$. We count parameters of the corrected encoder layer (self-attention + FFN + two LayerNorms), matching the previous solution where W_q, W_k, W_v have *no* bias (as in the provided code), while W_o and the FFN linears use biases.

Self-attention projections

- $W_q, W_k, W_v : \ \mathbb{R}^D \to \mathbb{R}^D$, bias=False:

$$3 \cdot (D \cdot D) = 3D^2$$

- Output projection $W_o : \ \mathbb{R}^D \to \mathbb{R}^D$ (with bias):

$$D \cdot D + D = D^2 + D$$

So attention parameters: $3D^2 + (D^2 + D) = 4D^2 + D$. (Notice this does *not* depend on n_{heads} as long as the projections are $D \to D$.)

Feed-forward network

- $W_1 : \ \mathbb{R}^D \to \mathbb{R}^{d_{\text{ff}}}$ (with bias): $D\,d_{\text{ff}} + d_{\text{ff}}$
- $W_2 : \ \mathbb{R}^{d_{\text{ff}}} \to \mathbb{R}^D$ (with bias): $d_{\text{ff}}\,D + D$

So FFN parameters:

$$(Dd_{\text{ff}} + d_{\text{ff}}) + (d_{\text{ff}}D + D) = 2Dd_{\text{ff}} + d_{\text{ff}} + D.$$

LayerNorms Each LayerNorm over D has a scale and shift $(\gamma, \beta) \in \mathbb{R}^D \times \mathbb{R}^D$, i.e. $2D$ parameters. With two LayerNorms: $4D$ parameters.

Total

$$N(D, d_{\text{ff}}) = (4D^2 + D) + (2Dd_{\text{ff}} + d_{\text{ff}} + D) + 4D = 4D^2 + 2Dd_{\text{ff}} + d_{\text{ff}} + 6D.$$

Evaluation for $D = 128$, $d_{\text{ff}} = 512$.

$$N = 4 \cdot 128^2 + 2 \cdot 128 \cdot 512 + 512 + 6 \cdot 128 = 65536 + 131072 + 512 + 768 = 197888.$$

If all four attention linears had biases), add $3D$ extra parameters (biases for W_q, W_k, W_v), i.e. $197888 + 3 \cdot 128 = 198272$.

Coding

For this problem set, we do not provide Gradescope autograding, because the expected outputs are plots and other visualizations, which are difficult to assess automatically. Instead, we will include reference plots in the model solution and review them during the tutorial sessions.

Exercise 7 (Iterating max pooling - $\star$).

In this exercise we study what happens if we repeatedly apply max pooling *without* downsampling. You will compare this to iterated Gaussian blurring. We work with a sample image `sample_image.png` represented as a tensor of shape $[1, 1, H, W]$ (batch size 1, one channel). A skeleton Python file `iter_maxpool.py` is provided. You only need to fill in the parts marked `# TODO`. In `helper_maxpool.py`, you can find functions that are provided to you and do not need to be edited by you.

1. Implement a function `maxpool_iter(x, k=3, iters=1)` in PyTorch that
   - takes an input tensor `x` of shape `[B, C, H, W]`,
   - applies 2D max pooling with kernel size k, stride 1 and padding $k/2$ (use `torch.nn.functional.max_pool2d`) (note that because stride is 1, this operation simply replaces each pixel by the maximum value in its $k \times k$ neighbourhood, but it does not downsample the image),
   - repeats this operation `iters` times,
   - and returns a tensor of the same shape as `x`.
2. Implement a small 2D Gaussian blur and an iterator `gaussian_blur_iter(x, k=5, sigma=1.0, iters=1)`:
   - First write a helper `gauss_kernel_2d(k, sigma)` that builds a $k \times k$ Gaussian kernel with standard deviation `sigma` and normalises it so that the entries sum to 1. More details can be found in the skeleton code.
   - Then implement `gaussian_blur_iter` that applies this kernel `iters` times using `torch.nn.functional.conv2d` with appropriate padding. More details can be found in the skeleton code.
3. Load a grayscale image from disk, normalise it to $[0, 1]$, convert it to a tensor of shape `[1, 1, H, W]` in `load_image_as_tensor()`:
4. In `tensor_to_numpy_image)()`, convert a tensor of shape [1, 1, H, W] to a 2D numpy image of shape [H, W].
5. We provided a `run_experiment()` function that you can find in `helper_maxpool.py`. It plots the results in a 2×5 grid: the first row shows max pooling for different iteration counts, and the second row shows Gaussian blur for the same iteration counts. Run the `main` block and inspect the results. Briefly answer:
   - What happens to fine details and textures as you increase the number of max pooling iterations?
   - What kind of filter does this correspond to?
   - How does the result differ from iterated Gaussian blur (for example at edges and in bright regions)?
   - Are both filters linear?

Reference plot:

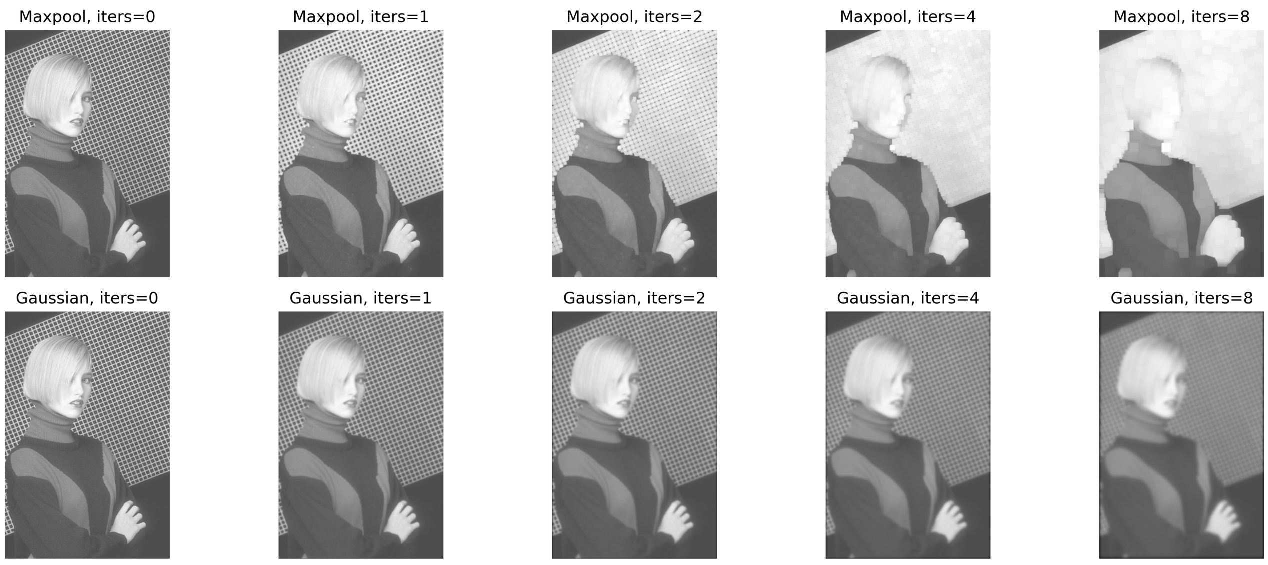

Figure 1: Reference solution plot for the maxpool exercise.

Qualitative observations:

1. Low-pass effect:
   - As the number of maxpool iterations increases, fine details and high-frequency textures disappear from the image.
   - Large homogeneous regions become flatter and more uniform.
   - This shows a low-pass behaviour: high frequencies are suppressed more and more.
2. Differences between maxpool and Gaussian blur:
   - Max pooling is nonlinear and tends to ”grow” bright regions outward. Bright edges become thicker and small bright spots expand.
   - Gaussian blur is linear and averages both bright and dark values. Edges become smoother but do not expand in the same way.
   - Max pooling preserves the maximum in each local neighbourhood, so local contrast is often higher than in the Gaussian case, where everything is averaged.
3. Receptive field intuition:
   - Each application of 3x3 stride-1 pooling looks at a 3x3 neighbourhood.
   - After many iterations, the value at a pixel depends on a larger and larger neighbourhood of the original image.
   - This effective neighbourhood size grows roughly with the number of iterations, which is why we see stronger smoothing as we iterate the pooling.

Conclusion: Repeated stride-1 max pooling behaves like a nonlinear low-pass filter: it removes small-scale variations and keeps only large-scale structure, but it does so in a different way than a linear Gaussian blur.

Exercise 8 (Gradients in input space - ⋆⋆)**.**

In this exercise you will study how a classifier changes with respect to its input by visualizing gradients in a two-dimensional input space. You will work with a simple synthetic classification problem in $\mathbb{R}^2$ and a small neural network classifier implemented in PyTorch. We consider a classifier $f_\theta : \mathbb{R}^2 \to \mathbb{R}^2$ which maps an input $x = (x_1, x_2) \in \mathbb{R}^2$ to two real-valued scores $f_\theta(x) = (s_0(x), s_1(x))$. From these scores we obtain class probabilities by a softmax:

$$p_\theta(y = c \mid x) = \frac{\exp(s_c(x))}{\exp(s_0(x)) + \exp(s_1(x))}, \quad c \in \{0, 1\}.$$

We are interested in the gradient of the class probability with respect to the input,

$$\nabla_x p_\theta(y = 1 \mid x) \in \mathbb{R}^2.$$

and in how this gradient field looks in the input plane. You are given a Python skeleton file in `gradients2d.py` that contains stubs for the functions you need to implement. In `helper.py`, you can find functions that are provided to you and should not be edited by you. One of these functions generates the toy dataset used in this exercise and the other function is used for plotting later in this exercise.

Information about the toy dataset generated in `make_toy_dataset()`: We denote by $x^{(i)} = \left(x_1^{(i)}, x_2^{(i)}\right)$ the i-th point in the dataset and collect all points in a matrix

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(N)} & x_2^{(N)} \end{bmatrix} \in \mathbb{R}^{N \times 2}.$$

This matrix X is returned by the `make_toy_dataset()` function. Additionally, it returns the class labels y (0 or 1).
Tasks:

1. Inspect the toy dataset returned by `make_toy_dataset()` (e.g. scatter plot). You do not need to implement this function. If you want, you can try out different seeds and see how the dataset changes.
2. Train a small neural network classifier :
   (a) Complete the forward method in `class MLP2d(nn.Module)`. It should apply self.net to the input.
   (b) Implement a simple training loop in `train_classifier()`. Please use torch.utils.data.TensorDataset and DataLoader. As learning algorithm, please use torch.optim.SGD or torch.optim.Adam. As a loss function please use nn.CrossEntropyLoss. Report the training accuracy at the end. Make sure your classifier reaches clearly better than random accuracy on the toy dataset.
3. Evaluate the class probability $p_\theta(y = 1|x)$ on a regular grid in the (x_1, x_2) plane. Implement the following steps in `evaluate_on_grid()`:
   (a) Create two one dimensional tensors xs and ys with steps equally spaced values between the given min and max.
   (b) Create a meshgrid of coordinates (XX, YY) of shape [steps, steps].
   (c) Stack the grid into a tensor grid of shape [steps * steps, 2] to feed into the model.
   (d) Compute the class scores for all grid points (`logits`), apply softmax in the class dimension (`probs`), and extract the probability for class 1 (`p1`).
   (e) Reshape the results back to a two dimensional array P of shape [steps, steps] that aligns with XX and YY.
4. Compute gradients of $p_\theta(y = 1|x)$ with respect to the input coordinates $x = (x_1, x_2)$ for each point on the grid. Implement the following steps in `compute_input_gradients()`:

(a) Create a meshgrid XX, YY as before (shape [steps, steps]) and stack into a tensor grid of shape [steps * steps, 2].

(b) Enable gradient tracking for grid by calling `grid.requires_grad_(True)`.

(c) Pass grid through the model to obtain scores of shape [steps * steps, 2].

(d) Apply softmax to obtain probabilities. Extract the probability for class 1 as a tensor p of shape [steps * steps] or [steps * steps, 1].

(e) Compute the gradient of the sum of these probabilities with respect to grid:

i. Call `model.zero_grad()`.

ii. Call `p1.sum().backward()`.

iii. The tensor `grads` has shape [steps * steps, 2] and contains $\nabla_x p_\theta(y = 1 \mid x)$ at each grid point. Define the first column of `grads` as GX of shape [steps, steps] and the second column as GY of shape [steps, steps].

5. In the `"__main__"` block: Pass the required inputs into the provided plotting function `plot_data_boundary_and_gradients()` that you can find in `helper.py`. You do not need to implement this function yourself. In this exercise, we use the standard rule "predict class 1 if $p_\theta(y = 1 \mid x) > 0.5$", so the decision boundary is the set of points where $p_\theta(y = 1 \mid x) = 0.5$.

6. Interpretation questions:

(a) Where in the input plane are the gradients $\nabla_x p_\theta(y = 1 \mid x)$ largest in magnitude? Are they closer to the decision boundary or far away from it?

(b) In regions where the model is very confident for class 0 or class 1, do the gradients tend to be large or small? Why does this make sense?

(c) Consider a point x that lies close to the decision boundary. If you move a small step in the direction of the gradient, does the probability $p_\theta(y = 1 \mid x)$ increase or decrease? How is this visible in your plot?

Reference plot:

```
Training accuracy: 94.5%
```

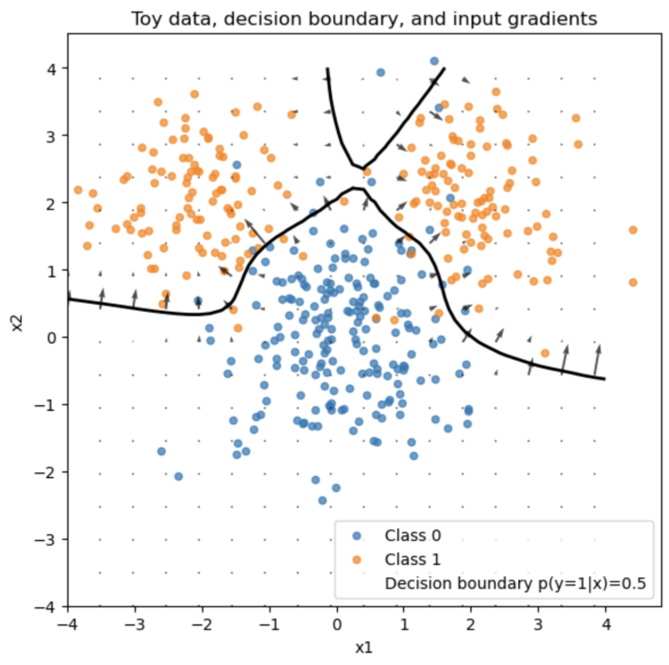

Figure 2: Reference solution plot for exercise 8.

Possible answer to the interpretative questions:

1. The gradients $\nabla_x p_\theta(y = 1 \mid x)$ are largest mainly in regions where the classifier is uncertain, that is, close to the decision boundary. There the probability changes quickly from “more class 0” to “more class 1”, so a tiny change in x has a strong effect on $p_\theta(y = 1 \mid x)$. Far away from the boundary the gradient magnitude is usually much smaller.

2. In regions where the model is very confident for class 0 or class 1, the gradients tend to be small. There the softmax output is close to 0 or close to 1. Changing the input slightly does not change the output probability much, so the derivative with respect to the input is small.

3. For a point x close to the decision boundary, moving a small step in the direction of the gradient $\nabla_x p_\theta(y = 1 \mid x)$ increases the probability $p_\theta(y = 1 \mid x)$. By definition, the gradient points in the direction of steepest increase of the function. In the plot this is visible because the arrows near the boundary point from the side with lower class-1 probability toward the side with higher class-1 probability, that is, from the “class 0 region” toward the “class 1 region”.

Exercise 9 (Explainability of Deep Networks- ⋆⋆⋆)**.**

Modern neural networks contain numerous layers and components. These networks work extremely well, but their inner workings are often inscrutable.

To improve the models, design new architectures, ensure safety and certification in critical applications (e.g., medical systems or systems vulnerable to adversarial attacks), and build trust, it is important to look for strategies to explain how the networks “make decisions”.

Explainability is still a developing research area. In this exercise, we explore one of the earliest approaches for explaining deep image recognition networks by identifying which parts of an input image were most responsible for the model’s predicted class. One natural idea is to examine the gradient of the output with respect to the input image (this is different from the gradient used during training). The gradient tells us how sensitive the network’s output is to changes in each pixel. Using a first-order Taylor expansion around the input x

$$f(x + \Delta x) = f(x) + \nabla_x f(x)^T \Delta x + \mathcal{O}(\Delta x) \tag{1}$$

Observe that $\nabla_x f(x)$ is the same size as that of the input x, in our case it is the image. Thus, we see that if the value $\nabla_x f(x)[i, j]$ is large at location (i, j), then that pixel strongly influences the class score. In practice, however, raw gradients can be quite noisy because of the many ReLU nonlinearities in modern networks. To reduce this noise, Springenberg *et al.* [2] proposed a modified version of ReLU for the backward pass known as guided backpropagation. In this exercise, you will compute these modified gradients for a ResNet-50 classifier to visualize which pixels in an image most strongly contribute to the predicted class.

Tasks:

1. **Load a pretrained ResNet-50 model:** Use `torchvision.models` to load ResNet-50 with pretrained weights (`IMAGENET1K_V1`), which outputs a 1000-dimensional vector for the 1000 ImageNet classes.

2. **Load and preprocess the provided image:** The image contains a bulldog and a cat, and the bulldog corresponds to the ImageNet class ”bull mastiff” (index 243). Convert the image into a PyTorch tensor, resize/crop it appropriately, and apply ImageNet normalization. (Look at how the models were trained and transform the provided image accordingly)

3. **Replace standard ReLU with the modified (guided) ReLU:** Use the provided implementation of the modified ReLU and substitute it for all ReLUs in the network so that the backward pass uses guided gradients.

4. **Compute the gradient of the class score w.r.t. the input:** Perform a forward pass, then call `backward` on the output at index 243 to backpropagate gradients from that specific class.

5. **Visualize the input gradient:** Plot the gradient of the input tensor as an image.

Observe which regions are highlighted and discuss why the network identifies those parts as important.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[2] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.