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

Theory

We start with a few more problems about convnets.

Exercise 1 (Why "transposed convolutions" - ⋆⋆)**.** 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 $\boldsymbol{x} = [x_0, x_1, x_2, x_3]^T$, which is padded with zeros to give $\tilde{\boldsymbol{x}} = [0, x_0, x_1, x_2, x_3, 0]^T$, and a filter with parameters $\boldsymbol{w} = [w_0, w_1, w_2]$.

(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 $\boldsymbol{z} = [z_0, z_1]^T$ 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.

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

Exercise 3 (Temperature in softmax - ⋆)**.** 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 $\boldsymbol{z} = [z_1, z_2, \ldots, z_n]$. The temperature-scaled softmax is defined as:

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

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.

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 $\omega_1, \ldots, \omega_{d/2}$:

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

(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.

(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.

(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).

(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 $\boldsymbol{W}_q, \boldsymbol{W}_k \in \mathbb{R}^{d \times d}$.

Exercise 5 (Debugging Transformers ⋆ ⋆ ⋆)**.** 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.

```
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
```

Exercise 6 (Counting parameters in a Transformer layer - ⋆)**.** 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 $d_{\text{ff}} = 512$.

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?

Exercise 8 (Gradients in input space - $\star\star$)**.**

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?

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

Modern neural networks—such as those used in language models, image generators, and even in smartphones often contain very deep architectures with many stacked layers. These networks work extremely well, but they are also difficult to understand internally.

To improve these models, design new architectures, or ensure safety in critical applications (e.g., medical systems or systems vulnerable to adversarial attacks), it is important to study explainability:

Why did a neural network make a particular prediction?

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

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

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.