Machine Learning Methods

Exercise 5

Last Update: 12.1.2025

1 General Description

This exercise centers on implementing and analyzing probabilistic models (Gaussian and Uniform Mixture Models) and a small-scale GPT2 transformer for language modeling. You will explore how to represent data as a mixture of distributions, optimize parameters using stochastic gradient descent, and evaluate model performance using visualization and statistical techniques. In the transformer section, you'll delve into attention mechanisms and autoregressive modeling to predict sequences.

You are expected to hand in a report, no longer than 8 pages, describing your experiments and answers to the questions in the exercise. The submission guidelines are described in Sec. 6, please read them carefully.

Note 1: You must explain your results in each question! Unless specified otherwise, an answer without explanation will not be awarded marks.
Note 2: When you are asked to plot something, we expect that plot to appear in the report.

2 Seeding

You should seed your code as in the previous exercises. Specifically, we will need to see both numpy and pytorch as follows:
(i) `np.random.seed(42)`
(ii) `torch.manual_seed(42)`
NOTE: It doesn't matter which seed you just that your results are reproducible!

3 Prerequisites

Before delving into implementing mixture models and Self-Attention, ensure an understanding of the mathematical theory. That way you will also gain the most

from the exercise. We provide skeleton code in two files: `mixture_models.py` and `transformer.py`. In addition, we provided you with `dataset.py` to help you with data handling.

4 Mixture Models

4.1 Data

As in the previous exercises, we will use the countries of Europe dataset.
Important: Normalize your data to a mean of zero and a standard deviation of one for each dimension, the model will not work otherwise

Gaussian Mixture Model

A Gaussian Mixture Model (GMM) is a probabilistic model used to represent data as a mixture of several Gaussian distributions. Essentially the probability space is broken down into multiple Gaussian. Using the law of total probability together with Bayes' rule, we can express the PDF $p(\mathbf{x})$ as:

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x}|k),$$

Where:

- $p(k) = \pi_k$: The prior probability of the k-th Gaussian.
- $p(\mathbf{x}|k) = \mathcal{N}(\mathbf{x}|\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\}$: The conditional distribution of $\mathbf{x}$ given component k.

Each Gaussian component is defined by a mean vector μ_k and a covariance matrix Σ_k:

$$p(\mathbf{x}|k) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}_k|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_k)\right).$$

In this exercise, **we will limit ourselves to a 2D diagonal covariance matrix**, therefore the conditional can be expressed only in terms of the mean $\begin{bmatrix}\mu_{k1}\\ \mu_{k2}\end{bmatrix}$ and the variance $\begin{bmatrix}\sigma_{k1}^2\\ \sigma_{k2}^2\end{bmatrix}$:

$$p(\mathbf{x}|k) = \frac{1}{2\pi\sigma_{k1}\sigma_{k2}} \exp\left(-\frac{1}{2}\left[\frac{(x_1 - \mu_{k1})^2}{\sigma_{k1}^2} + \frac{(x_2 - \mu_{k2})^2}{\sigma_{k2}^2}\right]\right),$$

The overall model is weighted by mixing coefficients π_k such that $\sum_k \pi_k = 1$.

The goal of training a GMM is to optimize the parameters $\{\mu_k, \sigma_k^2, \pi_k\}$ for $k = 1, \ldots, K$, where K is the number of Gaussian components. Typically, the Expectation-Maximization (EM) algorithm is used for parameter optimization, but here we will use Stochastic Gradient Descent (SGD) instead.

You must fill in the missing code in *mixture_model.py*, where you'll implement a GMM class and optimize its parameters.
You will optimize: *self.logits* (the probability of each Gaussian before applying the Soft-max operation), self.means (the mean of each Gaussian), and $self.log\_variances$ (the log of the variance of each Gaussian)
A few notes about optimization:

- In previous exercises we've optimized the weights of models without any restrictions on the weights. Here we need to restrict multiple parameters for the GMM to be valid.
- We need to guarantee $p(k) = \pi_k$ represents a valid distribution, for this we need to ensure $\pi_k \geq 0$ and $\sum_{k=1}^{K} \pi_k = 1$. To tackle this, we'll optimize logits instead of the probabilities themselves. The logits have no restrictions on their values (thus can be optimized using GD) and after Soft-max, represent a valid probability distribution.
- A valid covariance matrix must be symmetric and PSD (Positive Semi-Definite), therefore directly optimizing the covariance matrix will result in invalid covariance matrices. By exploring only diagonal matrices we are guaranteed the covariance matrix will be valid as long as the entries are positive. To enforce positivity we won't optimize the entries of the diagonal directly, instead we'll optimize their log. The log-variance has no restriction therefore suitable for GD.
- When maximizing the log-likelihood we're maximizing an expression of the form: $\log p(\mathbf{x}) = \log \sum_{k=1}^{K} p(k)p(\mathbf{x}|k)$.
  $p(k)$, and $p(\mathbf{x}|k)$ can be very small, multiplying them can cause underflow. To avoid numerical issues we express $p(k)p(\mathbf{x}|k)$ with logs:

  $$\log p(\mathbf{x}) = \log \sum_{k=1}^{K} p(k)p(\mathbf{x}|k) = \log \sum_{k=1}^{K} \exp[\log p(k) + \log p(\mathbf{x}|k)]$$

  Notice that $\log p(\mathbf{x}) + \log p(\mathbf{x}|k)$ can result in large negative values, i.e. $p(x|k) = 10^{-1000} \rightarrow \log(p(x|k) = -1000$, exp on such values will cause underflow, therefore to avoid these sort of numerical issues, PyTorch has a special function called torch.logsumexp Use it when optimizing

Task

.
Fill in the missing code in the GMM module in *mixture_model.py*:

- Initialize logits, means, and logvar in the code according to the dimension of the data and the number of Gaussians.

- Fill the forward function, the function receives a batch of samples and computes the log-likelihood of each sample in the batch compute

$$\log p(\mathbf{x}) = \log \sum_{k=1}^{K} p(k)p(\mathbf{x}|k) = \log \sum_{k=1}^{K} \exp[\log p(k) + \log p(\mathbf{x}|k)]$$

  Use $nn.functional.log\_softmax$ for $\log p(k)$ and the expression below for the conditional

$$\log p(\mathbf{x}|k) = -\log(2\pi) - \log(\sigma_{k1}^2) - \log(\sigma_{k2}^2) - \frac{1}{2}\left[\frac{(x_1 - \mu_{k1})^2}{\sigma_{k1}^2} + \frac{(x_2 - \mu_{k2})^2}{\sigma_{k2}^2}\right]$$

  Remember to use logsumexp over the logs of the conditional.

- Fill the loss function, it should return the negative mean *log_likelihood*: $-\frac{1}{n}\sum \log p(\mathbf{x}_n; \theta)$

Sampling:
To sample from a GMM you first sample a Gaussian according to $k \sim p(k)$ with *torch.multinomial*. After sampling a Gaussian you can sample $\mathbf{x} \sim p(\mathbf{x}|k)$ using torch.randn to sample from a standard Normal distribution, i.e. $z \sim \mathcal{N}(0, I)$ and then using $x = \mu + z * \sigma$ independently along each dimension. This results in $\mathbf{x} \sim p(\mathbf{x})$

- Fill the sample function. First sample a Gaussian using the logits and then sample according to that Gaussian. You should return either a numpy array or a torch tensor of shape (n, d).

- Fill the conditional_sample function, it takes as an argument the index k, and returns a sample from the conditional distribution of the kth Gaussian.

Unless specified otherwise, your default parameters should be the ones given in the code
Answer the following:

1. For $n\_components = [1, 5, 10, n\_classes^1]$ optimize the GMM:

   (a) Display a scatter plot with 1000 samples from the GMM (use 'sample' function)

   (b) Display a scatter plot with 100 samples from each Gaussian (use 'conditional_sample'). You need to loop over the different components. For each iteration, sample 100 samples. Add all $100 * n\_components$ resulting samples in a single scatter plot, and color the samples according to their corresponding Gaussian.

[1] The number of classes in your datasets, i.e. countries in Europe

2. For $n\_components = n\_classes$ optimize the GMM.

   (a) For each epoch in $[1, 10, 20, 30, 40, 50]$, display the same two plots as before.

   (b) Plot the training and testing mean $log\_likelihood$ vs. epoch (Don't confuse with the loss).

   (c) Repeat (a+b), this time you will initialize the means as the mean location of each country in Europe (the mean of Gaussian k is the mean coordinate of country k in the training dataset). Compare your results to the previous question, did you do better or worse?

Uniform Mixture Model

In this section, you will implement a mixture model just as before, but instead of the mixture being composed of Gaussians, it should be composed of 2D uniform distributions. as before: $p(\mathbf{x})$ as:

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x}|k),$$

Where:

- $p(k) = \pi_k$: The prior probability of the k-th Gaussian.
- $p(\mathbf{x}|k) = \mathcal{U}(\mathbf{x}|\{\boldsymbol{c}_k, \mathbf{s}_k\})$: The conditional distribution of $\mathbf{x}$ given component k.
- $\mathbf{c}_k = \begin{bmatrix} c_{k1} \\ c_{k2} \end{bmatrix}$, and $\mathbf{s}_k = \begin{bmatrix} s_{k1} \\ s_{k2} \end{bmatrix}$ represent the center, and the size of the uniform distribution accordingly

Each Uniform component is defined by a center vector c and a size vector s, in the 2D case:

$$f_{\mathbf{x}}(\mathbf{x}) = \begin{cases} \frac{1}{s_1 s_2} & \text{if } c_x - \frac{s_1}{2} \leq x_1 \leq c_x + \frac{s_1}{2},\ c_y - \frac{s_2}{2} \leq x_2 \leq c_y + \frac{s_2}{2} \\ 0 & \text{otherwise} \end{cases}$$

The goal of training is to optimize the parameters $\{\mathbf{c}_k, \mathbf{s}_k, \pi_k\}$ for $k = 1, \ldots, K$ where K is the number of components.
You will optimize: self.logits (the probability of each uniform before applying the Soft-max operation), self.centers (the center of each uniform), and $self.log\_sizes$ (the log of the size of each uniform)
A few notes about optimization:

- For the same reason as you've seen with the logits in GMM we'll optimize the logits

- For ensuring positivity in the diagonal entries in the covariance matrix, we'll parameterize the $\log \mathbf{s}_k$ instead of $\mathbf{s}_k$ directly, ensuring $\mathbf{s}_k > 0$
- if a sample is not in any component, the log-likelihood of that sample will be $-\infty$, this will lead to nan values in your loss. To avoid this, don't assign $-\infty$ to log probabilities, instead assign a large negative number, i.e. $-1e^6$

Task

Fill in the missing code in the UMM (Uniform Mixture Module) module in *mixture_model.py*:

- Initialize logits, mins, and *log_sizes* in the code according to the dimension of the data and the number of uniforms.
- Fill the forward function, similar to the Gaussian case. What changes here is $\log p(\mathbf{x}|k)$:

$$\log p(\mathbf{x}) = \begin{cases} -\log(s_1) - \log(s_2) & \text{if } c_x - \frac{s_1}{2} \leq x_1 \leq c_x + \frac{s_1}{2},\ c_y - \frac{s_2}{2} \leq x_2 \leq c_y + \frac{s_2}{2} \\ -\infty & \text{otherwise} \end{cases}$$

- Fill in the rest functions in the class similar to what you did with GMM.
- **TIP**: You can use $torch.distributions.Uniform$ to sample from a uniform distribution.

Unless specified otherwise, your default parameters should be the ones given in the code.
Answer the following:

1. Repeat the same questions 1-2 as in the GMM section.
2. Look at the scatter plot with the different colors, do you notice a certain trend in the different uniforms' support [2] as we advance through the epochs? Do you notice a trend in the centers of the uniform components? Why do you think it happens? What is the problem with gradient descent in the Uniform Mixture Model that we don't have in the Gaussian Mixture Model?

Transformer

4.2 Data

We provided a dataset of Shakespeare writings, alongside a class for handling the data. We recommend you read through it and understand how it works.

[2] The support of a random variable is the set of values where its probability density function (PDF) or probability mass function (PMF) is non-zero

Inside it, you'll find how we encode and decode the characters into and from characters to tokens, the vocabulary (a list of all characters in the corpus), and the different datasets. Different from previous exercises, the epoch will not be defined as a pass through all of the data, instead, it will be a predefined number of samples from the data.

4.3 Objective

In this exercise, your task is to predict the next character given the previous characters, i.e. Auto Regressive model. The number of characters your model will consider is dynamic but is bounded by *block_size*, i.e. $p(x_{i+1}|x_i,..,x_1) = p(x_{i+1}|x_i,...x_{i-block\_size+1})$.
Given a sequence $X = [x_1, x_2, ...x_T]$ the model will output $Y = [y_1, y_2, .., y_T]$, where y_i is the prediction for the next character given $[x_1, ..., x_i]$. You will then use Cross-Entropy loss, comparing $output = [y_1, .., y_T]$ to $target = [x_2, .., x_{T+1}]$ (in practice y_i is the logits over the possible characters for the next token, so use Cross Entropy just as we did for classification).

Accuracy: In this exercise you will measure the ratio between the number of sentences in which the model correctly predicted the **last** character in the input sentence vs. the total number of sentences, i.e. $\frac{\sum_{i=1}^{n} \mathbb{1}_{y_n^T = x_n^{T+1}}}{n}$.
You will select the predicted token as the token with the highest score.

4.4 Model

In this exercise, we will use a small modified version of a GPT2 transformer. The model will consist of several *causal-self-attention* blocks, stacked one after the other, composing our transformer architecture *GPT*.
Everything here will be on a small scale so we don't even need a GPU for training.
Your model implementation will involve building a **CausalSelfAttention** layer, the causal self-attention is similar to regular self-attention but we add a mask on the attention matrix.
why causal?: Notice that the model aims to predicts $output = [y_1, .., y_T]$. The input vector is $input = [x_1, ..., x_T]$, and the target is $target = [x_2, .., x_{T+1}]$. This means that For every $i < T$ the target token is present in the input vector. By adding a mask to the self-attention matrix we restrict the layer so that for every $i \leq T$ the output embedding y_i can only depend on the embeddings $[x_1, ..., x_i]$

4.5 Causal Self Attention

Given $input = [x_1, ..., x_T]$ where x_i is the ith token and T is the sequence length (number of characters is the sequence) the *causal-self-attention* performs:

1. **Key, Query, and Value Projections**:

$$\text{For } \mathbf{x} \in \mathbb{R}^{B \times T \times input\_dim}$$
$$q, k, v = \text{Linear}(x) \in \mathbb{R}^{B \times T \times 3C}$$

This is equivalent to defining 3 separate Linear layers for q, k, v.
Notations:

- $C = n_h \times d_h$, i.e. for ith token $q_i = [q_{i1}, ..., q_{in_h}]$.
  Similarly for k and v.
- n_h = number of heads
- d_h = dimension of head

2. **Splitting Heads**: After projection, q, k, and v are reshaped and split into n_h heads:

$$q, k, v \in \mathbb{R}^{B \times T \times C} \rightarrow \mathbb{R}^{B \times n_h \times T \times d_h}$$

where $d_h = \frac{C}{n_h}$ is the dimensionality of each head. This is done by reshaping and transposing:

$$q = q.\text{view}(B, T, n_h, d_h).\text{transpose}(1, 2)$$

Similarly for k and v.

3. **Multi-Head Attention**: The attention mechanism operates on the split heads:

$$\text{Attention}(q_i, k_i, v_i) = \text{Softmax}\left(\frac{q_i k_i^\top}{\sqrt{d_h}} + \text{Mask}\right) v_i$$
$$q_i, k_i, v_i \in \mathbb{R}^{d_h}$$
$$\rightarrow \text{Attention}(q, k, v) \in \mathbb{R}^{B \times n_h \times T \times d_h}$$

TIP: In PyTorch, you can perform matrix multiplication on tensors of higher dimensions, by default, it will consider all dimensions before the last two in the tensor, as a batch, i.e. $(B, n_h, T, d_h)@(B, n_h, d_h, T) \rightarrow (B, n_h, T, T)$

4. **Causal Mask**: A lower triangular matrix is added to enforce causality:

$$\text{Mask}[i, j] = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{otherwise} \end{cases}$$

5. **Re-arranging Heads**: After applying the attention mechanism, the outputs from the multiple heads are concatenated back into their original shape:

$$y \in \mathbb{R}^{B \times n_h \times T \times d_h} \rightarrow \mathbb{R}^{B \times T \times C}$$

6. **Output Projection**:
Finally, the output is projected back to the original embedding dimensionality:

$$y = \text{Linear}(y) \in \mathbb{R}^{B \times T \times input\_dim}$$

4.6 Generation

This model is a **Auto-Regressive** model, we can use it both for estimating the probability of a character given the previous characters, i.e. $p(\mathbf{c}_i|\mathbf{c}_{i-1}, ..., \mathbf{c}_1)$, and for generating sentences by sampling from the distribution using *torch.multinomial*. To generate a sentence we sample character after character, each time conditioning on the characters we sampled up to that point.
Note: if the number of characters in the condition is larger then the $block\_size$ you will use only the last $block\_size$ tokens, i.e. $p(\mathbf{c}_i|\mathbf{c}_{i-1}, ..., \mathbf{c}_{i-block\_size})$

Top-k-sampling trick: In cases where the vocabulary is very large, sampling directly from $p(\mathbf{c}_i|\mathbf{c}_{i-1}, ..., \mathbf{c}_1)$ can cause noisy results. This happens because even if the likelihood of sampling an unlikely character is very small, i.e. $p(\text{b}|\text{I love my Mothe}) \ll 1$, because our vocabulary is large, the sum of probabilities of all unlikely characters is high. Therefore it's likely we'll select some unlikely character.
A simple solution for this is to zero out all unlikely characters by assigning zero to all probabilities that aren't in the top k probabilities, and normalizing to get a valid probability, i.e. divide by $\sum_{l=1}^{vocab\_size} p'(\mathbf{c}_i^l|\mathbf{c}_{i-1}, ..., \mathbf{c}_1)$ where

$$p'(\mathbf{c}_i^l|\mathbf{c}_{i-1}, ..., \mathbf{c}_1) \begin{cases} p(\mathbf{c}_i^l|\mathbf{c}_{i-1}, ..., \mathbf{c}_1) & \text{if l is in top k} \\ 0 & \text{otherwise} \end{cases}$$

4.7 Task

1. Fill in the missing code for CausalSelfAttention in transformer.py
2. Train the model with the default hyper-parameters in the code and plot the accuracy and the loss of the train and test set.
3. You will visually measure your model's generation ability.
   After each epoch generate 3 sentences using your model. By default, we'll start each sentence with "the " and generate the next 30 characters but you're welcome to experiment and start with a different beginning.
4. Repeat the same experiments as before but use the **Top-k-sampling** trick with $k = 5$. Present your generated sentences. Do you notice a difference in the quality of the results?

5 Ethics

We will not tolerate any form of cheating or code sharing. You may only use the `numpy`, `mathplotlib`, `torch`, `tqdm` libraries.

6 Submission Guidelines

You should submit your report, code, and README for the exercise as $ex5\_\{YOUR\_ID\}.zip$ file. **Other formats (e.g. $.rar$) will not be accepted!**.

The README file should include your name, cse username, and ID.

Reports should be in PDF format and be no longer than 8 pages in length. They should include any analysis and questions that were raised during the exercise. **You should submit your code, Report PDF, and README files alone without any additional files.**

6.1 Submission

Please submit a zip file named $ex5\_\{YOURID\}.zip$ that includes the following:

1. A README file with your name, cse username, and ID.
2. The *.py* files with your filled code.
3. A PDF report.

Congradulations on completing all exercises in IML! Good Luck with your exams