Advanced Deep Learning
Winter 2024/25
Florian Kordon, Jonas Utz, Vincent Christlein, Katharina Breininger

Sheet 3, starting from January 14th, 2025, due February 2nd, 2025, 14:00

Topic 3: Generative and Predictive Modeling with Energy-based Models

In this exercise, we will investigate the characteristics of **Energy-based Models (EBM)**, which can be seen as a quite fundamental and generally applicable approach to generative probabilistic modeling. We will work on an image dataset that contains scans of historical printed glyphs, i.e. individual lower- and uppercase characters that stem from one of the earliest letterpress printings.

We will start off by covering the basic theory and explore how we can use EBMs to model a distribution of image data. Besides allowing us to sample from the approximate distribution to synthesize new examples, we will further analyze how we can utilize this notion for identifying out-of-distribution (OOD) samples. Lastly, will explore how we can repurpose the logits from a conventional classifier to build a Joint Energy-based Model (JEM) that provides us with a very elegant way to perform joint predictive *AND* generative modeling at high quality and little computational overhead [2].

1. Energy-based Models (EBM)

Let's consider a set of observations $\mathbf{x} \in \mathbb{R}^D$, e.g., images, numerical measurements, text lines, or video frames. In *predictive* modeling, we are interested in finding the most discriminative features within those observations that allow us some degree of decision-making, e.g., in the form of a classifier or regressor. In comparison, in *generative* modeling, we want to capture (and potentially uncover) the generating rules that describe the characteristics and commonalities of data that show a high likelihood under some distribution. More formally, we want to find a statistical model $q_\theta(\mathbf{x})$ that estimates the true density $p(\mathbf{x})$.

1.1 Modeling the probability distribution

An EBM seeks to find a good estimate $q_\theta(\mathbf{x})$ by establishing a relation between the probabilities and different scalar energy states. Intuitively, we want to find a mapping function that assigns low-energy states to data points with a high likelihood and high-energy states to very unlikelihood observations.

We can express this relation with a Helmholtz-Gibbs distribution.

$$q_\theta(\mathbf{x}) = \frac{\exp\left(-E_\theta(\mathbf{x})\right)}{Z_\theta} \tag{1}$$

$E_\theta(\mathbf{x}) : \mathbb{R}^D \mapsto \mathbb{R}$ is called the *energy function* which takes a data sample and outputs a single scalar energy value. $Z_\theta = \int_{\mathbf{x}} \exp\left(-E_\theta(\mathbf{x})\right) d\mathbf{x}$ is called the normalization constant, or *partition function*, which makes sure that we obtain a valid probability distribution integrating to 1. θ are the parameters that are used to tune and adapt the energy function.

An especially intriguing and powerful aspect of this formalism is that we can parameterize $E_\theta(\mathbf{x})$ with about any function type, e.g., a neural network with weights θ. In EBM-related literature, we then oftentimes write this neural network-based parameterized function as $f_\theta(\mathbf{x}) = -E_\theta(\mathbf{x})$! Note that with this notation, the network models the negative energy, which makes notation easier but needs to be considered in your implementation.

1.2 The problem of the intractable partition function

As we have seen, the energy function can be easily represented using a conventional neural network, e.g., a ResNet with a single output neuron. However, getting an accurate or approximate estimate for the partition function Z_θ is difficult or almost impossible in most scenarios. Especially for high-dimensional data like images, solving Z_θ becomes computationally intractable. On the other hand, if we would solve for $q_\theta(\mathbf{x})$ without Z_θ, we could not guarantee that the training observations become more likely than any other observation in the input space since the unknown Z_θ would constantly change.

For problems that can be expressed based on ratios between energies (e.g., image denoising), we do not need to know Z_θ. Sometimes it is also feasible to decompose a complex problem into multiple smaller ones, which can be described using individual energy functions with an analytically solvable partition function (i.e., a product of experts). For most cases, however, we got to approximate the normalization constant as best as possible.

2. Maximum likelihood estimation (MLE) with EBMs

We will now study how we can optimize EBMs with standard maximum likelihood estimation (MLE) and how we can approximate the partition function. Accordingly, for the observations $\mathbf{x} \in \mathbb{R}^D$, we would like to minimize the following negative log-likelihood-based loss functional:

$$\mathcal{L}_{\mathrm{ML}}(\theta; p) = \mathbb{E}_{p(\mathbf{x})}\left[-\log q_\theta(\mathbf{x})\right] \tag{2}$$

Intuitively, we want to maximize the likelihood of our training observations by minimizing the corresponding energy function's output and maximizing it for all other feature combinations in the input space.

If we now look at the gradient of this loss functional, we can derive a very interesting contrastive expression [4]. This expression encourages maximizing the likelihood of samples from the true distribution $p(\mathbf{x})$ while reducing the likelihood of samples from the modeled distribution $q_\theta(\mathbf{x})$. The optimum (minimum) is reached when both distributions arrive at the equilibrium $p(\mathbf{x}) = q_\theta(\mathbf{x})$.

$$\begin{aligned}
\frac{\partial \mathcal{L}_{\mathrm{ML}}(\theta; p)}{\partial \theta} &= -\mathbb{E}_{p(\mathbf{x})}\left[\frac{\partial \log q_\theta(\mathbf{x})}{\partial \theta}\right] \\
&= \mathbb{E}_{p(\mathbf{x})}\left[\frac{\partial E_\theta(\mathbf{x})}{\partial \theta}\right] - \frac{\partial \log Z(\theta)}{\partial \theta} \\
&= \mathbb{E}_{p(\mathbf{x})}\left[\frac{\partial E_\theta(\mathbf{x})}{\partial \theta}\right] - \mathbb{E}_{q_\theta(\mathbf{x})}\left[\frac{\partial E_\theta(\mathbf{x})}{\partial \theta}\right] & (3) \\
&\approx \frac{1}{n}\sum_{i=1}^{n}\left[\frac{\partial E_\theta(\mathbf{x})}{\partial \theta}\right] - \frac{1}{\tilde{n}}\sum_{i=1}^{\tilde{n}}\left[\frac{\partial E_\theta(\mathbf{x})}{\partial \theta}\right] & (4)
\end{aligned}$$

In practice, we approximate the expectations using a finite set of n training examples (first term) and a finite set of $\tilde{n}$ samples that have high likelihood under $q_\theta(\mathbf{x})$ (second term). In the next section, we study one particularly useful and widely employed approach to synthesizing such samples!

3. Sampling the model with Stochastic Gradient Langevin Dynamics (SGLD)

We have seen that we can circumvent the calculation of $Z(\theta)$ by approximating it with synthesized samples from $q_\theta(\mathbf{x})$. For that purpose, we typically utilize Markov Chain Monte Carlo (MCMC) that allows us to stochastically explore the model distribution $q_\theta(\mathbf{x})$ and arrive at points of high probability. While Vanilla MCMC theoretically works, we oftentimes see that the Markov Chain gets stuck in local areas of high probability, such that our distribution collapses to rather trivial solutions.

A more sophisticated approach that originally describes a physical phenomenon is to use gradient-based Langevin dynamics. A Langevin dynamic consists of a gradient term and an additive noise term. The gradient term allows us the chain to iteratively update an initial state along a trajectory toward high probability under $q_\theta(\mathbf{x})$. The noise term originates from Brownian motion and ensures that the Markov chains will explore the complete parameter space. We can express this idea in the following Stochastic Gradient Langevin Dynamics (SGLD) update procedure:

$$
\begin{aligned}
\mathbf{x}^0 &\sim \pi(\mathbf{x}) \\
\mathbf{x}^{k+1} &= \mathbf{x}^k + \frac{\eta}{2}\frac{\partial \log q_\theta(\mathbf{x}^k)}{\partial \mathbf{x}^k} + \epsilon, \quad \epsilon \sim \mathcal{N}(0,\eta) \\
&= \mathbf{x}^k - \frac{\eta}{2}\frac{\partial E_\theta(\mathbf{x}^k)}{\partial \mathbf{x}^k} - \underbrace{\frac{\partial \log Z(\theta)}{\partial \mathbf{x}^k}}_{=0} + \epsilon \\
&= \mathbf{x}^k - \frac{\eta}{2}\frac{\partial E_\theta(\mathbf{x}^k)}{\partial \mathbf{x}^k} + \epsilon\ .
\end{aligned} \tag{5}
$$

$\pi(\mathbf{x})$ is a prior distribution from which a sample can easily be drawn, η is the step size, $k \in [0, K-1]$ is the number of MC mixing steps, and ϵ is an additive Gaussian noise. In literature, there are multiple proposals for $\pi(\mathbf{x})$:

1. **Contrastive Divergence**: $\pi(\mathbf{x}) = p_{\text{train}}(\mathbf{x})$ — Start from observed training samples.
2. **Persistent Chain**: $\pi(\mathbf{x}) = q_\theta^{\text{current epoch-1}}(\mathbf{x})$ — Start from synthesized samples from last epoch(s).
3. **Non-persistent Short-run MCMC**: $\pi(\mathbf{x}) = \mathcal{N}(\mu, \sigma)$ — Start from Gaussian noise.

4. Training tricks to improve stability

While EBMs are a rather old idea, they only recently regained some track in scientific research. Since the above optimization scheme is computationally expensive and is susceptible to stability problems where the contrastive loss collapses, researchers came up with a selection of useful ideas that help both ongoing research as well as practical application.

4.1 Reservoir sampling (also replay buffer)

Reservoir sampling is a simple yet very powerful idea that builds upon the idea of persistent chains but at the same time retains the benefits of non-persistent short-run MCMC. Instead of always starting with a *fresh* $\mathbf{x}^0$ sampled from Gaussian noise, we instead continue the chain using a sample from the last epoch with a certain probability. E.g. for a minibatch of 10 data points, we could take eight (=80%) synthesized examples from the last epoch and sample two new ones (=20%). In practice, this can be done by having a buffer (array, list, etc.) that stores a certain number of examples from the preceding epoch(s). The newly synthesized samples of the current epoch are then added to that buffer after each epoch to build up a large reservoir of Markov chains. This approach helps to stabilize training and further drastically improves the convergence speed of the optimization process.

4.2 L2 regularization of the energy magnitudes

The contrastive setup in Eq. 3 encourages the energies on the data manifold to be close to zero and wants to maximize all other energies of the manifold. This can result in steep and non-smooth energy surfaces where gradient-based inference becomes problematic. Also, without any specific target value range for the energy magnitudes, the energies will oscillate quite heavily, which renders the loss functional unstable. An intuitive solution to this problem is constraining the Lipshitz constant so that the energy gradients become bounded. Du et al. [1] propose to apply L2 regularization to both expectation terms (i.e., approximations with finite sum), making sure that the energy magnitudes are close to zero (training samples) or rather small (synthesized samples).

5. Data synthesis: Using the EBM as a latent generative model

During training, we alternate between two interrelated steps: 1) First, we perform gradient-based inference to obtain samples of the current model distribution q_θ. We perform noisy gradient ascent (i.e., Eq. 5) toward data points with high probability/low energy values. 2) We use these samples to approximate the expectation of our model distribution and optimize the model parameters θ using the contrastive loss functional given in Eq. 3.

Once the training procedure is completed, we can use the EBM in two major ways.

1. Given an input sample $\mathbf{x}$, we can use the parameterized energy function to obtain the associated energy value as $E_\theta(\mathbf{x})$. If the energy is close to zero, we can assume that the data is very similar to the real distribution $p(\mathbf{x})$ (or at least the training distribution p_{train}). If the energy is noticeably larger than zero, we assume that $\mathbf{x}$ is an out-of-distribution (OOD) observation.

2. We can use the EBM's interpretation as a latent variable model and the inherent gradient-based inference procedure to synthesize new samples. Especially if the model is trained using a variant of non-persistent short-run MCMC, we can generate new and quite realistic images with high likelihood/low energy by starting with Gaussian-distributed noise and iteratively adapting the data features toward a lower energy state.

6. Joint energy-based model (JEM)

Generative models, while having compelling properties (see above), often struggle when being tasked with solving explicit functions, e.g., classification or regression. For such tasks, we tend to use predictive modeling instead. More formally, in predictive modeling, we are typically interested in finding a mapping that models the conditional distribution $q_\theta(y|\mathbf{x})$ with data points $\mathbf{x} \in \mathbb{R}^D$ and labels $y \in \mathbb{R}$ for a number of distinct classes C.

$$q_\theta(y|\mathbf{x}) = \frac{\exp(f_\theta(\mathbf{x})[y])}{\sum_{y'} \exp(f_\theta(\mathbf{x})[y'])} \tag{6}$$

Here, $\exp(f_\theta(\mathbf{x})[y])$ is the y^{th} logit.

Intriguingly, Gratwohl et al. [2] could show that both those ideas can be combined into a single hybrid EBM that is capable of strong generative as well as predictive performance. Specifically, we can repurpose the logits of any classifier as the unnormalized densities of the joint distribution $q_\theta(\mathbf{x}, y) = \exp(f_\theta(\mathbf{x})[y])/Z(\theta)$ with $f_\theta(\mathbf{x})[y] = -E_\theta(\mathbf{x}, y)$.

If we now factorize the log of the joint density model in a smart way, we arrive at a very elegant optimization objective [2, 3]:

$$\log q_\theta(\mathbf{x}, y) = \log q_\theta(\mathbf{x}) + \log q_\theta(y|\mathbf{x})\,. \tag{7}$$

The first term resembles the standard density model we can train using the MLE objective and SGLD-based synthesis (cf. Eq. 5). The second term is the standard categorical/conditional distribution of a predictive classifier which we can optimize using a cross-entropy loss function.

7. Out-of-distribution (OOD) detection with EBMs

The probabilistic foundation of EBMs naturally makes them a competitive detector for OOD samples. For example, we can use the trained EBM and compute the energy magnitudes for all training samples. Afterward, we use these magnitudes as the energy upper bound (energy-based threshold) that allows us to distinguish in-distribution (ID) and OOD samples (i.e., if the energy is higher than the threshold, the samples are considered OOD).

In other words, we can define a score function $s_\theta(\mathbf{x}) \in \mathbb{R}$ that, based on its output value, acts as a binary classifier that separates ID and OOD samples into two score distributions. Besides directly using the energy value, there exist other feasible types of score functions (cf. Gratwohl et al [2]).

Exercise 3.1: General understanding, code setup, data

Similar to the last exercises, we will provide you with the main structure for this exercise. Please make yourself familiar with the provided code in `ex3_main.py`. The code is based on PyTorch Lightning which reduces the amount of boilerplate code that we need to implement ourselves. Also, large parts of the code rely on the course *Deep Energy-based Models* by Philipp Lippe, which can be accessed here: (`https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial8/Deep_Energy_Models.html`). You can make use of this resource to solve the subsequent tasks.

In detail, the main code structure encapsulates the following steps:

1. Loading the historical glyph data and creating corresponding PyTorch datasets.
2. Setup of the function $f_\theta(\mathbf{x}) = -E_\theta(\mathbf{x})$ (and later on $f_\theta(\mathbf{x}, y)$) parameterized by a small convolutional neural network (CNN)
3. Defining the gradient-based inference procedure/sampling via SGLD.
4. Defining the main training object (pl.LightningModule) that encapsulates the contrastive loss functional and implements the abovementioned best practices for training (L2 regularization, reservoir sampling)
5. Trainer object that specifies the hyper-parameters for model training.
6. Inference object that allows performing the model sampling.
7. Code to perform detection of OOD samples based on a score function $s_\theta(\mathbf{x})$.

Your task: Please take a look at the above theoretical recap on EBMs. Make yourself familiar with the given code (`ex3_main.py`, `ex3_model.py`, `ex3_ood.py`, and `ex3_data.py`) to be able to implement all subsequent tasks efficiently and successfully. Feel free to take a look at the glyph dataset and visualize some examples to get a feeling for the characteristics of the data that you are working with.

After studying the material (including the lecture slides), you should have a good understanding of the following aspects (you don't have to provide answers to us to these questions):

- What are EBMs, what can I do with them?
- Why do we need a partition function, can't I just optimize the energy function?
- What can we do to approximate the oftentimes unknown partition function?
- What is the difference between inference and training of EBMs?
- How can we make sure that we explore the complete parameter space of the current model distribution?

Data: The data corpus you are going to work with contains 16,171 images distributed across 42 classes that represent different lowercase and uppercase letters. The data is split into train/validation/test subsets according to a 70%/15%/15% ratio (11,319/2,426/2,426 images). The images have different spatial resolutions and are brought to a common shape of 56×56 px during data loading to facilitate data handling and training. Although the images are rather small, training might take a couple of hours due to the computational demands of the inference procedure via SGLD.

Quota & co: We will provide the datasets for this exercise also in the project folder that we use for the environment. This way, the dataset does not (additionally) add to your quota. You can find the glyph dataset at `/proj/aimi-adl/GLYPHS/`.

Exercise 3.2: The energy function: Parameterizing E_θ with a neural network

We have seen that the choice of energy function in an EBM is very flexible. Especially for high-dimensional and more intricate problems like images, neural networks with weights θ provide us with a powerful tool for modeling the parameterized energy function $E_\theta(\mathbf{x})$.

Your task - Implementation: In this task, you will have to implement the forward function of a shallow CNN that serves as the energy function. More specifically, you will implement the `def forward(x,y=None)` and `def get_logits(x)` methods in `ex3_model.py`.

(a) Implement the `get_logits` method that returns the logits across all output nodes. This function will be handy for subsequent tasks. Feel free to also use this method to aid the implementation of the `forward` method.

(b) Accommodate all three use cases (EBM, uncond. JEM and cond. JEM) in `forward`. Implementation-wise you just need to consider two cases: (1) EBM / unconditional JEM and (2) conditional JEM. Consider using adaptive average pooling to convert between the 2D features and a linear representation. In the subsequent tasks, we will utilize the JEM-based forward function for multiple classes. In the case of a vanilla EBM that models $E_\theta(\mathbf{x})$, the forward method takes the input tensor $\mathbf{x}$ (=input image) and outputs a single scalar value corresponding to the assigned energy level.

In the case of the JEM model, we have two ways to calculate the energy score:

a) *Unconditional* — Calculate $\log \sum_y \exp(f_\theta(\mathbf{x})[y])$. Here, we don't need a specific class label and can calculate the logsumexp across all class logits.

b) *Class-conditional* — Calculate $f_\theta(\mathbf{x})[y]$. Here, we need the label information that determines the image's class membership.

Exercise 3.3: The inference procedure: Gradient-based sampling with SGLD

We have learned that we can approximate the normalization constant/partition function using MCMC. Specifically, we often use Langevin dynamics with a gradient-based score function, i.e., Stochastic Gradient Langevin Dynamics (SGLD).

For this task, we want to implement a Sampler class that performs SGLD-based sampling. Given an EBM with a known function signature (PyTorch nn.Module), we want to perform non-persistent short-run MCMC where we take some random Gaussian-distributed 2D noise and iteratively update the noise image toward lower energy magnitudes. We also make use of reservoir sampling to stabilize the training.

Your task - Implementation: Complete the `MCMCSampler` class in `ex3_main.py` where we perform SGLD via non-persistent short-run MCMC and reservoir sampling. Feel free to adapt the method signature and encapsulate certain parts of the functionality to your liking. The implementation of the sampling process involves the following steps.

(a) Define how the initial samples are calculated. As we use reservoir sampling, we sample a certain percentage of initial starting points $\mathbf{x}^0$ from a small zero-mean Gaussian distribution (e.g., $\mu = 0, \sigma = 0.01$). The remaining starting points are selected from the synthesized results from the last run of SGLD, requiring you to set up a data structure that can hold these synthetic images across epochs. Keep in mind that memory-related issues may occur if you don't preset a fixed maximum size for the reservoir buffer.

(b) Evaluate the energy function of the EBM $E_\theta(\mathbf{x})$ (and $E_\theta(\mathbf{x}, y)$ for conditional JEM) and perform back-propagation to obtain the gradient information at the input images.

(c) Perform gradient ascent (updating the image pixels) toward an image with higher likelihood under the current modeled distribution (=gradient descent if we consider the energy function).

Exercise 3.4: JEM: Contrastive loss, classification loss, and training

We now want to finally train our network that models the joint density of images and class labels $q_\theta(\mathbf{x}, y)$. For that, we are going to utilize the factorization given in Eq. 7, where we split training into a joint (1) density estimation and (2) classification.

Your task - Implementation: Complete the `JEM` class in `ex3_main.py`.

(a) Implement the `px_step` method that corresponds to estimating $q_\theta(\mathbf{x})$. First, we need to synthesize images using your implementation of the MCMC sampler. We can then use these together with the training samples from the minibatch to calculate the contrastive loss. In addition, it is usefule to constrain the Lipshitz constant of the energy surface to fascilitate image synthesis and the overall convergence behavior. This can be done using L2 regularization on the energy terms of the training samples and the synthesized samples, which makes sure that the energy levels remain rather steady and do not oscillate too much across different update iterations (example given in the template code).

(b) Implement the `pyx_step` method that corresponds to estimating $q_\theta(y|\mathbf{x})$. Consider using the get_logits() function you implemented earlier in the CNN model in `ex3_model.py`.

(c) Use both of your implementations to complete the whole training step in `training_step` method. (This should boil down to the factorization in Eq. 7).

Exercise 3.5: Synthesis of new images

After training, we can use our EBM/JEM to generate new examples.

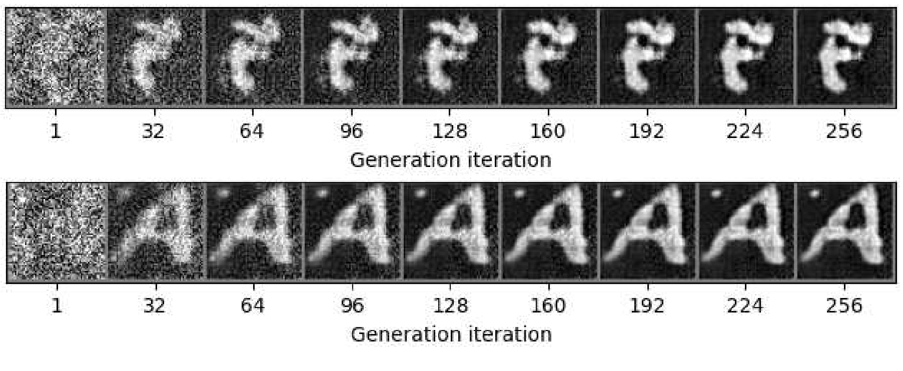

Abbildung 1: Example visualization of the energy score distribution of the ID test data (real) and different OOD data.

Your task - Adaption and Evaluation: We have provided you with some template code for image synthesis in the `run_generation` method in `ex3_main.py`. Please adapt the code according to your needs and your own implementation and then generate new images with your trained JEM model.

(a) Adapt the template code in `run_generation` according to your needs and your own implementation, and then generate new images with your trained JEM model. We synthesize new images starting from small Gaussian-distributed noise (as you have already done during training) and update the image towards lower energy levels. Instead of using a small number of steps in MCMC as done during training, you can use a much larger mixing time (e.g., $K = 256$ or $K = 512$ steps) to improve image quality.

(b) Use the code to visualize and evaluate the generation process at different steps for a single image or visualize some fully-synthesized (i.e., after all K steps have been completed) examples.

Exercise 3.6: OOD detection

Once trained, we can use our EBM/JEM to distinguish between ID and OOD data. In this task, we are going to evaluate how well the energy magnitude can serve this purpose.

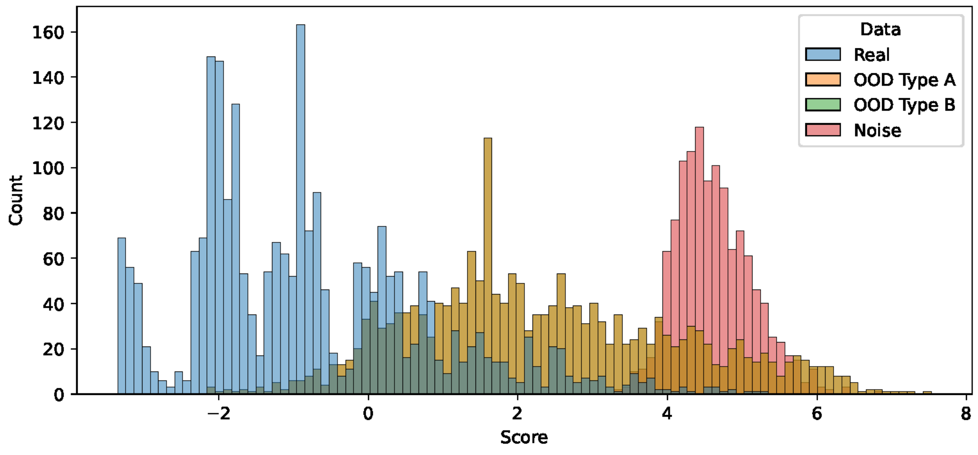

Abbildung 2: Example visualization of the energy score distribution of the ID test data (real) and different OOD data.

Your task - Implementation: Complete the `run_ood_analysis` method in `ex3_main.py`.

(a) Evaluate the score function $s_\theta = E_\theta(\mathbf{x})$ for all samples of the test distribution and two types of OOD distributions (see `ex03_ood.py` for implementation of the score function that you can use).

(b) Visualize the distributions, e.g., in a histogram and qualitatively evaluate whether you could identify a meaningful energy threshold (see Fig. 2 for an example).

(c) Perform a binary classification based on the energy scores. This can be done by assigning the test distribution samples a target label of 0 and the OOD samples a target label of 1. Evaluate the threshold-independent AUROC metric to evaluate the discrimination performance. (It is enough to use one of the OOD distributions.)

Additional information: Using a prepared conda environment

Since the quota on the cip-pool machines is limited, and we may need additional packages during the semester, we have prepared a conda environment `adl23_2` that you can use for this exercise. To use this environment, you can do the following:

Option 1: Activate the environment directly:

```
conda activate /proj/aimi-adl/envs/adl23_2
```

Option 2: Adapt your conda config: Check if you have a file `.condarc` in your home directory (`/.condarc`). If not, run

```
conda config
```

If you have it or after running the above command, open the file in your favorite editor and add the following line(s) to the file (amend if you already have the `envs_dirs` option):

```
envs_dirs:
- /proj/aimi-adl/envs/
```

You should then be able to activate the environment using

```
conda activate adl23_2
```

Let us know if you encounter any issues!

Literatur

[1] Yilun Du and Igor Mordatch. Implicit generation and modeling with energy based models. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[2] Will Grathwohl, Kuan-Chieh Wang, Jörn-Henrik Jacobsen, David Duvenaud, Mohammad Norouzi, and Kevin Swersky. Your classifier is secretly an energy based model and you should treat it like one. In *International Conference on Learning Representations*, 2020.

[3] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *J. Mach. Learn. Res.*, 22(57):1–64, 2021.

[4] Oliver Woodford. Notes on contrastive divergence. *Department of Engineering Science, University of Oxford, Tech. Rep*, 2006.