Exercise 1 - PyTorch & Variational Inference

67912 - Advanced Course in Machine Learning

Last Updated: 19.4.2025

Due Date: 1.5.2025

1 Exercise Overview

In this exercise you will first practice PyTorch [1] in a simple supervised classification task. Then, you will implement a Variational Auto-Encoder (VAE) [2] from scratch, on a small-scale image dataset.

We will use the MNIST dataset [3], which is a dataset of black and white handwritten digits. To first practice implementation in PyTorch, you will train a supervised classifier on MNIST. In this part of the exercise you will be thoroughly guided, filling in pre-written code.

The second part of the exercise is to simply train a VAE on MNIST. This part will not have a code template.

You will submit a PDF report of your work (as well as **all** of your code). In this PDF you will answer the questions, and present the requested findings described in Sec. 2 & 3. If you have additional interesting findings you are welcome share them as well, at the end of the report.

2 Supervised Classification (30 pts)

In this first part of the exercise you will implement a supervised classifier, by completing the code skeleton given to you in `sup_classification.py`.
NOTE: Don't add additional code files for this part of the exercise.

2.1 PyTorch Background: A Powerful Library for Gradient Based Methods

What is PyTorch? PyTorch (named `torch` in code) is a library similar to `numpy`. Its arrays are called tensors, and they are operated in a similar way.

Why `torch` over `numpy`? Long story short: Pytorch saves the computational graph of tensors, and can automatically compute the gradient of each tesnor w.r.t any loss function).
Now in some more details: Pytorch is designed in consideration of tracking another feature of tensors, which is their gradient w.r.t some (unknown) function. To be able to do so, for any function, It keeps track of where tensors came from.

Meaning, for each tensor it can automatically remember what are the mathematical operations that lead to its computation. This tracking graph is called the computational graph. Pytorch retains this computational graph for its tensors, at any time. Then, in a single command, pytorch can use this computational graph to automatically compute the gradient of any tensor with respect to any defined function (as long as all the operations done are derivable). This is the true power of pytorch. You can simply write your algorithm and "loss" function, and it automatically computes for you how to change the algorithm's parameters to fit this function. If this was not clear for you, we highly recommend attending the reception hour of Jonathan, where this will be explained in person if needed.

Tip: First, train a network for a small number of epochs to debug your code. Only than run the experiments for many epochs, when you already know there are no major bugs. This tip could save you a lot of time!

2.2 Supervised Classification

We will train a neural network for classification using Stochastic Gradient Descent (SGD). I.e., we (i) Iterate over the dataset in batches (ii) Get a prediction for each image in the batch (iii) Compute the gradients of all weights based on some pre-defined loss function (iv) Update the parameters of the network by going in the opposite direction to the gradients. We will use the Cross-Entropy loss:

$$\mathcal{L}_{CE} = -\sum_{y \in \mathcal{Y}} p(y) \cdot log(q(y)) \tag{1}$$

where p(y) is the ground truth, and q(y) is the probability given by the model for the label y.

You will see that PyTorch makes the implementation of such a network extremely easy, as it will compute the gradients of each parameter automatically.

2.3 Requirements

You are required to train a CNN to classify the MNIST dataset. To reduce runtime, you will train on $20,000$ images. Our code subsamples the training set automatically, with $2,000$ samples from each class (digit). We will keep the validation set the same. To build a CNN network use the modules `torch.nn.Conv2d` for the convolutional layers, `torch.nn.Linear` for fully-connected layer, `torch.nn.MaxPool2d` for pooling layers, and `torch.nn.ReLU` as the activation function. For an example of a neural architecture you may visit this tutorial.

The main purpose of this part is to give you a hands-on practice in pytorch. If you have previous experience with pytorch, this part should be extremely simple for you.

You are already given a skeleton code at `sup_classification.py` which you need to fill out. Don't submit additional files for this part of the exercise, and only fill in you code in the marked spots.

Please see Sec. 4.1 for guidelines in solving the exercise, and Sec. 4.2 for computational resources.

2.3.1 Task

Complete the code in `sup_classification.py`, and train a supervised classifier on the MNIST dataset.

At the end of each epoch compute the average validation loss and accuracy of the model, and plot them through the epochs (obviously, present these 2 figures in your report). At the end of training, your classifier should reach **at least** 97% accuracy on the test set.

3 VAE (70 pts)

3.1 Background

3.1.1 The Generative Model

We define a latent-variable model for the data as:

$$p(x) = \sum_z p(z) \cdot p(x|z) = \mathop{\mathbb{E}}_{z \sim p} p(x|z) \tag{2}$$

Such that $p(z)$ is the prior distribution of the latent variable, and $p(x|z)$ is the posterior. Here, we choose a standard Gaussian prior $P(z) = \mathcal{N}(0, \mathcal{I})$. The posterior distribution is also Gaussian, with the mean $\mu_z = G(z)$ and the variance being $\sigma_p^2 I$. The generator is a function $G : R^d \to R^{N_p}$, where d is our latent dimension, and N_p is the number of pixels. We will implement G using a neural network. Overall, the posterior distribution is given by $P(x|z) = \mathcal{N}(G(z), \sigma_p^2 \mathcal{I})$.

3.1.2 ELBO

We train a VAE for sampling new data and for estimating point likelihood. We need to succeed at two tasks: i) compute the log-likelihood $log(p(x))$ ii) Find the optimal generator parameters such that our training data have maximum average log likelihood under the model. Task (i) is computationally infeasible as it requires an infinite sum. Computing it via Monte-Carlo may have high variance unless we sample a very large number of latents. The solution will be to use Variational Inference [4]. We will use a proposal distribution $q(z)$, and perform importance sampling:

$$p(x) = \mathop{\mathbb{E}}_{z \sim q} w(z) \cdot p(x|z) \tag{3}$$

where $w(z) = \frac{p(z)}{q(z)}$ are our importance weights.

Taking the log of each side, along with Jensen's inequality gives us the following lower bound:

$$log(p(x)) \geq \mathop{\mathbb{E}}_{z \sim q} log(p(x|z)) - KL(q \| p) \tag{4}$$

We call this the Evidence of Lower BOund (ELBO), and it is true for every proposal distribution q we might choose. By optimizing to maximize the ELBO, we produce a tighter bound for $log(p(x))$, with the best result when $q_x(z) = p(z|x)$.

3.1.3 Choosing q

How will we choose the right q then? by optimizing it! We assume the proposal distribution q_i for each image x_i is a Gaussian with diagonal covariance. Therefore the 2 vectors μ_i and σ_i paramerize q_i as $q_i = \mathcal{N}(\mu_i, \sigma_i)$.

We have 2 options to optimize the q_i distributions.

1. Amortized: Learning an encoder $S : R^{|pixles|} \rightarrow R^{2d}$ which takes an image x_i and outputs its μ_i, σ_i.
2. Latent Optimization: Learning $\mu_1, ..., \mu_N$ and $\sigma_1, ..., \sigma_N$ directly as parameters.

In this exercise you will experiment with both.

3.1.4 The re-parametrization trick

When optimizing q, we note that we sample from q is order to approximate the expectation in the first term, which is not well defined. We avoid differentiating through a sampling process by the re-parametrization trick. All it says is that we first sample $\epsilon \sim \mathcal{N}(0, I)$ then treat μ_i and σ_i as deterministic by defining the random variable z_i as $z_i = \mu_i + \sigma_i \odot \epsilon$. We will draw a new ϵ for each sample during training.

Note: Due to numerical stability issues, we won't be optimizing σ directly, but instead $r = log(\sigma^2)$. When doing the re-parametrization trick don't forget to transform r into σ.

3.1.5 Optimization Objective

As proposed by [2], when choosing a prior of $z \sim \mathcal{N}(0, I)$ and $p(x|z) = \mathcal{N}(G(z), \sigma_p^2 I)$, we can maximize the ELBO by the following optimization objective:

$$\mathcal{L}_{VAE} = \min_{S,G} \sum_x \|x - G(\mu_x + \sigma_x \odot \epsilon_x)\|_2^2 + KL(\mathcal{N}(\mu_x, \sigma_x^2 I) \| \mathcal{N}(0, I))$$
$$\text{where} \quad \mu_x, \sigma_x = S(x) \tag{5}$$

and for a Latent Optimization solution:

$$\mathcal{L}_{LO} = min_{\mu_1,...,\mu_N,\sigma_1,...,\sigma_N,G} \sum_{i \in N} \|x_i - G(\mu_i + \sigma_i \odot \epsilon_x)\|_2^2 + KL(\mathcal{N}(\mu_i, \sigma_i^2 I) \| \mathcal{N}(0, I)) \tag{6}$$

where in both cases we can compute the KL divergence precisely by:

$$KL(\mathcal{N}(\mu, \sigma^2 I) \| \mathcal{N}(0, I)) = \frac{1}{2}\big(\mu^2 + \sigma^2 - log(\sigma^2) - 1\big) \tag{7}$$

Remember that in both versions ϵ_x is drawn by $\epsilon_x \sim \mathcal{N}(0, I)$ independently for every sample in every iteration.

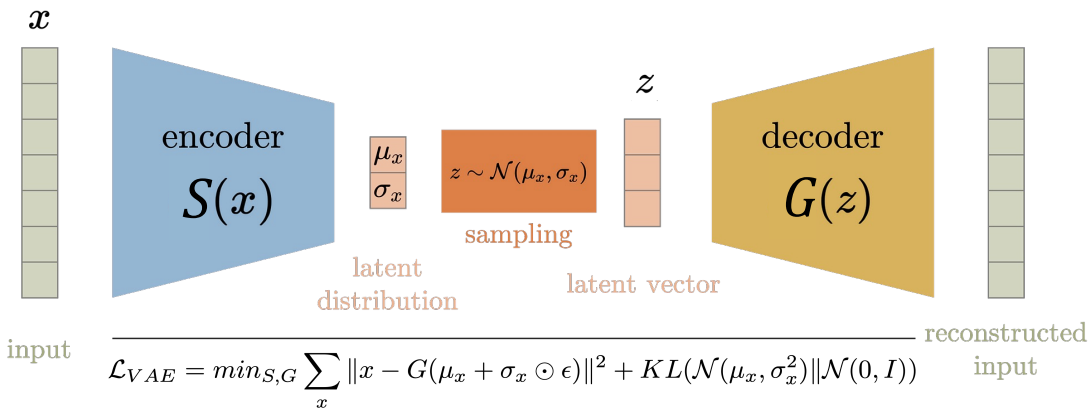

Figure 1: **The VAE Architecture**

3.1.6 Computing the Probability of a Sample (Amortized only)

To estimate the log likelihood log(p(x)) for a trained mode, you do not need to use Jensen's inequality. Instead, use Eq. 3 directly, and estimate its value using monte-carlo with $M = 1000$ random values of z sampled from q_x. In practice, this will be numerically unstable, as the probabilities are vanishingly small. Working in log-space will solve this problem. Specifically, we will compute the following:

$$log(p(x)) = log\big(\mathop{\mathbb{E}}_{z\sim q} w(z)\cdot p(x|z)\big) \approx log\big(\frac{1}{M}\sum_{i=1}^{M}\frac{p(z_i)\cdot p(x|z_i)}{q(z_i)}\big) \text{ , where } z_i \sim q_x \tag{8}$$

Taking the exponent of the log of each item in this sum then yields:

$$log(p(x)) \approx log\sum_{i=1}^{M} exp\big[log(p(z_i)) + log(p(x|z_i)) - log(q(z_i))\big] - log(M) \text{ , where } z_i \sim q_x \tag{9}$$

We can now use pytorch's `torch.logsumexp` which solves our numerical issues as it is numerically stable. When asked to, please evaluate $log(p(x))$ as described in Eq. 9.

Hint: Computing the PDF of a Multivariate Gaussian is numerically unstable, therefore you will have to implement a function that computes the log PDF where the unstable parts are already in log-space.

3.2 Important Implementation Details

- **Optimizing σ.** For numerical stability, it is common to define $r = log(\sigma^2)$, and optimize r instead of instead of σ. You can write the functions of $sigma$ in terms of r and vice versa. The optimization might not work for you if you choose to optimize σ directly. In your Latent Optimization experiments, make sure to also initialize r accordingly. I.e., initialize $r = log(\sigma^2)$ where $\sigma \sim \mathcal{N}(0, I)$.
- **KL aggregation.** The MSE can be easily computed by the loss class `torch.nn.MSE`. This averages the squared error in the pixel space. To match that, make sure to take the average KL losses of all latent dimensions as well. I.e.,

Figure 2: Expected reconstruction quality of an amortized VAE. Real images are in the first row. Matching reconstructed versions are at the second row.

the KL divergence is computed separately at every latent dimension, and you should average these losses and only then combine with the MSE loss.

- **Hyper-parameters.** We recommend trying the following hyper-parameters: $\texttt{epochs} = 30, \texttt{learning\_rate} = 0.001, \texttt{optimizer} = Adam, \texttt{latent\_dimension} = 200, \sigma_p = 0.4$.
- **Network Arch.** We provided you with the Architecture for your AutoEncoder in `model.py`. Although this architecture is highly recommended, you may choose another one if you wish. For the latent optimization parts, you only need the Decoder. The provided architecture, contains a linear layer mapping from the latent dimensions to the one required by the decoder. Please do not remove.
- **Learning Rate for Latent Optimized Vectors.** In general, Latent optimized vectors change much more sparsely than a shared network (as they are appear in a single iteration at each epoch). Therefore, we typically use a $10x$ learning rate for them. Specifically, for our latent optimization experiments use a learning rate of 0.001 for the decoder and 0.01 for the latent vectors.

3.3 Requirements

You are required to train a VAE model over the MNIST dataset. As before, you will work on a subset of MNIST keeping $20,000$ images for training. Re-use the code we gave you to create the training set.

Answer the following questions / assignments. You will be required to perform a few ablation studies on different parts of the algorithm, and analyze the results. You are required to explain all of your answers.

Please see Sec. 4.1 for guidelines in solving the exercise, and Sec. 4.2 for computational resources.

Note: These models are not perfect. In Fig. 2, we display the expected reconstruction quality of a few training set samples, when training an amortized VAE.

3.4 Question & Assignments

For each experiment in the following questions, save the weights from the final epoch. Notice if other checkpoints are needed for follow-up questions.

1. **(Q1: Amortized VAE.)** Train an amortized VAE on your MNIST subset, for 30 epochs. Plot the loss values after each epoch. Additionally, choose 10 random validation images (one from each class) and plot them and their reconstructions at epochs $1, 5, 10, 20$ and 30. Do the same for 10 random images from the training set. Did the Auto-Encoder overfit the training data? Explain.

2. **(Q2: Sampling from a VAE.)** Sample 10 latent variables from your prior distribution, pass them in the generators from epochs $1, 5, 10, 20$ and 30. Plot the generations from each epoch, and observe how the generator changed over-time (No explanation needed).

3. **(Q3: Latent Optimization.)** Train a generator by Variational Inference, using Latent Optimization for optimizing the q vectors instead of a shared encoder. Initialize the q vectors by sampling from a gaussian distribution of $q \sim \mathcal{N}(0, \mathcal{I})$. This will be our prior distribution for this experiment. Use the same dimensions for q as in Q1 and Q2.

    - Plot the reconstructions of 10 images (one from each class) from the training set, at epochs $1, 5, 10, 20$ and 30. Compare these reconstructions to the ones from Q1. Which method proposed better q vectors? Explain.
    - Sample from your new model, by inputting it 10 latent vectors sampled from the prior distribution. Compare these to the samples from Q2. Was our initialization sufficient to establish a good prior distribution for this problem? Explain.

4. **Q4: Computing the log-probability of an image.** For each digit $(0 - 9)$ sample 10 images: 5 images from the training set and 5 from the test set. Compute the log-probability of each image as described in Eq. 9.

    (a) Plot a single image from each digit, with its log-probability.

    (b) Present the average log-probability per digit. Which digit is the most likely? Why do you think that is the case? Explain.

    (c) Present the average log-probability of the images from the (i) training set (ii) test set. Are images from the training set more or less likely? Explain your answer.

4 Grading & Requirements

4.1 Ethics & Limitations

This is an advanced course, therefore we will have 0 tolerance for any kind of cheating. We are stating it very clearly that you are forbidden from doing the followings:

- Using any external github repository.

- Sharing any part of your code with other students.
- Using any external library other than `numpy`, `scipy`, `sklearn`, `pytorch`, `torchvision`, `matplotlib`, `tensorboard`, `wandb`, `pandas`, `plotly`, and `tqdm`. (You are of course allowed to use all basic "native" python packages, such as `math` or `os`).

4.1.1 Github Copilot & ChatGPT

We **allow** (and even encourage) using automatic tools, such as github copilot and even the infamous ChatGPT, for the exercise. Saying that, we do have a few restrictions for these tools: We require to note and explain (in the submitted code itself) every part of the code that was written by these tools. Also, in your PDF (as a separate section) explain how did you use these tools, and for which aspects of the exercise you found them useful.

4.2 Resources

You should be able to run this exercise on your personal computer. If you run into a problem with this subject please reach out to us as soon as possible by Moodle / reception hours.

4.3 Submission Guidelines

4.3.1 PDF Report

The PDF report should be 5 pages at max. Answer the questions from Sec. 2 & 3 in it, marking clearly where are the answers to each questions. We are note requiring any format. Explain your results when needed and analyze them.

4.3.2 Code

Submit all of your code used to perform the experiments and evaluations of this exercise.

Jupyter Notebooks. You are allowed to use jupyter notebooks in the submission.

4.3.3 Submission

In your submission should be a zip file including the following:

- A README file with your name and and cse username.
- Your filled-in code the supervised classification part (2).
- Your code for the VAE part (3).
- A PDF report answering the questions / requests from Sec. 2 & 3.

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[2] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 2013.