Exercise 2 - Flow Models

67912 - Advanced Course in Machine Learning

Last Updated: 20.6.2024

Due Date: 4.7.2024

1 Overview

In this exercise you will experiment with 2 methods for flow based generative modelling: Normalizing Flows and Flow Matching. Such probability flow models are generative models which transform a distribution into another by deforming its coordinates using a sequence of transformations. You will work on 2D data, which will enable you to analyze these algorithms from a distribution matching perspective.

The main challenge of this exercise is truly understanding the equations, then putting them together to create flow models from scratch. Therefore, we do not supply you with any code. You are free to choose your implementation, as long as it follows the equations correctly. As normalizing flows tend to be a bit more complex, we guide you through their implementation in Sec. 3.1.6. You are also free to choose the architecture, optimizer, learning rate, learning rate scheduler, number of sampling steps (T), batch size, number of epochs, and etc. However, for each type of model, we supply you with working parameters tested by us.

You are expected to submit a PDF report of your work, in which you answer the questions, and present the requested findings described in Sec. 3.3 & 4.3. If you have additional interesting findings you are welcome to describe them in the report as well.

Tip: You may find these functions especially useful for this exercise:

```
torch.distributions.MultivariateNormal
```

2 Data & Resources

2.1 Data

For both parts of the exercise you will work on 2D points. This allows to use smaller architectures, and have fast running times. You should be able to run all of the experiment on you personal computer, at ease.

To assemble your data, sample points from the Olympic logo, using the provided code in `create_data.py`. This will be your data, which you will train flow based models on. We will guide you how many samples are needed for each

experiment. Meaning, we want to create a model that samples points in a 2D space. Don't hesitate the visualize the data during training or sampling, for understanding your errors. For unconditional models use the function `create_unconditional_olympic_rings` and for conditional ones use `create_olympic_rings`.

2.2 Resources

The exercise was built to be extremely light-weight. Therefore, you absolutely **dont** need to pay for any external resources for this exercise. You should be a able to run all the experiments of this exercise on your private computer in about 40 minutes. Extrenal resouces are absolutely unnecessary and wont speed your runtimes by much, as the data and models are very simple and lightweight. If you run into slow runtimes of over a few minutes, please check your implementation for excessive looping or come see us at the reception hours.

3 Normalizing Flows (50 pts)

3.1 Background

3.1.1 Normalizing Flows

In normalizing flows, we assume an invertible function exists between our latent and data distributions. We then define ϕ which is an **invertible** function that transforms the latent distribution of choice P_z into our data distribution. In other words we are looking for an invertible function ϕ such that:

$$x = \phi(z) \quad \forall \; z \in P_z \tag{1}$$

where it is common to choose $P_z = \mathcal{N}(0, \mathcal{I})$. We will follow that choice for the purpose of this exercise.

We optimize ϕ by Maximum Likelihood Estimation (MLE) as seen in other models e.g. Auto-Regressive and Variational Inference. As ϕ is invertible, the solution *seems* trivial: maximizing the likelihood of the inverse mapping for each data point. In class you saw why the trivial solution is wrong, and misses an important density normalization factor. Hence, using the change-of-variable formula we can correct this to consider this normalization factor, having:

$$arg\max_{\phi} E_{x\sim q} log(p(x)) = arg\max_{\phi} E_{x\sim q}\big[log(p_z(\phi^{-1}(x))) + log(|\det \frac{\delta\phi^{-1}(x)}{\delta x^T}|)\big] \tag{2}$$

Basically, this is normalizing flows. We will now remind you which about the affine coupling layer which we use to build flow models, since they match our specific requirements of both (i) invertability (ii) simple normalization term.

3.1.2 Affine Coupling Layer

We require each of our models layers to (i) be fully invertible (ii) have an easy to compute Determinant of the Inverse functions Jacobian (second right-hand term in Eq. 2). Sadly, a straightforward Fully-connected layer does not match any of these requirements as it is not certainly invertible and can have a very complex Inverse Jacobian Determinant. Instead,

independently scaling and shifting each of the inputs axes does match both of our requirements, i.e. $z = z \odot s + b$ (s, b being learnable parameters). However, this manipulation is too simple, and is unable to learn complex flow patterns such as from a normal distribution to a distribution of images.

In class you saw how a simple trick can overcome this linear limitation, widely extending the range of representable flows. Each layer will split its input into two equal parts: $z = (z_l, z_r)$. It then transform z_l by a learnable function (e.g. any neural network) into a shift and log-scale parameters, which we will operate on z_r:

$$y_r = e^{log(\hat{s})} \odot z_r + b \ , \ \ f(z_l) \rightarrow (log(\hat{s}), b) \tag{3}$$

The output of the layer will be (z_l, y_r) which will enable us to make this layer easily invertible.

This clever manipulation is not only invertible, but also sets the Inverse of the Jacobian to have a simple determinant.

3.1.3 Inversion of an Affine Coupling Layer

Given a vector y we wish to invert, we will split it in half obtaining (y_l, y_r). $y_l = z_l$ by definition,. Therefore, we can get $\hat{s}, b$ by passing it through f, i.e. $f(y_l) = (log(\hat{(}s), b)$ then $\hat{s} = e^{log(\hat{s})}$. Obtain z_r is then simple:

$$z_r = (y_r - b)/\hat{s} \tag{4}$$

Computing the Inverse Jacobian Determinant is also very simple. The Inverse Jacobian becomes:

$$J = \begin{bmatrix} I & \frac{\delta z_l}{\delta y_r} \\ 0 & diag(s^{-1}) \end{bmatrix} \tag{5}$$

Denoting the entire layer as h, its inverse jacobian determinant is simply:

$$|\det \frac{\delta h^{-1}(y)}{\delta y^T}| = \prod_{k=1}^{|z_r|} |s_k^{-1}| \tag{6}$$

which is the multiplication along the diagonal axis of Eq. 5.

3.1.4 Composition of Layers

For a forward pass, the composition is trivial, performing the next layer on top of the previous layers output. For optimization, we need to compose the inverse mapping as well. Since each layer by itself is invertible this can be done to the inverse mappings as well. Now we are left to take care of combining the inverse jacobian determinants of all layers. Keeping in mind that the determinant of multiplied matrices is the multiplication of their determinants, we will simply multiply these determinants one after another.

Formally, we define the flow up to the layer t to be $\phi_t(z)$ and each individual layer as h_t, i.e. $\phi_{t+1}(z) = h_t(\phi_{t-1}(z))$. The inverse flow from the data up to layer t is denoted as ψ_t, hence $\psi_t(x) = h_t^{-1}(\psi_{t+1}(x))$. Then, the Inverse Jacobian determinant up to a time t is:

$$|\det \frac{\delta \psi_t(x)}{\delta x^T}| = \prod_{i=L}^{t} |\det \frac{\delta h_i^{-1}(\psi_{i+1}(y))}{\delta \psi_{i+1}(y)}| \tag{7}$$

where L is the number of composed layers and $\psi_L(x) = x$.

The inverse jacobian determininat of the entire normalizing flow ϕ then becomes:

$$|\det \frac{\delta \phi^{-1}(x)}{\delta x^T}| = \prod_{t=L}^{1} |\det \frac{\delta h_t^{-1}(\psi_{t+1}(y))}{\delta \psi_{t+1}(y)}| \tag{8}$$

where the left side expression is the product of Eq. 6 over all layers.

3.1.5 Permutation Layers

Each layer only modifies half of its input. In order to use all parameters of our input, we introduce permutation layers after each affine coupling layer (except the last one). They are easily invertible and have a inverse jacobian determinant of 1, hence do not alter our previous result.

3.1.6 Optimization Objective

The full optimization objective is as describe in Eq. 2:

$$\mathcal{L}(x) = -log(p_z(\phi^{-1}(x))) - log(|\det \frac{\delta \phi^{-1}(x)}{\delta x^T}|) \tag{9}$$

where $p_z = \mathcal{N}(0, \mathcal{I})$.

3.1.7 Implementation Guideline

You may choose your own way to implement a normalizing flow. However, this is our recommended recipe for implementing one:

1. **Building an affine coupling layer -** h_t. This class should have: (a) An initialization function, which initializes a neural network, that maps z_l to $(log(\hat{s}), b)$. (b) A forward function that computes $h_t(z)$ given z. (c) An inverse function that computes $h_t^{-1}(y)$ given y. (d) A function for computing the **log** inverse jacobian determinant of this layer as in Eq. 6. . For numerical stability, we recommend you compute the log determinant without implicitly computing the determinant itself (i.e., summing over s^{-1}).

2. **Building a permutation layer.** This should be straightforward and you should only implement a forward and a **matching** inverse function which inverses the permutation.

3. **Construct your flow model.** Assemble your model from a composition of interleaving affine coupling layer and permutation layers. The model should have: (a) A forward function (trivial). (b) An inverse function for $\phi^-1(x)$, using the inverse mapping of each layer this also becomes trivial. (c) A function for Eq. 8, using the inverse jacobian

determinants functions of each affine coupling layer. As the permutation layers have an inverse jacobian determinant of 1 you may simply ignore them.

4. **Training Loop.** Write the training loop of your model. Use the inverse mapping of the entire flow and the inverse jacobian determinant function of it to compute the loss as in Eq. 9.
   Tip: Dont forget to take the log of this determinant, and the log PDF of the point $\phi^{-1}(x)$ before combining them.

3.2 Expected Results & Runtime

To guide you, we provide you a glimpse of what a good solution is expected to achieve. In Fig. 1 we show a sampling of 2000 points from our normalizing flow model.

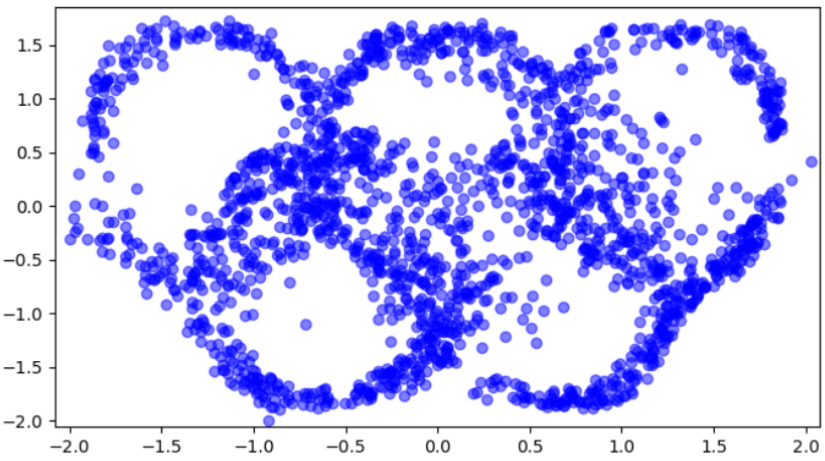

Figure 1: Expected sampling results of a Normalizing Flow model

Regarding runtimes, in slower private computers, training this normalizing flow model can take up to 2 minutes per epoch, hence 40 minutes overall. Saying that, in stronger computers (such as Apples new processors) it may be much faster, about 15 seconds per epoch. To speed debugging times, we recommend you train your model for 7 epochs in the debugging phase, before training it for the full 20 epochs. Seeing the progress (i.e., the sampled points distribution gets closer to the Olympic rings) after every epoch could greatly aid you to verify whether you are in the "right place".

3.3 Questions & Assignments

You are required to train an unconditional normalizing flow model over the 2D shape described at Sec. 2.1.

We recommend you start from the following:

- **Affine coupling layers:** We recommend to use 5 consecutive linear layers with a hidden size of 8 neurons, and `nn.LeakyReLU` layers between them. Each affine coupling layer should have 2 of these, one for b and another one for $log(s)$.
- **Arch.:** 15 affine coupling layers, with permutation layers between them.

- **Training Hyper-parameters:** `optimizer=torch.optim.Adam`, `learning_rate=`$1e-3$, `number_of_data_points=`$250,000$, `epochs=`20, `batch_size=`128, and a Cosine annealing learning rate decay. Since we are working with a larger quantity of points, do not try to visualize them all at once, instead visualize a subsampling of them.

During the building of your model, we ask you to present the followings:

1. **Q1: Loss.** Present the validation loss over the training epochs. Additionally, plot the log-determinant and the $log(p_z(x))$ components of this loss in separate lines at the same figure.
2. **Q2: Sampling.** Present a 3 figures with different samplings of 1000 points, using 3 different seeds.
3. **Q3: Sampling over time.** Your Normalizing flow model should keep the dimension of the input at all times. Sample 1000 points out of it, and plot them after each layer of the model (in separate figures) showing how the distribution progresses. Please plot up to 6 figures, if your model has more than 5 layers, choose a layer at every $1/5$ of the way from your model.
4. **Q4: Sampling trajectories.** Sample 10 points from your model and present the forward process of them layer by layer, as a trajectory in a 2D space. Color the points according to their time t.
5. **Q5: Probability estimation.** For 5 points of your choice, present the inverse process layer by layer, as a trajectory in a 2D space. Choose 3 points from inside the olympic logo and 2 outside of it. Color the points according to their time t. Compute the $log(p_z(x))$ of both points. Are the points inside the Olympic logo more or less likely from outside of it? Why? Explain.

4 Flow Matching (50 pts)

One recent breakthrough in generative modelling has been text-to-image models such as DALL-E 3 [1], Imagen [2], Stable Diffusion 2 (Re-implementation of Rombach et al. [3]) and many more. Most of these methods are based on diffusion models. Lately, a newer and simpler formulation was proposed by Lipman et al. [4], using flow matching. The most recent Stable Diffusion model, version 3, already uses this formulation, and more are likely to follow. In this exercise you will experiment with training a flow matching model from scratch by yourselves, according to the relatively simple formulation you saw in class.

4.1 Background

As in normalizing flows, we wish to find a model that transforms the coordinates such that our prior probability distribution slowly changes into a different probability distribution. Here, we will progress the flow in time by operating our model over and over again on the current state, unlike normalizing flow models which progress the flow in time by different layers. Additionally, a powerful advantage of flow matching models, is the ability to choose any function approximation we wish to. As you will soon see, flow matching is extremely easy to implement, making it very convenient to work with.

We will not dive into the proofs of flow matching, and instead focus on how to implement them.

4.1.1 Learning the Vector Field

In flow matching, we assume there is a continuous flow $\phi(z) = x$, that we wish to approximate. We will approximate the flow ϕ by approximating its derivative, the vector field u_t, where:

$$\frac{d\phi_t(y)}{dt} = u_t(\phi_t(y)) \tag{10}$$

Therefore our optimization objective is:

$$arg\min_{v_t} E_{y\sim p_t}\|v_t(y) - u_t(y)\|^2 \tag{11}$$

In practice we dont know u_t, and cant sample from it.

4.1.2 Conditional Flows

In order to approximate u_t we will shift to a conditional flow view. The conditioning will be on the target data point y_1 (y at time 1). This will define a much simpler probability path of $p(y|y_1)$ (rather than $p(y)$) where we typically define $p_t(y|y_1) = N(y|\mu_t(y_1), \sigma_t^2 I)$.Keeping in mind that the continuity equation sets a special relationship between a vector field and the probability path:

$$\frac{dp_t(y)}{dt} + \sum_{i=1}^{d} \frac{d(p_t(y)v_t^i(x))}{dy^i} = 0 \tag{12}$$

one can show that optimal solution to Eq. 11 is the same solution obtained from the following conditional version of it:

$$arg\min_{v_t} E_{y_1\sim q, y\sim p_t(.|y_1)}\|v_t(y) - u_t(y|y_1)\|^2 \tag{13}$$

(We will not delve into this proof here)

This key result will later allow us to choose a simple form of conditional linear flows, which have a constant derivative in time.

4.1.3 Linear Flows

We will choose to look for conditional linear flows only, i.e., flows in which:

$$\phi_t(y_t|y_1) = (1-t)y_0 + ty_1 \tag{14}$$

Looking at the time-derivative of this flows:

$$\frac{d\phi_t(y_t|y_1)}{dt} = y_1 - y_0 \tag{15}$$

we can easily notice it is constant in time! Combined with Eq. 13, this makes our job much easier, as we can approximate u_t by simply optimizing the following:

$$arg\min_{v_t} E_{y_1 \sim q, y_0 \sim p_0} \|v_t(y_t) - (y_1 - y_0)\|^2 \tag{16}$$

And that's it, this is flow matching. We summarize the exact process of flow matching training in Alg. 1.

Algorithm 1 Flow Matching - Training

Input: Data point $\mathcal{X} = (x_0, ..., x_N)$, Prior distribution p_Z, Neural Network $V(.,.)$, `epochs`

for $i = 1, ...$`epochs` **do**

  for $X_b \subseteq \mathcal{X}$ **do** ▷ Iterate over the data in batches

    Sample $\epsilon_b \sim p_Z$ ▷ Sample a batch of random noise in the same size as X_b

    Sample $t \sim \mathcal{U}_{[0,1]}$

    $y = t \cdot X_b + (1 - t) \cdot \epsilon_b$

    $\hat{v_t} = V(y, t)$

    `loss` $= \|\hat{v_t}(y) - v_t(y)\|^2$

    `loss.backward()`

    `optimizer.step()`

  end for

end for

return V

4.1.4 Sampling from a Flow Matching Model

To sample from a flow matching model, we can simply sample a vector from our prior $y_0 \sim p_0$, and slowly propagate it through time as follows:

$$y_{t+\Delta t} = y_t + v_t(y_t)\Delta t \tag{17}$$

Eq. 18 can be used in the same way to inverse a the flow,by switching the sign:

$$y_{t-\Delta t} = y_t - v_t(y_t)\Delta t \tag{18}$$

We summarize the sampling process in Alg. 2

4.2 Expected Results & Runtime

To guide you, we provide you a glimpse of what a good solution is expected to achieve. In Fig. 2 and Fig. 3 we show a sampling of 2000 points from our flow matching models with and without class conditioning.

Algorithm 2 Flow Matching - Sampling

Input: Prior distribution p_0, Neural Network $V(.,.)$

Sample $y \sim p_0$

for $t = 0, \Delta t, ..., 1 - \Delta t, 1$ **do** ▷ $(dt > 0)$

$\mathbf{y} = \mathbf{y} + V(y, t) \cdot \Delta t$

end for

return $\mathbf{y}$

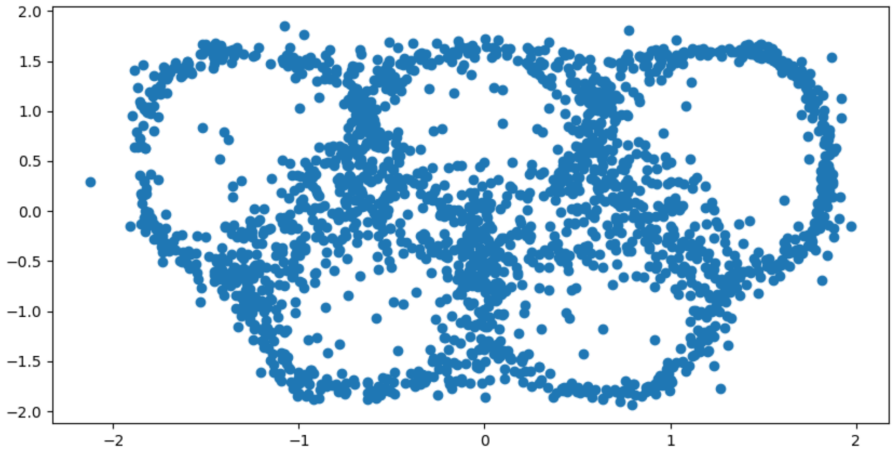

Figure 2: Expected sampling results of an Unconditional Flow Matching model

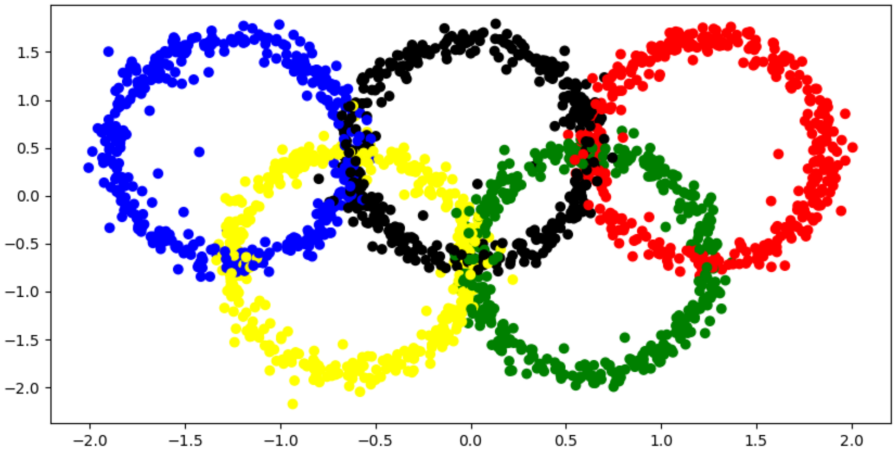

Figure 3: Expected sampling results of a Conditional Flow Matching model

Training this flow matching model should be very quick even on your personal computer: about 25 seconds per epoch, hence a few minutes overall. In stronger computers this may be much faster. To speed debugging times, we recommend you to train for 7 epochs in the debugging phase, and only then train for the full 20 epochs.

4.3 Assignment

You are required to train a 2 flow matching models over a 2D as describe in Sec. 2.1, one unconditional and one conditional, then answer the following questions / assignments. We recommend you start from the following:

- **Arch.:** a simple network, with $4-5$ Fully-Connected layers with Leaky-ReLU layers between them. Each Fully-Connected layer should have a width of at least 64 neurons.
- **Hyper-parameters:** `optimizer=torch.optim.Adam`, `learning_rate=`$1e-3$, $\Delta t = \frac{1}{1000}$, `number_of_data_points=`$250,000$, `epochs=`$20$, `batch_size=`$128$, and a Cosine annealing learning rate decay.

During the building of your models, we ask you to present the following:

Unconditional Flow Matching:

1. **Q1: Loss.** Present the loss function (Eq. 16) over the training batches.
2. **Q2: Flow Progression.** Your flow matching model should keep the dimension of the input at all times. Sample 1000 points out of it, and plot them after each of $t = 0, 0.2, 0.4, 0.6, 0.8, 1$ in separate figures, showing how the distribution progresses.

   Compare this to Q3 of Sec. 3.3, in which aspects is the distribution flow different between the models? Explain.
3. **Q3: Point Trajectory.** Sample 10 points of your choice from your model and present the their forward process as a 2D trajectory. Color the points according to their time t.

   Compare this to Q4 of Sec. 3.3. Which models flow is more consistent over time? Explain (you may answer this and the previous question together).
4. **Q4: Time Quantization.** Sample 1000 points from the models using $\Delta t = 0.002, 0.02, 0.05, 0.1, 0.2$. Plot the results in separate figures. How does this quantization of the flow affect the resulted distribution? Explain.
5. **Q5: Reversing the Flow.** Pick the same 5 points from Q5 of Sec. 3.3. Insert them to the reverse sampling process of your flow matching model, and plot their trajectories in a 2D space.

   Compare this to Q5 of Sec. 3.3.

   - Are the outputs the same? Explain why.
   - Re-enter the inverted points back into the forward model. Did you get the same points? Explain why / why not.
   - Say you would re-enter the normalizing flow inversion back to the normalizing flow. Would you then get the same points back? Explain.

Conditional Flow Matching:

Train a conditional version of your flow matching model. Our code already samples class conditioned input, where each Olympic ring is a different class. Insert the class input to the network with an embedding layer (look at *torch.nn.Embedding*), followed by a Fully-Connected that mixes the input with the class vector. The subspace of the point shall represent its class.

1. **Q1: Plotting the Input.** Plot your input coloring the points by their classes. Which equation did you change to insert the conditioning?

2. **Q2: A Point from each Class.** Sample 1 point from each class. Plot the trajectory of the points, coloring each trajectory with its class's color. Validate the points reach their class region.

3. **Q3: Sampling.** Plot a sampling of at least 3000 points from your trained conditional model. Plot the sampled points coloring them by their classes.

(Optional) Bonus (10 pts): Make one of your flow matching models (conditional or unconditional), output the point (4, 5) which is far away from the rings, as its sampled output. How did you do it? Describe your attempts including unsuccessful ones. Trivial solutions will receive less points than more clever ones (keep in mind clever doesn't necessarily mean complicated). Plot the trajectory of the sampled point over time.

5 Grading & Requirements

5.1 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 libraries other then `numpy`, `pytorch`, `matplotlib`, `tensorboard`, `wandb`, `pandas`, `plotly`, `tqdm`.

5.1.1 Github Copilot & ChatGPT

We allow 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.

5.2 Submission Guidelines

5.2.1 PDF Report

The PDF report should be 12 pages at max. Answer the questions from Sec. 3.3 & 4.3 in it, marking clearly where are the answers to each questions. We are not requiring any format, but please keep your explanations concise and to the point.

5.2.2 Code

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

Jupyter Notebooks. TL;DR No jupyter notebooks in the submission.
We will not accept any code inside a Jupyter Notebook. Meaning any jupyter notebook submitted will simply be ignored. You can develop the code for the exercise using notebooks but due to very short runtimes this is not recommended. For the submission convert any jupyter notebook file to a standard python files.

5.2.3 Submission

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

- A README file with your name and and cse username.
- Your code for the normalizing flow and flow matching models including their construction, training and any ablations and evaluations.
- A PDF report answering the questions / requests from Sec. 3.3 & 4.3.

References

[1] James Betker, Gabriel Goh, Li Jing, † TimBrooks, Jianfeng Wang, Linjie Li, † LongOuyang, † JuntangZhuang, † JoyceLee, † YufeiGuo, † WesamManassra, † PrafullaDhariwal, † CaseyChu, † YunxinJiao, and Aditya Ramesh. Improving image generation with better captions.

[2] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, et al. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in Neural Information Processing Systems*, 35:36479–36494, 2022.

[3] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[4] Yaron Lipman, Ricky TQ Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. *arXiv preprint arXiv:2210.02747*, 2022.