Hardest Homework in the entire course

Preface

Below is a concise, self-contained derivation and explanation of how the commonly used Mean-Squared Error (MSE) reconstruction loss in Variational Autoencoders (VAEs) can be seen as a special case of the Gaussian negative log-likelihood (NLL). We also define Bits-Per-Dimension (BPD) and the KL divergence on the latent codes.

1. From Gaussian NLL to MSE Loss

In a Variational Autoencoder, the decoder $p_\theta(\mathbf{x} \mid \mathbf{z})$ is often assumed to be a multivariate Gaussian with diagonal covariance. For simplicity, let us assume:

$$p_\theta(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}\big(\mathbf{x}; \boldsymbol{\mu}_\theta(\mathbf{z}), \sigma^2 \mathbf{I}\big),$$

where $\boldsymbol{\mu}_\theta(\mathbf{z})$ is the decoder's output (mean) and $\sigma^2 \mathbf{I}$ is an isotropic covariance matrix. The **negative log-likelihood (NLL)** of $\mathbf{x}$ under this Gaussian model is:

$$-\log p_\theta(x|z) = \frac{1}{2\sigma^2}||x - \mu_\theta(z)||^2 + \frac{D}{2}\log(2\pi\sigma^2)$$

where D is the dimensionality of $\mathbf{x}$.

Often in practice, we drop constants that do not depend on θ or $\mathbf{z}$. Thus, ignoring $\frac{D}{2}\log(2\pi\sigma^2)$ and setting $\sigma^2 = 1$ for simplicity, the term we minimize reduces to:

$$\|\mathbf{x} - \boldsymbol{\mu}_\theta(\mathbf{z})\|^2.$$

This is precisely the **Mean-Squared Error (MSE)** reconstruction term. Hence, the MSE loss is a direct consequence of assuming an isotropic Gaussian decoder with unit variance.

2. Bits-Per-Dimension (BPD)

When training or evaluating generative models, we often measure performance in **bits per dimension (BPD)**. Suppose our negative log-likelihood is in **nats**. Then, for a single data point $\mathbf{x}$:

$$\text{BPD}(\mathbf{x}) = \frac{-\log p_\theta(\mathbf{x})}{D \cdot \log(2)},$$

where:

- $\log(\cdot)$ denotes the natural logarithm (base e) ,
- D is the dimensionality of $\mathbf{x}$ (for an image of size $H \times W \times C$, $D = H \times W \times C$),
- $\log(2)$ converts nats into bits.

In order to get the entire dataset or batch **BPD** we compute the average **BPD**.

3. Computing BPD with Variational Inference

In VAEs, we don't have direct access to $\log p_\theta(\mathbf{x})$. Instead, we use the Evidence Lower BOund (ELBO), which gives us a lower bound on the log likelihood:

$$\log p_\theta(\mathbf{x}) \geq \mathcal{L}(\theta, \phi; \mathbf{x}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$$

Therefore, we can compute an upper bound on BPD using:

$$\text{BPD}(\mathbf{x}) \leq \frac{-\mathcal{L}(\theta, \phi; \mathbf{x})}{D \cdot \log(2)}$$

To compute this in practice:

First compute the reconstruction term using our Gaussian NLL from Section 1:

$$\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \approx \frac{-1}{2\sigma^2}||\mathbf{x} - \boldsymbol{\mu}_\theta(\mathbf{z})||^2 - \frac{D}{2}\log(2\pi\sigma^2)$$

where $\mathbf{z}$ is sampled from $q_\phi(z|x)$

Then compute the KL divergence term. For the standard VAE with diagonal Gaussian encoder we dervie the formula in the following section:

4. KL Divergence on the Latent Codes

The second term in the VAE loss comes from the **Kullback–Leibler divergence** between the approximate posterior $q_\phi(\mathbf{z} \mid \mathbf{x})$ and the prior $p_\theta(\mathbf{z})$. The VAE objective (ELBO) for a single data point $\mathbf{x}$ is:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p_\theta(z))$$

The KL term specifically is:

$$D_{\mathrm{KL}}\big(q_\phi(\mathbf{z} \mid \mathbf{x}) \,\big\|\, p_\theta(\mathbf{z})\big) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\Big[\log \frac{q_\phi(\mathbf{z} \mid \mathbf{x})}{p_\theta(\mathbf{z})}\Big].$$

Gaussian Prior and Posterior

Often, we take a standard normal prior $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ and a diagonal Gaussian approximate posterior:

$$q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\big(\mathbf{z}; \boldsymbol{\mu}_\phi(\mathbf{x}), \boldsymbol{\sigma}^2_\phi(\mathbf{x})\,\mathbf{I}\big).$$

For this case, the KL divergence has a closed-form:

$$D_{\mathrm{KL}}\Big(\mathcal{N}\big(\mathbf{z}; \boldsymbol{\mu}_\phi, \boldsymbol{\sigma}^2_\phi \mathbf{I}\big) \,\big\|\, \mathcal{N}(\mathbf{0}, \mathbf{I})\Big) = \frac{1}{2}\sum_{j=1}^{d}\Big(\sigma^2_{\phi,j} + \mu^2_{\phi,j} - 1 - \log \sigma^2_{\phi,j}\Big),$$

where d is the dimensionality of the latent space representation $\mathbf{z}$ of a single datapoint $\mathbf{x}$.

In order to get the entire dataset or batch **KL divergence** we compute the average **KL divergence**.

5. Predicting Log-Variance in the Decoder

In many practical scenarios, rather than fixing $\sigma^2 = 1$, we allow the decoder to **predict** the variance (or log-variance). This is sometimes referred to as a "learned variance" or "heteroscedastic" decoder. In that case, the decoder outputs both $\boldsymbol{\mu}_\theta(\mathbf{z})$ and $\log \boldsymbol{\sigma}^2_\theta(\mathbf{z})$ for each dimension of $\mathbf{x}$.

5.1 Gaussian NLL with Predicted Log-Variance

If the decoder outputs $\mu_{\theta,d}(\mathbf{z})$ for each dimension d along with $\log \sigma^2_{\theta,d}(\mathbf{z})$, then the decoding distribution is:

$$p_\theta(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}\Big(\mathbf{x}; \boldsymbol{\mu}_\theta(\mathbf{z}), \operatorname{diag}\big(\boldsymbol{\sigma}^2_\theta(\mathbf{z})\big)\Big).$$

Hence, the negative log-likelihood for a single data point $\mathbf{x}$ is:

$$-\log p_\theta(x|z) = \frac{1}{2}\sum_{d=1}^{D}[\log(2\pi\sigma^2_{\theta,d}(z)) + \frac{(x_d - \mu_{\theta,d}(z))^2}{\sigma^2_{\theta,d}(z)}]$$

We typically let the network predict $\alpha_d(\mathbf{z}) = \log \sigma^2_{\theta,d}(\mathbf{z})$ (i.e., the log-variance), so that $\sigma^2_{\theta,d}(\mathbf{z}) = \exp\big(\alpha_d(\mathbf{z})\big)$. Substituting this in, the NLL becomes:

$$-\log p_\theta(x|z) = \frac{1}{2}\sum_{d=1}^{D}[(\log(2\pi) + \alpha_d(z)) + \frac{(x_d - \mu_{\theta,d}(z))^2}{\exp(\alpha_d(z))}]$$

5.2 Minimizing the Full Gaussian NLL

When training a VAE with a learned variance (or log-variance), we do **not** drop the variance term, because it now **depends** on the decoder's parameters. Minimizing this term encourages the decoder to learn an appropriate variance for each pixel (or each dimension), balancing how "confident" it is in each reconstruction versus the magnitude of reconstruction error.

1. If the predicted variance is too small, the second term $\frac{(x_d-\mu_{\theta,d})^2}{\exp(\alpha_d)}$ can become very large.
2. If the predicted variance is too large, the first term $\alpha_d(\mathbf{z})$ in the log-likelihood penalizes that as well.

Thus, by learning both the mean and the log-variance, the model can adaptively weight reconstruction errors and produce more probabilistically calibrated outputs.

Tasks

Task 1 - Training the model

For this homework you will have to implement a VAE on the **CelebA** dataset which you can already find implemented in `torchvision`.

1. For the purpose of this homework - mainly computational efficiency - you will work with images resized to `64 x 64`.
2. You will use the train and test splits that are native to the dataset.
3. You can opt either for the variance predicting decoder, or the isotropic variance decoder (i.e. you just predict the RGB values / means)
4. *Optional* You can use another distribution for the decoder - i.e. Gaussian Mixture, Beta Dsitribution, etc - but beware, you'll need to explicitly mathematically model them. The first bit of demonstrations we have in this document is to showcase that minimizing the MSE is implicit to modelling a Gaussian.

In order to make it easier for you we provide the following tips, such that you can quickly get to a minimally working VAE (since this is just half of the homework)

- You will want to use something that is called *KL annealing* - i.e. you gradually increase the weight of the KL term in the full loss as training progresses.
- Your latent factor dimension (i.e. the dimensionality of $\mathbf{z}$) will play a **HUGE** role in the quality of your reconstructions / generations alongside the value of your *KL Divergence*. A larger dimensionality will allow for an easier reconstruction - because of the lower compression rate - but at the same time accumulate more error in the *KL term* which could lead into drowning the gradient signal coming from the reconstruction loss.
- Another trade-off that stems from this is the `latent identification` one. As the latent grows larger and larger it becomes more and more difficult to identify which latent dimensions and values are correlated with a particular attribute of the data. This is called posterior collapse. It makes it exponenitally harder to identify which changes should be made to a latent in order to edit an already existing image for a desired outcome *(spoiler alert for part 2 in the homework)*
- You might want to use a *Perceptual Loss* in order to speed-up training. Any model will work, both a ResNet or a Clip will work. This will speed-up convergence time for reconstruction quality.
- You don't want to use a giant batch size - it will be prone to mode collapse. TL;DR for this, unde MSE optimization, the mini-batch mean becomes more stationary as we increase the batch size - i.e. the variance between the means of mini-batches will become smaller and smaller which will encourage the network to output the naive solution which is the dataset mean. If your reconstructions look oddly similar, try toning down the batch size.
- In order to go from features to latent we strongly suggest you do something as follows -> `nn.Linear(H * W * channels, latent_dims * 2)` for the encoder and `nn.Linear(latent_dims, H * W * channels)` for the decoder. If you use `GlobalAveragePooling` you are going to lose **A LOT** of information. In order to overcome that, you will need to over-parametrise both the encoder and the decoder. This `nn.Linear` approach will require you to strike a balance between how much you spatially compress the image and how large you make the latent in order to not blow up its size.
- For those of you that opt for the variance predicting decoder: There is a high likelihood of your decoder to output artifacts in the reconstructions - those are pixels / regions for which your decoder outputs high values for $\alpha_d(z)$. You should clamp that output head to have a bounded range, say `(-3, 3)` (no, this are not our refference implementation values). You can simply do this with a `torch.clamp`.
- For those of you that opt for the variance predicting decoder: The artifacts might go away after sufficient training or by toning down your perceptual loss term weight.

- For those of you that opt for the variance predicting decoder: You are going to get significantly better NLLs, and by extension BPDs *(spoiler alert for bonus conditions)* but potentially worse *KL divergences*

Model definition

Safest bet is to use something similar (i.e. a simillar architecture, not copy-pasted code) to the stable diffusion encoder/decoder blocks. You can find some refferences here and here.

There is a high likely-hood that using attention might be too "expensive" for you. We have a StableDiffusion inspired reference implementation that we can configure to work well enough in the range of `3–9 GB VRAM` with `bfloat16` training at a batch-size of 32 (`float16` will have a simillar memory cost). There is plenty of wiggle room for you to experiment with sizing based on your hardware configuration.

In order to also help you with this stage, we provide the architecture guide-lines we've used:

- Residual Bottlenecks with 3 `ConvNormActs` like in ResNeXt. We've set it up such that we always have 32 groups, or at 4 channels per group.
- `SiLU` and `GroupNorm` just like in StableDiffusion.
- Squeeze and Excitation gating for the Residual Bottleneck
- Layer Scale on top of the Squeeze and Excitation operation
- Non-parametric residuals for upscaling and downscaling ops like in DC-AE (hint: Page 4, Section 3.2)
- Downscaling is a strided convolution, Upscaling is Bilinear upsampling followed by a convolution.

With this architecture, we managed to get working implementations for latent dimensions between 128 and 2048 factors. With `torch.compile` our largest variant takes about 10 minutes per epoch on a single A100, with the main bottleneck being dataset loading and processing alongside logging. You can look into WebDataset format in order to speed up your workloads (reduces disk pressure significantly). We also suggest you look into performing as much of the preprocessing when setting up the `.tar` files initially in order to avoid doing them at training time. We did not use this optimization, in order to not give ultra-optimised reference times.

Health-checks

In order to make sure you're heading in the right dirrection we suggest you check the following 2 things, using visual plots. NLLs and BPD require some degree of interpretation (their scale is highly affected by your KLD which affected by your latent size, whilst also being affected by the data dimensionality D)

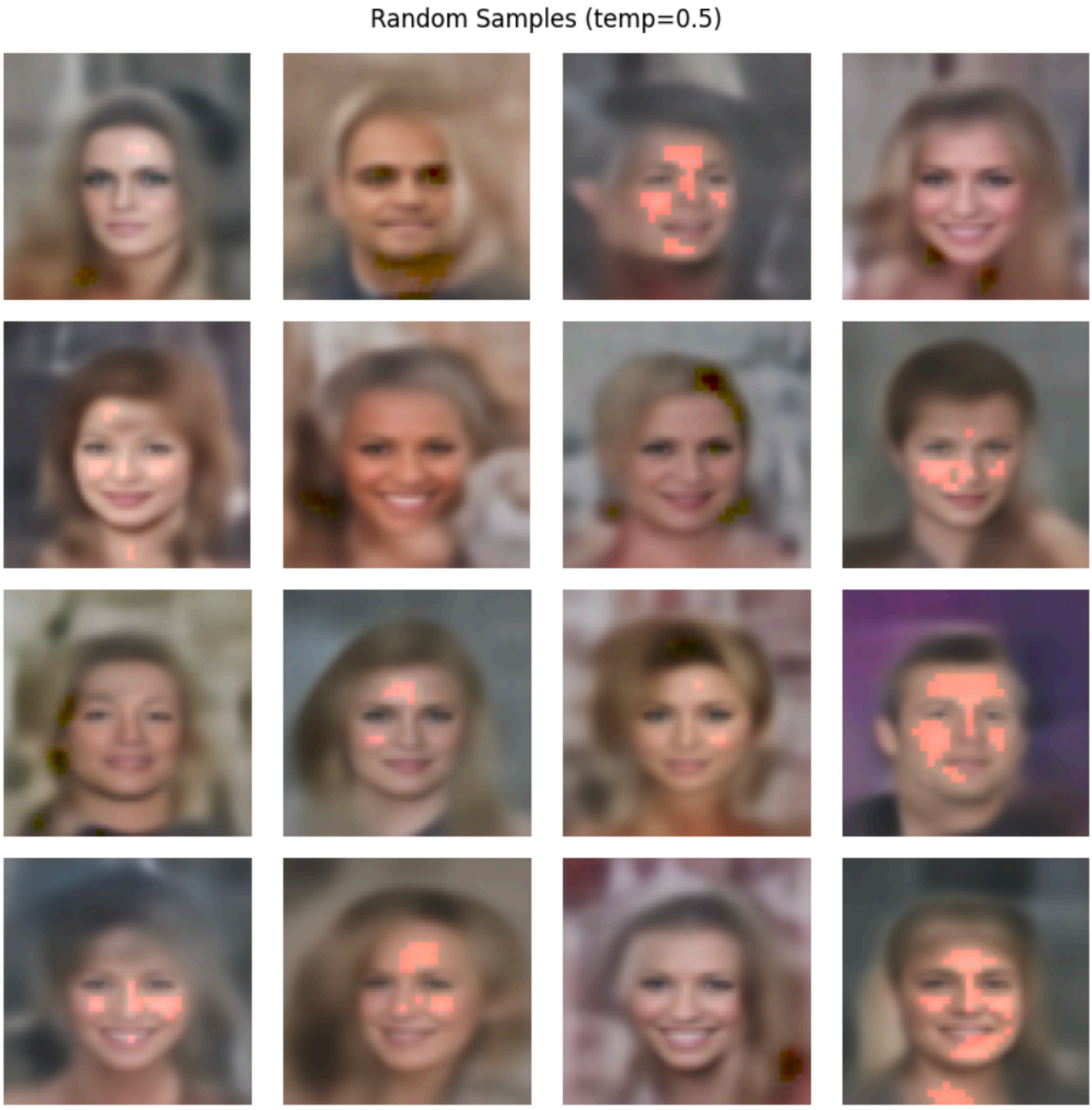
- Check that your model can interpolate between two samples.
- Check that you can sample from the posterior and not have completely degenerate results.

Bellow is an example of how your model should be abble to interpolate between two samples. In order to achieve this, you should use your encoder in order to get μ_1 and μ_2 out of your two samples and treat them as as the actual latent code $\mathbf{z_1}$ and $\mathbf{z_2}$ without accounting for any variance. Then, you want to treat your interpolated code as $\mathbf{z_i} = \mathbf{z_1} \cdot \alpha + (1 - \alpha) \cdot \mathbf{z_2}$

When sampling from the posterior remember that you have to account for $\mu(\mathbf{x})$ and $\log \sigma^2(\mathbf{x})$ under the reparametrization trick of $\mathbf{z} = \mu(\mathbf{x}) + \sigma(\mathbf{x}) \cdot \boldsymbol{\epsilon}$. For simplicity we can set $\mu(\mathbf{x})$ to a constant and vary $\sigma(\mathbf{x})$ in a predetermined range such as $[0.2, 0.5, 1.0, 1.5]$

As you increase $\sigma(\mathbf{x})$ for a predetermined $\mu(\mathbf{x})$ you should gradually observe details being added. Don't be afraid if for high variances your images look very hectic, that's a normal phenomenom for an unoptimised VAE. In our pictures $temp = \sigma(\mathbf{x})$. We obtain the latent codes by sampling $z \sim \mathcal{N}(0; 1)$ and then performing $z = z \cdot temp$

Random Samples (temp=0.2)



Task 2 - Editing Pictures

In this task you will have to perform 3 types of image edits.

1. Feature amplification
2. Label guidance
3. Identity transfer

We denote the latent code $\mathbf{z} = [z_1, z_2, ..., z_d]$. In order to amplify a component we can take a component $z_i' = z_i + \alpha$ where α is a scalar whilst keeping the rest of the latent unchanged. Ideally some feature will be amplified in the reconstructed sample $\mathbf{x} = \mathbf{dec}(\mathbf{z})$ is amplified as α increases and diminishes if α decreases.

- For `feature amplification` you have to discover 4 such meaningful components. A meaningful components means that varying its intensity produces an observable change in the reconstructed image (think hair length, bear presence, skin color, etc.).
- You have to plot the observed effect across 10 values of α for each of the `4` identified component z_i. You have to showcase this effect across 8 different samples.
- Feel free to cherry pick your best results.

`CelebA` contains for each image 40 attribute labels such as `Arched_Eyebrows`, `Bags_Under_Eyes`. `Attractive`, etc. (the labels are dirrectly available in the `torchvision` dataset as well).

- For `label guidance` you will need to find a way such that according to the actual data and label distribution - $p_{x\sim\mathbf{X}}(\mathbf{label} \mid x)$ you can `steer` a latent such that $p(\mathbf{label} \mid \mathbf{dec}(\mathbf{z})) = 1$. In layman's terms, you need to change the latent representation of an already existing sample such some classification procedure assings some given label to that image.
- **BIG HINT** - assume you have some classifier of form $f(x) = label$, you set $\mathbf{z}$ as an optimizable parameter in some optimizer of choice and you do gradient descent such that $f(\mathbf{dec}(\mathbf{z})) = label'$ where $label'$ is the modified label vector of the original sample.
- You have to showcase the label guided latents for 4 different label changes on 8 samples each - i.e. a sample that originally had 0 for `Bags_Under_Eyes` will be modified to have 1 for this particular label.

`CelebA` also contains some `IDs` that indicate if a picture is of a particular person.

- For `Identity Transfer` you should choose 3 "people" which will serve as anchors.
- You should then find 8 different subjects and try to morph their picture such that it resembles as much as possible the 3 "anchor people". You should aim to preserve as much as possible of the original subject picture (i.e. if they were looking left, they should still be looking to the left)
- **BIG HINT** - Encode multiple images of each anchor identity into your latent space. Find the average latent representation for each anchor identity. The average vectors should capture the 'essence' of each person's identity while averaging out pose/expression variations. You can try interpolating in between a subject and the average vector.
- **BIG HINT** - You can try using Integrated Gradients on the latent space.
- **BIG HINT** - You can try using some sort of PCA-like method on the latent space.

Grading

- By this point you got used to us asking of you to produce some sort of report. This time is no different. The report should minimaly contain your primary design choices for the network architecture and optimization hyper-parameters and other necessary bits of information.
  --
- [**7p**] for implementing a VAE that has sensible visulizations for the 3 image grids we included. In this .pdf
- Additionally you have to add "loss plots" for you **KL Divergence** & **Gaussian NLL** on the train/validation splits
- We want to see that your model is capable of interpolating whilst being able to reconstruct something that resembles a human face.
- When you perform the "temperature" sampling we want to see that for the lowest temperature the model produces samples looking like an "average face" and that the next level of temperature adds some more detail to those images.
  --
- [**3p**] for producing the vizualizations we asked of you in picture editing task
- We're interested in simply observing the desired effect, not how realistically looking it is (we're not trying to make deepfakes here)

Bonus structure

Since it's a late homework, we put out an entire **1p** of final grade bonus. This point is split into two categories.

- Optimizing for BPD - because BPD takes in the reconstruction and the KLDiv, you could get better BPD by just overoptimising one term at the detriment of the other.
- Therefore you will have to implement a harmonic mean between the two normalized metrics, NLL & KL Div for tracking this objective. Since the formulas are using sum terms, the normalized scores will divide the KL Div by the number of latent components, and the NLL by the number of pixels. We will showcase during the course how this metric should be implemented. In your submission you have to introduce your model checkpoint, and a file `bonus_1.py` which will compute this metric on the test set. Your file will need to be runnable in order to validate your reported results and ensure fairness as we will be checking your metric implementation as well.
- Top 5 scores will receive, **0.5p, 0.4p, 0.3p, 0.2p, 0.1p**

- Optimizing for picture edits - we will reward the most faithful and best looking edits with **0.5p, 0.4p, 0.3p, 0.2p, 0.1p**
- Although this is very subjective - we will apply blind majority voting, and we'll be broadly looking at the fooling things when grading them
- Image quality - this will be the most dominant factor, the better reconstructions you can get, the better your performance will be weighted.

- Transfer quality - this will be the following important factor, say you add sunglasses to someone's face. If they look like actual sunglasses instead of 2 giant black spots on top of the subjects eyes - you will be graded higher.
- Perturbation amount - When performing any type of edit, we will highly reward if you are able to produce a faithul looking modification, without unnecessary perturbations - i.e. if you change someone's hair color, we'd want to only see that change, not their skin colour getting a slight tint change as well.
- Remember that this evaluation is subjective, as such, the better of a case (i.e. thorough / hefty report) you make in order to showcase the quality of your generated samples - the higher you will be ranked for this task.