Deep Learning 1 - Homework 3

Pedro M.P. Curvo
MSc Artificial Intelligence
University of Amsterdam
pedro.pombeiro.curvo@student.uva.nl

Part 1

1.1

To sample an image using the decoder f_θ from the generative model described in this section, we first need to sample a latent vector z from the prior distribution $p(z)$. In this case, the prior distribution is a standard multivariate Gaussian distribution, so we can sample z from a standard normal distribution, i.e., $z \sim \mathcal{N}(0, I_D)$.

Then, we need to compute the pixel probabilities using the decoder f_θ. For this, we pass the sampled latent vector z through the decoder f_θ, which is a neural network parameterized by θ. This will map the latent vector z to the probabilities of the Categorical distribution for each pixel x_m in the image. $f_\theta(z) \rightarrow (p_1, p_2, ..., p_k)^M$. Here: $p_m = (p_{m1}, p_{m2}, ..., p_{mk})$ are the event probabilities for pixel m being in one of the k categories. M is the number of pixels in the image.

Then, we can sample pixel values x_n for each pixel m by sampling from the Categorical distribution $Cat(x_m|f_\theta(z)_m)$, where, $x_m \sim Cat(p_m)$. This will generate the image by sampling each pixel value independently based on the probabilities computed by the decoder.

Finally we combine the pixel values x_m to from the image x_n.

1.2

Monte Carlo integration with samples from $p(z_n)$ can approximate the expectation of the log-likelihood, but it is inefficient for training VAEs because it requires a large number of samples to achieve an accurate estimate. The inefficiency arises from the high-dimensional nature of the latent space z.

As the dimensionality D of the latent space increases, the prior $p(z)$ becomes increasingly sparse, and the true posterior $p(z|x)$ often concentrates in specific regions of the latent space (illustrated by the blue contours in Figure 2). Consequently, a very large number of samples is required to adequately cover the latent space and accurately estimate the log-likelihood. This results in the computational cost scaling exponentially with the dimensionality of z, making it impractical for high-dimensional latent spaces.

For this reason, VAEs use an alternative approach, such as the variational posterior $q(z|x)$, which focuses sampling in regions of high posterior density, improving efficiency and scalability.

1.3

From Equation 10, we have:

$$\log p(x_n) - KL(q_\theta(z_n|x_n)||p(z_n|x_n)) = \mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)] - KL(q_\theta(z_n|x_n)||p(z_n))$$

Rearranging the equation we get:

$$\log p(x_n) = \mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)] - KL(q_\theta(z_n|x_n)||p(z_n)) + KL(q_\theta(z_n|x_n)||p(z_n|x_n))$$

Now, we know the KL divergence is always non-negative, since it is a measure of the difference between two distributions by measuring how much one probability distribution diverges from a second, expected probability distribution. It is zero if and only if the two distributions are the same. Mathematically, we have that:

$$\begin{aligned} KL(q_\theta(z_n|x_n)||p(z_n)) &\geq 0 \\ KL(q_\theta(z_n|x_n)||p(z_n|x_n)) &\geq 0 \end{aligned}$$

With this we have that:

$$\begin{aligned} \log p(x_n) &= \mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)] - KL(q_\theta(z_n|x_n)||p(z_n)) + KL(q_\theta(z_n|x_n)||p(z_n|x_n)) \\ &\geq \mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)] - KL(q_\theta(z_n|x_n)||p(z_n)) \end{aligned}$$

Therefore, the right-hand side of the equation $\mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)] - KL(q_\theta(z_n|x_n)||p(z_n))$ is always less than or equal to the true log-likelihood $\log p(x_n)$. Hence, the right-hand side of the equation is a lower bound on the true log-likelihood.

1.4

ELBO consists of two terms:

$$ELBO(x_n) = \mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)] - KL(q_\theta(z_n|x_n)||p(z_n))$$

The first term, $\mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)]$, represents the expected log-likelihood of the data x_n given the latent variable z_n. However, this value stays the same regardless of how $q(z_n|x_n)$ is chosen. The second term, $KL(q_\theta(z_n|x_n)||p(z_n))$, represents the KL divergence between the approximate posterior $q_\theta(z_n|x_n)$ and the prior $p(z_n)$. As $q_\theta(z_n|x_n)$ approaches the true posterior $p(z_n|x_n)$, the KL divergence term decreases towards zero. This is because the KL divergence is minimized when the two distributions are the same, $q(z_n|x_n) = p(z_n|x_n)$. With that being said, as the KL divergence term decreases, the ELBO increases. becoming closer to the true log-likelihood $\log p(x_n)$. Ideally, when $q(z_n|x_n) = p(z_n|x_n)$, the ELBO is equal to the true log-likelihood $\log p(x_n)$.

So:

$$\begin{aligned}
\lim_{q_\theta(z_n|x_n)\to p(z_n|x_n)} ELBO(x_n) &= \lim_{q_\theta(z_n|x_n)\to p(z_n|x_n)} \mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)] - KL(q_\theta(z_n|x_n)||p(z_n)) \\
&= \mathbb{E}_{p(z_n|x_n)}[\log p(x_n|z_n)] - \lim_{q_\theta(z_n|x_n)\to p(z_n|x_n)} KL(q_\theta(z_n|x_n)||p(z_n)) \\
&= \mathbb{E}_{p(z_n|x_n)}[\log p(x_n|z_n)] - KL(p(z_n|x_n)||p(z_n)) \\
&= \mathbb{E}_{p(z_n|x_n)}[\log p(x_n|z_n)] - 0 \\
&= \mathbb{E}_{p(z_n|x_n)}[\log p(x_n|z_n)] \\
&= \log p(x_n)
\end{aligned}$$

And that is why the main goal of training a VAE is to maximize the ELBO, as it is a lower bound on the true log-likelihood $\log p(x_n)$. And this corresponds to minimizing the KL divergence between the approximate posterior $q_\theta(z_n|x_n)$ and the prior $p(z_n)$.

1.5

The names **reconstruction** loss and **regularization** loss are used because:

- **Reconstruction Loss**: The term $\mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)]$ is the expected log-likelihood of the data x_n given the latent variable z_n. Hence, it is a measure of how well the model can reconstruct the observed data x_n from the latent variable z_n. Meaning, it directly corresponds to the task of reconstructing the original input, being the main objective of the model. Therefore, it is called the **reconstruction** loss.
- **Regularization Loss:** The term $KL(q_\theta(z_n|x_n)||p(z_n))$ is the KL divergence and ensures that the learned posterior distribution $q_\theta(z_n|x_n)$ is close to the prior distribution $p(z_n)$. This term regularizes the model by preventing the posterior distribution from deviating too much from the prior distribution. Therefore, it is called the **regularization** loss.

1.6

When the prior distribution $p(z_n)$ and variational posterior $q_\theta(z_n|x_n)$ are not Gaussian, the KL divergence term $KL(q_\theta(z_n|x_n)||p(z_n))$ cannot be computed simply in closed form. To overcome this, we can use the Monte Carlo approximation to estimate the regularization term.

The regularization term in the VAE is the KL divergence between the variational distribution $q_\theta(z_n|x_n)$ and the prior distribution $p(z_n)$. Mathematically, this term is defined as:

$$D_{KL}(q_\theta(z_n|x_n)||p(z_n)) = \mathbb{E}_{q_\theta(z_n|x_n)}[\log \frac{q_\theta(z_n|x_n)}{p(z_n)}]$$

Assuming the closed-form expression for the KL divergence is intractable, we can use the Monte Carlo approximation to estimate the KL divergence. First, we sample L latent vectors $z_n^{(l)}$ from the variational distribution $q_\theta(z_n|x_n)$, where $l = 1, 2, ..., L$. Then, we compute the log-ratio of the variational distribution and the prior distribution for each sample:

$$\log \frac{q_\theta(z_n^{(l)}|x_n)}{p(z_n)}$$

where $q_\theta(z_n^{(l)}|x_n)$ is the probability density of the variational distribution evaluated at the sample $z_n^{(l)}$. Finally, we compute the Monte Carlo estimate of the KL divergence as:

$$D_{KL}(q_\theta(z_n|x_n)||p(z_n)) \approx \frac{1}{L}\sum_{l=1}^{L} \log \frac{q_\theta(z_n^{(l)}|x_n)}{p(z_n)}$$

Then, we can use this approximation in the loss function:

$$\mathbb{L}_{regularization,n} \approx \frac{1}{L}\sum_{l=1}^{L} \log \frac{q_\theta(z_n^{(l)}|x_n)}{p(z_n)}$$

This methods allows us to estimate the KL divergence term when the prior and variational posterior are not Gaussian. The more samples we use, the more accurate the estimate will be, but it will also increase the computational cost of training the model. This method can be used for any arbitrary prior and variational posterior distributions, as long as we can sample from the variational distribution and compute the log-ratio of the two distributions.

It is important to note that in this case, sampling from $q_\theta(z_n|x_n)$ is significantly more efficient than using Monte Carlo integration to estimate the log-likelihood by sampling directly from $p(z_n)$. Sampling from the prior distribution $p(z_n)$ can be inefficient because the prior may not accurately capture the structure of the data. As a result, many samples may be drawn from regions in the latent space z_n that contribute little or nothing to $p(x_n|z_n)$, leading to a poor estimate of the log-likelihood. For instance, when $p(z_n)$ is a standard normal distribution, it spans the entire latent space, whereas the posterior distribution $p(z_n|x_n)$ is typically concentrated in specific regions of the latent space, as illustrated by the blue contours in the image.

By contrast, sampling from the variational distribution $q_\theta(z_n|x_n)$ is much more effective. This distribution is designed to approximate the true posterior $p(z_n|x_n)$, and as such, it focuses sampling on the regions of the latent space that are most relevant to the data. Samples drawn from $q_\theta(z_n|x_n)$ are concentrated in areas where the posterior is high, meaning they are more likely to contribute meaningfully to the log-likelihood estimate. This targeted sampling reduces the number of samples needed for accurate estimation, making Monte Carlo integration more efficient. In other words, leveraging $q_\theta(z_n|x_n)$ as the sampling distribution ensures that computational resources are focused on regions of the latent space that matter most, leading to better performance and efficiency.

1.7

When sampling directly from a distribution $q_\theta(z_n|x_n)$ to estimate the expectation $\mathbb{E}_{q_\theta(z_n|x_n)}[\log p(x_n|z_n)]$, the sampling process itself is non-differentiable with respect to the parameters θ. This is because sampling introduces randomness that disrupts the smooth computation graph required for gradient-based optimization. As a result, the gradients of the loss function with respect to the encoder parameters θ cannot be computed directly using standard backpropagation.

The **reparameterization trick** resolves this issue by making the sampling process differentiable. Instead of sampling z_n directly from $\mathcal{N}(\mu_\theta(x_n), \Sigma_\theta(x_n))$, the random variable z_n is reparameterized as a deterministic transformation of a noise variable $\epsilon \sim \mathcal{N}(0, I_D)$. Specifically, z_n is expressed as:

$$z_n = \mu_\theta(x_n) + \sigma_\theta(x_n) \odot \epsilon$$

where $\mu_\theta(x_n)$ and $\sigma_\theta(x_n)$ are the encoder outputs parameterized by θ, and $\odot$ denotes element-wise multiplication.

This reformulation ensures that the randomness is isolated in ϵ, which is independent of θ. Consequently, the sampling operation becomes differentiable with respect to θ, allowing gradients of the loss function to propagate through the encoder network via backpropagation.

1.8

Test BPD
0.518

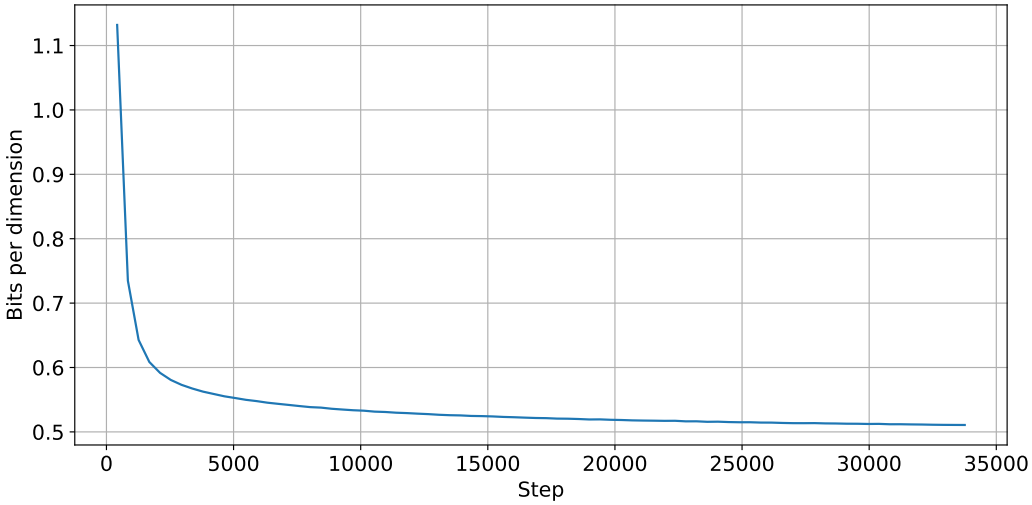

Figure 1: Train BPD for VAE with latent dim=20

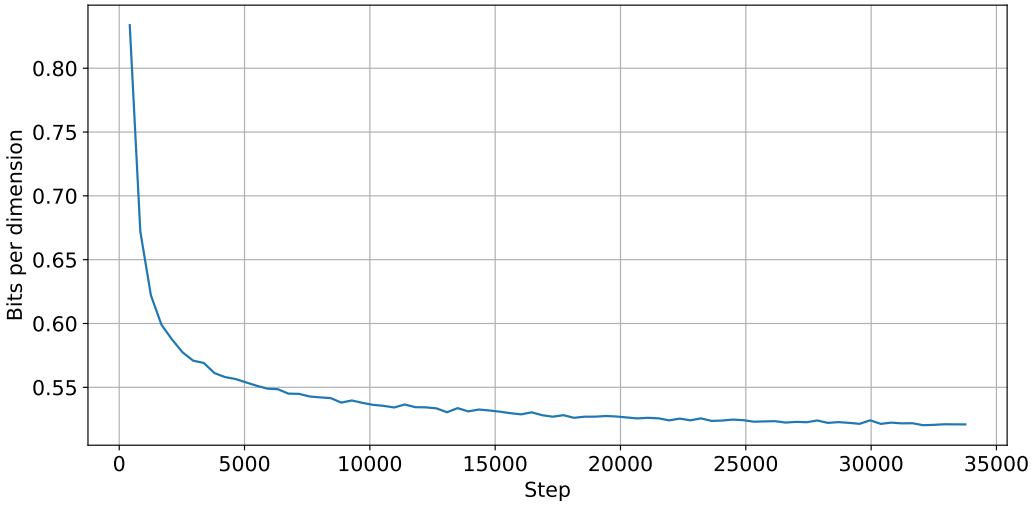

Figure 2: Validation BPD for VAE with latent dim=20

1.9

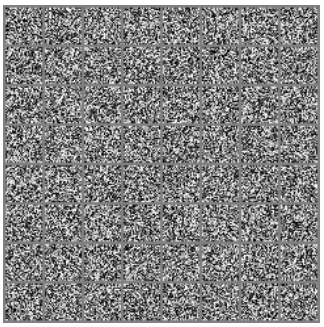

Figure 3: Epoch 0

Figure 4: Epoch 10

Figure 5: Epoch 80

Figure 6: Visualization of the VAE training progress at epochs 0, 10, and 80.

By looking at the images generated by the VAE model before training, middle of training, and after training, we can see that initially we only have noise in the images, as the model has not learned to generate meaningful images yet. As the model trains, the images start to become more recognizable, and we start to see some numbers appearing in the images resembling the digits in the MNIST dataset, but with some imperfections. At the end of training, I see that the images are more refined and resemble the digits in the MNIST dataset more closely, with fewer imperfections and noise, showing the model has learned to generate more accurate and realistic images as it trains.

1.10

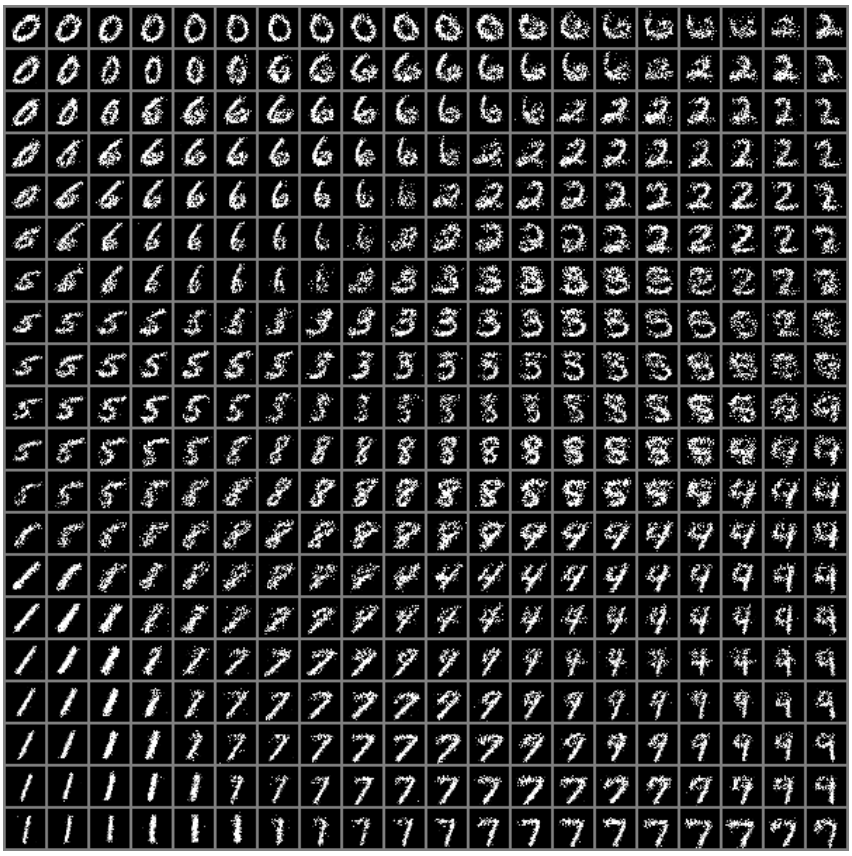

Figure 7: Visualization of the data manifold learned by the VAE. The manifold shows the reconstructed images from the 2-dimensional latent space.

The manifold illustrates the latent space learned by the VAE, decoded into MNIST digits. Points in the latent space that correspond to similar digits seemed to be grouped into clusters, occupying distinct regions in the grid. This clustering shows that the VAE encodes similar digits closer together in the latent space. For example, digits that are visually alike are positioned near each other, resulting in smooth transitions between neighboring points.

Some areas contain blurry or incomplete digits, indicating regions where the model struggles to represent the data distribution accurately, specially in the boundaries between different digits.

Part 2

2.1

Argument	**batch_size**	**valid_ratio**	**augmentations**	**pretrained**	**num_epochs**	**train_strats**
Value	64	0.75	False	True	30	['standard']
Argument	**visualise**	**epsilon_fgsm**	**alpha_fgsm**	**epsilon_pgd**	**alpha_pgd**	**num_iter_pgd**
Value	True	0.1	0.5	0.01	2	10

Table 1: Configuration used for a pretrained ResNet18 with and without FGSM attack

Attack Test Type	**Parameters**	**Test Accuracy**
Without	-	0.93
FGSM	Alpha: 0.5, Epsilon: 0.1	0.41
PGD	Alpha: 2, Epsilon: 0.01, Num Iter: 10	0.4228

Table 2: Testing results for **standard** training loss for different attacks using a **pretrained** ResNet18

- **i)** Adding a random perturbation of size ϵ does not target the specific vulnerability of the model, but rather adds noise to the input data. FGSM, on the other hand, is a targeted attack, since it uses the gradient of the loss function to find the direction in which the input data should be perturbed to maximize the loss for a given input. This targeted attack is more effective at fooling the model than adding random noise. It also ensures that the perturbation aligns with the model decision boundary, making it more likely to fool the model. Random noise, on the other hand, is unlikely to align with the decision boundary, making it less effective at fooling the model.

- **ii)** This phenomenon occurs because both models A and B, despite being trained on different subsets of the data, are optimized for the same task and learn to extract similar features from the data. As a result, they develop similar high-level representations and decision boundaries. Adversarial perturbations exploit weaknesses in these representations by pushing inputs in directions that maximize the loss, perturbation that align with the models' decision boundary (and in this case A and B will have similar decision boundaries). These directions often correspond to vulnerabilities shared by both models due to the shared structure of the data and the shared task.

  In high-dimensional spaces, even small perturbations designed for one model are likely to remain effective on another one because of the inherent similarities in their learned feature spaces and gradients. This shared susceptibility leads to the transferability of adversarial examples between models trained for the same task.

- **iii)** Models trained with data augmentation are more robust to adversarial examples because data augmentation introduces noise and perturbations to the training data, which makes the model more robust to small perturbations in the input. When the model is trained with augmented data, such as adding noise, rotations, or translations to the training images, the model learns to generalize better and consequently the model's decision boundary becomes smoother and less sensitive and reactive to small, targeted perturbations. This makes it harder for an attacker to generate adversarial examples that fool the model.

2.2

Argument	**Values**					
batch_size	64					
valid_ratio	0.75					
num_epochs	30					
epsilon_fgsm	0.1					
alpha_fgsm	0.5					
epsilon_pgd	0.01					
alpha_pgd	2					
num_iter_pgd	10					
visualise	True					
Device	GPU					
augmentations	**-**	**False**	**False**	**True**	**False**	**True**
pretrained	**True**	**False**	**False**	**True**	**False**	**False**
train_strats	[standard]	[standard]	[fgsm]	[fgsm]	[pgd]	[pgd]
Attack	**Test Accuracy**					
Without	93.00%	67.00%	61.00%	87.00%	53.00%	
FGSM	41.00%	17.36%	23.20%	56.24%	30.32%	
PGD	42.48%	15.88%	24.52%	62.24%	25.32%	
Train Loss		0.2033	1.0450	0.2839	0.6887	
Train Accuracy		0.9548	0.7514	0.9949	0.7705	
Validation Loss		0.9880	1.0957	0.3586	1.7670	
Validation Accuracy		0.6694	0.6103	0.8810	0.5058	
Best Validation Accuracy		0.6727	0.6171	0.8859	0.5373	

Table 3: Trained models (the columns that matter to this question are the second and third where we can see the results for with/without defense and with/without attacks) **Note:** We observe that by using a pretrained model and then using augmentation we get the best accuracy. This might be since we are finetunning the model with the adversial augmentation.

Without defense the model focuses solely on minimizing the loss on clean data, leading to high accuracy on natural, unperturbed inputs. However, it remains highly vulnerable to adversarial attacks, as it lacks mechanisms to handle small, targeted perturbations. That's why we see that without an attack we get 67% accuracy, but with FGSM we get 17%.

With defense, the model gains robustness to adversarial attacks by incorporating adversarial training. With this, the model learns smoother decision boundaries and learns how to adapt to small perturbations in the input data. This can reduce its accuracy on clean data (we see a drop to 61%) because the model shifts its focus from purely optimizing performance on the clean dataset to balancing performance on both clean and adversarial examples. This can lead to a decrease in accuracy on clean data. Hence, the models with defense often sacrifice better generalization on clean data to gain robustness to adversarial attacks (we observe better accuracy on adversarial attack - 23% compared to without defense).

This trade-off occurs because neural networks have **limited capacity** and must allocate resources to learn to generalize. Allocating some of this capacity to handle adversarial perturbations reduces the focus on patterns in clean data, leading to a decrease in accuracy on clean data. Besides this, defending against adversarial attacks often involves smoothing or reshaping decision boundaries to make them robust to perturbations, However, this might **oversimplify the boundary**, causing the model to misclassify some clean data points. During adversarial training, it might also be possible that the model might overfit to specific types of adversarial perturbations, as the FGSM, reducing the accuracy on clean data.

2.3

Argument	**Values**						
batch_size	64						
valid_ratio	0.75						
num_epochs	30						
epsilon_fgsm	0.1						
alpha_fgsm	0.5						
epsilon_pgd	0.01						
alpha_pgd	2						
num_iter_pgd	10						
visualise	True						
Device	GPU						
augmentations	**-**	**False**	**False**	**True**	**False**	**True**	**True**
pretrained	**True**	**False**	**False**	**True**	**False**	**False**	**True**
train_strats	[standard]	[standard]	[fgsm]	[fgsm]	[pgd]	[pgd]	[pgd]
Attack	**Test Accuracy**						
Without	93.00%	67.00%	61.00%	87.00%	53.00%		
FGSM	41.00%	17.36%	23.20%	56.24%	30.32%		
PGD	42.48%	15.88%	24.52%	62.24%	25.32%		
Train Loss		0.2033	1.0450	0.2839	0.6887		
Train Accuracy		0.9548	0.7514	0.9949	0.7705		
Validation Loss		0.9880	1.0957	0.3586	1.7670		
Validation Accuracy		0.6694	0.6103	0.8810	0.5058		
Best Validation Accuracy		0.6727	0.6171	0.8859	0.5373		

Table 4: Trained models

We observe that with pgd we get a better accuracy than with fgsm in adversarial attacks, but we loose overall accuracy wihtout any attacks, which is expected since PGD applies an iterative approach and can find more effective adversarial examples than FGSM. We also observe that the model trained with augmentation performs better than its counterpart without augmentation, which is expected as augmentation helps the model generalize better and become more robust to adversarial attacks.

i)

Comparing adversarial loss and adding adversarial examples to the batch:

- **Adversarial loss**: Adversarial loss incorporates adversarial examples directly into the objective function. This is done by adding adversarial perturbations to the loss term, mixing clean and adversarial gradients during each training step. This approach allows the model to learn from adversarial examples during the forward pass while still using the original input for calculating the primary loss. This method is efficient and straightforward, since it integrates adversarial examples directly into the loss function. It also separates the clean and adversarial data by using distinct loss terms for each. This separation can help prevent overfitting/underfitiing to adversarial perturbations, as the overall loss is a combination of clean and adversarial components. By smoothing one, the other may remain unaffected, preserving a balanced loss landscape. However, implementing adversarial loss requires modifying the loss function, which can lead to more finetunning.
- **Adding adversarial examples to the batch**: In this approach, the adversarial examples are explicitly generated and added to the batch during training, along with the original clean and samples. This results in the model training on a larger batch that contains both clean and adversarial examples, forcing the model to correctly classify both types of examples. It has the pros of the model seeing both types of inputs (clean and adversarial) in each iteration, increasing robustness. It has the cons of an increasing computational cost, because we are increasing the batch size. Besides that, since we are now using the same loss function for both clean and adversarial examples, the model might overfit to the adversarial examples

leading to a decrease in accuracy on clean data or oversmoothing the loss landscape (which now is just one overall), hurting the model's performance on clean data.

Difference: The key difference is that the adversarial loss modifies the optimization process directly by changing the objective function, while adding adversarial examples to the batch increases the diversity of the training data.

Alignment: These two methods may align when the adversarial loss function effectively integrates perturbations in a manner similar to how adversarial examples are added to the batch.

Divergence: They could diverge in scenarios where the adversarial loss focuses more on optimizing the network's response to perturbations, while adding adversarial examples to the batch gives the model a broader exposure to such attacks without necessarily optimizing its response to them directly. Adversarial loss is computationally more efficient, while batch augmentation demands greater resources due to increased batch sizes.

ii)

One advantage of the FGSM is the **speed**. FGSM is a single-step attack that computes the gradient once and adds a small perturbation based on that, making it very fast and computationally efficient compared to iterative attacks. However, its **effectiveness** is a limitation. FGSM produces more noticeable perturbations because it only takes a single step, and the model is more likely to recognize the pattern of the perturbation.

The PGD has the advantage of **robustness**. PGD is an iterative process which computes small gradient-based steps and projects the perturbed data back within the allowable ϵ-ball. This allows it to find more effective adversarial examples that are harder for the model to detect and often less perceptible to human observers. However, its disadvantage is being **slower**. The iterative nature makes it computationally expensive and significantly slower than FGSM.

The overall trade-off between FGSM and PGD depends on the specific use case and requirements, that is, **Speed vs Robustness**. FGSM is faster and computationally less demanding, making it suitable for quick attacks or scenarios where computational resources are limited. However, it is less effective than PGD in generating adversarial examples that cause consistent misclassifications, resulting in lower robustness to adversarial examples.

PGD is computationally more expensive due to its iterative nature but generates more precise adversarial examples that are harder for the model to classify correctly. This makes it more effective at improving a model's robustness to adversarial attacks.