HW3 - Solutions

1.1

To sample an image given a decoder f_θ, you first sample a latent vector $z \sim \mathcal{N}(0, I_D)$. You then feed z through the decoder, which produces for each pixel a vector of k numbers that represent how likely each intensity value is for that pixel (typically implemented as logits followed by a softmax to turn them into valid probabilities). Using these probability vectors, you draw a discrete value for each pixel from $\{0, \dots, k-1\}$. In practice this can be done independently per pixel or in a vectorized way with a categorical/multinomial sampler. Collecting all sampled pixel values gives one generated image x.

1.2

Monte-Carlo with samples from the prior $p(z)$ is inefficient because, for a given datapoint x_n, the integral

$$p(x_n) = \int p(x_n \mid z) p(z) dz \tag{1}$$

is concentrated on a small region of latent space where $p(x_n \mid z)p(z)$ is very large. By Bayes rule, this is proportional to the posterior $p(z \mid x_n)$, so Fig. 2 in the assignment PDF shows that most of the useful mass is concentrated in the blue posterior, while the prior (red contours) spreads its mass over a wider area. If we randomly sample z from the prior, most samples fall far outside the posterior region and contribute almost nothing to the Monte-Carlo average, leading to a very noisy, high-variance estimate of $\log p(x_n)$. As the dimensionality D of z increases, the posterior region occupies an exponentially smaller fraction of the prior's volume, due to the curse of dimensionality, so the probability that a random prior sample lands in this region shrinks exponentially in D. Therefore, the number of samples L needed to get a good estimate grows fast with D, making this approach impractical for training VAEs.

1.3

Equation (10) basically says

$$\log p(x_n) - D_{\mathrm{KL}}(q(z_n \mid x_n) \| p(z_n \mid x_n)) = \mathrm{ELBO}(x_n) \tag{2}$$

By definition, the KL divergence is nonnegative. Therefore

$$\log p(x_n) - D_{\mathrm{KL}}(\cdot \| \cdot) \leq \log p(x_n), \tag{3}$$

and both sides are equal only where the KL term is zero. Since the RHS of Equation (10) looks like $\log p(x_n) - D_{\mathrm{KL}}(\cdot\|\cdot)$, it will always be lower or equal than $\log p(x_n)$. Therefore, ELBO must be a lower bound on $\log p(x_n)$.

1.4

Again, from Equation (10) we get

$$\log p(x_n) - \text{ELBO}(x_n) = D_{\text{KL}}(q(z_n \mid x_n) \| p(z_n \mid x_n)) \tag{4}$$

As the variational distribution $q(z_n \mid x_n)$ becomes closer to the true posterior $p(z_n \mid x_n)$, this KL tends to zero. Therefore, ELBO converges to $\log p(x_n)$ and the LHS tends to zero. In other words, when the encoder's distribution matches the true posterior, maximizing the ELBO is equivalent to maximizing the true data log-probability.

1.5

The term reconstruction is appropriate because, with one latent sample $z_n \sim q_\phi(z_n \mid x_n)$, the loss becomes the negative log-likelihood of the observed image under the decoder conditioned on that latent code. Under the per-pixel categorical model, this reduces to the usual cross-entropy between the original pixel values and the decoder's predicted probabilities, which directly measures how well the model can reconstruct x_n from z_n.

The term regularization is appropriate because the KL penalty $D_{\text{KL}}(q_\phi(z_n \mid x_n) \| p(z_n))$ constrains the encoder's distribution to remain close to the prior. This stops the model from learning any arbitrary complex latent encoding, and encourages it to be smooth and structured. As this adds constraints, limiting the model's flexibility, its considered a form of regularization.

1.6

If the prior $p(z)$ is changed to a more complex distribution such that the KL has no closed form, we can estimate the regularization term with Monte-Carlo sampling from the encoder:

$$\begin{aligned} D_{\text{KL}}(q_\phi(z \mid x_n) \| p(z)) &= \mathbb{E}_{z \sim q_\phi(z \mid x_n)}[\log q_\phi(z \mid x_n) - \log p(z)] && (5) \\ &\approx \frac{1}{L} \sum_{l=1}^{L} (\log q_\phi(z^{(l)} \mid x_n) - \log p(z^{(l)})) && (6) \end{aligned}$$

where $z^{(l)} \sim q_\phi(z \mid x_n)$. Then we can use this as an estimate for the KL term, as long as we can evaluate (or approximate) $\log q_\phi$ and $\log p$ for sampled z.

1.7

If we sample $z \sim q_\phi(z \mid x)$ directly, the sampling step is non-differentiable. z is a random draw from a distribution whose parameters depend on ϕ, but there is no simple way for backpropagation to compute $\partial z / \partial \phi$. As a result, $\nabla_\phi \mathcal{L}$ would need an estimator like the REINFORCE algorithm, though that suffers from very high variance.

The reparameterization trick solves this by writing the sample as a deterministic function of ϕ and a noise variable independent of ϕ:

$$z = \mu_\phi(x) + \sigma_\phi(x)\,\varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, I). \tag{7}$$

Now the randomness is only in ε, and z is differentiable with respect to $\mu_\phi(x)$ and $\sigma_\phi(x)$, so we can find $\nabla_\phi \mathcal{L}$ using backpropagation.

1.8

For this question we implemented a CNN-based VAE with latent dimensionality $z_{\text{dim}} = 20$ using the default parameters, simply finished the skeleton. We trained the model for 80 epochs and tracked the bits-per-dimension (bpd) estimate of the negative ELBO on both the training and validation sets.

Fig. 1 shows the evolution of train and validation bpd as training progresses. Both curves drop sharply in the first few epochs (from well above 1 bpd, epoch 0 not shown, to roughly 0.65 by epoch 3), then it decreases and plateaus, such that the train bpd reaches roughly 0.52, and validation plateaus roughly at 0.54 bpd.

On the test set, the final model achieves a test bpd of ≈ 0.56, which is similar to the validation bpd, meaning the model generalizes quite well.

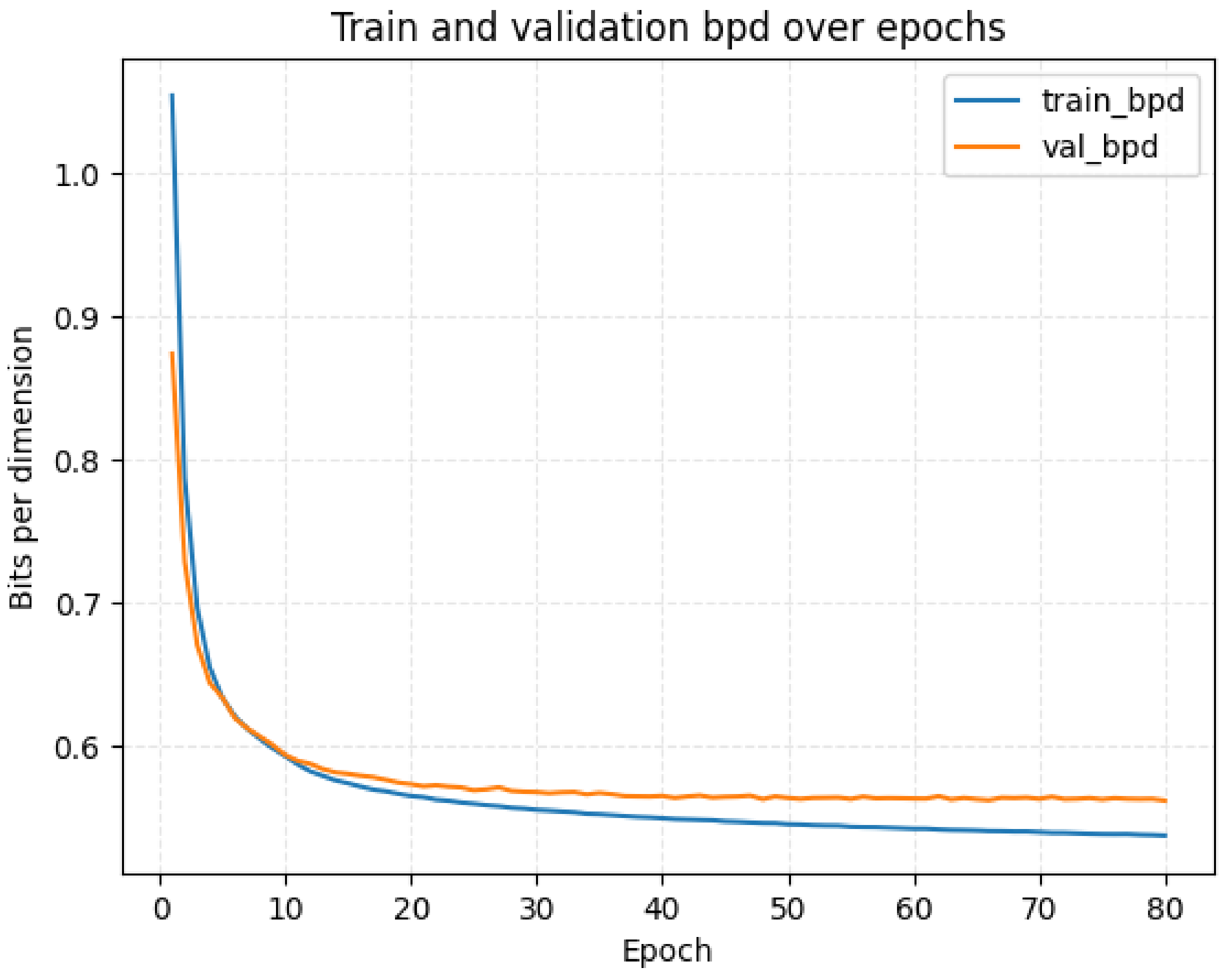
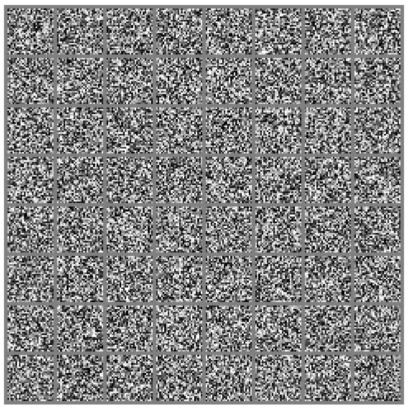
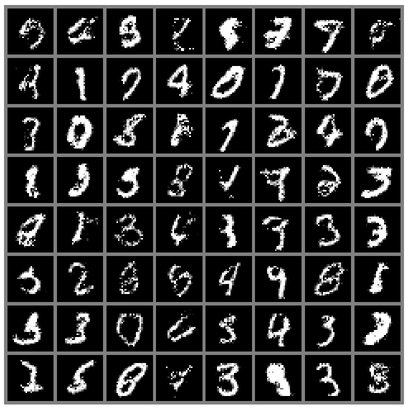


Figure 1: Bits-per-dimension (bpd) of the negative ELBO on the training and validation sets as a function of the training epoch for the CNN-based VAE with latent dimensionality $z_{\text{dim}} = 20$.

1.9

At epoch 0, all generated samples look like random noise with no structure whatsoever, as the decoder is completely untrained. After 10 epochs, most samples are clearly recognizable digits, but many have thin or fragmented strokes and noisy backgrounds, and a few are still ambiguous or distorted. By epoch 80, the digits are much better, lines are thicker and more continuous, backgrounds are cleaner, and only occasional digits are malformed. The improvement from epoch 10 to 80 is smaller than from 0 to 10, suggesting that sample quality has mostly converged.


(a) Epoch 0


(b) Epoch 10

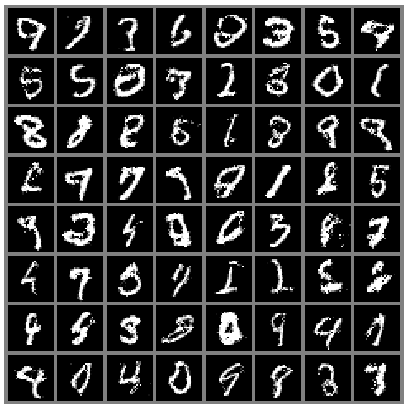

(c) Epoch 80

Figure 2: 64 generated samples (8×8 grid) from the VAE at different training stages.

1.10

In Fig. 3 we show the 20×20 manifold obtained from the VAE with a 2-dimensional latent space. Each tile corresponds to the decoder mean at a different point $z = (z_1, z_2)$, neighbouring tiles differ only slightly in z.

Horizontally, we see a clear progression of digit features. The left-most columns consist almost entirely of "0"-like digits, basically single closed loops with no tails. As we move right, a small opening and tail appears on the loop, turning the "0"s into "6"s and "9"s. Further toward the centre of the grid, these one-loop digits morph into more complex shapes with two loops or multiple bends, creating "8"s, "3"s and "5"s. Moving still further right, the shapes become simpler, and they shrink and straighten into a main vertical stroke with a small "hook", basically "7"s, and at the far right it becomes simply a vertical stroke, which looks like a "1". So, digits with similar structures (closed loops, tails, vertical stems) occupy neighbouring horizontal regions, and transitions between classes are gradual rather than abrupt.

Vertically, the manifold mostly changes the style and orientation of the digits. For example, for a fixed column of "0"s on the left, the circle rotates and deforms from top to bottom. In the centre of the image, the more complex numbers (2/3/5/8/9) change thickness, curvature and sometimes class as we move down the grid (for example, a "9" at the top of a column flattens into a "4" or "7", then turns into more rounded "8"/"3"/"5"). On the right-hand side, the top rows show "7"s, which transition into "1"s, then into vertical lines.

Therefore, the more complex shapes with multiple strokes all appear towards the centre of the latent space, while simpler shapes (0, 1, straight slashes) appear toward the edges. This means that the VAE has learnt smooth, structured, 2-d latent space.

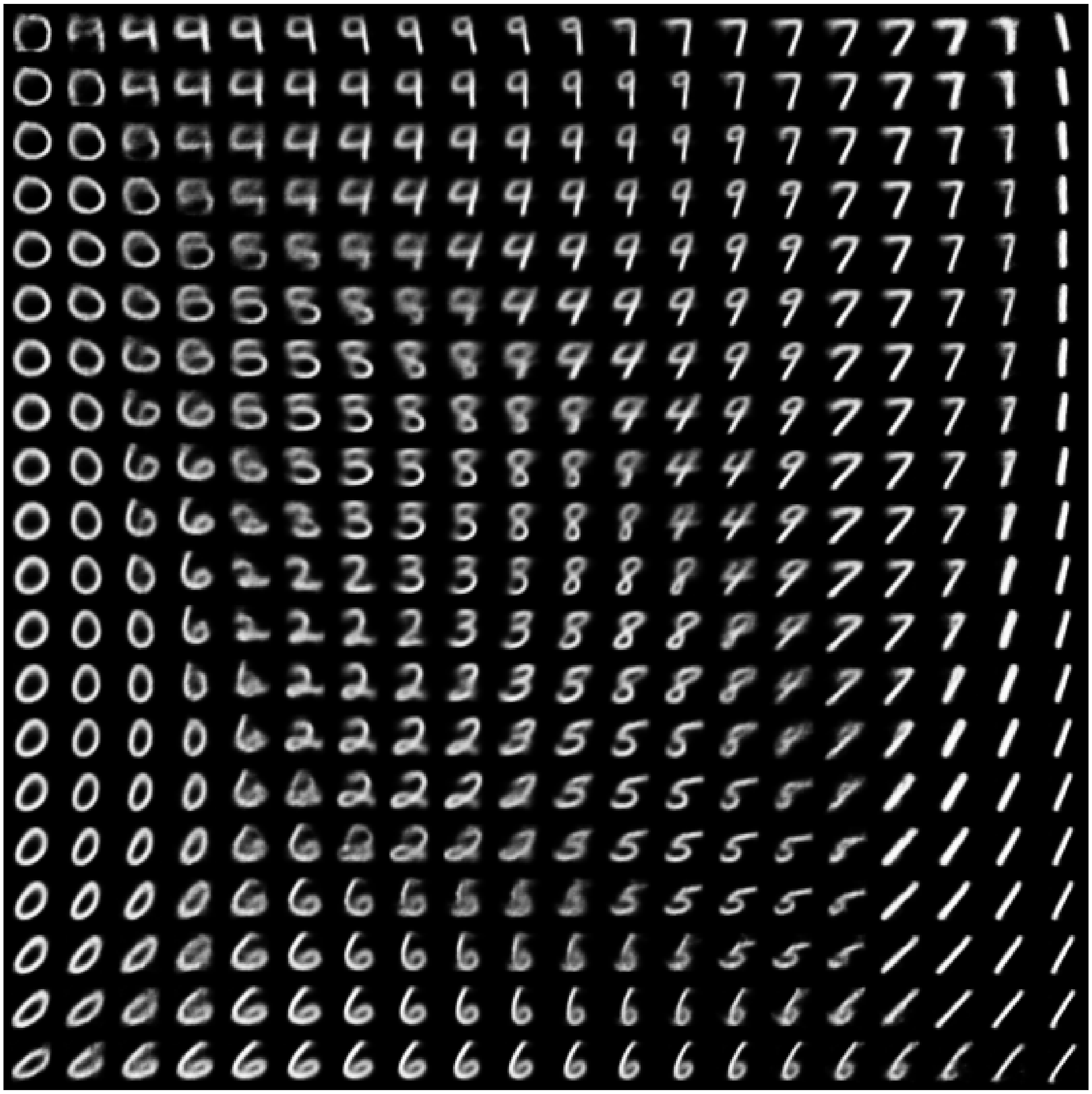


Figure 3: Learned 2D latent manifold of the MNIST VAE ($z_{\text{dim}} = 2$). Each tile shows the decoder mean for a different point in latent space, arranged on a regular grid of Gaussian percentiles. Moving horizontally from left to right, digits evolve from "0"-like shapes through "6/9" and multi-loop "2/3/5/8/9" digits to "7"s and almost straight "1"-like strokes. Moving vertically mainly changes style and orientation (e.g. rotation, stroke thickness). Digit classes occupy contiguous regions and morph smoothly into one another, indicating that the model has learned a coherent 2D latent manifold for the data.

1.11

As we increase β in Equation (15), we strengthen the KL regularization term relative to the reconstruction term. This forces the approximate posterior $q_\phi(z_n \mid x_n)$ to stay closer to the Gaussian prior $p(z_n) = \mathcal{N}(0, I)$ and reduces the information capacity of the latent channel, therefore applying a stronger regularization effect. For values of β slightly larger than 1, this regularizing effect encourages the model to use each latent dimension more efficiently, so individual coordinates of z tend to align with distinct generative factors (e.g. position, rotation, thickness).

However, if β is increased too much, the KL term dominates the loss and the easiest way to minimise it is to make $q_\phi(z_n \mid x_n) \approx p(z_n)$ for all x_n. In this case, the encoder output no longer depends meaningfully on the input, so the latent variables carry little or no information about the data, the decoder then tries to model $p(x)$ while basically ignoring z, which harms reconstruction quality. When $\beta < 1$ the KL term is less relevant, so the model is allowed to push $q_\phi(z_n \mid x_n)$ far away from the prior and pack a lot of information into z_n. This means less regularization, so the reconstruction is better, but it leads to more a entangled space and worse sampling.

2.1

(a)

For the main FGSM experiment we used a pretrained ResNet18 on CIFAR-10. We did not perform further training, and evaluated the model on the test set under three conditions: no attack, FGSM, and PGD. The data were normalised using the default CIFAR-10 statistics, with no data augmentation, batch size 128, and validation ratio 0.75. For FGSM we used $\varepsilon_{\text{FGSM}} = 0.1$ and $\alpha = 0.5$, and for PGD we used $\varepsilon_{\text{PGD}} = 0.01$, $\alpha = 2$, and 10 iterations.

For part 2.1(c) we trained two ResNet18 models from scratch, one without any data augmentation, and one with the provided random horizontal flips and random crops. In both cases we used the same attack parameters as above when evaluating FGSM and PGD. The resulting accuracies for all runs are summarised in Table 1.

Model / setting	Pretrained	Augment.	Attack	ε	# iters	Test accuracy
ResNet18	yes	no	none	–	–	93.0%
ResNet18	yes	no	FGSM	0.1	1	40.16%
ResNet18	yes	no	PGD	0.01	10	42.36%
ResNet18	no	no	none	–	–	62.0%
ResNet18	no	no	FGSM	0.1	1	13.76%
ResNet18	no	no	PGD	0.01	10	10.92%
ResNet18	no	yes	none	–	–	60.0%
ResNet18	no	yes	FGSM	0.1	1	12.36%
ResNet18	no	yes	PGD	0.01	10	13.96%

Table 1: Configuration and test accuracies for ResNet18 on CIFAR–10 under no attack, FGSM and PGD, for the pretrained model and the two models trained from scratch with and without data augmentation.

We observe that the pretrained model achieves high accuracy on clean test images (93%), but its performance degrades sharply under adversarial perturbations (around 40–42% under FGSM and PGD). For the models we made for 2.1(c), accuracies for the normal test data are lower (about 60–62%) and accuracies to attacks are also lower, and more importantly the effect of augmentaiton on robustness against attacks is either zero (for FGSM), or very small, but potentially non-negligeble (for PGD).

(b)

Let $J(\theta, x, y)$ be the loss at input x. For a small perturbation η we can use a first-order Taylor expansion:

$$J(\theta, x+\eta, y) \approx J(\theta, x, y) + \eta^\top \nabla_x J(\theta, x, y) \tag{8}$$

For a fixed-size $\eta \leq \epsilon$, the perturbation that maximizes this increase in loss matches the PDF's equation in Section 2.1:

$$\eta = \epsilon \cdot \operatorname{sign}(\nabla_x J(\theta, x, y)) \tag{9}$$

which is exactly the FGSM update. This aligns the change on each pixel with the sign of the gradient, so it maximizes the product $\eta^\top \nabla_x J(\theta, x, y)$, meaning the loss increases as much as possible for the given "budget" ϵ. In contrast, a random perturbation η_{rand} with the same norm (size ϵ) has no reason to align with $\nabla_x J$, and in complex environments, the components of η_{rand} could be as likely to align or not with the gradient sign, so the expected dot product is close to zero (little change in loss). Therefore, random noise behaves more like some form of data augmentation and leads to much smaller changes in the loss and prediction, while FGSM is explicitly built to be a worst-case perturbation within the same magnitude constraint.

(c)

When we split the original training set into two large, stratified subsets and train models A and B with the same architecture and a similar initialization, both models are effectively trained on the same underlying data distribution. Even though they do not see exactly the same examples, each model receives a representative sample of every class, and then the optimization (with same loss, same architecture, similar initial weights) induces very similar inductive biases. As a consequence, A and B tend to learn very similar feature representations and decision boundaries in input space.

An adversarial perturbation constructed from the gradients of model A is design to exploit weaknesses of A's decision boundary. Because model B has learned a very similar boundary, and uses on similar features, that same perturbation usually moves the input in a direction that is also harmful for B. Therefore, the observed transferability is likely due to this shared training distribution and similar learned features.

(d)

For the effect of data augmentation, we trained two standard ResNet18 models from scratch: one without any augmentation and one with the provided random flips and crops. Without augmentation, the clean test accuracy was 62% and the FGSM accuracy (with $\varepsilon = 0.1$) was ≈ 13.76. With augmentation, the clean test accuracy was 60% and the FGSM accuracy was $\approx 12.36\%$.

These values are almost identical, so for FGSM there is essentially no practical difference between training with or without these augmentations. This makes sense, as these augmentations enforce invariance to natural transformations (flips, crops), which does not directly protect against per-sample gradient-based perturbations.

We repeated the same experiment with PGD attacks ($\varepsilon = 0.01$, 10 iterations). Here, the model without augmentation scored $\approx 10.92\%$, while the augmented model scored $\approx 13.96\%$. Thus, augmentation does protect the model a bit from PGD, but has basically no impact on FGSM.

Overall, these results show that (these) data augmentations alone do not provide a meaningful defense against adversarial attacks.

2.2

(a)

We trained a ResNet18 using the FGSM-based adversarial loss using the same pretrained weights as in Question 2.1 and using the same FGSM attack parameters at test time ($\varepsilon = 0.1$, $\alpha = 0.5$). The defended model reached a best validation accuracy of 0.9053 and achieved 92% accuracy on the clean test set. Under the FGSM attack, its test accuracy was $1506/2500 \approx 60.2\%$.

For comparison, the standard pretrained model from Question 2.1 (no adversarial loss) achieved 93% accuracy on clean test data and 40.16% accuracy under the same FGSM attack. The FGSM defense therefore, has no (or a very small) impact on the normal accuracy while drastically improving robustness to the attack, increasing FGSM accuracy from roughly 40% to about 60%.

(b)

An undefended model optimizes purely for performance on the clean data distribution, and can therefore exploit highly predictive, but potentially fragile, features, overfitting on noise that FGSM attacks later exploit, meaning they can yield a slightly higher accuracy on the normal dataset, but a much poorer performance under a FGSM attack. In contrast, a FGSM-defended model is trained to minimise loss on both, clean and perturbed inputs, which will constrain the solution space to parameter settings that are relatively insensitive to these FGSM perturbations. This improves robustness, but reduces the model's ability to fit the clean data as aggresively as before, leading to a small drop in clean accuracy. With finite data and limited capacity, the netwrok can not simultaneously be optimal for both objectives, so optimizing for robustness to this attack will inevitably hurt performance. Our results discussed above show exactly this, though the performance drop on the normal dataset is actually very small, whereas the improvement on FGSM attacks is much larger.

2.3

(a)

We implemented the PGD attack as an iterative variant of FGSM, starting from the original image and repeatedly taking a step of size α in the sign of the gradient, followed by projection back onto the ball of radius ϵ around the original input and clamping to the valid pixel range. In `test_attack`, we added a block for PGD that uses this function to generate adversarial examples on the test set and then re-classifies the perturbed images. For the defense, we implemented another section within `train`, that generates PGD adversarial examples for each batch, concatenates them with the normal inputs, and trains the model on this bigger dataset using the cross-entropy loss.

For this part we trained a ResNet18 using this defense, starting from CIFAR–10 pretrained weights, with data augmentation enabled and using the default PGD parameters. The best validation accuracy during training was 0.8582, and on the test set the PGD-defended model achieved 87% accuracy on normal data. Under the PGD attack with the same parameters as used in training, it scored a test accuracy of $1724/2500 \approx 68.96\%$.

For comparison, the standard pretrained model from Question 2.1 (no adversarial training) with the same attack parameters reached 93% accuracy on clean data but only 42.36% accuracy under PGD. Thus, the PGD defense significantly improves robustness to PGD (from about 42% to about 69%) at the cost of a normal test accuracy (from 93% down to 87%).

(b)

In our implementation, “using an adversarial loss” refers to the FGSM-based defense of Question 2.2, where for each we compute an adversarial example and minimize the loss in Section 2.1 of the assignment PDF. And by adding adversarial examples to the batch, we do this PGD-based defense, where for each batch we generate PGD adversarial examples and then simply concatenate them to the clean batch, optimising the standard loss over all $2N$ samples (clean and adversarial) with equal weight.

Conceptually, both approaches are very similar, as they they both train the model to minimise loss simultaneously on clean and adversarially perturbed inputs. In the special case where we generate exactly one adversarial example per clean example using the same attack and weight them equally ($\alpha = 0.5$ in the equation given), adding adversarial examples to the batch and using an adversarial loss are mathematically equivalent, as both minimise the mean of $J(\theta, x, y)$ and $J(\theta, x_{\text{adv}}, y)$ over the training distribution. In practice, however, they can diverge because (i) the ratio between clean and adversarial samples can be chosen differently, and (ii) the attack used to generate x_{adv} may be stronger or weaker (through multiple steps). In our case, FGSM uses a single step and PGD uses multiple steps (10 by default), so both aim to improve the model's robustness but PGD defense targets a stronger adversary effect.

(c)

FGSM and PGD represent a tradeoff between computational cost and attack strength. FGSM is a single-step method that requires only one backward pass per batch. It is therefore very efficient and simple to implement, making it a much better choice when we have little compute, or we need a quick measure of how robust the model is to adversarial attacks. However, because it takes only one step in the gradient direction, it can underestimates the worst-case loss within the ϵ-sized ball and will be easier to defend against compared to PGD.

In contrast, PGD iterates over multiple gradient steps and projects it back into the ϵ-sized ball. This makes it a more accurate approximation of the best worse-case perturbation, so PGD produces better adversarial examples. The problem is that PGD is much more expensive, as it performs several forward and backward passes per batch per optimization step. Therefore FGSM is better for speed and lower complexity, whereas PGD can find stronger adversarial examples, and provide more meaningful robustness guarantees, though at a higher expense.