Deep Learning Dry Report

Shirah Hassan - 322694019 and Hillah Hassan - 209583574

April 2025

1 Theoretical Questions

1. The latent vector is essentially our compressed representation of an image. If we start with a small latent dimension, we're asking the encoder to cram all the important features of the image into a tiny space. This is tough to pull off, especially for more complex images like those in CIFAR-10. Essentially, we'd lose too much information, making it hard for the classifier to figure out what's in the image, and the accuracy would likely suffer.

   On the other hand, increasing the latent dimension gives the encoder more space to capture the image's features. This works well at first — the latent vector becomes more expressive, and the classifier has access to more meaningful features, which leads to higher accuracy. But if we keep increasing the latent dimension, problems start to crop up. Eventually, it gets so large that we're no longer really compressing the data — we're just storing big chunks of the original data in the latent space. This undermines the whole point of representation learning. Plus, larger latent spaces can add noise or redundancy, forcing the classifier to sift through irrelevant or repetitive features, which could end up hurting performance. So, at some point, we'd see the classification accuracy start to drop again.

2. **(a)** The Universal Approximation Theorem (UAT) tells us that a neural network with just one hidden layer can, in theory, approximate any continuous function to any level of accuracy, as long as it has enough neurons and uses a suitable activation function (for example ReLU or sigmoid). That means, given a function $f(\mathbf{x})$ and a small error tolerance $\epsilon > 0$, there exists a one-hidden-layer network $\phi(\mathbf{x})$ such that

   $$|\phi(\mathbf{x}) - f(\mathbf{x})| < \epsilon$$

   for every input $\mathbf{x}$ in a bounded domain. So in the context of MLPs, this result supports the idea that a single-layer network is in theory *capable* of achieving optimal error on any dataset and any loss function.

(b) Even though the Universal Approximation Theorem suggests that one hidden layer is theoretically enough, my friend's conclusion that we should never use more than one hidden layer doesn't really hold up in practice. The key issue here is efficiency. While a shallow network might be able to represent the function we're after, it could require an absurdly large number of neurons to do so. This makes the network harder to train, more prone to overfitting, and overall, less efficient. On the other hand, deeper networks can often learn the same functions with far fewer parameters by building features layer by layer. Each additional hidden layer captures increasingly abstract patterns, which not only makes the model more compact but also easier to optimize.

3. **(a)** Bob's right — a CNN is the way to go for image classification. The big advantage of a CNN over an MLP is that CNNs are built to take full advantage of the spatial structure in image data. When we think of an image, it's essentially a 2D grid of pixels, and nearby pixels tend to be really similar. This makes sense semantically too — like, the pixels on a water bottle in one area are probably going to share similar colors. A CNN works by using small filters called kernels to process little patches of the image. This helps it detect basic stuff like edges, corners, and textures. Then, as we move through the layers, it starts combining these lower-level features into more complex ones, like shapes and objects.

   On the other hand, an MLP treats the input as just a flat vector. So, when you want to feed an image into an MLP, you have to flatten it into a 1D vector, which completely loses the spatial relationships between pixels. Because of this, the MLP has to figure out all those pixel relationships from scratch, with no built-in knowledge about how images are structured. As a result, MLPs need way more parameters and training data to perform well, and they're more likely to overfit on image data. Plus, since every neuron in an MLP is connected to every pixel, it ends up being way more computationally expensive for high-dimensional data like images.

   (b) Alice is right that convolution is a linear operation, but her conclusion misses the point. The key difference is in how the linear operation is structured. In a fully connected layer, every output neuron is connected to *all* the input neurons. But in a convolutional layer, each output is only connected to a small, localized region of the input (the receptive field), and the same filter is applied across different parts of the image. This concept is known as weight sharing, and it results in translation invariance, which is super useful in image classification. So, even though both operations are technically linear, convolutional layers incorporate two important biases that fully connected layers don't: local connectivity and weight sharing. These biases make CNNs way more efficient and effective for image data. Essentially, CNNs generalize better, require fewer parameters, and train faster on image tasks compared to MLPs.

4. If we swapped EMA for simple linear averaging in optimization algorithms,

we'd probably see a decrease in efficiency. EMA places more weight on recent gradients, which helps optimizers like Adam or RMSprop respond quickly to changes during training. This quick adaptation is crucial for navigating complex loss landscapes—it smooths out fluctuations and ensures more stable updates. In contrast, linear averaging treats all past gradients the same, which might seem fair, but in reality, it causes the optimizer to react more slowly to new trends. This slower response can result in sluggish convergence or even getting stuck in suboptimal areas, because older, less relevant gradients end up diluting the impact of newer, more valuable ones. So, in general, EMA works better because it strikes a good balance between stability and adaptability, while linear averaging lacks that dynamic edge.

5. PyTorch requires the loss to be a scalar because backpropagation is designed to work with a single scalar output. It uses the chain rule to compute gradients, starting from this scalar loss and working its way backward through the computation graph.

   When the loss is a scalar, everything is straightforward. PyTorch knows how to compute the gradient of this scalar with respect to all model parameters, providing the direction needed to update the weights during training. If the loss were a tensor with more than one element, the gradient would no longer be just a vector of partial derivatives; it would become a matrix. Essentially, we'd be computing partial derivatives for every element in the output tensor with respect to every model parameter, which results in a Jacobian matrix. This would require much more computation, and since we're only aiming to minimize a single objective (the loss), it just seems like overkill.

   The tirgul we saw explained this - `.backward()` needs a scalar to initiate the gradient computation. If the tensor isn't scalar, PyTorch actually throws an error, unless you manually provide a `gradient` argument, which simulates what the gradient of a scalar loss would have been if it depended on that tensor.

6. **a.** Inductive bias in machine learning refers to the set of assumptions or prior knowledge that a learning algorithm uses to generalize beyond the training data. In other words, it is any built-in preference for one hypothesis over another, independent of the observed data. For example, in the cousrse intro to AI, we learned about Occam's Razor, which is an inductive bias which assumes simpler hypotheses are more likely correct.

   b. CNNs are built on the idea that local patterns, like edges and textures, are more important than distant relationships. That's why each neuron connects to just a small patch of the input, or a receptive field. This design helps the network focus on local features and build up more complex representations over layers. We can think of this as a bias towards locality. Another key bias is translation invariance: the same filter is applied across the image, no matter where the feature appears. So, if there's a balloon in

the top-left corner, it's still considered a balloon if it's in the bottom-right corner.

c. Some pros: this built-in structure helps models learn more efficiently and generalize well, especially when data is scarce. It acts as a form of regularization by limiting the space of possible solutions to only those that make sense, which is probably why CNNs work so well on images. Their assumptions line up perfectly with the way visual data is structured. But, the downside is that the wrong kind of bias can be limiting. If the data doesn't fit the assumptions—say, if position really matters and you're using a translation-invariant CNN—the model might underfit or miss key patterns. In cases where you have lots of data and computing power, models with weaker biases (like Transformers) might actually be better, as they can learn structure directly from the data. So, while strong inductive bias is useful when you have limited data and prior knowledge, with enough data, more flexible models might perform better.

7. **a.** Let's look at the attention operation:

$$\text{Softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) \cdot V$$

where $Q, K, V \in R^{n \times d}$, and n is the sequence length, and d is the feature dimension.

Let's break it down, innermost first. Computing $QK^\top$ involves multiplying matrixes, the first with dimensions $n \times d$ and the second with dimensions $d \times n$. This takes $O(n^2 d)$ time. We'll assume the $1/\sqrt{d}$ is $O(1)$ time. The softmax is then applied row-wise over the resulting $n \times n$ matrix and costs $O(n^2) = O(n^2 d)$. Then, multiplying the $n \times n$ attention matrix with $V \in R^{n \times d}$ also costs $O(n^2 d)$.

Altogether we got:

$$O(n^2 d)$$

b. Let's look at Bob's suggestion:

$$\text{Softmax}\left(\frac{Q}{\sqrt{d}}\right) \cdot (K^\top V)$$

We analyze the time complexity of this computation in three steps:

1. Compute $K^\top V \in R^{d \times d}$, which requires multiplying $K^\top \in R^{d \times n}$ with $V \in R^{n \times d}$. This takes $O(nd^2)$ time.

2. Compute the softmax over $Q/\sqrt{d}$. Assuming softmax is applied row-wise over each d-dimensional vector in $Q \in R^{n \times d}$, this costs $O(nd) = O(nd^2)$.

3. Multiply the softmax output $\in R^{n \times d}$ with $K^\top V \in R^{d \times d}$, which requires $O(nd^2)$ time.

So the altogether we get:

$$O(nd^2)$$

This is significantly cheaper than the original attention complexity $O(n^2 d)$ assuming $d \ll n$.

That said, Bob's version doesn't really behave like proper self-attention. In the original formulation, each query vector gets to compare itself to *every* key vector in the sequence, which allows the model to decide what to pay attention to, on a token-by-token basis. That's what makes self-attention so powerful-it's dynamic and context-aware. But in Bob's version, the interaction between keys and values is collapsed into a single fixed $d \times d$ matrix, which is then applied the same way to every token. So while it's definitely more efficient, the model loses the ability to focus on different parts of the sequence depending on the query.

8. **a.** Each pixel in the attention map shows how much the model focused on a specific input word (in English) while generating a specific output word (in French). The rows represent output positions, while the columns represent input positions. A pixel at a given row and column indicates the level of attention the model gave to a particular English word when producing the corresponding French word. Brighter pixels correspond to higher attention weights, while darker pixels show lower attention. This visualization enables us to see how the model dynamically selects the most relevant input context during the generation process.

   b. A row with only one non-zero pixel means the model relied almost entirely on a single English word to generate the corresponding French word. This can happen with words that have a strong one to one mapping, or words that translate directly. For example, names (direct translation) or numbers (strong mapping).

   c. When a row has several non-zero pixels, it means that the model distributed its attention across multiple input words while generating a single word in the output sentence. This can happen when translation depends on the context. For example "break a leg" or "bite the bullet". The model doesn't just look at the meaning of the individual words, but the sentence as a whole. To do that, it gathers information from multiple positions in the input to generate a fluent and meaningful translation.

   d. This happens because of normalization and visualization of attention weights. Attention values are usually normalized using the softmax function, and the sum of each row's weights equals one. When attention is concentrated on a single input word, that word receives a weight close to one and shows up as a white pixel, ie high focus. And when attention is more evenly distributed among several input words, each receives a smaller weight, showing up as lower intensity pixels.

9. **(a)** The reason we can drop the KL-divergence term is because it's always non-negative, i.e. $D_{\mathrm{KL}}(q\|p) \geq 0$. So, even if we don't include it, the

ELBO still gives us a lower bound on the true log-likelihood $\log p_\theta(x_0)$. That means if we optimize the ELBO, we're still pushing that lower bound up, getting closer to the actual likelihood, even without having access to the full expression. Basically, it's a valid and tractable objective we can optimize, unlike the full thing.

(b) We can't compute the KL term because the two distributions involved (especially $p_\theta(x_{1:T} \mid x_0)$) are intractable. This distribution represents what the model believes about the full latent diffusion path given the observed data, but it's too complex to compute directly. It requires integrating over all possible trajectories, which is computationally impossible. So we don't bother trying to calculate it and just focus on the part we can actually work with.

(c) That last KL term, the one between $q(x_T \mid x_0)$ and the prior $p_\theta(x_T)$, is usually ignored in training because it doesn't matter much. The prior is something simple, like a standard Gaussian, and the forward process already makes sure x_T ends up close to that anyway. So optimizing that term doesn't really give us much extra benefit — it's just noise. Dropping it simplifies the training and doesn't hurt performance.

10. **a.** Bob is wrong because a regular autoencoder isn't actually a generative model. The decoder learns how to reconstruct inputs from the specific latent vectors it saw during training, but it wasn't trained to handle random ones. So if you just feed it a random vector, there's no guarantee the output will look like anything meaningful. In other words, the model only knows how to decode latent vectors that came from its own encoder, so even though it seems like you could use the decoder to "generate" new data, that's not really what it was built for.

    b. Alice is definitely onto something. Unlike a standard autoencoder, which maps inputs to fixed points in latent space, a Variational Autoencoder (VAE) maps each input to a distribution, typically a Gaussian. During training, the model samples from that distribution and reconstructs the original input, much like a regular autoencoder. However, the key difference is the addition of a KL-divergence term. This term encourages the latent distributions to stay close to a standard normal distribution. The benefit of this is that, at inference time, when we sample a random vector from the normal distribution, the decoder can generate something meaningful. This is because the VAE was trained under the assumption that any vector drawn from that latent space should correspond to a valid output.