Deep Learning - DRY

1. (a) The UAT states that regardless of the choice of non-linear activation function, it is always possible to construct a neural network with L > 1 layer that can approximate any continuous function up to any specified precision. This means that even with a single hidden layer between the input and output neurons, the network can learn to represent any target function. This supports the claim because, according to the UAT, a sufficiently wide single-layer MLP (with enough neurons) is capable of achieving optimal error (precision of 100%) on any dataset (since the dataset can be represented by a function), provided the network is properly tuned.

(b) His conclusion is wrong because it overlooks the practical efficiency of using a single hidden layer. Approximating complex functions with just one hidden layer may require an extremely large number of neurons, making the network computationally inefficient or even impractical. In contrast, MLPs with multiple hidden layers can approximate the same functions more efficiently by using fewer neurons per layer and distributing the complexity across several layers.

2. (a) The main advantage of using a CNN over an MLP in this case is that CNN exploits the local spatial structure of the images, allowing the learning process to reduce the number of parameters by sharing weights across different regions. This approach, through the use of convolutions, enables CNNs to efficiently extract important features with lower computational demands. Alice's main difficulty is Large number of parameters. The MLP would need to connect every input pixel to every neuron in the next layer, leading to an enormous number of parameters(especially with images). This makes the network difficult to train(and even impossible) and prone to overfitting.

(b) I disagree with Alice's claim. While both MLPs and CNNs apply linear operations followed by non-linear activations, CNNs are specifically designed to exploit the spatial structure of the images. This allows CNNs to learn features more efficiently and effectively than MLPs, and with far fewer parameters. CNNs use the same filters (weights) across different regions of the input, which drastically reduces the number of parameters and improves generalization. It's similar to an MLP, but with many weights being either shared or set to zero. Additionally, convolutional layers apply kernels to the input to extract relevant features, while also reducing the spatial dimensions of the data.

3. Yes, using momentum can still help the optimization process, even when the loss function is convex (and therefore has a single minimum value) because

- Momentum can help to accelerate the convergence to the minimum value and speed up the optimization process, by allowing the updates to move more quickly in the direction of the optimal solution. So, It reduces the number of iterations needed to reach the minimum and also Improving Efficiency.
- Momentum can help to reduce oscillations and smooth out these oscillations (Because it Smooths out weight updates), leading to faster convergence and more stable updates.

4. Backpropagation essentially combines the forward pass and AD on the computational graph. In reverse-mode AD, we trace the computation backwards to calculate the gradients with respect to each parameter. The reason PyTorch demands the loss tensor to be a scalar in order to perform backpropagation is that backpropagation (using the chain rule) needs a single value to compute how each parameter contributes to the loss, because in order to take a gradient, the derivative must be with respect to a single value. If the output were not scalar, PyTorch would need to store gradients for every output(then the Jacobian-vector product would be computed), which is computationally expensive and impractical. By demanding the output to a scalar, PyTorch simplifies the gradient computation, allowing it to efficiently compute the gradient of each parameter with respect to that loss.

5. First, we can see that the kernel size is 32X32, which means the convolution will cover the entire image in one step. Also, the size of y_hat is [1,1,1](scalar). So, when we compute y_hat - y, we use the broadcasting mechanism. we can see $y\_hat = \sum_{i,j} W_{i,j} X_{i,j}$, where W is the Kernel and X is the input. $loss = \sum_{i,j} (y\_hat - y_{i,j})^2$.

$$\frac{\partial loss}{\partial W_{i,j}} = \frac{\partial loss}{\partial y\_hat} * \frac{\partial y\_hat}{\partial W_{i,j}} = 2 * (y\_hat - y_{i,j}) * \frac{\partial y\_hat}{\partial W_{i,j}} = 2 * (y\_hat - y_{i,j}) * X_{i,j}$$

6.

(a) No, it's not directly possible to add positional embeddings in the same way they are used in transformers when using an RNN model. In transformers, positional embeddings are added to the input sequence to provide the model with information about the position of each token in the sequence. This is crucial for transformers, as they process the input sequence in parallel and do not have an inherent notion of sequence order. RNNs, on the other hand, process the input sequence sequentially, one token at a time. This means that the RNN already has an implicit understanding of the sequence order. As a result, adding positional embeddings to an RNN would be redundant and could potentially introduce noise into the model.

(b) While RNNs inherently understand sequence order, adding positional embeddings can still be beneficial for long sequences or tasks where explicit positional information is crucial. This can be achieved by directly concatenating positional embeddings with the input or modifying the RNN architecture to incorporate attention mechanisms, positional encoding, or using specialized units like GRUs or LSTMs. The choice of whether to use positional embeddings and which adaptation to choose depends on the specific task and the desired trade-off between computational cost and performance.

7.

(a) Each pixel in the attention map represents the attention score between a specific word in the source sequence and a word in the target sequence. In a translation task, for example, it shows how much importance the model assigns to a particular word in the source sentence when predicting a word in the target sentence.

(b) A row with only one non-zero pixel indicates that the model is focusing all of its attention on a single word in the source sequence when predicting the corresponding target word. This suggests a strong one-to-one mapping between the source and target words for that prediction.

(c) Rows with several non-zero pixels indicate that the model is distributing its attention across multiple words in the source sequence. This happens when the model needs to attend to several words in the source sentence to predict a target word, which is common in complex language structures or when word order differs significantly between languages.

(d) Rows with only one non-zero pixel have white pixels because the attention is highly concentrated on a single word, indicating maximum attention, which is represented by a high score. The other rows have gray pixels because the attention is spread across multiple words, leading to lower attention scores for each word, hence the gray shading.

8.

The basic GAN employs a generator and a discriminator, with the generator creating fake data and the discriminator distinguishing between real and fake. However, this adversarial setup often suffers from instability, like mode collapse. WGAN addresses these issues by using the Wasserstein distance, a more stable metric for measuring the difference between distributions. By enforcing a Lipschitz constraint using techniques like weight clipping or gradient penalty, WGAN ensures smoother gradients and prevents instability, leading to improved training stability and higher-quality generated samples.

9.

(a) The mathematical basis for ignoring the KL-divergence term lies in the variational inference framework. In variational inference, we approximate a complex posterior distribution $p_\theta(x_{1:T}|x_0)$ with a simpler variational distribution $q(x_{1:T}|x_0)$. The goal is to minimize the KL divergence between these two distributions. However, since we cannot directly compute the posterior distribution, we instead maximize the Evidence Lower Bound (ELBO).

The ELBO is defined as: $ELBO(\theta) = E_q[log(\frac{p_\theta(x_0:T)}{q(x_{1:T}|x_0)})]$. By maximizing the ELBO, we indirectly minimize the KL divergence. The KL-divergence term itself is a constant with respect to the variational parameters. This means that optimizing the ELBO is equivalent to minimizing the KL divergence. Therefore, we can safely ignore the KL-divergence term during training, as it does not affect the optimization process.

(b) We cannot directly compute the KL-divergence term because it involves the true posterior distribution $p_\theta(x_{1:T}|x_0)$, which is often intractable. In many cases, the exact posterior is too complex to calculate. The variational inference framework aims to address this issue by approximating the intractable posterior with a simpler variational distribution $q(x_{1:T}|x_0)$. By maximizing the ELBO, we indirectly minimize the KL divergence between the approximate and true posteriors, even without explicitly calculating the KL-divergence term.

(c) The term $-D_{KL}(q(x_T|x_0) \,||\, p_0(x_T))$ is ignored during training because it is constant with respect to the learnable parameters of the model. In the variational inference framework, the goal is to optimize the variational parameters by minimizing the KL divergence between the approximate posterior $q(x_{1:T}|x_0)$ and the true posterior $p_\theta(x_{1:T}|x_0)$, which is equivalent to maximizing the ELBO. Since the term $-D_{KL}(q(x_T|x_0) \,||\, p_0(x_T))$ does not involve any trainable parameters and only depends on the fixed prior and data, it provides no useful gradient information during optimization. As a result, we can safely ignore it without affecting the optimization process, allowing the remaining terms in the ELBO to guide the training towards a better approximation of the true posterior.

10. (a) Vanishing Gradients - the gradients used to update the weights during backpropagation become very small (close to zero and even become zero) as they are propagated back through the layers of the network during training. This can effectively stop the weights from changing, leading to little or no learning in the early layers of the network.
Exploding Gradients - the gradients used to update the weights during backpropagation become excessively large as they are propagated back through the layers of the network during training.This large gradients cause updates to the model's weights to be excessively high (even become NaN) , destabilizing the learning process, and causing the network to fail to converge or diverge entirely.

(b) Example to Vanishing Gradients:
activation function: $\sigma(x) = \frac{1}{1+e^{-x}}, \sigma'(x) = \sigma(x) * (1 - \sigma(x))$
Suppose we have 6 layers, with weights of 0.1 and input of 1.
$Layer\ 1 : \sigma(0.1 * 1) \approx 0.525, Layer\ 2 : \sigma(0.1 * 0.525) \approx 0.513$
$Layer\ 3 : \sigma(0.1 * 513) \approx 0.5128, Layer\ 4 : \sigma(0.1 * 0.5128) \approx 0.5128$
$Layer\ 5 : \sigma(0.1 * 0.5128) \approx 0.5128, Layer\ 6 : \sigma(0.1 * 0.5128) \approx 0.5128$
Backpropagation:
$Layer\ 6 : \sigma'(0.5128) \approx 0.25, Layer\ 5 : \sigma'(z_6) * 0.25 \approx 0.0624$
$Layer\ 4 : \sigma'(z_5) * 0.0624 \approx 0.0156, Layer\ 3 : \sigma'(z_4) * 0.0156 \approx 3.9x10^{-3},$
$Layer\ 2 : \sigma'(z_3) * 3.9x10^{-3} \approx 9.75x10^{-4}, Layer\ 1 : \sigma'(z_2) * 9.75x10^{-4} \approx 2.44x10^{-4}$
As we can See, At each layer, the gradient shrinks as it propagates backward. By the time we reach the first layer, the gradient is very small . if we take the same network with 20 layers, the gradient of the first layer will be almost zero(~10x-10).
Example to Exploding Gradients:
activation function: $ReLU(x) = max(0, x)$
Suppose we have 20 layers, with weights of 5 and input of 1.
$Layer\ 1 : ReLU(5) = 5, Layer\ 2 : ReLU(5 * 5) = 25$
$Layer\ 3 : ReLU(5 * 25) = 125, Layer\ 4 : ReLU(125 * 5) = 625 \ \ldots$
$Layer\ 10 : ReLU(5 * 5^9) = 5^{10}$
Backpropagation:
we will get that $\frac{\partial L}{\partial x} = 5^{10}$

(c) MLP- Vanishing Gradients - Use ReLU as the activation function in place of sigmoid or tanh. Unlike sigmoid and tanh, which compress their output values towards the extremes (near 0 or 1), ReLU remains unsaturated for positive inputs. This property enables gradients

to pass through the network more easily during backpropagation, decreasing the chances of them shrinking excessively.
CNN - Use Batch normalization. This helps maintain stable gradients during training. By controlling the activation values and preventing them from becoming too large or too small, it reduces the risk of exploding or vanishing gradients in deep CNNs networks.
RNN - Exploding Gradients - Gradient clipping limits the maximum norm value of gradients during backpropagation to a pre-defined number, preventing them from growing too large and causing instability. If the gradient gets too large, we rescale it to keep it small. Gradient clipping ensures the gradient vector has norm at most c, preventing the Exploding Gradients issue.