1. 1. The Universal Approximation Theorem states that a feedforward neural network with a single hidden layer containing a sufficient number of neurons can approximate any continuous function to an arbitrary degree of accuracy, given enough training data. The UAT implies that the expressiveness of a single hidden layer is enough to model any function, regardless of the task's complexity. This supports the idea that, in theory, more than one hidden layer is unnecessary for achieving optimal performance, as a single hidden layer with enough neurons can represent the same functions that deeper networks can.
   2. While the UAT asserts that a single hidden layer is theoretically sufficient, it does not address the practical aspects. A single hidden layer may require an exponentially large number of neurons to approximate complex functions, making the network inefficient and difficult to train. Deeper networks are also more practical because they can learn complex features in a layered fashion, where each layer captures different levels of abstraction, which allows them to generalize better and solve more complex tasks more efficiently.
   3. The main advantage of a CNN is its ability to exploit the spatial structure of images. CNNs use convolutional layers to automatically learn spatial hierarchies of features, such as edges, textures, and object parts. Kernels capture local patterns in images, making CNNs very effective in recognizing objects regardless of their position within the image. Additionally, the weight sharing in convolutional layers reduces the number of parameters, making CNNs more efficient for large, high-dimensional inputs like images.

In contrast, an MLP ignores the spatial relationships between neighboring pixels. This leads to two main difficulties. First, MLPs require a significantly larger number of parameters to capture the same level of information, since they do not share weights across spatial regions, which results in increased computational costs and higher memory usage. Second, MLPs cannot inherently recognize that the same object appears in different locations or orientations within the image without additional mechanisms.

* 1. I disagree with Alice’s reasoning. An explained in section a, in CNNs convolutional layers use local receptive fields to capture spatial features and employ weight sharing, which reduces parameters and detects patterns across the image. CNNs also have spatial invariance, recognizing objects regardless of their position. MLPs on the other hand, flatten the input, ignoring pixel relationships and leading to a larger number of parameters. As a result, they cannot efficiently capture the spatial patterns necessary for image classification.

1. An important benefit of using momentum is to avoid local minima. The momentum continues to affect the optimization process after reaching a minimum, and if the local minimum is “shallow” enough, the process can escape it and reach a lower minimum.

In the case of a single minimum value there are no local minima to avoid and the process will always lead to the global minimum, so in that sense using momentum would not help the process reach a better result.

However, momentum could help speed up the process. By gradually increasing the friction coefficient, the momentum would help the process reach the minimum value faster and the momentum’s effect would decrease over time, letting the process settle at the minimum point.

1. PyTorch requires the loss tensor to be a scalar for backpropagation because a scalar loss produces well-defined gradients with respect to each parameter. If the loss were a vector, it would be unclear how to compute and propagate gradients through the network, as there would be no clear way to reduce or aggregate multiple loss values into a single objective for optimization
2. Firstly, we need to understand what the convolutional layer does. The layer is given an image of size 32\*32 with one channel and processes it with a kernel of size 32\*32, with dilation=1 and padding=0. This means the kernel examines the image in only one orientation, covering all of it. So y\_hat, the output of the layer, will be a scalar with a value of: , where ki,j is the kernel’s value in row i and column j, imi,j is the value of the corresponding pixel, wi,j is the weight of the corresponding kernel value, and bi,j is the corresponding bias.

When the loss is calculated, broadcasting is used to calculate “y\_hat-y”, as “y\_hat” is a scalar and “y” is a tensor. y\_hat is viewed as a tensor of size 1\*32\*32, where the original scalar value is copied to every element of the tensor. Thus, the formula for the loss is:

The learnable parameters of the model are the weights and the biases.

Given , the gradient of y\_hat w.r.t to wi,j is . So the gradient of for any is . And the gradient of the loss, being a derivative of a sum and thus a sum of derivatives will be:

.

Given , the gradient of y\_hat w.r.t to bi,j is 1. So the gradient of for any is . And the gradient of the loss will be: .

* 1. Adding positional embedding to an RNN model is possible, although the process is different depending on the model.

Generating the positional encoding can be done in the same way as was learned in the tutorial, based on sine and cosine functions with different frequencies. The difference lies in where we add the embedding.

In a Multiple-Single model, the embedding could be added to the input in the regular way since we have a sequence of multiple inputs.

In a Single-Multiple model however, like for example image captioning, there is a single input and multiple outputs so adding positional embedding to the input is impossible. However adding the embedding to the each iteration of the hidden state is possible and logical, since it can be thought of as the “input” that is used to generate each new word.

* 1. Adding positional embedding to RNN models would most likely be unnecessary, because the input of RNN models is already implicitly encoded in the model by the hidden state.

Each iteration of the network (either when processing input or generating output) will provide different results based on the position of the input/output because the hidden state contains information from previous steps and thus also, implicitly, the position.

* 1. Each pixel in the map represents how much the output word (row) focuses on the input word (column). This shows the how strongly each input word affects each output word.
  2. A row with only one non-zero pixel means that the output word of that row is affected by only one input word. For example, in the sentence given in the question, the word “agreement” is translated into “accord” without any consideration of the rest of the input.
  3. A row with more than one non-zero pixels means that the output word of that row was affected by more than one input word. For example, in the sentence given in the question, the row of the word “la” has two non-zero pixels: “the” which is the meaning of the word, and “area” which determined the gender of the word.
  4. The brightness of the pixels represents the importance of the input word in the translation to the output word. A fully white pixel signifies a one to one translation, meaning other input words are not involved in the translation. A row that has more than one non-zero pixel means the output word is dependent on more than one input word, and so no single word has 100% importance.

1. The difference between a basic GAN and a WGAN is the loss functions they use. WGANs use the Wasserstein distance and basic GANs use Jensen-Shannon divergence.

WGANs can provide more stable training and prevent vanishing gradients because Wasserstein distance can still provide a gradient when the distributions of the discriminator and the generator have no overlap.

WGANs are also less prone to mode collapse. By minimizing the Wasserstein distance, the generator is encouraged to explore more of the target distribution, producing a wider variety of samples.

10. a. Vanishing gradients and exploding gradients refer to issues that can arise during the training of deep neural networks, particularly when using gradient-based optimization methods like backpropagation.

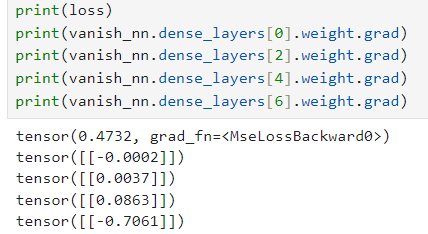
Vanishing gradients occur when the gradients of the loss function w.r.t the parameters shrink as they are propagated back through the layers of the network. This causes the weights in the earlier layers to update very slowly, leading to poor learning in those layers.

The opposite, exploding gradients, occur when the gradients become too large during backpropagation. This can cause the weights to undergo very large updates, leading to instability.

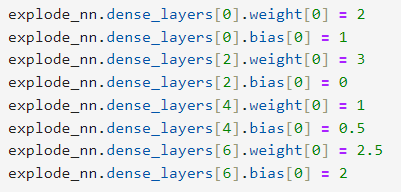
b. To demonstrate vanishing gradients we created a network consisting of 4 linear layers, each with 1 neuron, and a sigmoid activation function between each two layers. An input tensor [1] was used, and a random goal to compare the output to.

With a loss of 0.4732, the gradients of the linear layers were:

* Layer 4: -0.7061
* Layer 3: 0.0863
* Layer 2: 0.0037
* Layer 1: -0.0002

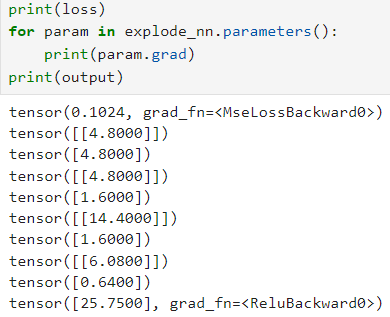


To demonstrate exploding gradients, we will use a simple example of a network with 4 linear layers, each with 1 neuron, and a ReLU activation function. We will manually set the network’s weights and biases to be too large:



This will cause the gradients calculated during backpropagation to be far too large for the model to converge.

Even after adjusting the expected output to receive a loss of only 0.1, we can see the gradients of the weights and biases exploded:



c.

* MLP: For MLPs we can use batch normalization to mitigate vanishing or exploding gradients. The idea of batch normalization is to use the differentiability of the shifting and rescaling operations to make the network learn how to normalize the data. Batch normalization reduces internal covariate shift, which helps stabilize and maintain the gradients.
* CNN: For CNNs we can use residual connections to mitigate vanishing or exploding gradients. Residual connections allow gradients to flow directly through the network without being diminished or amplified by deep layers.
* RNN: One simple and effective way to prevent exploding gradients in RNNs is gradient clipping. The basic principle is to shrink any gradient that becomes too large. Mathematically, for any layer L with a gradient matrix G, if the norm of G is larger than a certain threshold, , we perform the operation .