Dry part

CS236781:Deep learning

Submitting:

316353176
313283657

(1)

(a)

The Universal Approximation Theorem (UAT) states that given enough parameters, a multi-layer perceptron (MLP) with one hidden layer and a non-linear activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary precision.
More formally, for a continuous function $f: K \to \mathbb{R}^m \; where \; K \subseteq \mathbb{R}^n \; is \; compact$ there exists a function of the form $C\big(\sigma \circ (A \cdot x + b)\big)$ that can approximate f to any given precision, where $k \in \mathbb{N}, A \in \mathbb{R}^{kxn}, b \in \mathbb{R}^k, C \in \mathbb{R}^{mxk}$ are parameters that adjust to the required level of precision and σ nonlinear activation function.
This theorem supports the claim that you can achieve optimal error over any dataset and loss criterion. If there exists an optimal function f^* that minimizes the loss on a given dataset, the UAT guarantees that there exists a single hidden layer MLP that can approximate $f^*$to any desired precision. This means that such an MLP can approximate
f^* closely enough to behave in the same way on the dataset. As a result, using an MLP with one hidden layer provides a potential pathway to achieving the minimal loss.

*If the loss function does not have a finite minimum, we can refer to f^* as the function that achieves the desired level of loss.

(b)

The conclusion that one should "never use more than one hidden layer" is incorrect because the Universal Approximation Theorem (UAT) is primarily an **existence theorem**. While it guarantees that a single-hidden-layer neural network can approximate any continuous function on a compact domain, it does not offer practical guidance on how to construct or train such a network for real-world applications. Here are several reasons why using more than one hidden layer is often necessary:

- **Existence vs. Practical Implementation**: The UAT is an **existence theorem**. It guarantees that a single-hidden-layer network **can** approximate any continuous function given the right parameters and enough neurons. However, it doesn't provide a practical method for constructing such a network or specify how to find the optimal weights and biases needed for the approximation.
- **Unknown and Potentially Large Number of Neurons**: The theorem does not indicate how many neurons are required to achieve the desired level of precision. In practice, approximating complex functions might require an extremely large number of neurons in the hidden layer. This makes the network **impractically wide**, leading to high computational costs, increased training time, and greater resource consumption.
- **Optimization Challenges** The optimization process can get stuck in local minima, saddle points, or flat regions in the error landscape, preventing the network from finding the optimal solution and achieving the desired approximation.
- **Hierarchical Feature Learning**: Deep neural networks with multiple hidden layers are capable of learning **hierarchical representations** of data. Early layers capture low-level

features, while subsequent layers build upon them to recognize more complex patterns. This layered learning is crucial for tasks involving high-dimensional data with intricate structures, such as image and speech recognition. Single-hidden-layer networks lack the depth to model these hierarchical and compositional patterns effectively.

- **Empirical Success of Deep Networks**: Empirically, deep networks have been found to outperform shallow networks across various domains. They tend to **generalize better** to unseen data and achieve higher performance metrics. This success is attributed to their ability to model complex, non-linear relationships more effectively than shallow networks, even those with a large number of neurons in a single hidden layer.

(2)

(a)

Bob can convince Alice by explaining that the main advantage of using a Convolutional Neural Network (CNN) over a Multilayer Perceptron (MLP) for image classification is that CNNs are specifically designed to exploit the spatial structure of images. CNNs utilize local connectivity and weight sharing to effectively capture spatial hierarchies and local patterns such as edges, textures, and shapes, which are crucial for understanding and classifying images.

Main Difficulties Alice Might Encounter If She Uses an MLP:

- **Loss of Spatial Relationships**: MLPs require images to be flattened into one-dimensional vectors, which destroys the two-dimensional spatial arrangement of pixels. This loss of spatial information makes it challenging for the network to learn and recognize patterns that depend on the relative positions of pixels, leading to poorer performance in image classification tasks.
- **High Computational Complexity and Number of Parameters**: flattening high-resolution images results in extremely large input vectors. An MLP would need to connect each input pixel to neurons in the next layer, leading to a massive number of weights and biases. This not only increases computational requirements and memory usage but also makes the network more difficult to train effectively.
- **Risk of Overfitting**: with such a large number of parameters, the MLP is more prone to overfitting the training data. It may learn to memorize the training images rather than generalizing to new, unseen images, resulting in poor performance on validation or test datasets.

(b)

I disagree with Alice's conclusion. While it's true that the convolution operation is linear, the crucial difference lies in how these linear operations are structured and utilized within the network architectures, which significantly impacts their capabilities. As mentioned earlier, CNNs are designed to **preserve the spatial structure of images** through local connectivity and weight sharing. This architectural design enables CNNs to efficiently learn spatial hierarchies and patterns essential for image classification, such as edges and textures, regardless of their position in the image.

In contrast, MLPs require images to be flattened into one-dimensional vectors, which destroys the spatial relationships between pixels. This loss of spatial information prevents MLPs from effectively learning patterns that depend on pixel proximity. Additionally, MLPs do not employ

weight sharing, resulting in a much larger number of parameters compared to CNNs, which can lead to increased computational complexity and a higher risk of overfitting.
Therefore, even though both CNNs and MLPs involve linear operations between activations, the **architectural differences**, specifically, how these linear operations are applied, make CNNs far more effective for image classification tasks. CNNs can capture and utilize spatial features in images that MLPs cannot, due to their ability to maintain spatial hierarchies and apply learned features across different parts of the image. This fundamental difference means that CNNs and MLPs are not equivalent in practice for image classification, despite the linearity of their operations.
It is theoretically possible for sparse MLP layers to exhibit behavior like CNNs, since convolution matrices are indeed a subset of all possible matrices. However, it is highly unlikely for this behavior to emerge naturally during the optimization process.

(3)

Yes, using momentum can help the optimization process when dealing with a convex loss function that has a single global minimum. momentum accelerates convergence by incorporating an exponential moving average of past gradients into the current update, adjusting parameters based on both current and previous gradients.
In convex optimization, where the negative gradient consistently points toward the global minimum, this accumulation of gradients leads to faster and more direct movement toward the minimum.
Momentum also smooths out oscillations that may occur when gradients change direction between steps, especially if the learning rate is large, resulting in a more stable optimization process. For example, in steepest descent methods applied to least squares problems, gradients can become nearly perpendicular between iterations, causing the optimizer to zigzag and slow down.
Introducing momentum helps combat this issue by accumulating the direction of previous gradients, allowing the optimizer to maintain a more consistent and directed path toward the global minimum.
However, it’s important to note that momentum is not always superior to standard gradient descent. If the learning rate or momentum factor is not well-tuned, it can lead to overshooting or slower convergence.

(4)
The loss function is a scalar value that summarizes the performance of the model. When training a model, we usually want to minimize this scalar loss, which measures how well the model is doing (e.g., mean squared error, cross-entropy loss). The goal of calling .backward() is to compute the gradient of the loss with respect to the model's parameters.
The gradient of a scalar with respect to a vector of parameters is well-defined: it's a vector (or matrix for multiple layers).
However, if the output is not a scalar (i.e., it's a vector or matrix), then technically, the gradient would be a tensor of higher dimensions (i.e., you'd get a Jacobian matrix rather than just a

gradient vector). This would involve calculating and storing the gradient for every output with respect to every parameter, which would be much more complex and expensive.
So, when the output is scalar, you calculate a gradient (a vector), but if the output were a vector, you'd have to calculate a much more complex and expensive structure (i.e. Jacobian matrix), PyTorch simplifies this by requiring a scalar output when calling backward().

(5)

Given single channel image with 32×32 pixels denote as x, a model consists from 2D convolutional layer with input and output layer equal to 1 and kernel 32×32.
For each input pixel $x_{i,j}$at position (i,j) in the image, the convolution layer applies a set of learnable filters to compute an output $\hat{y}_{i,j}$
Since the **kernel size** is 32×32 , the filter covers the **entire input image** in a single step, therefore the output $\hat{y}$ is a scalar (size of 1×1).
Let:
- The kernel $W \in \mathbb{R}^{32,32}$ be:

$$W = \begin{bmatrix} w_{1,1} & \cdots & w_{1,2} \\ \vdots & \ddots & \vdots \\ w_{32,1} & \cdots & w_{32,32} \end{bmatrix}$$

- The input $X \in \mathbb{R}^{32,32}$ be:

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,2} \\ \vdots & \ddots & \vdots \\ x_{32,1} & \cdots & x_{32,32} \end{bmatrix}$$

- The bias is a scalar $b \in \mathbb{R}$.
The model output is given by:

$$\hat{y} = \sum_{i=1}^{32} \sum_{j=1}^{32} w_{ij} x_{ij} + b \in \mathbb{R}.$$

Since the loss computes the difference between $\hat{y}$ an y, and $\hat{y}$ is a scalar while y is a 32x32 matrix, broadcasting is applied. Broadcasting repeats the scalar $\hat{y}$ over all cells of the matrix y during the subtraction.
The loss term is therefore:

$$L = \sum_{k=1}^{32} \sum_{l=1}^{32} (\hat{y} - y_{k,l})^2$$

Using the chain rule :

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_{ij}} = \left(\sum_{k=1}^{32} \sum_{l=1}^{32} 2(\hat{y} - y_{k,l}) \right) \cdot x_{ij} = \sum_{k=1}^{32} \sum_{l=1}^{32} 2x_{ij}(\hat{y} - y_{k,l})$$

$$= 2x_{ij} \left(\hat{y} \sum_{k=1}^{32} \sum_{l=1}^{32} 1 - \sum_{k=1}^{32} \sum_{l=1}^{32} y_{k,l} \right) = 2x_{ij} \left(32^2 \hat{y} - \sum_{k=1}^{32} \sum_{l=1}^{32} y_{k,l} \right)$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = \left(\sum_{k=1}^{32} \sum_{l=1}^{32} 2(\hat{y} - y_{k,l}) \right) \cdot 1 = \sum_{k=1}^{32} \sum_{l=1}^{32} 2(\hat{y} - y_{k,l})$$
$$= 2\left(\hat{y} \sum_{k=1}^{32} \sum_{l=1}^{32} 1 - \sum_{k=1}^{32} \sum_{l=1}^{32} y_{k,l} \right) = 2\left(32^2 \hat{y} - \sum_{k=1}^{32} \sum_{l=1}^{32} y_{k,l} \right)$$

(6)

(a + b)

Yes, it is possible to add positional embeddings when using a Recurrent Neural Network (RNN) model.

The idea behind positional encoding is to add a vector to each token's embedding that represent its position in the sequence.

In the context of RNNs, you can augment the input embeddings with positional embeddings before feeding them into the RNN layers, this means that for each time step t instead of feeding only the input x_t, you feed $x_t + p_t$, where p_t is the positional embedding corresponding to position t.

However, while it is possible to add positional embeddings to an RNN, it is generally not necessary because RNNs naturally incorporate positional information due to their sequential processing of tokens.

RNNs read the input sequence one token at a time, and the hidden state at each time step inherently depends on all the previous tokens and their positions in the sequence. This sequential processing ensures that the model "knows" the position of each token based on when it is processed, making explicit positional embeddings redundant. However, if desired, positional embeddings could still be added to the input before feeding it into the RNN, though this would likely offer limited benefit.

Note: The Transformer processes input sequences in parallel, meaning that it has no inherent notion of the order of tokens. To give the model a sense of order, positional encoding is added to the token embeddings.

7

(a)

Each pixel in the map represents the attention weight between a word in the source sequence (English) and a word in the target sequence (French). The attention weight shows how much influence the English token has when creating the context-based representation for the French token. The attention weight is based on the similarity between the query (target token) and the keys (source tokens). A higher weight means the English token played a more significant role in forming the representation of the French token. The color of the pixel represents the strength of attention: lighter colors indicate stronger relationship (higher attention weights), while darker colors indicate weaker relationship.

(b)

Rows with only one non-zero pixel mean that the attention for that particular French word is focused entirely on a single English word. This typically occurs when the model identifies a

direct, straightforward translation between the French and English tokens. In this case, the model does not need to distribute attention across multiple words and has clearly identified that the English token is the most relevant for translating the French token. As shown in the image, these single-focus attention rows often correspond to one-to-one word translations between the two languages.

(c)

Rows with several non-zero pixels mean that the attention for that particular French word is distributed across multiple English tokens. This indicates that the translation of the word is not a straightforward one-to-one relationship, or that more context is needed to understand the French token. For example, in the case of "L'", the attention is distributed between the matching word in English ("the") and the objects it refers to, such as "agreement" and "signed". This shows that the model is using information from multiple English words to form the correct French translation.

(d)

The attention weights for each French token must sum to 1 (the weights are normalized during calculation using SoftMax). When a row has only one non-zero pixel, that single token receives all the attention (with a weight of 1), which is why the pixel appears white, indicating maximum attention. In contrast, when the attention is distributed across multiple tokens in the source sequence, each token is assigned a fraction of the total attention, with individual weights less than 1. Since the attention is divided, these pixels will have lower values and therefore appear as darker pixels (gray) rather than white.

(8)

A Generative Adversarial Network (GAN) is a neural network architecture with two parts: a generator that creates fake data and a discriminator that distinguishes between real and fake data. GANs are powerful for generating high-quality, realistic outputs, making them useful for tasks like image generation and data augmentation.

The main difference between a basic GAN and WGAN lies in the loss function and the output of the discriminator.

In regular GAN- the discriminator outputs a probability (between 0 and 1) of whether an input is real or fake.

The objective function of GAN is formulated as:

$$\min_G \max_D \mathbb{E}_{x\sim P_{real}} \log D(x) + \mathbb{E}_{z\sim P_z}[\log\left(1 - D\big(G(z)\big)\right)]$$

The discriminator maximizes its ability to distinguish between real and fake data.

The generator minimizes the discriminator's ability to distinguish the fake data from real data.

In WGAN, the discriminator is called a **critic** and outputs a real-valued score instead of a probability.

The objective function of WGAN is formulated as:

$$\min_G \max_{C\in Lip_1} \mathbb{E}_{x\sim P_{real}} C(x) - \mathbb{E}_{z\sim P_z}[\mathrm{C(G(z))}]$$

The critic tries to maximize the difference between these scores to approximate the Wasserstein distance between the real and generated data distributions.

The generator then minimizes this estimated distance, gradually improving the quality of the generated samples so that the generated distribution will be closer to the real distribution.
The main advantages of WGAN are:
GANs can be challenging to train, often facing instability and issues like mode collapse, where the generator produces limited variations of data. Wasserstein GAN (WGAN) improves upon basic GANs by replacing the loss with the Wasserstein distance, offering smoother gradients and more stable training.
In addition, WGAN often requires fewer training tricks and adjustments, making the training process easier and more robust.

9.

(a)

We saw that $\log p_\theta(x_0) = KL + ELBO$, meaning the log-likelihood of p_θ is constructed from a KL-divergence term and an ELBO term. The KL divergence is always non-negative. Therefore, we can conclude that $\log p_\theta(x_0) \geq ELBO$. This means the ELBO serves as a lower bound for $\log p_\theta(x_0)$.
Since our goal is to maximize $\log p_\theta(x_0)$ (in line with the maximum likelihood estimation approach), if we maximize the ELBO term, we also maximize what we want to optimize: the likelihood.

(b) To compute the KL-divergence between $q(x_1, x_2, \dots x_T|x_0)$ and $p_\theta(x_1, x_2, \dots x_T|x_0)$, we can either estimate it empirically (by sampling from the distributions) or use an analytical solution if the distributions are explicitly defined. In this case, we don't have an exact form for these distributions.

While we could attempt to approximate $q(x_1, x_2, \dots x_T|x_0)$ using Markov rules, this would require estimating intermediate values $x_i \; for \; 1 \leq t \leq T$, which is generally avoided during training. Instead, we typically compute $q(x_T|x_0)$ directly, as it's more practical.
On the other hand, $p_\theta(x_{t-1}|x_t)$ is estimated through a neural network. This is hard to compute since p_θ models the reverse process $x_T, x_{T-1} \dots ., x_0$.
Even if we use formulas like $p_\theta(x_1, x_2, \dots x_T|x_0) = \frac{p_\theta(x_0, x_1, \dots x_T)}{p_\theta(x_0)}$ we will still stuck with problematic terms we don't have like $p_\theta(x_0)$ (which is essentially what we try to compute).
Moreover, even if we had these distributions, computing the expectation required for the KL divergence itself can be difficult. The expectation involves integrating over all possible sequences, which can be computationally expensive or intractable, particularly when working with high-dimensional distributions.

(c) We ignore this term because it is constant with respect to the learned parameters θ. Since $q(x_T|x_0)$ is not learned and $p_\theta(x_T) \sim N(0, I)$ is a known prior, this KL-divergence does not depend on θ. Therefore, its gradient is zero, meaning it does not affect the optimization process, allowing us to safely ignore it during training.

(10)
(a + b)

Vanishing gradients:

Vanishing gradients refer to a problem in training deep neural networks where the gradients used to update the weights become exceedingly small as they are backpropagated to earlier layers. This causes the network to learn very slowly or stop learning altogether in the initial layers, hindering the ability to capture long-range dependencies.

An example:

Denote MLP network with 3 layers where $x \in \mathbb{R}$:
$MLP(x) = \sigma(w_3(\sigma(w_2(\sigma(w_1 \cdot x))))$
and sample (x_1, y_1).
Denote: $z_2 = \sigma(w_2(\sigma(w_1 \cdot x_1))),\ z_1 = \ \sigma(w_1 \cdot x)\ , \widehat{y_1} = \ MLP(x_1)$

$$\frac{\partial L}{\partial w1} = \frac{\partial L}{\partial \widehat{y_1}} \cdot \frac{\partial \widehat{y_1}}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} = \frac{\partial L}{\partial \widehat{y_1}} \cdot \sigma'(w_3 z_2) \cdot w_3 \ \cdot \sigma'(w_2 z_1) \cdot w_2 \cdot \sigma'(w_1 \cdot x) \cdot x$$

The maximum value of $\sigma'(x)$ is 0.25, which occurs when $\sigma(x) = 0.5$.
Suppose during backpropagation, the gradient of the loss with respect to the output layer is 1.
In the example we gave, we want to compute the gradient of the loss with respect to $w1$.
At each layer, the gradient is multiplied by the derivative of the activation function and the weight connecting to the next layer. For simplicity, assume all weights are 1 and the activation derivatives are 0.25.
Since $\sigma'(x)$ (the derivative of the sigmoid) is at most 0.25, each multiplication of these derivatives reduces the gradient's magnitude.
If the weights and input x are fixed and small, the overall product can become extremely small. This effect compounds as the number of layers increases, particularly affecting early weights like $w1$, leading to vanishing gradients.

Exploding gradients:

Exploding gradients occur when the gradients grow exponentially large during backpropagation. This leads to massive weight updates, causing numerical instability and making the training process erratic or diverging altogether, which prevents the network from learning effectively.

An example:

Denote MLP network with 3 layers where $x \in \mathbb{R}$:
$MLP(x) = \mathrm{RELU}(\mathrm{RELU}(\sigma(\mathrm{RELU}(\sigma(\mathrm{RELU} \cdot x))))$

and sample (x_1, y_1).
Denote: $z_2 = \text{RELU}(w_2(\text{RELU}(w_1 \cdot x_1)),\ z_1 = \ \text{RELU}(w_1 \cdot x)\ , \widehat{y_1} = \ MLP(x_1)$

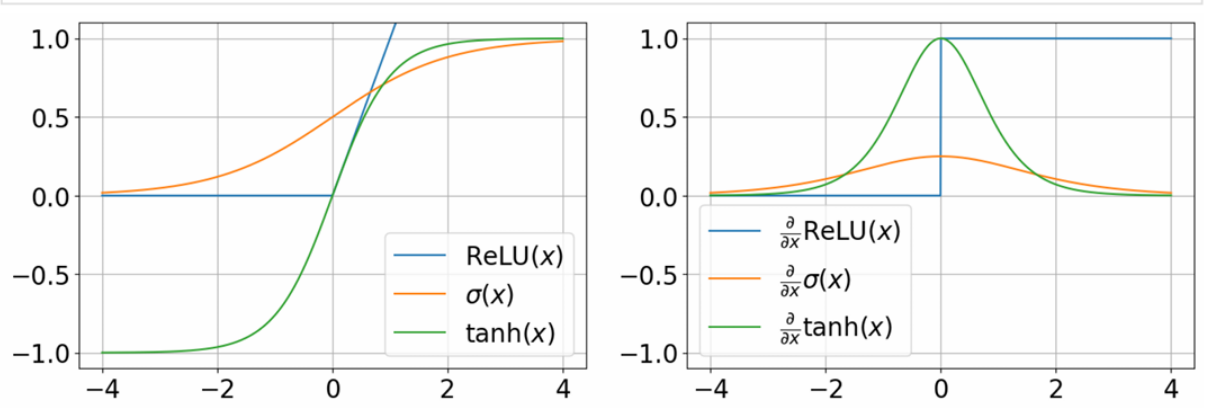
$$\frac{\partial L}{\partial w1} = \frac{\partial L}{\partial \widehat{y_1}} \cdot \frac{\partial \widehat{y_1}}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} = \frac{\partial L}{\partial \widehat{y_1}} \cdot \text{RELU}'(w_3 z_2) \cdot w_3 \ \cdot \sigma'(w_2 z_1) \cdot w_2 \cdot \text{RELU}'(w_1 \cdot x) \cdot x$$

The gradient of RELU when x>0 is 1.
Suppose during backpropagation, the gradient of the loss with respect to the output layer is 1.
In the example we gave, we want to compute the gradient of the loss with respect to $w1$.
At each layer, the gradient is multiplied by the derivative of the activation function and the weight connecting to the next layer. Since the derivative is 1 , if the weights are large, we will get a large gradient as a result of this multiplication.
This may lead to exploding gradients, where the updates to the weights become excessively large. The network becomes unstable, and the weights may change drastically in each iteration.

(c)

Multilayer Perceptron (MLP):

Vanishing and exploding gradients occur during the training of deep networks with many layers, as gradients are backpropagated to update weights, they are calculated by multiplying derivatives across layers. Activation functions like sigmoid or tanh have derivatives less than one, causing gradients to shrink exponentially as they move backward through the network—a phenomenon known as the vanishing gradient problem. This makes it difficult for the network to learn from early layers, hindering overall training. (Conversely, if derivatives are greater than one, gradients can grow exponentially, leading to exploding gradients that cause unstable updates and divergence during training)
To combat vanishing gradients in MLPs, **one distinct technique is using the Rectified Linear Unit (ReLU) activation function**. ReLU outputs the input directly if it's positive and zero otherwise, with a derivative of one for positive inputs. **This property prevents gradients from shrinking exponentially during backpropagation**, ensuring they remain substantial across layers.
Additionally, ReLU does not suffer from vanishing gradients when the input x is far from zero (though the gradient can still be zero when x≤0). It is also much faster to compute than sigmoid and tanh functions due to its simpler mathematical operations, accelerating the training process. Furthermore, ReLU promotes sparse activations because neurons with x≤0 become "dead" and do not activate, leading to sparsity in the activations of the next layer. This sparsity can improve computational efficiency and reduce overfitting by encouraging the network to learn more robust features.



Convolutional Neural Network (CNN):

CNNs, particularly deep ones with many layers, can suffer from vanishing and exploding gradients, as gradients are backpropagated through multiple convolutional and pooling layers, the repeated multiplication of small derivatives can cause them to vanish, slowing down or halting the training of initial layers responsible for detecting basic features.

Exploding gradients can occur due to improper weight initialization or large learning rates, leading to instability and divergence during training.

Additionally, increasing the depth of CNNs complicates the optimization process because the parameter space becomes larger, making it harder for optimization algorithms like stochastic gradient descent to find the optimal solution, even if a better solution theoretically exists.

To address these gradient issues in CNNs, a distinct method is implementing **batch normalization**.

Batch normalization normalizes the inputs of each layer by adjusting and scaling the activations based on the mean and variance of the current mini-batch,

this (stabilization) ensures that the inputs to each layer maintain a consistent scale and distribution, which helps keep the gradients within a suitable range during backpropagation.

By preventing gradients from becoming too small (vanishing) or too large (exploding), batch normalization allows for more stable and efficient training of deep networks.

It enables the use of higher learning rates and reduces sensitivity to initialization, accelerating convergence and improving overall model performance.

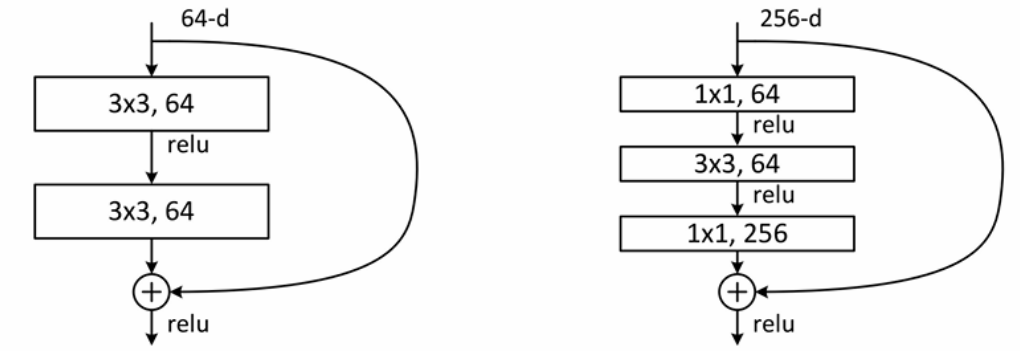
Furthermore, as we saw in the tutorial, another approach to combat vanishing and exploding gradients in CNNs is the use of Residual Networks (ResNets) with **shortcut connections**.

For image-related tasks, deeper networks can learn more complex features, and theoretically, adding more layers should improve or at least maintain the network's accuracy since extra layers could act as identity maps.

ResNets address these issues by building a network architecture composed of convolutional blocks with added shortcut connections, also known as skip connections.

These shortcut connections bypass one or more layers by directly connecting the input of a block to its output, enabling gradients to flow freely backward through the network without diminishing. This mitigates the vanishing gradient problem and simplifies optimization by allowing each residual block to learn a residual mapping (the difference from the identity

function), making it easier for the network to optimize parameters. Together, these techniques facilitate the training of much deeper CNNs that can learn more complex features without suffering from gradient-related issues



Recurrent Neural Network (RNN):

RNNs are particularly prone to vanishing and exploding gradients due to their sequential data processing and the use of backpropagation through time.

As gradients are propagated backward through many time steps, they can exponentially decrease (vanish) or increase (explode) because of repeated multiplication of derivatives. Vanishing gradients hinder the network's ability to learn long-term dependencies, while exploding gradients cause numerical instability and erratic training.

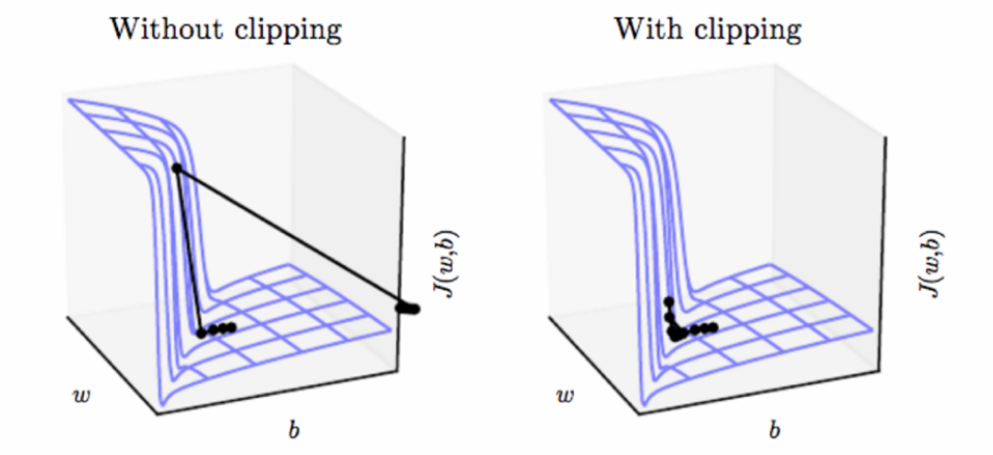
To combat **exploding gradients** in RNNs, a distinct and effective technique is **gradient clipping**. The idea is straightforward yet specifically efficient for preventing exploding gradients (though it doesn't address vanishing gradients).

During backpropagation through time, for each layer L with gradient matrix G, we calculate the norm of the gradients ‖G‖, If this norm exceeds a predefined threshold C, we scale down the gradients to ensure they do not exceed this threshold.

The process is as follows:

- **Compute the Gradient Norm:** Calculate ‖G‖.
- **Check Against Threshold:** If ‖G‖≥C, proceed to clip.
- Scale the Gradients: $G = \frac{G}{||G||} \times C$.

by constraining the gradients within a reasonable range, gradient clipping ensures stable and controlled weight updates, enabling the RNN to learn effectively from long sequences without the instability caused by exploding gradients.



In addition, we saw in tutorials and lectures we can combat gradient problems by using architectures designed to handle long-term dependencies like LSTM and GRU. Both LSTMs and GRUs are designed to keep the gradients flowing, even over long sequences, making them much more resistant to the vanishing gradient problem compared to standard RNNs.