Summer Project — Programming Club

Illusion Craft

Documentation

Aarush Singh Kushwaha— Anirudh Singh— Aayush Jhawar— Hardik Tiwari—
Ikrima Badr Shamim Ahmed— Parnika Mittal— Prithviraj Ghosh

Mentors

Abhijit Singh Jowhari —Abhishek Kumar— Raghav Manglik—Rudradeep Datta

Contents

1 Basics of Machine Learning **5**

1.1 Common Types of Machine Learning . . . 5

1.1.1 Supervised Learning . . . 5

1.1.2 Unsupervised Learning . . . 5

1.2 Supervised Machine Learning Techniques . . . 5

1.2.1 Linear Regression . . . 5

1.2.2 Logistic Regression . . . 6

1.3 Cost Function and Loss Function . . . 6

1.3.1 What is Loss? . . . 6

1.3.2 Difference between Loss and Cost Function . . . 6

1.3.3 Some Common Cost Functions . . . 6

1.4 Back-propagation . . . 7

1.5 Hyperparamters . . . 8

1.6 Gradient Descent . . . 8

1.6.1 Learning rate . . . 8

1.6.2 Different ways of doing Gradient Descent . . . 8

1.7 What Is Underfitting and Overfitting in Machine Learning? . . . 9

1.7.1 Underfitting . . . 9

1.7.2 Overfitting . . . 9

2 Activation functions **11**

2.1 What are Activation functions? . . . 11

2.2 Linear Activation Function . . . 11

2.3 Binary Step Function . . . 12

2.4 Sigmoid Activation Function . . . 12

2.5 Tanh Activation (Hyperbolic Tangent) . . . 13
2.6 ReLU Function . . . 13
2.7 Softmax Activation . . . 14

3 Optimizers **16**
3.1 What is optimisation? . . . 16
3.2 Vanilla Gradient Descent . . . 16
3.2.1 Mathematical formulation . . . 16
3.3 Stochastic Gradient Descent . . . 17
3.3.1 Mathematical formulation . . . 17
3.4 Adam (Adaptive Moment Estimation) . . . 18
3.5 Detailed Steps . . . 19
3.6 Advantages and Disadvantages of Adam Optimization . . . 20
3.7 Implementation of Adam Optimization in Machine Learning Libraries . . . 20
3.7.1 TensorFlow . . . 20
3.7.2 PyTorch . . . 20
3.7.3 Keras . . . 21

4 CNN **22**
4.1 What are CNNs? . . . 22
4.2 Why to use CNNs? . . . 22
4.3 Parts of CNN . . . 22
4.3.1 Input Layers . . . 22
4.3.2 Convolutional Layers . . . 22
4.3.3 Pooling Layers . . . 23
4.3.4 Stride . . . 23
4.3.5 Padding . . . 24
4.3.6 Cov2DTranspose Layer . . . 25
4.3.7 Flatten layer . . . 25
4.3.8 Fully Connected Layers . . . 25

5 DCGANs **26**
5.1 Introduction . . . 26
5.2 Architecture . . . 26
5.2.1 The Generator . . . 26
5.2.2 The Discriminator . . . 26
5.3 Training . . . 27

5.3.1 Discriminator Training . . . 27
5.3.2 Generator Training . . . 28
5.3.3 Loss functions . . . 28
5.4 Tools in GANs . . . 28
5.4.1 KL Divergence . . . 28
5.4.2 Jensen-Shannon Consistency Loss . . . 29

6 Stack-GANs **30**
6.1 Introduction . . . 30
6.2 Stage-I GAN . . . 30
6.2.1 Text Embedding . . . 30
6.2.2 Training Process . . . 30
6.2.3 Reparameterization Trick . . . 31
6.2.4 Model Architecture . . . 31
6.3 Stage-II GAN . . . 32
6.3.1 Training process . . . 32
6.3.2 Differences . . . 33
6.3.3 Model Architecture . . . 33

7 Image Augmentation **34**
7.1 What is Image Augmentation? . . . 34
7.2 Why do we need Image Augmentation? . . . 34
7.3 Classical Techniques for Image Augmentation: . . . 34
7.3.1 Flipping and Rotating: . . . 34
7.3.2 Cropping and Resizing: . . . 34
7.3.3 Color Jittering: . . . 34
7.3.4 Adding Noise: . . . 34
7.3.5 Image Warping: . . . 35
7.3.6 Random Erasing: . . . 35
7.4 Advanced Techniques for Image Augmentation: . . . 35
7.4.1 Cutout . . . 35
7.4.2 Mixup Augmentation . . . 35
7.4.3 Cutmix . . . 36
7.5 Augmix . . . 37

8 Assignments **38**
8.1 Assignment 1 . . . 38

8.1.1 Objective 38
8.1.2 Learning 38
8.1.3 Results 38
8.2 Assignment 2 40
8.2.1 Objective 40
8.2.2 Learning Outcomes 40
8.2.3 Results 40

1 Basics of Machine Learning

1.1 Common Types of Machine Learning

1.1.1 Supervised Learning

In supervised learning, the model is trained using labeled data, where each example includes both input features and the corresponding target values. The objective is to develop a mapping function that can accurately predict the target values for new, unseen data.

1.1.2 Unsupervised Learning

When we have unclassified and unlabeled data, the system attempts to uncover patterns from the data. There is no label or target given for the examples. One common task is to group similar examples together, called clustering.

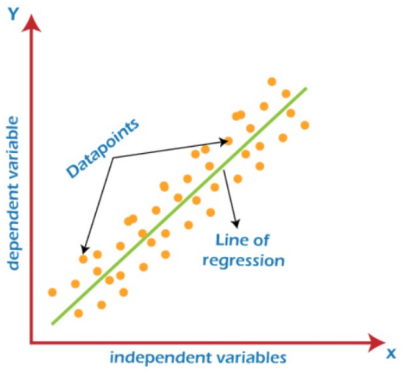
1.2 Supervised Machine Learning Techniques

1.2.1 Linear Regression

Linear Regression is one of the simplest and most popular machine learning algorithms. It is used for predictive analysis by making predictions for real variables such as experience, salary, cost, etc. It is a statistical approach that represents the linear relationship between two or more variables, either dependent or independent, hence called Linear Regression.

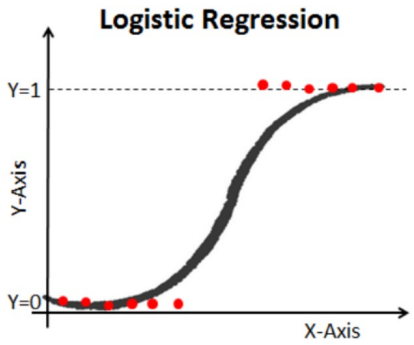
In the case of linear regression, the output is a linear function of the input. Let $\hat{y}$ be the output our model predicts:

$$\hat{y} = WX + b$$



1.2.2 Logistic Regression

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, logistic regression is a predictive analysis. It is used to describe data and to explain the relationship between one dependent binary variable and one or more independent variables.



1.3 Cost Function and Loss Function

1.3.1 What is Loss?

'Loss' in Machine learning helps us understand the difference between the predicted value and the actual value.The Function used to quantify this loss during the training phase in the form of a single real number is known as the "Loss Function".

1.3.2 Difference between Loss and Cost Function

Loss function: Used when we refer to the error for a single training example.
Cost function: Used to refer to an average of the loss functions over an entire training data.

1.3.3 Some Common Cost Functions

1. **Mean Absolute Error (MAE)**:
   - Mean Absolute Error (MAE) is a common cost function used for regression tasks. It measures the average of the absolute errors, which are the absolute differences between the actual and predicted values. The formula for MAE is:
   $$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$
2. **Mean Squared Error (MSE)**:
   - The Mean Squared Error (MSE) is another common cost function used for regression tasks. It measures the average of the squares of the errors, which are the differences between the actual

and predicted values. The formula for MSE is:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

3. **Binary Cross-Entropy**:

   - Binary Cross-Entropy loss is used in binary classification tasks, with only two possible classes or labels: positive and negative or true and false. The formula for Binary Cross-Entropy is:

$$\text{L}_{\text{BCE}} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

   where:

   – n is the number of samples,

   – y_i is the actual binary label (0 or 1) for sample i,

   – $\hat{y}_i$ is the predicted probability that sample i is of the positive class.

1.4 Back-propagation

The idea of the backpropagation algorithm is, based on error (or loss) calculation, to recalculate the weights array w and the bias value b in the last neuron layer, and proceed this way towards the previous layers, from back to front, that is, to update all the weights w and bias b in each layer, from the last one until reaching the input layer of the network .
For this doing the backpropagation of the error obtained by the network. In other words, we calculate the error between what the network predicted to be and what it was indeed, then we recalculate all the weights values, from the last layer to the very first one,always intending to decrease the neural network error.

Backpropagation consists in the following parts:

1. Initialize all the weights with small random values Feed data into the network and figure out the value of the error function,obtained by comparison with the expected output value.
2. Since the learning process is supervised, we know beforehand the value of the correct answer.
3. It's important that the error function is differentiable.In order to minimize the error, the gradients of the error function with respect to each weight is calculated.
4. It's known, from Calculus, that the gradient vector indicates the direction of highest increase of a function,therefore we want to move the weights in the direction of highest decrease of the error function(ie substract the gradient from the weights)

$$w^{(l)} \leftarrow w^{(l)} - \eta \frac{\partial L}{\partial w^{(l)}} \tag{1}$$

Where:

$\eta-$ is the learning rate

$\frac{\partial L}{\partial w^{(l)}}-$is the gradient of L wrt w

1.5 Hyperparamters

Hyperparameters are higher-level parameters that describe structural information about a model that must be decided before fitting model parameters. Examples include: Learning rate (alpha), batch size.

1.6 Gradient Descent

In the gradient descent algorithm, we start with random model parameters and calculate the error for each learning iteration, keep updating the model parameters to move closer to the values that results in minimum cost. The process can be summarized as:

$$\theta := \theta - \alpha \nabla J(\theta)$$

where:

- θ represents the model parameters,
- α is the learning rate,
- $\nabla J(\theta)$ is the gradient of the cost function $J(\theta)$.

Repeat this process until the cost function reaches its minimum value.

1.6.1 Learning rate

The learning rate is a hyperparameter in machine learning that controls the step size at which the parameters of a neural network are updated during training. It specifies the amount by which the model's parameters are altered in the direction opposite to the gradient of the loss function.

1.6.2 Different ways of doing Gradient Descent

There are three ways of doing gradient descent:

1. **Batch Gradient Descent**: Batch gradient descent is an optimization algorithm used to minimize the cost function by iteratively updating the model parameters based on the gradients of the entire training dataset.

2. **Mini-Batch Gradient Descent**: Mini-batch gradient descent is a variation of gradient descent where the training dataset is divided into smaller batches, and the model parameters are updated based on the gradients computed on each batch.

3. **Stochastic Gradient Descent (SGD)**: In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, hence the term "stochastic" in stochastic Gradient Descent

   The advantage of using SGD is its computational efficiency, especially when dealing with large datasets. By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset..

1.7 What Is Underfitting and Overfitting in Machine Learning?

1.7.1 Underfitting

When the model has fewer features, it isn't able to learn from the data very well. This means the model has a high bias.

1.7.2 Overfitting

When the model has complex functions, it's able to fit the data but is not able to generalize to predict new data. This model has high variance. There are three main options to address the issue of overfitting:

1. Hold-out (data):

   - Rather than using all of our data for training, we can simply split our dataset into two sets: training and testing. A common split ratio is 80% for training and 20% for testing. We train our model until it performs well not only on the training set but also for the testing set. This indicates good generalization capability since the testing set represents unseen data that were not used for training. However, this approach would require a sufficiently large dataset to train on even after splitting.

2. Data Augmentation (data):

   - A larger dataset would reduce overfitting. If we cannot gather more data and are constrained to the data we have in our current dataset, we can apply data augmentation to artificially increase the size of our dataset. For example, if we are training for an image classification task, we can perform various image transformations to our image dataset (e.g., flipping, rotating, rescaling, shifting).

3. Early Stopping (model):

- We can first train our model for an arbitrarily large number of epochs and plot the validation loss graph (e.g., using hold-out). Once the validation loss begins to degrade (e.g., stops decreasing but rather begins increasing), we stop the training and save the current model. We can implement this either by monitoring the loss graph or set an early stopping trigger. The saved model would be the optimal model for generalization among different training epoch values.

2 Activation functions

2.1 What are Activation functions?

The activation function defines the output of a node based on a set of specific inputs in machine learning, deep neural networks, and artificial neural networks. The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output. The purpose of an activation function is to add non-linearity to the neural network.
In the domain of deep learning, a neural network absent of an activation function resembles a linear regression model. These functions drive a neural network's ability to handle intricate tasks by performing crucial non-linear computations.

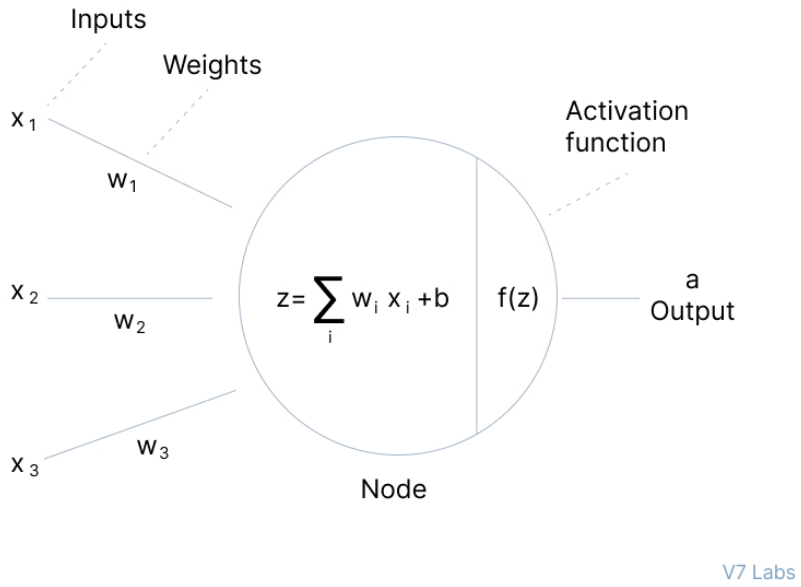


Figure 1: Activation Function

Let us see a few Activation Functions :

2.2 Linear Activation Function

The linear activation function, also known as "no activation," or "identity function" (multiplied x1.0), is where the activation is proportional to the input.
The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.

2.3 Binary Step Function

The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.

Binary step

$$f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geqslant 0 \end{cases}$$

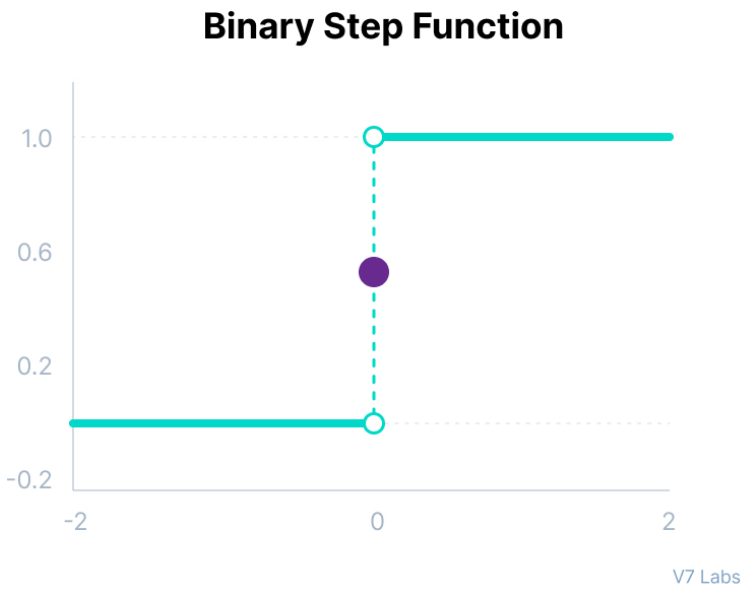


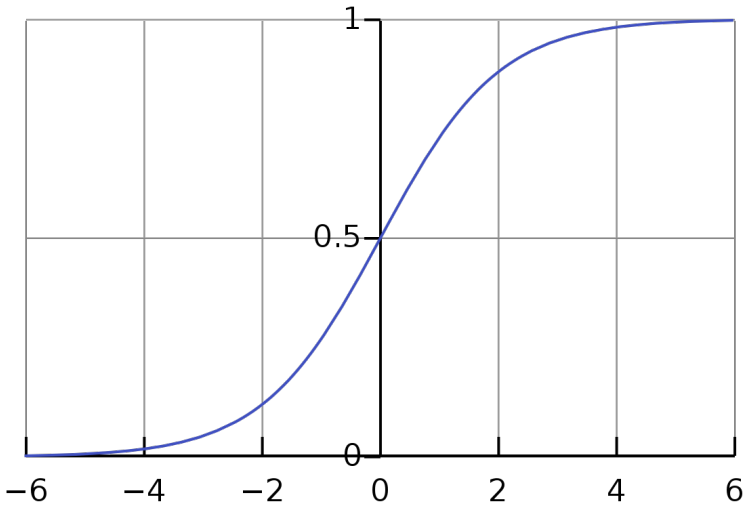
Figure 2: Binary step function depends on a threshold value that decides whether a neuron should be activated or not.

2.4 Sigmoid Activation Function

This function takes any real value as input and outputs values in the range of 0 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0, as shown below.

Sigmoid / Logistic

$$f(x) = \frac{1}{1+e^{-x}}$$

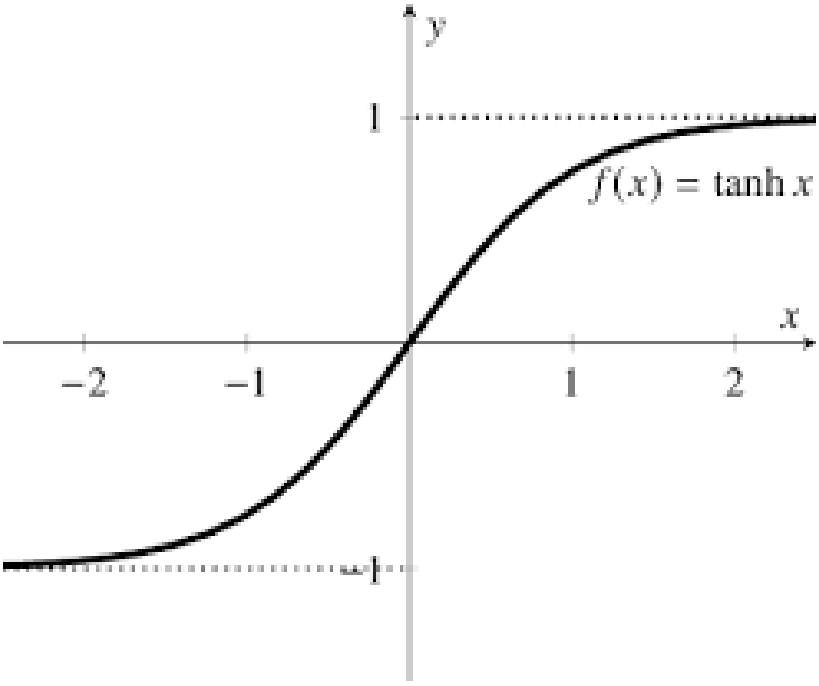


2.5 Tanh Activation (Hyperbolic Tangent)

Tanh function is very similar to the sigmoid/logistic activation function and even has the same S-shape with the difference in output range of -1 to 1. In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

Tanh

$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$



2.6 ReLU Function

ReLU stands for Rectified Linear Unit.

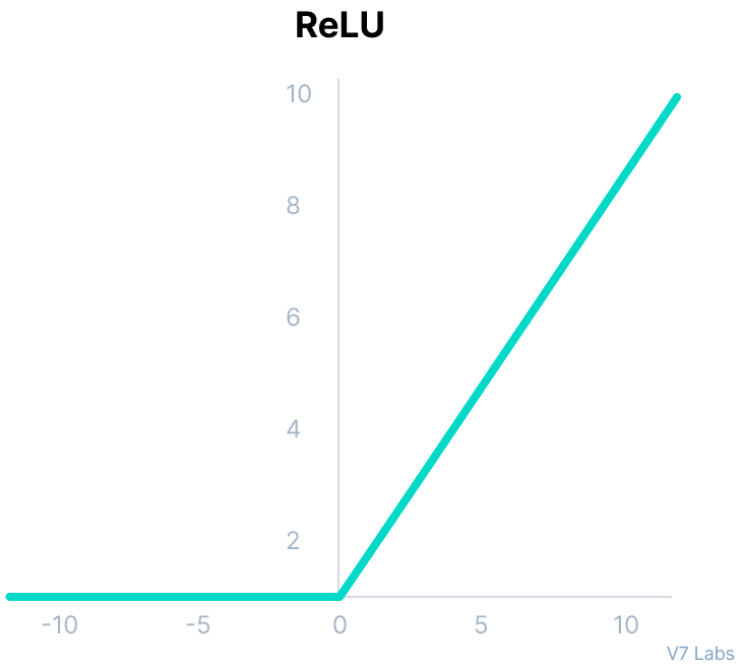
Although it gives the impression of a linear function, ReLU has a derivative function and allows for back-propagation while simultaneously making it computationally efficient.

The main catch here is that the ReLU function does not activate all the neurons at the same time.

The neurons will only be deactivated if the output of the linear transformation is less than 0.

$$ReLU$$

$$f(x) = max\,(0, x)$$



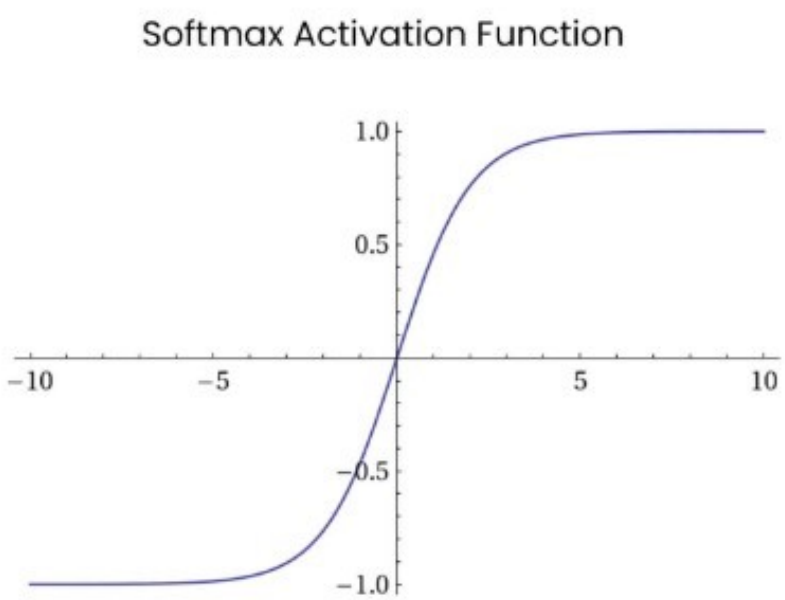
2.7 Softmax Activation

The Softmax function is described as a combination of multiple sigmoids. It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class. It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.

Softmax

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

V7 Labs



3 Optimizers

3.1 What is optimisation?

The ultimate goal of the ML model is to reach the minimum of the loss function. After we pass input, we calculate the error and update the weights accordingly. This is where the optimizer comes into play. It defines how to tweak the parameters to get closer to the minima.
So essentially, optimization is a process of finding optimal parameters for the model, which significantly reduces the error function.
Let us discuss a few optimizers:

3.2 Vanilla Gradient Descent

It is the most fundamental among the class of optimizers in deep learning. This optimization algorithm uses calculus to consistently modify the values and achieve the local minimum.

3.2.1 Mathematical formulation

Given a loss function $L(\theta)$, where θ represents the parameters of the model, the goal is to find the parameters θ that minimize L.

- **Loss Function**: $L(\theta)$
- **Gradient**: The gradient of the loss function $\nabla_\theta L(\theta)$ is a vector of partial derivatives of L with respect to θ.
- **Update Rule**:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

  where η is the learning rate.

Gradient descent works as follows:

1. It starts with some coefficients, and sees their loss.
2. It updates the parameters in the direction of the negative gradient of the loss function with respect to the parameters.
3. The process repeats until the local minimum is reached. A local minimum is a point beyond which it can not proceed.

Pros and Cons of Gradient Descent

Pros:

- Always converge
- Easy to compute

Cons:

- Slow
- Easily get stuck in local minima or saddle points
- Sensitive to the learning rate

3.3 Stochastic Gradient Descent

To tackle the challenges large datasets pose, we have stochastic gradient descent, a popular approach among optimizers in deep learning. The term stochastic denotes the element of randomness upon which the algorithm relies. In stochastic gradient descent, instead of processing the entire dataset during each iteration, we randomly select batches of data. This implies that only a few samples from the dataset are considered at a time, allowing for more efficient and computationally feasible optimization in deep learning models.

3.3.1 Mathematical formulation

- **Loss Function:**

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} L_i(\theta)$$

  where N is the number of training samples and L_i is the loss for the i-th sample.

- **Stochastic Gradient:** The gradient is estimated using a single sample or a mini-batch:

$$\nabla_\theta L(\theta) \approx \nabla_\theta L_i(\theta)$$

- **Update Rule:**

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L_i(\theta_t)$$

The procedure is first to select the initial parameters w and learning rate n. Then randomly shuffle the data at each iteration to reach an approximate minimum.
Since we are not using the whole dataset but the batches of it for each iteration, the path taken by the algorithm is full of noise as compared to the gradient descent algorithm. Thus, SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation

time increases. But even after increasing the number of iterations, the computation cost is still less than that of the gradient descent optimizer.

3.4 Adam (Adaptive Moment Estimation)

Adam optimizer, short for Adaptive Moment Estimation optimizer, is an optimization algorithm commonly used in deep learning. It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training.

The name "Adam" is derived from "adaptive moment estimation," highlighting its ability to adaptively adjust the learning rate for each network weight individually. Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments.

The creators of Adam optimizer incorporated the beneficial features of other optimization algorithms such as AdaGrad and RMSProp.

Mathematical Formulation

Adam maintains two moving averages of the gradient: the first moment (mean) and the second moment (uncentered variance).

1. **Initialize Parameters:**

$$m_0 = 0 \quad \text{(first moment vector)}$$
$$v_0 = 0 \quad \text{(second moment vector)}$$
$$t = 0 \quad \text{(time step)}$$

2. **Hyperparameters:**

α : Learning rate

β_1 : Decay rate for the first moment (typically 0.9)

β_2 : Decay rate for the second moment (typically 0.999)

ϵ : A small constant to prevent division by zero (typically 10^{-8})

3. **Update Rule:**

- Increment time step:

$$t = t + 1$$

- Compute gradient:

$$g_t = \nabla_\theta L(\theta_t)$$

- Update biased first-moment estimate:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

- Update biased second-moment estimate:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- Correct bias in the first moment:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

- Correct bias in the second moment:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Update parameters:

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

3.5 Detailed Steps

1. **Initialization:** Initialize the parameters θ, m_0, v_0, and set $t = 0$.
2. **Iterate:** For each step:
   - Increment t.
   - Compute the gradient g_t.
   - Update the biased first and second-moment estimates:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t,$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2.$$

   - Compute bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t},$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

   - Update the parameters:

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

3. **Repeat:** Continue until convergence or for a predetermined number of iterations.

3.6 Advantages and Disadvantages of Adam Optimization

- **Advantages:**
  - **Adaptive Learning Rate:** Adjusts the learning rate for each parameter individually, which can improve convergence.
  - **Bias Correction:** Provides bias-corrected first and second moment estimates ($\hat{m}_t$ and $\hat{v}_t$), resulting in more accurate parameter updates.
  - **Efficiency:** Combines the benefits of AdaGrad (effective for sparse gradients) and RMSProp (effective for non-stationary objectives), making it suitable for a wide range of optimization problems.
  - **Robustness:** Well-suited for large-scale problems and high-dimensional parameter spaces, where it can converge faster than other optimization algorithms.
- **Disadvantages:**
  - **Complexity:** More complex to implement and understand compared to simpler algorithms like Stochastic Gradient Descent (SGD).
  - **Resource Intensive:** Requires more memory to store the moment estimates m_t and v_t, which can be a consideration for memory-limited environments.

3.7 Implementation of Adam Optimization in Machine Learning Libraries

Modern machine learning libraries like TensorFlow, PyTorch, and Keras provde built-in implementations of optimization algorithms. Here are examples of implementing Adam optimization in these libraries:

3.7.1 TensorFlow

```
import tensorflow as tf

# Define the optimizer
optimizer = tf.optimizers.Adam(learning_rate=0.001)
```

3.7.2 PyTorch

```
import torch
```

```
# Define the model and optimizer
model = YourModel()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

3.7.3 Keras

```
from tensorflow import keras

# Define the optimizer
optimizer = keras.optimizers.Adam(learning_rate=0.001)
```

4 CNN

4.1 What are CNNs?

- CNN stands for Convolution Neural Network
- Just like ANNs, CNNs are also deep neural networks, which are predominantly used to extract the features from grid-like matrix datasets like images
- Each layer of CNNs comprises of several filters(or kernels) (of n x m dimensions) which are used to detect features and patterns in the grid dataset.

4.2 Why to use CNNs?

1. Patterns or objects in an image generally have their pixels next to each other, thus the pixels in the neighbor of the central pixel of a pattern are involved in the classification of the pattern. Simple fully connected networks are unable to detect this relationship easily, whereas CNNs due to using a n x n filter (or kernel) are also able to take into account the neighbors of a pixel.
2. CNNs require relatively lesser number of trainable parameters (weights and biases) to classify the images with the same accuracy as a Fully Connected Network. Thus CNN models are less complex, small, and give better accuracy.

4.3 Parts of CNN

A complete Network comprises several layers, following are the description and working of these layers:

4.3.1 Input Layers

- It's the layer in which we give input to our model. In CNN, generally, the input will be an image or a sequence of images.

4.3.2 Convolutional Layers

- In convolutional neural networks (CNNs), a convolutional layer consists of a set of learnable filters, also known as kernels.
- These kernels are small-sized matrices (typically 3x3 or 5x5) that slide across the input image or feature map, and convolution operation happens between the input image pixels and the kernel.
- The convolution operation involves element-wise multiplication between the kernel and a local region of the input, followed by summing up the results to produce a single output value.

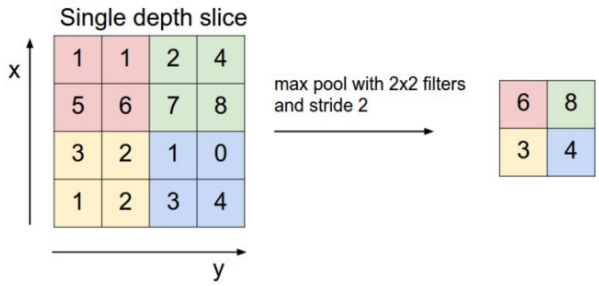


Figure 3: Max Pooling— Takes the maximum value out of each color

- This process is repeated across the entire input image or feature map, resulting in a new output feature map.

4.3.3 Pooling Layers

Pooling layers are used to downsample the image after the convolution layer.

Why to do Pooling?

1. Helps to reduce the number of sliding operations over the image by the kernel in the next layer. This helps reduce the training and the prediction time.
2. Method like Max Pooling helps to preserve the important features(as we will preserve the maximum value within a kernel thus preserving the classified pixels) while simultaneously reducing the dimension.
3. The reduction of dimensions by Pooling reduces the number of parameters passed to the fully connected layer, thus reducing the number of parameters required to be trained.

Types of Pooling:

1. **Max Pooling:**
   Reduces the dimensionality of feature maps by selecting the maximum value from a predefined window (filter) that slides across the input. [Figure 3]
2. **Average Pooling:**
   Calculates the average value within the window, capturing a broader sense of the features in that region.[Figure 4]
   Can be useful for tasks where spatial information is important or when dealing with noisy data.

4.3.4 Stride

Stride defines the step size or the number of pixels by which the kernel moves across the input image during convolution.

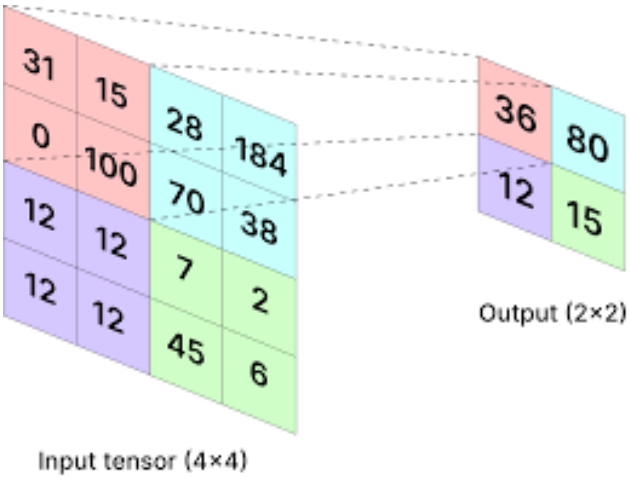


Figure 4: Enter Caption

Having a stride of 1 helps to capture the maximum detail from the image, but it would take a lot of time if we have for example a very high-definition image, where we may have to compromise for a bigger stride (thus compromising over relevant pattern detection) to improve training and prediction time.
Therefore, it is taken to be 1, generally, except when the image is of high resolution. We can use heuristics to judge the size of the stride.

4.3.5 Padding

Zero Padding, adds rows and columns of 0 (depending on the size of padding) to the borders of the image
Padding helps mitigate the loss of spatial data at the image borders during convolution, leading to potentially better feature extraction and model performance. If padding is not done, pixels at the border will be convoluted less number of times by the kernel as compared to pixels at the center, leading to a reduction of pattern detection at the edges.
Padding allows you to achieve desired output dimensions for your convolution layers, which helps prevent the change in dimension after convolution.

Dimensions of (n,n) image after convolution with a given padding and stride:

$$\text{Dimensions} = \left(\frac{n + 2p - f + 1}{s}, \frac{n + 2p - f + 1}{s} \right) \tag{2}$$

Where:
n - Width and height of the original input image (n x n).
p - Amount of padding added to each border (usually zero for zero padding).
f - Width and height of the filter kernel (assuming a square kernel).
s - Stride of the convolution operation (number of steps between filter applications).

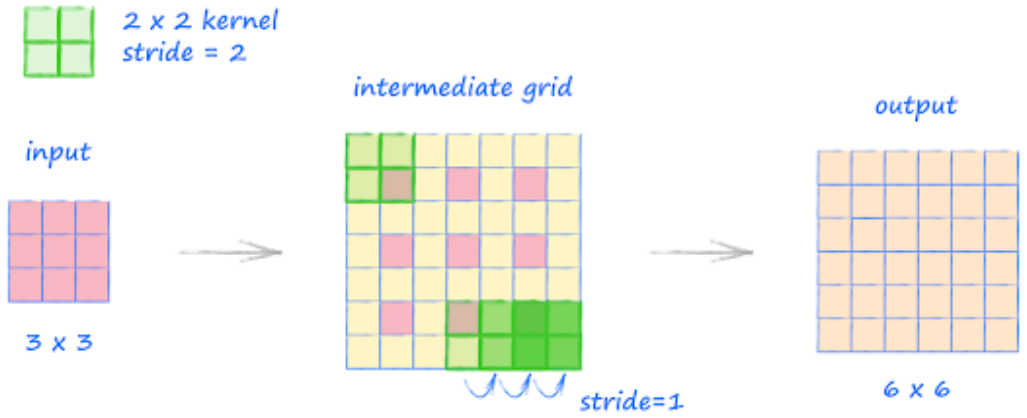


Figure 5: Working of Conv2Dtranspose layer— In the second image,the pink squares are the parts of the original image, and the yellow squares are part of rows and columns added in order to expand dimensions.

4.3.6 Cov2DTranspose Layer

The Conv2DTranspose layer, also sometimes called a deconvolutional layer (although technically not a true deconvolution)

The Conv2DTranspose layer uses a learned kernel (filter) to perform a transposed convolution operation. This operation expands the input feature map while capturing spatial information and potentially adding new features.

4.3.7 Flatten layer

The resulting feature maps are flattened into a one-dimensional vector after the convolution and pooling layers so they can be passed into a completely linked layer for categorization.

For example: If the previous layer output is of shape (width, height, channels) the new dimension will be (width x height x channels)

4.3.8 Fully Connected Layers

The output vector through the Flatten layer is passed through several fully connected layers, which ultimately classify the image.

Fully connected layers act like a classifier on top of the feature extraction capabilities of convolution layers.

5 DCGANs

5.1 Introduction

Deep Convolutional Generative Adversarial Networks (DCGANs) were developed to demonstrate the capabilities of Convolutional Neural Networks in unsupervised learning.
DCGANs represent a very strong neural network design used for synthesizing new and previously unknown data, mostly images. They are part of the Generative Adversarial Networks (GANs) known for their remarkable ability to discern the inherent structures within a dataset and produce fictional instances that closely resemble real data.

5.2 Architecture

DCGANs at their core also use a discriminator and generator model where the Generator tries to fool the Discriminator from distinguishing the image generated by the generator from a real one.

5.2.1 The Generator

The generator starts with a random noise vector and builds upon it, using it as the base for generating an image. The generator uses transpose convolutional layers to gradually change and enhance the initial noise vector. With every layer, the image expands step-by-step. Unlike pooling layers, which compress image data, transpose convolutional layers do the opposite. They add details to the given noise vector (or an image), resulting in a more intricate image. Throughout this process, information is processed, and patterns and features develop from the noise vector. These features gradually evolve and change, finally giving out a recognizable image.

Key features-

1. Training deep neural networks such as the generators and discriminators from DCGANs comes with many challenges such as exploding and vanishing gradients. Batch normalization helps in answering these challenges by normalizing the outputs of individual layers. This ensures a smoother learning process and protects the network from being stuck.

2. For adding non linearities in the generator the ReLU activation function is used. We will see that this is not the case for the discriminator.

5.2.2 The Discriminator

The discriminator is a binary classifier that classifies images as either fake (i.e., generated) or real (i.e., belonging to the dataset). It uses a series of convolutional layers to extract features from the image and then

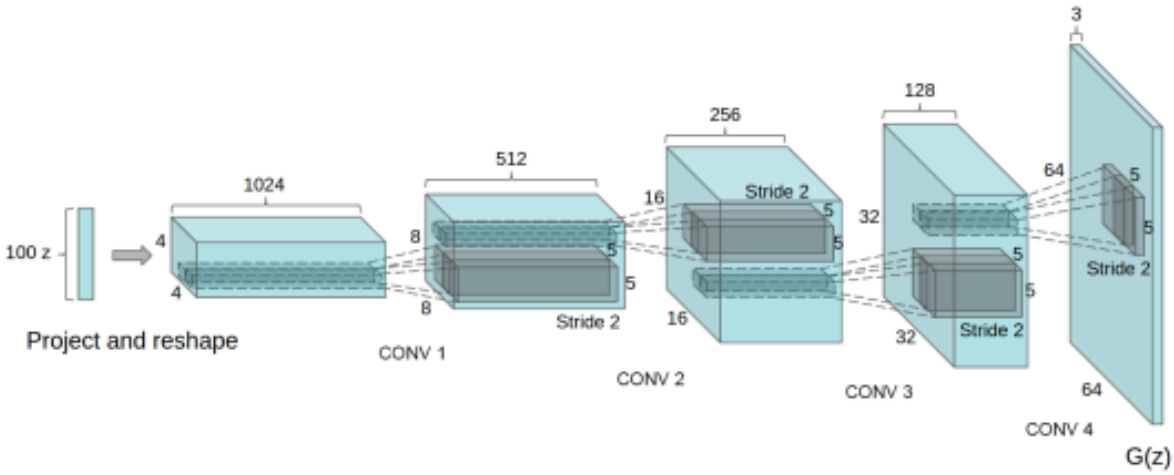


Figure 6: The Generator architecture in DCGANs

uses a few fully connected layers to finally output the probability of the image being real or fake.

Key features-

1. The discriminator is also a deep neural network and thus prone to vanishing and exploding gradients.To deal with these issues, the discriminator also uses batch normalization layers to stabilize training.

2. The activation function used in discriminators is Leaky ReLU, rather than ReLU. Leaky ReLU prevents neurons from becoming completely inactive, thus allowing them to learn from even small errors. This makes Leaky ReLU a more robust choice for the discriminator.

5.3 Training

Training a DCGAN is like a game of tug-of-war between the generator and the discriminator.
They both get progressively stronger, but you want to maintain a balance between them. This ensures they get stronger without overpowering each other.

5.3.1 Discriminator Training

The discriminator's goal is to correctly identify real and fake images. During initialization, it's trained on the image dataset and random noise images to distinguish real from fake. Afterward, it's continuously fed images from both the generator and the real dataset to progressively improve its ability to classify real and generated images.

5.3.2 Generator Training

The generator's goal is to successfully fool the discriminator, essentially tricking it into classifying the generated image as real. Based on the discriminator's feedback, the generator continuously refines itself, optimizing its ability to create increasingly realistic images.

5.3.3 Loss functions

In the DCGAN architecture the loss functions for both, discriminator and generator are derived for the value function.

$$V(D,G) = E_{x\sim pdata(x)}[logD(x)] + E_{z\sim pz(z)}[log(1 - D(G(z)))] \tag{3}$$

- **Discriminator Loss**

  The discriminator loss is taken as the entire value function where the discriminator tries to maximize the discriminator loss

$$L_D = -E_{x\sim pdata}[logD(x)] - E_{z\sim pz}[log(1 - D(G(z)))] \tag{4}$$

- **Generator Loss**

  The generator loss is taken as the part of value function affected by the generator. The generator tries to minimize the generator loss.

$$L_G = -E_{z\sim pz}[log(1 - D(G(z)))] \tag{5}$$

5.4 Tools in GANs

5.4.1 KL Divergence

Kullback-Leibler divergence metric (relative entropy) is a statistical measurement from information theory that is commonly used to quantify the difference or dissimilarity between one probability distribution from a reference probability distribution.
The formula for the KL divergence $D_{\mathrm{KL}}(P \parallel Q)$ of a distribution P from a distribution Q is given by:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In the continuous case, the formula is:

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)}\, dx$$

where:

- P and Q are the two probability distributions.
- $P(i)$ and $Q(i)$ are the probabilities of the i-th event according to distributions P and Q respectively.
- $p(x)$ and $q(x)$ are the probability density functions of the continuous random variable x under distributions P and Q respectively.

The KL divergence is non-negative and is zero if and only if P and Q are the same distribution. It is important to note that KL divergence is not symmetric, i.e., $D_{\mathrm{KL}}(P \parallel Q) \neq D_{\mathrm{KL}}(Q \parallel P)$.
In this project we have used this while defining the **Jensen-Shannon Consistency Loss**

5.4.2 Jensen-Shannon Consistency Loss

Jensen-Shannon Consistency (JSC) Loss is a metric used to measure the similarity between two probability distributions. It is particularly useful in tasks involving generative models, such as data augmentation or semi-supervised learning, where maintaining consistency between different versions or augmentations of data is important.

$$JS(p,q) = \frac{1}{2}\left(KL(p||m) + KL(q||m)\right)$$

$$L_{JS} = \lambda \cdot JS(p,q)$$

where $m = \frac{1}{2}(p+q)$ and λ is the weight of JSC in the loss.
HyperParameters :

- Number of augmented versions.
- Alpha: weight of JSC in loss.

6 Stack-GANs

6.1 Introduction

Stacked Generative Adversarial Networks were proposed for high-resolution image generation from text descriptions which employs a two-stage training process.

A StackGAN is named as such because it has two GANs that are stacked together where the first GAN generates low-resolution images with basic colors and rough sketches, conditioned on text-embedding, while the second GAN takes this image generated by the first GAN, corrects defects and adds compelling details conditioned in text-embedding, yielding a more realistic high-resolution image. The two GANs are called the two stages, namely the Stage-I GAN and the Stage-II GAN.

6.2 Stage-I GAN

6.2.1 Text Embedding

Using a pre-trained text encoder to convert the text description into a fixed-length vector is known as text embedding. This embedding captures the semantic meaning of the text. Let φ_t be the text embedding of the given description. The Gaussian conditioning variables $\hat{c}_0$ for text embedding are sampled from $\mathcal{N}(\mu_0(\varphi_t), \Sigma_0(\varphi_t))$ to capture the meaning of φ_t with variations.

6.2.2 Training Process

Stage-I GAN is trained like a typical GAN. In generator training, the generator aims to produce images that the discriminator will classify as real while the discriminator is trained to accurately distinguish between real images and generated images.

Conditioned on $\hat{c}_0$ and random variable z, Stage-I GAN trains the discriminator D_0 and the generator G_0 by alternatively maximizing the discriminator loss function $\mathcal{L}_{D_0}$ and minimizing the generator loss function $\mathcal{L}_{G_0}$,

$$\mathcal{L}_{D_0} = \mathbb{E}_{(I_0,t)\sim p_{data}}[\log D_0(I_0, \varphi_t)] + \mathbb{E}_{z\sim p_z, t\sim p_{data}}[\log(1 - D_0(G_0(z, \hat{c}_0), \varphi_t))], \tag{6}$$

$$\mathcal{L}_{G_0} = \mathbb{E}_{z\sim p_z, t\sim p_{data}}[\log(1 - D_0(G_0(z, \hat{c}_0), \varphi_t))] + \lambda D_{KL}(\mathcal{N}(\mu_0(\varphi_t), \Sigma_0(\varphi_t)) \parallel \mathcal{N}(0, I)), \tag{7}$$

I_0 - real image

t - text description from the true data distribution p_{data}

z - noise vector randomly sampled from Gaussian distribution p_z

λ - regularization parameter, $\lambda = 1$ for all our experiments

6.2.3 Reparameterization Trick

To allow gradient-based optimization methods to work through the random sampling process, the reparameterization trick is used. This trick involves expressing the Gaussian sampling in a way that separates the deterministic and stochastic parts:

$$\hat{c}_0 = \mu_0(\varphi_t) + \Sigma_0(\varphi_t) \odot \epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$ is a random variable drawn from a standard normal distribution, and $\odot$ denotes element-wise multiplication. Here's what each component represents:
- $\mu_0(\varphi_t)$ is the mean vector conditioned on the text embedding. - $\Sigma_0(\varphi_t)$ is the standard deviation (or covariance matrix) conditioned on the text embedding. - ϵ is a noise vector sampled from a standard normal distribution.
By reparameterizing $\hat{c}_0$ in this manner, the randomness is now introduced through ϵ, which does not depend on the network parameters. This allows gradients to be backpropagated through $\mu_0(\varphi_t)$ and $\Sigma_0(\varphi_t)$ which are learned jointly with the network.

6.2.4 Model Architecture

1. Generator G_0**-** the generator is typically a deep neural network composed of several layers that transform the input vector into an initial shape which is then progressively upscaled by subsequent layers to the desired low resolution(64 x 64). Each layer uses activation functions like ReLU (Rectified Linear Unit) to introduce non-linearity. Text embedding φ_t is fed into a fully connected layer to generate μ_0 and σ_0.
Conditioning vector $\hat{c}_0$ is computed by $\hat{c}_0 = \mu_0 + \sigma_0 \odot \epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$. $\hat{c}_0$ is concatenated with a noise vector to generate an image through up-sampling blocks.

2. Discrimator D_0 **-** The Discriminator network is responsible for evaluating the authenticity of generated image where the convolutional layers extract features from the images and text embeddings to distinguish real from fake images and outputs the probability of the image being real. It does so by compressing and spatially replicating the text embedding φ_t. The image is downsampled and concatenated with the text tensor. The combined tensor is fed into a 1×1 convolutional layer and a fully connected layer to produce the decision score.

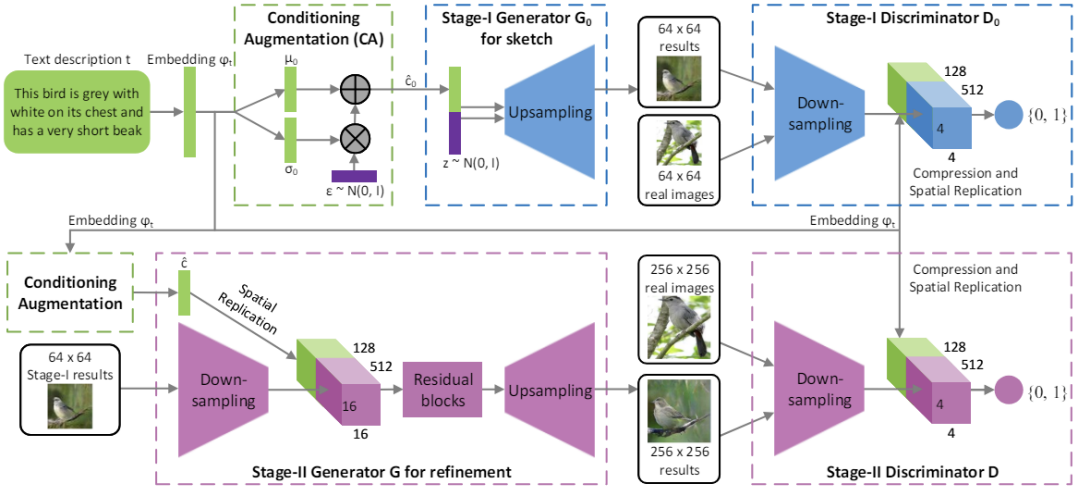


Figure 7: Architecture of the StackGAN.

6.3 Stage-II GAN

6.3.1 Training process

Low-resolution images generated by Stage-I GAN usually lack vivid object parts and might contain shape distortions. Some details in the text might also be omitted in the first stage, which is vital for generating photo-realistic images. Our Stage-II GAN is built upon Stage-I GAN results to generate high-resolution images. It is conditioned on low-resolution images and also the text embedding again to correct defects in Stage-I results. The Stage-II GAN completes previously ignored text information to generate more photo-realistic details.

Conditioning on the low-resultion result $s_0 = G_0(z, \hat{c}_0)$ and Gaussian latent variables $\hat{c}$, the discriminator D and generator G in Stage-II GAN are trained by alternatively maximizing generator loss function, $\mathcal{L}_D$, and minimizing discriminator loss function, $\mathcal{L}_G$.

$$\mathcal{L}_D = \mathbb{E}_{(I,t)\sim p_{data}}[\log D(I, \varphi_t)] + \mathbb{E}_{z\sim p_z, t\sim p_{data}}[\log(1 - D(G(z, \hat{c}), \varphi_t))], \tag{8}$$

$$\mathcal{L}_G = \mathbb{E}_{z\sim p_z, t\sim p_{data}}[\log(1 - D(G(z, \hat{c}), \varphi_t))] + \lambda D_{KL}(\mathcal{N}(\mu(\varphi_t), \Sigma(\varphi_t)) \parallel \mathcal{N}(0, I)), \tag{9}$$

6.3.2 Differences

Different from the original GAN formulation, the random noise z is not used in this stage with the assumption that the randomness has already been preserved by s_0. Gaussian conditioning variables $\hat{c}_0$ used in this stage and $\hat{c}_0$ used in Stage-I GAN share the same pre-trained text encoder, generating the same text embedding φ_t. However, Stage-I and Stage-II Conditioning Augmentation have different fully connected layers for generating different means and standard deviations. In this way, Stage-II GAN learns to capture useful information in the text embedding that is omitted by Stage-I GAN.

6.3.3 Model Architecture

1. Generator *G*- Stage-II generator is designed as a encoder-decoder network with residual blocks. Similar to the previous stage, the text embedding ϕ_t is used to generate the N_g-dimensional text conditioning vector $\hat{c}$, which is spatially replicated to form a $M_g \times M_g \times N_g$ tensor. Meanwhile, the Stage-I result s_0 generated by Stage-I GAN is fed into several down-sampling blocks (i.e., encoder) until it has a spatial size of $M_g \times M_g$. The image features and the text features are concatenated along the channel dimension. The encoded image features coupled with text features are fed into several residual blocks, which are designed to learn multi-modal representations across image and text features. Finally, a series of up-sampling layers (i.e., decoder) are used to generate a $W \times H$ high-resolution image. Such a generator is able to help rectify defects in the input image while adding more details to generate the realistic high-resolution image.

2. Discriminator *D*- For the discriminator, its structure is similar to that of Stage-I discriminator with only extra down-sampling blocks since the image size is larger in this stage. To explicitly enforce GAN to learn better alignment between the image and the conditioning text, rather than using the vanilla discriminator, we adopt the matching-aware discriminator for both stages. During training, the discriminator takes real images and their corresponding text descriptions as positive sample pairs, whereas negative sample pairs consist of two groups. The first is real images with mismatched text embeddings, while the second is synthetic images with their corresponding text embeddings

7 Image Augmentation

7.1 What is Image Augmentation?

Image augmentation is a process of creating new training examples from existing ones. To make a new sample, you slightly change the original image. For instance, you could make a new image a little brighter, you could cut a piece from the original image; you could make a new image by mirroring the original one, etc.

7.2 Why do we need Image Augmentation?

- Images can vary in many ways, such as lighting, pose, scale, and occlusion. These variations can make it difficult for computer vision models to generalize to new data.
- In many object detection and classification tasks, the number of images in each class is not balanced, and some classes may have only a few examples. This can make it difficult for models to learn to recognize all classes equally well.
- A model trained on one dataset or in one environment may not perform well when applied to new, unseen data or environments. This is due to the distribution of the data being different in the new environment.
- A larger set of learnable parameters in a deep learning model demands more data for training. If the number of parameters increases, the model may overfit by memorizing specific data points, leading to poor performance on new data.

7.3 Classical Techniques for Image Augmentation:

7.3.1 Flipping and Rotating:

Robust to changes in object orientation.

7.3.2 Cropping and Resizing:

To handle changes in object scale or position within the image.

7.3.3 Color Jittering:

To handle changes in lighting conditions and color variations in the dataset.

7.3.4 Adding Noise:

Beneficial to handle variations in image quality and simulate noisy environments.

7.3.5 Image Warping:

Helpful for increasing a model's ability to handle changes in object pose and simulate changes in camera viewpoint.

7.3.6 Random Erasing:

Useful to make it more robust to occlusions or clutter in the image, or when you want to simulate missing data.

7.4 Advanced Techniques for Image Augmentation:

7.4.1 Cutout

Cutout is an augmentation technique that randomly covers a region of an input image with a square.
Explanation: Co-adaptation in neural networks refers to a situation where some neurons become highly dependent on others, affecting model performance when independent neurons receive "bad" inputs. Cutout applies a similar method on images for CNNs by dropping contiguous sections of inputs rather than individual pixels or neurons.
Advantages:

- Helps in training models to recognize partial or occluded objects.
- Allows the model to consider more of the image context such as minor features rather than relying heavily on major features.

Limitations:

- It can completely remove important features from an image.
- It may not work well for images with complex backgrounds.
- Significantly reduces the proportion of informative pixels used in the training process.

Hyperparameter: Size and number of patches to be cut out from the image.

7.4.2 Mixup Augmentation

Mixup generates a weighted combination of random image pairs from the training data. Given two images and their ground truth labels: (xi, yi), (xj, yj), a synthetic training example (x, y) is generated as:

$$x = \lambda x_i + (1 - \lambda)x_j$$

$$y = \lambda y_i + (1 - \lambda)y_j$$

where λ is sampled from the Beta distribution.

Explanation: Mixup constructs virtual training examples, extending the training distribution by incorporating the prior knowledge that linear interpolations of feature vectors should lead to linear interpolations of the associated targets.

Advantages:

- Reduces overfitting by combining different features and labels.
- Provides a smoother estimate of uncertainty.
- Robustness to adversarial examples and stabilized GAN training.
- Domain-agnostic technique applicable to various data modalities.

Limitations:

- Only inter-class mixup.
- The examples are not real representations of classes.
- Does not work well with Supervised Contrastive Learning and label smoothing

7.4.3 Cutmix

Cutmix replaces a square region of an input image with a patch of similar dimensions from another image. The ground truth labels are mixed proportionally to the number of pixels from each image.

Explanation:

$$X = M \odot x_i + (1 - M) \odot x_j$$

$$y = \lambda y_i + (1 - \lambda) y_j$$

where M is a binary mask indicating the Cutout and fill-in regions from the two randomly drawn images.

Advantages:

- Avoids uninformative pixels during training.
- Efficient training process.

Limitations

- Complex to implement.
- May not work well with certain datasets.

7.5 Augmix

Augmix uses three separate chains of one to three randomly chosen augmentation operations, combined with the original image using different weights.

Explanation: Creates multiple sources of randomness, including the selection of operations, their intensity, the length of the chains, and the mixing weights. Combined with a consistency loss to ensure the augmented images are semantically meaningful.

Jensen-Shannon Consistency Loss

$$JS(p, q) = \frac{1}{2} \left(KL(p \| m) + KL(q \| m) \right) \tag{10}$$

$$L_{JS} = \lambda \cdot JS(p, q) \tag{11}$$

where $m = \frac{1}{2}(p + q)$ and λ is the weight of JSC in the loss.

8 Assignments

8.1 Assignment 1

8.1.1 Objective

The objective of this assignment is to understand and implement a simple neural network for data classification from scratch. The task involves building and training a neural network with various configurations to classify data points into their respective categories. The focus is on experimenting with different numbers of hidden layers and observing their impact on the model's accuracy.

8.1.2 Learning

By completing this assignment, we learnt:

- **Implement a Neural Network**: Develop a neural network from scratch using basic Python libraries such as NumPy.
- **Understand Network Architecture**: Gain insights into how different configurations (i.e., the number of hidden layers) affect the network's performance.
- **Train Neural Networks**: Learn the process of training neural networks, including forward and backward propagation and parameter updates.
- **Evaluate Model Performance**: Use accuracy as a metric to evaluate the performance of the neural network and understand the impact of model complexity.
- **Visualize Decision Boundaries**: Plot decision boundaries to visually assess the classification performance of different neural network configurations.

8.1.3 Results

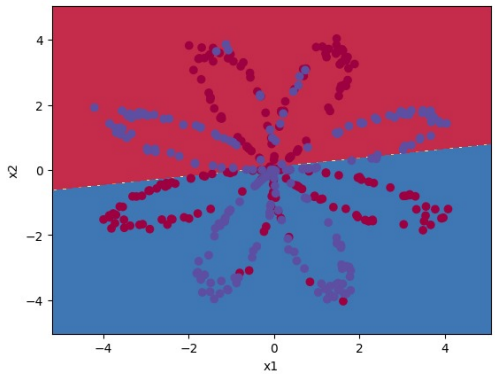


Figure 8: Decision Boundary while using Logistic regression

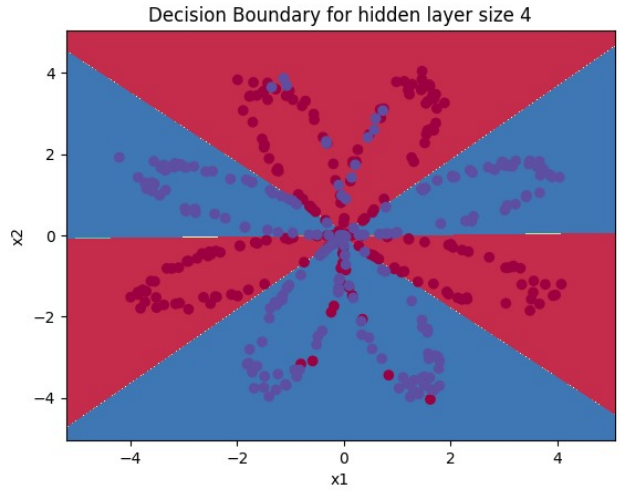


Figure 9: Decision Boundary while using a neural network which has 4 hidden layers

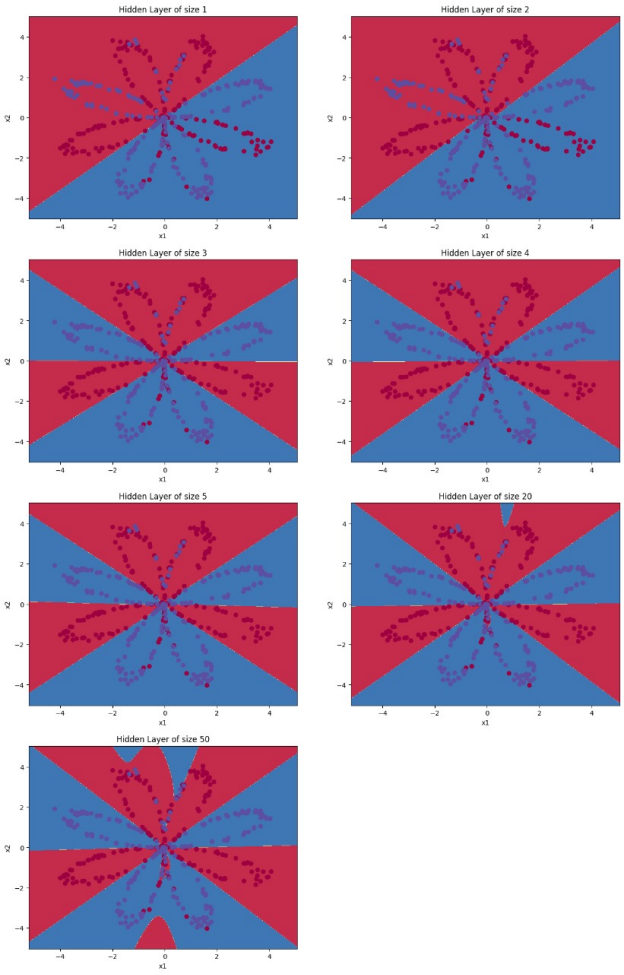


Figure 10: Decision Boundary while using a neural network with various different number of hidden layers

8.2 Assignment 2

8.2.1 Objective

Understanding how to build a basic CNN model, how to import the dataset, and preparing training and testing dataset.

Implement a Convolutional Neural Network (CNN) model to achieve an accuracy of 98 percent or above within 20 epochs on a given image dataset.

Utilize various techniques and optimizations to achieve the desired accuracy target.

8.2.2 Learning Outcomes

How to import a zip file, decompressing it, splitting it in training and test dataset.

Gain practical experience in building and training CNN models.

Understand the importance of hyperparameter tuning and optimization techniques in achieving high accuracy.

Develop problem-solving skills by exploring different approaches to improve model performance.

Discovered new techniques like adjusting learning rate, using early stopping, changing complexity of model to improve the accuracy.

8.2.3 Results

```
Epoch 10/19
8/8 [==============================] - 0s 12ms/step - loss: 0.2156 - accuracy: 0.9508 - val_loss: 0.5442 - val_accuracy: 0.8125
Epoch 11/19
8/8 [==============================] - 0s 15ms/step - loss: 0.1521 - accuracy: 0.9590 - val_loss: 0.4371 - val_accuracy: 0.8594
Epoch 12/19
8/8 [==============================] - 0s 18ms/step - loss: 0.1365 - accuracy: 0.9672 - val_loss: 0.4823 - val_accuracy: 0.8594
Epoch 13/19
8/8 [==============================] - 0s 16ms/step - loss: 0.0935 - accuracy: 0.9754 - val_loss: 0.3810 - val_accuracy: 0.8594
Epoch 14/19
8/8 [==============================] - 0s 14ms/step - loss: 0.0615 - accuracy: 0.9918 - val_loss: 0.5410 - val_accuracy: 0.8438
Epoch 15/19
8/8 [==============================] - 0s 15ms/step - loss: 0.0430 - accuracy: 0.9836 - val_loss: 0.3386 - val_accuracy: 0.9375
Epoch 16/19
8/8 [==============================] - 0s 15ms/step - loss: 0.0279 - accuracy: 0.9959 - val_loss: 0.3925 - val_accuracy: 0.9062
Epoch 17/19
8/8 [==============================] - 0s 14ms/step - loss: 0.0119 - accuracy: 1.0000 - val_loss: 0.3579 - val_accuracy: 0.8906
Epoch 18/19
8/8 [==============================] - 0s 14ms/step - loss: 0.0108 - accuracy: 1.0000 - val_loss: 0.3317 - val_accuracy: 0.9062
Epoch 19/19
8/8 [==============================] - 0s 14ms/step - loss: 0.0046 - accuracy: 1.0000 - val_loss: 0.3585 - val_accuracy: 0.9375
2/2 [==============================] - 0s 9ms/step - loss: 0.3585 - accuracy: 0.9375
Test accuracy: 0.9375
Total Training Time taken:  0 Minutes
```

Figure 11: Results of Model 1 — Basic CNN network

```
Epoch 1/20
31/31 [==============================] - 2s 13ms/step - loss: 2.8060 - accuracy: 0.0451 - val_loss: 2.7534 - val_accuracy: 0.0625
Epoch 2/20
31/31 [==============================] - 0s 7ms/step - loss: 2.7350 - accuracy: 0.1148 - val_loss: 2.7098 - val_accuracy: 0.1250
Epoch 3/20
31/31 [==============================] - 0s 9ms/step - loss: 2.5659 - accuracy: 0.1680 - val_loss: 2.2974 - val_accuracy: 0.2500
Epoch 4/20
31/31 [==============================] - 0s 7ms/step - loss: 2.1648 - accuracy: 0.3402 - val_loss: 1.7416 - val_accuracy: 0.5000
Epoch 5/20
31/31 [==============================] - 0s 7ms/step - loss: 1.4479 - accuracy: 0.5779 - val_loss: 1.0023 - val_accuracy: 0.7188
Epoch 6/20
31/31 [==============================] - 0s 7ms/step - loss: 0.7163 - accuracy: 0.7951 - val_loss: 0.5216 - val_accuracy: 0.8438
Epoch 7/20
31/31 [==============================] - 0s 7ms/step - loss: 0.3891 - accuracy: 0.9098 - val_loss: 0.2519 - val_accuracy: 0.9375
Epoch 8/20
31/31 [==============================] - 0s 7ms/step - loss: 0.1724 - accuracy: 0.9426 - val_loss: 0.1025 - val_accuracy: 0.9844
Epoch 9/20
31/31 [==============================] - 0s 7ms/step - loss: 0.1507 - accuracy: 0.9549 - val_loss: 0.1075 - val_accuracy: 0.9375
Epoch 10/20
31/31 [==============================] - 0s 7ms/step - loss: 0.0585 - accuracy: 0.9836 - val_loss: 0.2275 - val_accuracy: 0.9375
Epoch 11/20
31/31 [==============================] - 0s 7ms/step - loss: 0.1469 - accuracy: 0.9549 - val_loss: 0.3240 - val_accuracy: 0.8750
Epoch 12/20
31/31 [==============================] - 0s 9ms/step - loss: 0.2127 - accuracy: 0.9590 - val_loss: 0.1352 - val_accuracy: 0.9531
2/2 [==============================] - 0s 11ms/step - loss: 0.1025 - accuracy: 0.9844
Test accuracy: 0.9844
```

Figure 12: Results of Model 2 — After using Data Augmentation , Changing batch size and using early stopping