Regularization in Deep Learning with Python Code

shubham.jain
Last Updated : 03 Dec, 2024

221

One of the most common challenges encountered by **data science** professionals is the issue of overfitting. Have you ever experienced a scenario where your **machine learning** model or regression model excelled on the training data but struggled to make accurate predictions on unseen test data? This is often a trade-off between high bias (underfitting) and high variance (overfitting), impacting the model's ability to generalize effectively. **Overfitting** can lead to poor performance on new data, especially in the presence of outliers or noise in the training set. If so, this article addresses your concerns by exploring techniques such as regularization in **deep learning** and essential concepts of bagging, boosting, and stacking in **ensemble learning** to improve model generalization.

In this article, you will delve into the concept of regularization in **neural networks**, discover different regularization techniques, and understand how these approaches can greatly enhance model performance while preventing overfitting.

Note: *This article assumes that you have basic knowledge of neural networks and their implementation in Keras. If not, you can refer to the below articles first–*

- **Fundamentals of Deep Learning – Starting with Artificial Neural Network**
- **Tutorial: Optimizing Neural Networks using Keras (with Image recognition case study)**

Table of contents

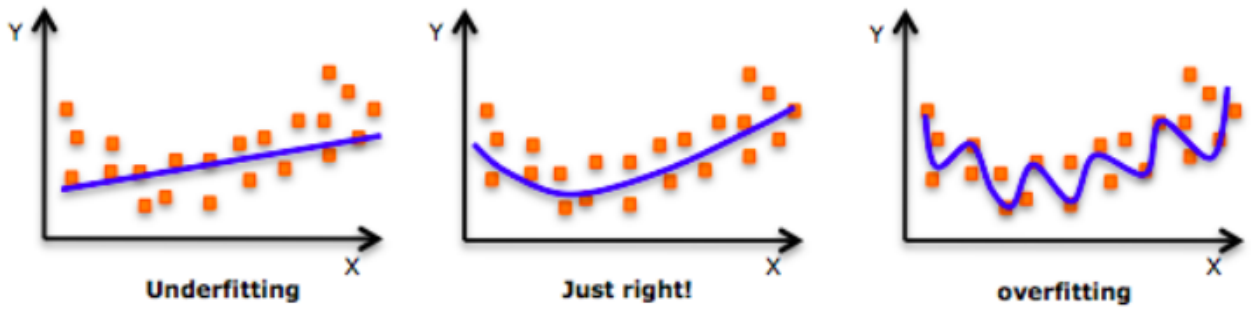
3. Different Regularization Techniques in Deep Learning
    - L2&L1 Regularization
    - Dropout
    - Data Augmentation
    - Early Stopping
4. A Case Study on MNIST Data with Keras
5. Conclusion
6. Frequently Asked Questions

Regularization is a technique used in machine learning and deep learning to prevent overfitting and improve a model's generalization performance. It involves adding a penalty term to the **loss function** during training.

This penalty discourages the model from becoming too complex or having large parameter values, which helps in controlling the model's ability to fit noise in the training data. Regularization in deep learning **methods** includes L1 and L2 regularization, dropout, early stopping, and more. By applying regularization for deep learning, models become more robust and better at making accurate predictions on unseen data.

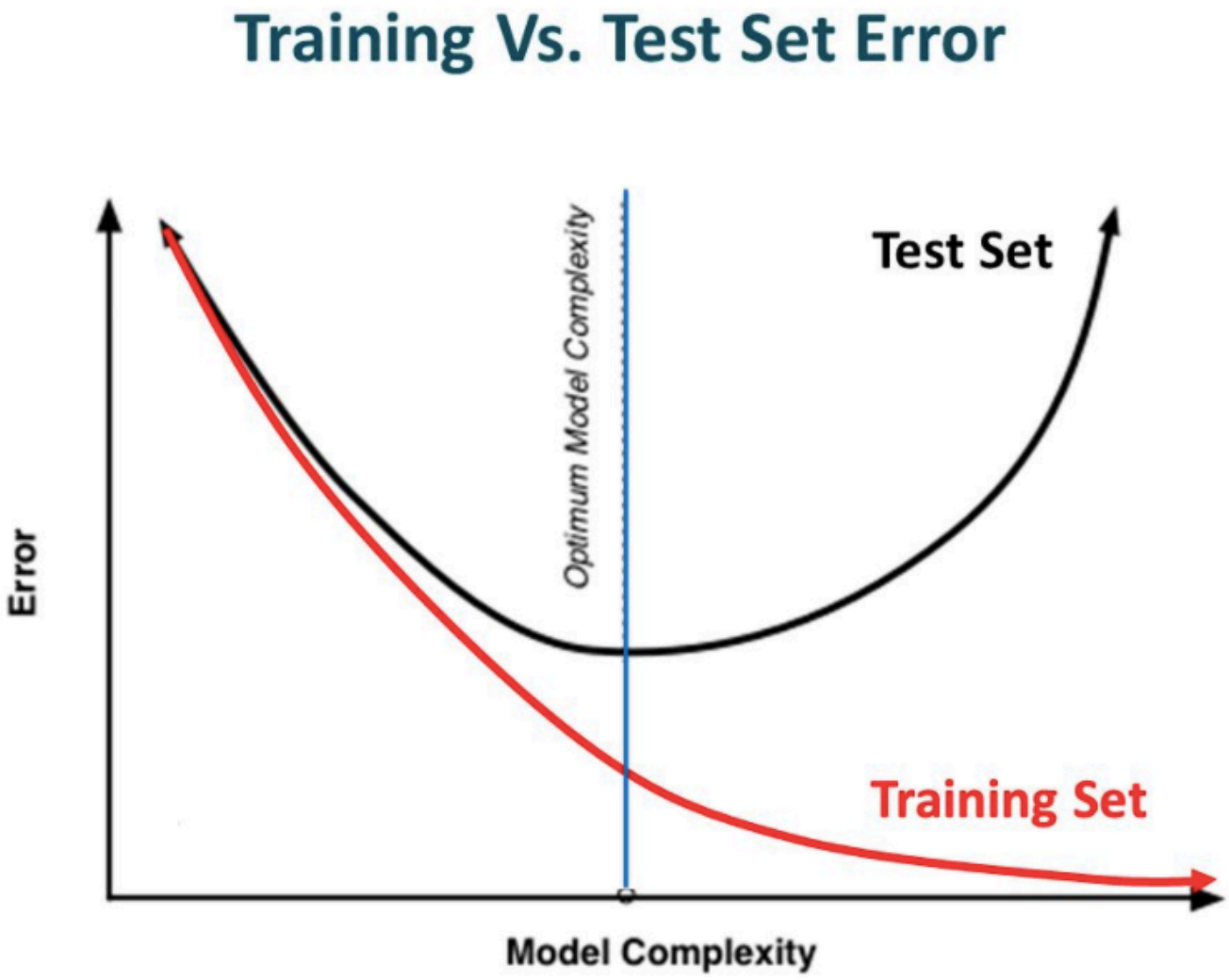
***Also Read:** Regularization in Machine Learning*

Before we deep dive into the topic, take a look at this image:

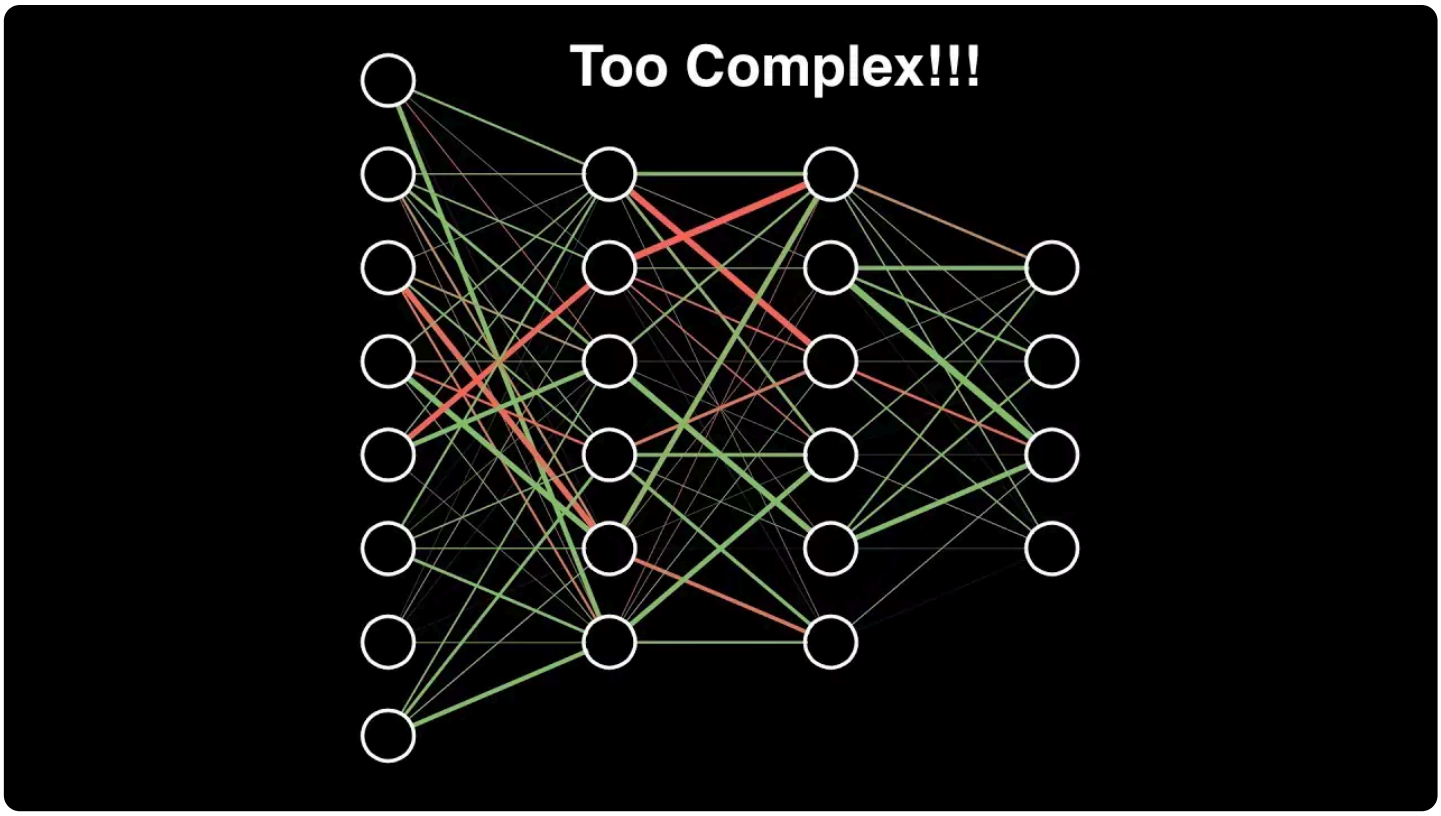


Have you seen this image before? As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, ultimately resulting in poor performance on the unseen data.

In other words, while going toward the right, the complexity of the model increases



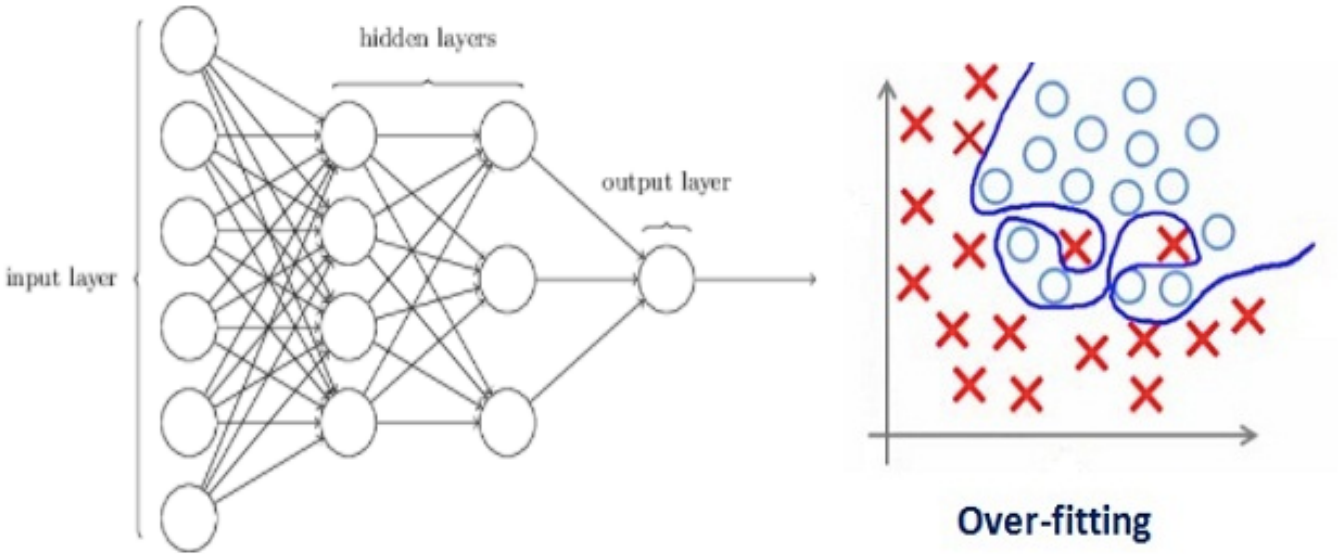
If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting.



Regularization is a technique that modifies the learning algorithm slightly so that the model generalizes better. This, in turn, improves the model's performance on unseen data as well.

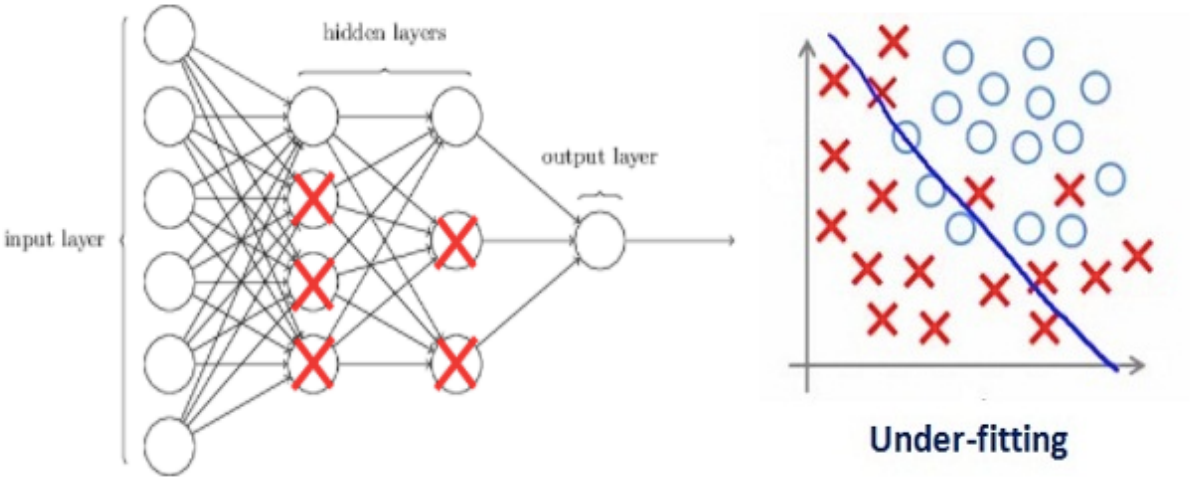
How does Regularization help Reduce Overfitting?

Let's consider a neural network that is overfitting on the training data as shown in the image below:

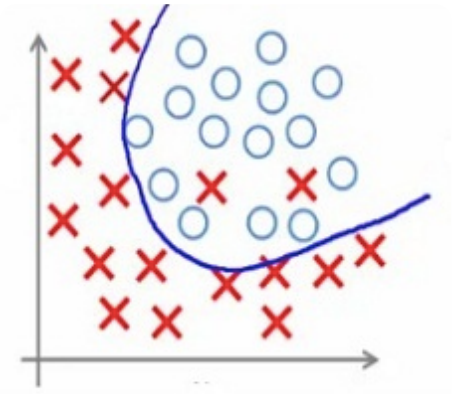


If you have studied the concept of regularization in machine learning, you will have a fair idea that regularization penalizes the coefficients. In deep learning, it penalizes the weight matrices of the nodes.

Assume that our regularization coefficient is so high that some of the weight matrices are nearly equal to zero.



Such a large value of the regularization coefficient is not that useful. We need to optimize the value of the regularization coefficient to obtain a well-fitted model as shown in the image below:



Appropriate-fitting

Also Read: ***How to avoid Over-fitting using Regularization?***

Different Regularization Techniques in Deep Learning

Now that we understand how regularization helps reduce overfitting, we'll learn a few different techniques for applying regularization in deep learning.

L2&L1 Regularization

L1 and L2 are the most common types of regularization deep learning. These update the general cost function by adding another term known as the regularization term.

- *Cost function = Loss (say, binary cross entropy) + Regularization term*

Due to the addition of this regularization term, the values of weight matrices decrease

For L2:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

For L1:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||$$

In L2, we have: ||w||^2 = Σ w_i^2. This is known as ridge regression, where lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero).

In L1, we having: ||w||=Σ |w_i|. In this, we penalize the absolute value of the weights. Unlike L2, the weights may be reduced to zero here. L1 regularization is also called lasso regression. **Hence, it is very useful when we are trying to compress our model. Otherwise, we usually prefer L2 over it.**

In *keras*, we can directly apply regularization for deep learning to any layer using the **regularizers**.

Below is the sample code to apply L2 regularization to a Dense layer:

Copy Code

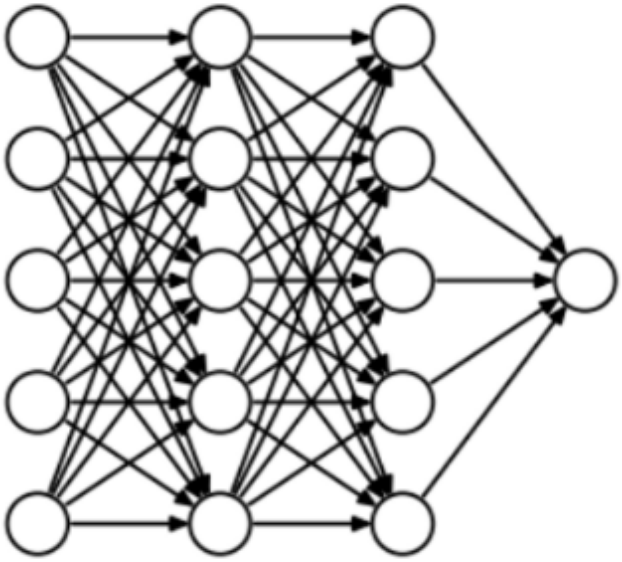
```
from keras import regularizers

model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01)))
```

Note: Here the value 0.01 is the value of regularization parameter, i.e. lambda, which

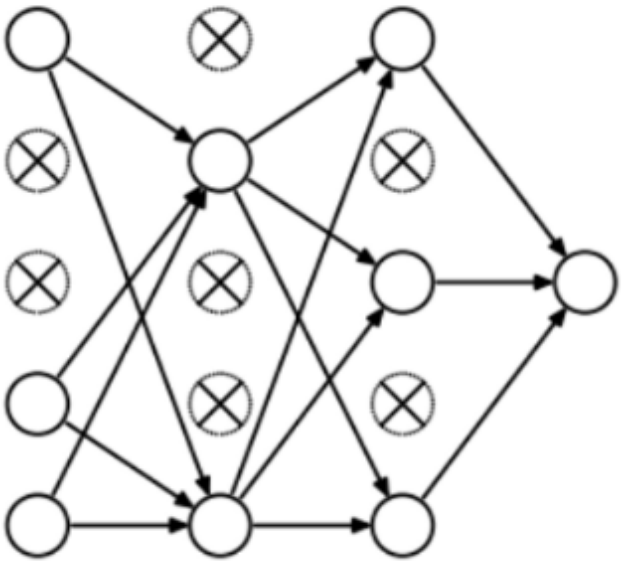
Dropout

This is one of the most interesting types of **regularization techniques**. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.

To understand dropout, let's say our neural network structure is akin to the one shown below:



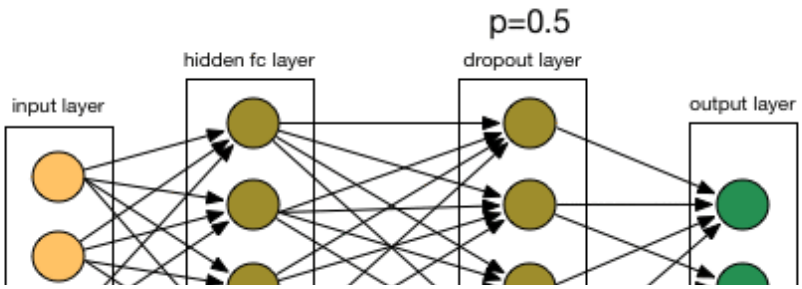
So what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below:



Each iteration has a different set of nodes, which results in a different set of outputs. This can also be thought of as an ensemble technique in machine learning.

Ensemble models usually perform better than a single model as they capture more randomness. Similarly, dropout models also perform better than normal neural network models.

This probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function. As seen in the image above, dropout can be applied to both the hidden layers as well as the input layers.



Due to these reasons, dropout is usually preferred when we have a large neural network structure to introduce more randomness.

In **Keras**, we can implement dropout using the **Keras core layer**. Below is the Python code for it:

Copy Code

```
from keras.layers.core import Dropout

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])
```

As you can see, we have defined 0.25 as the probability of dropping. We can tune it further for better results using the grid search method.

Data Augmentation

The simplest way to reduce overfitting is to increase the training data size. In machine learning, however, increasing the training data size was impossible as the labeled data was too costly.

But now, let's consider we are dealing with images. In this case, there are a few ways of increasing the size of the training data—rotating the image, flipping, scaling, shifting, etc. In the image below, some transformation has been done on the handwritten digits dataset.

This technique is known as data augmentation. It usually provides a big leap in improving the accuracy of the model, and it can be considered a mandatory trick to improve our predictions.

In *keras*, we can perform all of these transformations using **ImageDataGenerator**. It has a big list of arguments that you can use to pre-process your training data.

Below is the sample code to implement it:

Copy Code

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

Early Stopping

Early stopping is a **cross-validation** strategy in which we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model.

In the above image, we will stop training at the dotted line since, after that, our model will start overfitting on the training data.

In *keras*, we can apply early stopping using the **callbacks** function. Below is the sample code for it.

```
from keras.callbacks import EarlyStopping

EarlyStopping(monitor='val_err', patience=5)
```

Copy Code

Here, monitor refers to the quantity that you need to keep track of, and '**val_err**' refers to the validation error.

Patience denotes the number of epochs with no further improvement, after which training stops. For a better understanding, let's look at the above image again. After the dotted line, each epoch will result in a higher validation error value. Therefore, our model will stop 5 epochs after the dotted line (since our patience equals 5) because it sees no further improvement

Note: *After 5 epochs (the value generally defined for patience), the model might start improving again, and the validation error may decrease. Therefore, we need to take extra care while tuning this hyperparameter.*

Also Read: ***Prevent Overfitting Using Regularization Techniques***

A Case Study on MNIST Data with Keras

By this point, you should theoretically understand the different techniques we have

Copy Code

```
%pylab inline

import numpy as np
import pandas as pd
from scipy.misc import imread
from sklearn.metrics import accuracy_score

from matplotlib import pyplot

import tensorflow as tf
import keras

# To stop potential randomness
seed = 128
rng = np.random.RandomState(seed)
```

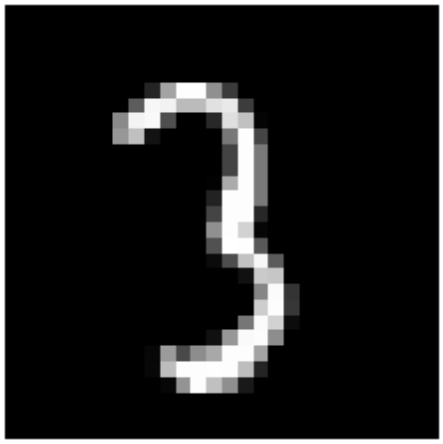
Now, let's load the dataset.

```
root_dir = os.path.abspath('/Users/shubhamjain/Downloads/AV/identify the digits/')
data_dir = os.path.join(root_dir, 'data')
sub_dir = os.path.join(root_dir, 'sub')

## reading train file only
train = pd.read_csv(os.path.join(data_dir, 'Train', 'train.csv'))
train.head()
```

Output:

	filename	label
0	0.png	4
1	1.png	9
2	2.png	1
3	3.png	7
4	4.png	3

```
img = imread(filepath, flatten=True)

pylab.imshow(img, cmap='gray')
pylab.axis('off')
pylab.show()
```



Copy Code

```
#storing images in numpy arrays
temp = []
for img_name in train.filename:
 image_path = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)
 img = imread(image_path, flatten=True)
 img = img.astype('float32')
 temp.append(img)

x_train = np.stack(temp)

x_train /= 255.0
x_train = x_train.reshape(-1, 784).astype('float32')

y_train = keras.utils.np_utils.to_categorical(train.label.values)
```

Create a validation dataset to optimize our model for better scores. We will use a 70:30 train-validation dataset ratio.

First, let's build a simple neural network with 5 hidden layers, each with 500 nodes.

Copy Code

```
# import keras modules
from keras.models import Sequential
from keras.layers import Dense

# define vars
input_num_units = 784
hidden1_num_units = 500
hidden2_num_units = 500
hidden3_num_units = 500
hidden4_num_units = 500
hidden5_num_units = 500
output_num_units = 10

epochs = 10
batch_size = 128

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dense(output_dim=hidden2_num_units, input_dim=hidden1_num_units, activation='relu'),
 Dense(output_dim=hidden3_num_units, input_dim=hidden2_num_units, activation='relu'),
 Dense(output_dim=hidden4_num_units, input_dim=hidden3_num_units, activation='relu'),
 Dense(output_dim=hidden5_num_units, input_dim=hidden4_num_units, activation='relu'),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])
```

Also Read: ***Train-Test-Validation Split in 2024***

Evaluating the Model

Note that we are just running it for 10 epochs. Let's quickly check the performance of our model.

Output:

```
Train on 34300 samples, validate on 14700 samples
Epoch 1/10
34300/34300 [==============================] - 11s - loss: 0.2926 - acc: 0.9118 - val_loss: 0.1854 - val_acc: 0.9444
Epoch 2/10
34300/34300 [==============================] - 10s - loss: 0.1145 - acc: 0.9643 - val_loss: 0.1279 - val_acc: 0.9627
Epoch 3/10
34300/34300 [==============================] - 10s - loss: 0.0792 - acc: 0.9765 - val_loss: 0.1114 - val_acc: 0.9671
Epoch 4/10
34300/34300 [==============================] - 10s - loss: 0.0616 - acc: 0.9814 - val_loss: 0.1213 - val_acc: 0.9671
Epoch 5/10
34300/34300 [==============================] - 10s - loss: 0.0468 - acc: 0.9855 - val_loss: 0.1280 - val_acc: 0.9665
Epoch 6/10
34300/34300 [==============================] - 10s - loss: 0.0379 - acc: 0.9882 - val_loss: 0.1299 - val_acc: 0.9665
Epoch 7/10
34300/34300 [==============================] - 10s - loss: 0.0404 - acc: 0.9874 - val_loss: 0.1203 - val_acc: 0.9692
Epoch 8/10
34300/34300 [==============================] - 11s - loss: 0.0351 - acc: 0.9897 - val_loss: 0.1200 - val_acc: 0.9716
Epoch 9/10
34300/34300 [==============================] - 10s - loss: 0.0263 - acc: 0.9918 - val_loss: 0.1089 - val_acc: 0.9736
Epoch 10/10
34300/34300 [==============================] - 12s - loss: 0.0237 - acc: 0.9931 - val_loss: 0.1308 - val_acc: 0.9699
```

Now, let's try the L2 regularizer over it and check whether it gives better results than a simple neural network model.

Copy Code

```
from keras import regularizers

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu',
 kernel_regularizer=regularizers.l2(0.0001)),
 Dense(output_dim=hidden2_num_units, input_dim=hidden1_num_units, activation='relu',
 kernel_regularizer=regularizers.l2(0.0001)),
 Dense(output_dim=hidden3_num_units, input_dim=hidden2_num_units, activation='relu',
 kernel_regularizer=regularizers.l2(0.0001)),
 Dense(output_dim=hidden4_num_units, input_dim=hidden3_num_units, activation='relu',
 kernel_regularizer=regularizers.l2(0.0001)),
 Dense(output_dim=hidden5_num_units, input_dim=hidden4_num_units, activation='relu',
 kernel_regularizer=regularizers.l2(0.0001)),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
Train on 34300 samples, validate on 14700 samples
Epoch 1/10
34300/34300 [==============================] - 15s - loss: 0.4862 - acc: 0.9092 - val_loss: 0.3283 - val_acc: 0.9531
Epoch 2/10
34300/34300 [==============================] - 14s - loss: 0.2887 - acc: 0.9635 - val_loss: 0.2822 - val_acc: 0.9644
Epoch 3/10
34300/34300 [==============================] - 14s - loss: 0.2388 - acc: 0.9738 - val_loss: 0.2534 - val_acc: 0.9696
Epoch 4/10
34300/34300 [==============================] - 12s - loss: 0.2010 - acc: 0.9806 - val_loss: 0.2418 - val_acc: 0.9678
Epoch 5/10
34300/34300 [==============================] - 13s - loss: 0.1807 - acc: 0.9830 - val_loss: 0.2470 - val_acc: 0.9655
Epoch 6/10
34300/34300 [==============================] - 13s - loss: 0.1707 - acc: 0.9826 - val_loss: 0.2458 - val_acc: 0.9669
Epoch 7/10
34300/34300 [==============================] - 12s - loss: 0.1524 - acc: 0.9865 - val_loss: 0.2125 - val_acc: 0.9718
Epoch 8/10
34300/34300 [==============================] - 14s - loss: 0.1365 - acc: 0.9886 - val_loss: 0.2254 - val_acc: 0.9697
Epoch 9/10
34300/34300 [==============================] - 13s - loss: 0.1392 - acc: 0.9863 - val_loss: 0.2070 - val_acc: 0.9695
Epoch 10/10
34300/34300 [==============================] - 13s - loss: 0.1209 - acc: 0.9912 - val_loss: 0.1952 - val_acc: 0.9739
```

Note that the value of **lambda** is equal to 0.0001. Great! We just obtained an accuracy that is greater than our previous NN model.

Now, let's try the L1 regularization technique.

Copy Code

```
## l1

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu',
 kernel_regularizer=regularizers.l1(0.0001)),
 Dense(output_dim=hidden2_num_units, input_dim=hidden1_num_units, activation='relu',
 kernel_regularizer=regularizers.l1(0.0001)),
 Dense(output_dim=hidden3_num_units, input_dim=hidden2_num_units, activation='relu',
 kernel_regularizer=regularizers.l1(0.0001)),
 Dense(output_dim=hidden4_num_units, input_dim=hidden3_num_units, activation='relu',
 kernel_regularizer=regularizers.l1(0.0001)),
 Dense(output_dim=hidden5_num_units, input_dim=hidden4_num_units, activation='relu',
 kernel_regularizer=regularizers.l1(0.0001)),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
trained_model_5d = model.fit(x_train, y_train, nb_epoch=epochs, batch_size=batch_size, va
```

```
Train on 34300 samples, validate on 14700 samples
Epoch 1/10
34300/34300 [==============================] - 13s - loss: 2.8190 - acc: 0.9019 - val_loss: 1.7565 - val_acc: 0.9455
Epoch 2/10
34300/34300 [==============================] - 13s - loss: 1.3347 - acc: 0.9541 - val_loss: 1.0242 - val_acc: 0.9565
Epoch 3/10
34300/34300 [==============================] - 12s - loss: 0.8235 - acc: 0.9628 - val_loss: 0.7109 - val_acc: 0.9554
Epoch 4/10
34300/34300 [==============================] - 15s - loss: 0.5809 - acc: 0.9661 - val_loss: 0.5493 - val_acc: 0.9580
Epoch 5/10
34300/34300 [==============================] - 15s - loss: 0.4472 - acc: 0.9721 - val_loss: 0.4364 - val_acc: 0.9637
Epoch 6/10
34300/34300 [==============================] - 14s - loss: 0.3720 - acc: 0.9754 - val_loss: 0.3725 - val_acc: 0.9659
Epoch 7/10
34300/34300 [==============================] - 16s - loss: 0.3208 - acc: 0.9778 - val_loss: 0.3323 - val_acc: 0.9690
Epoch 8/10
34300/34300 [==============================] - 14s - loss: 0.2893 - acc: 0.9790 - val_loss: 0.3114 - val_acc: 0.9688
Epoch 9/10
34300/34300 [==============================] - 16s - loss: 0.2619 - acc: 0.9820 - val_loss: 0.3068 - val_acc: 0.9673
Epoch 10/10
34300/34300 [==============================] - 15s - loss: 0.2418 - acc: 0.9828 - val_loss: 0.2921 - val_acc: 0.9667
```

This doesn't show any improvement over the previous model. Let's jump to the dropout technique.

Copy Code

```
## dropout

from keras.layers.core import Dropout
model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden2_num_units, input_dim=hidden1_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden3_num_units, input_dim=hidden2_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden4_num_units, input_dim=hidden3_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden5_num_units, input_dim=hidden4_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
trained_model_5d = model.fit(x_train, y_train, nb_epoch=epochs, batch_size=batch_size, va
```

```
 Dropout(0.25),
 Dense(output_dim=hidden2_num_units, input_dim=hidden1_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden3_num_units, input_dim=hidden2_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden4_num_units, input_dim=hidden3_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden5_num_units, input_dim=hidden4_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])


model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc
trained_model_5d = model.fit(x_train, y_train, nb_epoch=epochs, batch_size=batch_size, va
```

Copy Code

Output:

```
Train on 34300 samples, validate on 14700 samples
Epoch 1/10
34300/34300 [==============================] - 14s - loss: 0.4231 - acc: 0.8652 - val_loss: 0.1756 - val_acc: 0.9461
Epoch 2/10
34300/34300 [==============================] - 12s - loss: 0.1771 - acc: 0.9487 - val_loss: 0.1526 - val_acc: 0.9573
Epoch 3/10
34300/34300 [==============================] - 13s - loss: 0.1296 - acc: 0.9625 - val_loss: 0.1111 - val_acc: 0.9683
Epoch 4/10
34300/34300 [==============================] - 13s - loss: 0.1055 - acc: 0.9691 - val_loss: 0.1137 - val_acc: 0.9688
Epoch 5/10
34300/34300 [==============================] - 12s - loss: 0.0924 - acc: 0.9739 - val_loss: 0.1184 - val_acc: 0.9677
Epoch 6/10
34300/34300 [==============================] - 12s - loss: 0.0821 - acc: 0.9755 - val_loss: 0.1059 - val_acc: 0.9712
Epoch 7/10
34300/34300 [==============================] - 13s - loss: 0.0714 - acc: 0.9794 - val_loss: 0.1103 - val_acc: 0.9699
Epoch 8/10
34300/34300 [==============================] - 13s - loss: 0.0635 - acc: 0.9812 - val_loss: 0.0965 - val_acc: 0.9766
Epoch 9/10
34300/34300 [==============================] - 13s - loss: 0.0588 - acc: 0.9825 - val_loss: 0.1133 - val_acc: 0.9718
Epoch 10/10
34300/34300 [==============================] - 15s - loss: 0.0572 - acc: 0.9826 - val_loss: 0.1085 - val_acc: 0.9729
```

Not bad! Dropout also gives us a little improvement over our simple NN model.

Now, let’s try data augmentation.

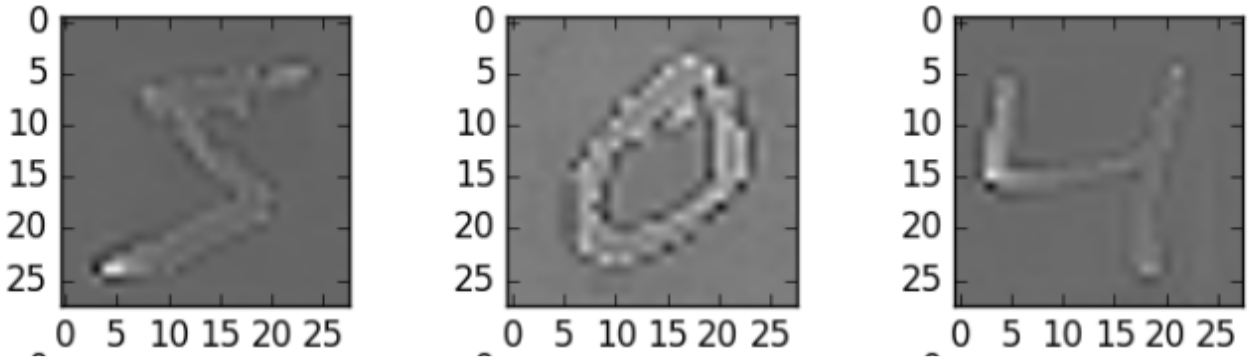
```
temp = []
for img_name in train.filename:
 image_path = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)
 img = imread(image_path, flatten=True)
 img = img.astype('float32')
 temp.append(img)

x_train = np.stack(temp)

X_train = x_train.reshape(x_train.shape[0], 1, 28, 28)

X_train = X_train.astype('float32')
```

Now, fit the training data in order to augment.

Copy Code

```
# fit parameters from data
datagen.fit(X_train)
```

Here, I have used **zca_whitening** as the argument, which highlights the outline of each digit as shown in the image below:



Copy Code

```
## splitting
y_train = keras.utils.np_utils.to_categorical(train.label.values)
split_size = int(x_train.shape[0]*0.7)

x_train, x_test = X_train[:split_size], X_train[split_size:]
```

Copy Code

```
## structure using dropout
from keras.layers.core import Dropout
model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden2_num_units, input_dim=hidden1_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden3_num_units, input_dim=hidden2_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden4_num_units, input_dim=hidden3_num_units, activation='relu'),
 Dropout(0.25),
 Dense(output_dim=hidden5_num_units, input_dim=hidden4_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])


model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc
trained_model_5d = model.fit(x_train, y_train, nb_epoch=epochs, batch_size=batch_size, va
```

Copy Code

Output:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 20s - loss: 0.1222 - acc: 0.9669 - val_loss: 0.0893 - val_acc: 0.9761
Epoch 2/10
60000/60000 [==============================] - 20s - loss: 0.0949 - acc: 0.9741 - val_loss: 0.0948 - val_acc: 0.9746
Epoch 3/10
60000/60000 [==============================] - 19s - loss: 0.0849 - acc: 0.9764 - val_loss: 0.0793 - val_acc: 0.9780
Epoch 4/10
60000/60000 [==============================] - 19s - loss: 0.0705 - acc: 0.9803 - val_loss: 0.0825 - val_acc: 0.9794
Epoch 5/10
60000/60000 [==============================] - 19s - loss: 0.0671 - acc: 0.9809 - val_loss: 0.0765 - val_acc: 0.9782
Epoch 6/10
60000/60000 [==============================] - 20s - loss: 0.0620 - acc: 0.9831 - val_loss: 0.0797 - val_acc: 0.9817
Epoch 7/10
60000/60000 [==============================] - 18s - loss: 0.0563 - acc: 0.9843 - val_loss: 0.0746 - val_acc: 0.9828
Epoch 8/10
60000/60000 [==============================] - 18s - loss: 0.0492 - acc: 0.9860 - val_loss: 0.0847 - val_acc: 0.9800
Epoch 9/10
60000/60000 [==============================] - 18s - loss: 0.0515 - acc: 0.9855 - val_loss: 0.0880 - val_acc: 0.9784
Epoch 10/10
60000/60000 [==============================] - 20s - loss: 0.0503 - acc: 0.9856 - val_loss: 0.0614 - val_acc: 0.9835
```

```
from keras.callbacks import EarlyStopping

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
trained_model_5d = model.fit(x_train, y_train, nb_epoch=epochs, batch_size=batch_size, va
 , callbacks = [EarlyStopping(monitor='val_acc', patience=2)])
```

Copy Code

Output:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 20s - loss: 0.0479 - acc: 0.9869 - val_loss: 0.0881 - val_acc: 0.9808
Epoch 2/10
60000/60000 [==============================] - 19s - loss: 0.0486 - acc: 0.9875 - val_loss: 0.0756 - val_acc: 0.9833
Epoch 3/10
60000/60000 [==============================] - 21s - loss: 0.0467 - acc: 0.9875 - val_loss: 0.0788 - val_acc: 0.9795
Epoch 4/10
60000/60000 [==============================] - 20s - loss: 0.0398 - acc: 0.9886 - val_loss: 0.0821 - val_acc: 0.9818
Epoch 5/10
60000/60000 [==============================] - 20s - loss: 0.0417 - acc: 0.9887 - val_loss: 0.0968 - val_acc: 0.9813
```

Our model stops after only 5 iterations as the validation accuracy does not improve. It gives good results in cases where we run it for a larger number of epochs, so it's a technique to optimize the number of epochs.

Also Read: ***Techniques To Prevent Overfitting In Neural Networks***

Conclusion

I hope you now understand regularization deep learning techniques in deep learning and the different methods required to implement them, not just for linear regression models but also for complex neural networks like CNN. Regularization is crucial when training deep learning models on large datasets to prevent overfitting and improve generalization. You can effectively employ techniques like L1/L2 regularization, dropout, data augmentation, and early stopping using gradient descent optimization

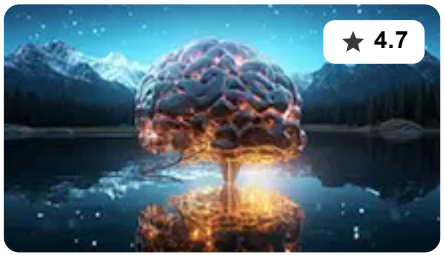
learning methods when dealing with a deep learning task, as they will help you expand your horizons, better understand the topic, and build more robust machine learning algorithms.

Hope you like the article and better understand regularization in deep learning. Regularisation techniques, such as L1 and L2 regularization, dropout, and data augmentation, prevent overfitting, and improve model performance.

shubham.jain

Deep Learning | Image | Intermediate | Python | Python | Technique | Unstructured Data

Free Courses



Generative AI - A Way of Life

Explore Generative AI for beginners: create text and images, use top AI tools, learn practical skills, and ethics.

Getting Started with Large Language Models

Master Large Language Models (LLMs) with this course, offering clear guidance in NLP and model training made simple.

Building LLM Applications using Prompt Engineering

This free course guides you on building LLM apps, mastering prompt engineering, and developing chatbots with enterprise data.

Improving Real World RAG Systems: Key Challenges & Practical Solutions

Explore practical solutions, advanced retrieval strategies, and agentic RAG systems to improve context, relevance, and accuracy in AI-driven applications.

Master MS Excel for data analysis with key formulas, functions, and LookUp tools in this comprehensive course.

RECOMMENDED ARTICLES

Ridge and Lasso Regression in Python

Regularization in Machine Learning

Dropout Regularization in Deep Learning

Complete Guide to Prevent Overfitting in Neural...

Complete Guide to Prevent Overfitting in Neural...

Understanding Overfitting in ConvNets

Prevent Overfitting Using Regularization Techni...

Tutorial: Optimizing Neural Networks using Kera...

Study of Regularization Techniques of Linear Mo...

Responses From Readers

What are your thoughts?...

Submit reply

Pramod

Thanks a lot bro. Immensely helpful.

Chetan

Could we achieve the same validation accuracy with much lesser architecture ? If so, then our Hyperparameters (especially the L2 lambda) must be quite high (say 0.05). Isn't it ? Or, is there any other way of combating this ??

Florido Meacci

Thank you for the great article. I'm trying to differentiate

Frequently Asked Questions

Q1. What is regularization in deep learning?

such as L1 and L2 regularization, dropout, and batch normalization help control model complexity and improve neural network generalization to unseen data.

Q2. What is L1 regularization and L2 regularization?

Q3. What is dropout in neural network?

Q4.What is L1 and L2 regularization technique?

Flagship Courses

GenAI Pinnacle Program | AI/ML BlackBelt Courses

Free Courses

Generative AI | Large Language Models | Building LLM Applications using Prompt Engineering | Building Your first RAG System using LlamaIndex | Stability.AI | MidJourney | Building Production Ready RAG systems using LlamaIndex | Building LLMs for Code | Deep Learning | Python | Microsoft Excel | Machine Learning | Decision Trees | Pandas for Data Analysis | Ensemble Learning | NLP | NLP using Deep Learning | Neural Networks | Loan Prediction Practice Problem | Time Series Forecasting | Tableau | Business Analytics

Generative AI Tools and Techniques

GANs | VAEs | Transformers | StyleGAN | Pix2Pix | Autoencoders | GPT | BERT | Word2Vec | LSTM | Attention Mechanisms | Diffusion Models | LLMs | SLMs | StyleGAN | Encoder Decoder Models | Prompt Engineering | LangChain | LlamaIndex | RAG | Fine-tuning | LangChain AI Agent | Multimodal Models | RNNs | DCGAN | ProGAN | Text-to-Image Models | DDPM | Document Question Answering | Imagen | T5 (Text-to-Text Transfer Transformer) | Seq2seq Models | WaveNet | Attention Is All You Need (Transformer Architecture)

Popular GenAI Models

Llama 3.1 | Llama 3 | Llama 2 | GPT 4o Mini | GPT 4o | GPT 3 | Claude 3 Haiku | Claude 3.5 Sonnet | Phi 3.5 | Phi 3 | Mistral Large 2 | Mistral NeMo | Mistral-7b | Gemini 1.5 Pro | Gemini Flash 1.5 | Bedrock | Vertex AI | DALL.E | Midjourney | Stable Diffusion

Data Science Tools and Techniques

Python | R | SQL | Jupyter Notebooks | TensorFlow | Scikit-learn | PyTorch | Tableau | Apache Spark | Matplotlib | Seaborn | Pandas | Hadoop | Docker | Git | Keras | Apache Kafka | AWS | NLP | Random Forest | Computer Vision | Data Visualization | Data Exploration | Big Data | Common Machine Learning Algorithms | Machine Learning