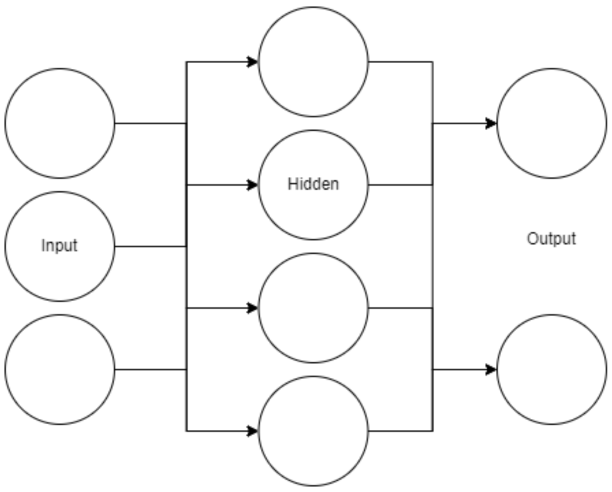
Machine Learning – in contrast from traditional explicit coding, the goal of machine learning is to use learning algorithms over data to train a model for specific tasks

Deep Learning – subset under Machine Learning, like biological neurons, artificial neurons receive inputs, process them and pass the output to the next layer. The complex neural network approach results at better feature extraction (such as edges and textures), scalable and allows for non-linear complex relationships.

Common deep learning architectures and algorithms are, CNN architecture which excels at processing grid-like data such as images and videos or Transformer type models which utilize NLP and self-attention mechanism for their operation, and Backpropagation algorithm which helps optimize models by reinforcing good weight & bias and discouraging bad weight & bias.

The most basic models are built sequentially, the model is built in layers where each layer is responsible for different needs. At minimum, a model requires at least 3 Layers, the input, output and hidden layers.



The input layer receives the raw data (This can be image, audio, video, text etc.) each neuron in the input layer corresponds to a feature of the data (Example 1, Image 28x28 = 784 pixels, therefore 784 neurons, each neuron receives a pixel. Example 2, Large text file, split into vectors, each neuron receives a vector.) it will then pass it on.

The hidden layers consist of a minimum of 1 layer, they process the data, add a weight and bias to them and preferably pass the result to an activation function. This is where patterns and complex relationships are learned. Each hidden layer captures different layers of abstraction. It will then pass it on.

The output layer is responsible for producing a result, this can be in the form of text, audio, video or an image (Example 1, in classification, each neuron in the output layer can represent a class and the output will be the predicted class. Example 2, when expecting a binary response its preferable to use binary cross-entropy for a binary result, 1 or 0).

An important aspect of deep learning is introducing non-linear functions (or activation functions). These functions help the model understand more complex variable relationships, they allow for a more flexible and adaptive model, they can handle complexities much better and handle data outliers.

Linear models can only compare variables in a straightforward way, each variable is considered isolated unlike nonlinear models.

Example for Linearly and nonlinearly models, a cat dressed as a dog needs to be identified, a linear model would see the tail of a dog, and perhaps the size or fur and base their predictions on that. A nonlinear model would be able to see the same number of variables, but it'll be able to use, compare and understand the relationship between them and understand it's a cat dressed like a dog.

Data pre-processing in model training is the process of preparing the data for the pipeline, resizing it, normalization (usually between 1 and 0), ensuring proper labeling and formatting (Such as one-hot encoding) for the model. Removing unnecessary data such as dropping color channels and handling missing / bad data.

After the pre-processing, it's recommended to include in the pipeline data augmentation.

Data augmentation helps to reduce overfitting and improves generalization with introduction of randomness to the data. It is done by making changes to image such as rotation, image flipping, changing pixels and more. These augmentations make the model more robust to variations in real-world data.

After pre-processing and augmentation it's a good practice to evaluate Batch size and number of epochs, Batch size refers to the number of samples (In CNN, images) to be processed by the model before updating the weights (Batch size of 50 in CNN would be 50 images).

An Epoch is one full pass through the entire training dataset (for example, 5 Epoches would mean the training data passes through the model 5 times) Smaller batch size (32,64) can lead to a more stable learning but might take longer to converge while larger batch sizes (128,256) can make faster progress but be more unstable.

More epochs can help the model learn more data, but too many epochs can lead to overfitting.

Early stopping prevents overfitting by halting training when progress stagnates, saving time and compute.

It is common to use optimization algorithms when model training, some of the more common ones are –

- SGD – Updates model weights iteratively with randomly selected subset of data (mini-batches), Basic optimizer.
- Adam – Versatile, works well with most models.
- RMSprop – effective for RNNs, good with non-stationary objectives.
- Adagrad – useful for sparse data or text-based models like NLP.
- Momentum – enhances SGD by adding momentum to help accelerate gradients, useful for deep networks.
- AdaDelta – works well with models that require minimal hyperparameter tuning.

One of the biggest issues for model training is overfitting, overfitting refers to models that preform well on training data but very poorly on unseen new data.

Another method to help counteract overfitting is Regularization.

There are several common regularization techniques that can prevent overfitting such as –

- L1/L2 Regularization – adds penalty to the loss function to prevent large weights.
- Dropout – randomly disables neurons during training forcing the model to rely on different subset of neurons.
- Early stopping is also considered a method of Regularization.

One of the most important aspects of AI model training is understanding it's a randomized trial and error process. This leads us to Hyperparameter tuning, the process of adjusting parameters that control the behavior of the model but aren't directly from the training data such as –

- Learning Rate – determines how fast the model learns (High – Risky & Fast, Low – Stable & Slow
- Batch size
- Number of epochs
- Dropout rate (How much of the network is randomly "Dropped out" during training).

Even slightly changing these numbers can significantly alter the model's performance.

After playing around with hyperparameters a good method of understanding if our model is accurate is via validation accuracy, this metric tells us how accurate our model is on new unseen before data.

In general, data is split into 2 distinct categories, Training data and validation data.

Validation gives us a realistic estimate of the model's effectiveness on real-world data.

There are several methods of validation such as –

- Hold-out Validation – simple split into training and validation sets (80-20 usually).
- Cross validation – the data is split into multiple subsets and the model Is trained and validated multiple times on different splits (Such as K-fold cross validation).
- Stratified Validation – useful for imbalanced datasets.

Several common metrics for validation are –

- Accuracy – % of correct predictions.
- Precision/Recall/F1 Score – useful for imbalanced datasets.
- Loss – the objective function the model is minimizing.

Another less common issue is underfitting, this likely occurs due to the model being too simple to find patterns in training data.

When data goes into the pipeline via the layers it is called forward pass, the data passes through all the layers (primarily the hidden layers) where different features are detected such as edges, sentiment, patterns etc. to eventually reach the output layer.

In each layer (Example input to hidden, hidden to output) the model calculates the weighted sum of the inputs for each neuron.

Z = W1X1 + W2X2 + … + WNXN + B

Where –

- Z = Weighted Sum
- W1, W2 … WN are weights
- X1, X2 … XN are features
- B = Bias

And the result is passed through an activation function (Non-linear function) such as ReLU or Sigmoid.

After these calculations are done (from input layer to hidden and hidden to output) the predicted output is compared to the actual output using a loss function (Such as cross-entropy for classification or mean squared error for regression)

The loss function output tells us how “Wrong” the prediction was (the difference between the expected to the prediction).

The most important part of training a model comes next, the backpropagation(Backward pass) is used to adjust weights to reduce the loss function in the future, this is done by calculating the gradient of the loss function with respect to each weight in the network.

Simply put, the backpropagation calculates how much each neuron contributed to the bad result.

This means if a particular weight had a more impactful hit for the accuracy, it can be adjusted more heavily compared to other weights.

Once all gradients are calculated, the weights are updated using gradient’s descent (Or a variant such as SGD, Adam etc.)

Wnew = Wold – N * (D / DX)

Where –

- Wold = Old weights.
- N = Learning rate
- (D / DX) = the gradient of loss with respect to weight.

Bias is also calculated similarly.

This process repeats itself for each training batch.

There are several issues which could occur during Backpropagation –

- Vanishing Gradient – the gradient is extremely small and as a result the early layers of the network (Closer to the input) learn very slowly or not at all leading to slow learning model (Much more common with Tanh and Sigmoid).
- Exploding Gradient – the gradient is extremely large causing significant changes which could lead to bad adjustments.

There are many activation functions, and each has their own properties and advantages –

- Sigmoid = outputs value between 1 to 0, good for binary classification.
- ReLU – outputs 0 for negative numbers and the input value for positive inputs. It is computationally efficient but can get stuck on negative inputs.
- Leaky ReLU – output positive values for positive inputs and small negatives slopes for negative inputs. Prevents being stuck on negative numbers.
- Tanh – outputs values range between -1 and 1 (Useful in RNNs).
- Softmax – produces a probability distribution (Output sum to 1), useful for multi-class classification tasks.
- Swish – a smoother, nonlinear output often used in deep networks and performs better compared to ReLU.

During backpropagation like we mentioned earlier several issues could arise, to solve these we can either use activation functions such as ReLU or a method called Batch Normalization, Batch normalization helps stabilize the inputs before they reach the neuron (Before weights and bias are applied), this ensures they remain at a stable range.

The normalization is independent and happens at every batch of data.

Input is flowing into the neuron, weights and bias are applied, it goes through Batch normalization and then into a non-linear activation function.

This ensures a normalized range of outputs and can prevent the issues.

Teaching a model comes in various methods, you can either teach the model using data in a traditional way or other methods such as Transfer Learning.

Transfer Learning is a method in which you can train your model using other pre-trained models, this can be highly effective and save time and computational resources.

For example, if you had a model which needed to identify Trucks on roads, instead of teaching your model using thousands of images you can use a pre-trained model which already learned large datasets for finding the best weight and bias to fit your specific task.

- ResNet – CNN, good for image recognition, object detection and feature extraction.
- GPT – Transformer, effective at Text generation, Conversational AI, Content summarization and creative writing.
- YOLO – CNN, ideal for real-time computer vision applications.
- MobileNet - Lightweight ResNet / VGG.

Interpretability is the concept of understanding the behaviour of models, or more specifically, their decision making and predications.

This becomes increasingly important with larger more complex models.

At a larger scope, it also important for trust and transparency, debugging, regulation and accountability and bias detection.

There are several methods to introduce interpretability to models –

- Feature Importance – helps rank input feature based on their impact on the model's output.
- LIME – for example, explaining why a CNN classified an image by highlighting the most influential pixels.
- SHAP – based on co-operative game theory, for individual prediction explanation.
- Saliency Maps – useful for CNN, highlighting influential regions.

The methods above and more can assist Engineers in creating a superior more understandable model, clearer debug paths and help users understand the reason for the prediction.

When training a model, it is always important to weigh the trade-offs between interpretability and model complexity.

GPU Acceleration is another crucial part of building deep learning models, this is due to the complexity and size of large deep learning models.

While CPUs could do the trick, GPUs are more efficient and offer higher memory bandwidth leading to better learning and inference completion times.

These can be further improved through dedicated libraries such as Tensorflow or PyTorch which utilize specialized GPU cores (Namely "CUDA").

GPUs excel at parallelizing operations like matrix multiplication and tensor computations, which are fundamental in deep learning tasks. This is especially important when scaling models to millions or billions of parameters.

For example, training a model on ResNet-50 on a CPU could theoretically take days, a GPU at the same class could drop it to hours.

This can be further apparent when using multi-GPUs or distributed GPU architectures.

Data Preprocessing and Augmentation can be done via Keras's pipeline, in the example below we utilize the "ImageDataGenerator" function to create the preprocess pipeline.

In the example we also use folders to split images for training and validation using flow_from_directory function in keras.

We also specify Batch Size in the ImageDataGenerator function.

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Create a separate ImageDataGenerator for training data with augmentation
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Rescale pixel values (normalization)
    rotation_range=40,  # Rotate images randomly up to 40 degrees
    width_shift_range=0.2,  # Shift the image horizontally by up to 20%
    height_shift_range=0.2,  # Shift the image vertically by up to 20%
    shear_range=0.2,  # Shear angle
    zoom_range=0.2,  # Randomly zoom into images
    horizontal_flip=True,  # Randomly flip images horizontally
    fill_mode='nearest',  # Strategy to fill pixels that are lost after
shifting/rotation
    validation_split=0.2  # Reserve 20% of data for validation
)

# Create a separate ImageDataGenerator for validation data without
augmentation (only rescaling)
validation_datagen = ImageDataGenerator(
    rescale=1./255,  # Only rescale validation data, no augmentation
    validation_split=0.2  # Reserve 20% of data for validation
)

# Load the training data with augmentation from the 'Data' folder
train_generator = train_datagen.flow_from_directory(
    'Data/',  # Path to the main data folder
    target_size=(128, 128),  # Resize all images to 128x128 pixels
    batch_size=32,  # Number of images to process in a batch
    class_mode='binary',  # Binary classification (e.g., cats/dogs)
    subset='training'  # Use this for training data (80%)
)

# Load the validation data without augmentation from the 'Data' folder
validation_generator = validation_datagen.flow_from_directory(
    'Data/',  # Path to the main data folder
    target_size=(128, 128),  # Same image size as training
    batch_size=32,  # Number of images in each batch
    class_mode='binary',  # Binary classification (e.g., cats/dogs)
    subset='validation'  # Use this for validation data (20%)
)
```

This example code for preprocess aimed at Transformer based models

```
# Import required libraries
import pandas as pd
from transformers import BertTokenizer

# Step 1: Load Text Data from a CSV File
# Assuming a CSV file with two columns: 'text' (input text) and 'label'
(target label)
data = pd.read_csv('path_to_your_csv_file.csv')

# Step 2: Load the BERT Tokenizer
# 'bert-base-uncased' is the pre-trained model we're using
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Step 3: Tokenization, Padding, and Truncation
# Function to preprocess text data using the tokenizer
def preprocess_texts(texts, max_length=128):
    return tokenizer(
        texts.tolist(),  # List of input texts
        padding=True,  # Pad the sequences to the max length
        truncation=True,  # Truncate sequences longer than max_length
        max_length=max_length,  # Max sequence length
        return_tensors='tf'  # Return TensorFlow tensors (can also be 'pt' for
PyTorch)
    )

# Apply the tokenizer to the 'text' column in your DataFrame
encoded_inputs = preprocess_texts(data['text'])

# Step 4: Check Output (Optional)
# View the tokenized input IDs and attention masks
print(encoded_inputs['input_ids'])  # Numeric representation of the tokens
print(encoded_inputs['attention_mask'])  # Mask for padding tokens

# The processed inputs now contain:
# - 'input_ids': the tokenized and padded/truncated text
# - 'attention_mask': mask indicating the actual text vs. padding
```

Keras doesn’t have a functionality to handle errors, to prevent them we can use the following code before the pipeline.

The example below shows a function that runs before the Pipeline to catch bad images and delete them from their folder.

```
import os
from PIL import Image

def check_and_remove_corrupted_images(directory):
    for root, _, files in os.walk(directory):
        for file in files:
            try:
                img_path = os.path.join(root, file)
                img = Image.open(img_path)
                img.verify()  # Verify if the image is intact
            except (IOError, SyntaxError) as e:
                print(f"Corrupted image found and removed: {file}")
                os.remove(img_path)  # Remove the corrupted image
```

In the example below, an error catcher for CSV files aimed for Transformer based models.

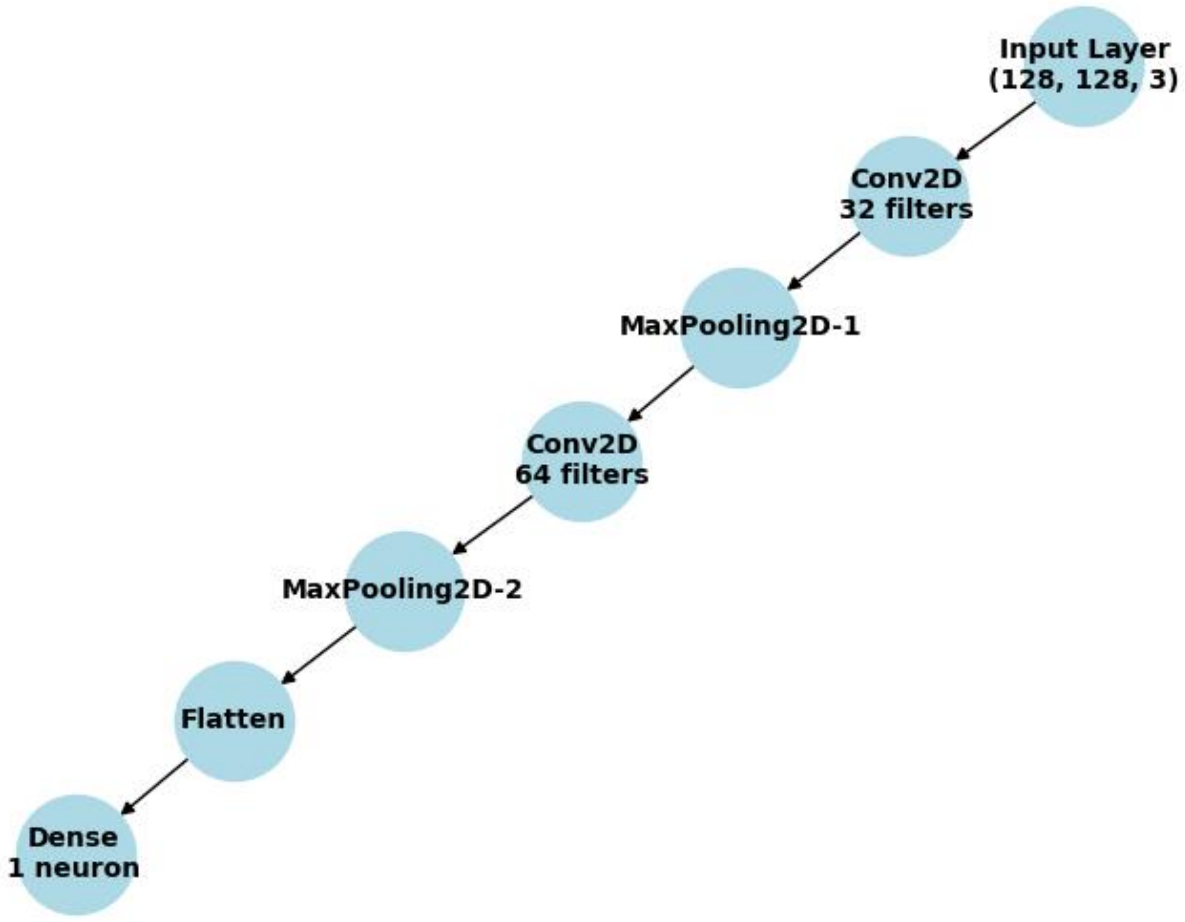
```
import pandas as pd

# Function to check and handle missing data in a CSV file
def check_and_handle_missing_data(csv_file, strategy='drop', output_file=None):
    # Step 1: Load the CSV into a DataFrame
    data = pd.read_csv(csv_file)

    # Step 2: Check for missing values
    print("Missing values before handling:")
    print(data.isnull().sum())  # Print the number of missing values for each column

    if strategy == 'drop':
        # Step 3: Drop rows with missing values
        cleaned_data = data.dropna()
        print(f"Dropped {data.shape[0] - cleaned_data.shape[0]} rows with missing 
values.")

    elif strategy == 'mean':
        # Fill missing values with the mean of each column
        cleaned_data = data.fillna(data.mean())
        print("Filled missing values with the mean.")

    elif strategy == 'median':
        # Fill missing values with the median of each column
        cleaned_data = data.fillna(data.median())
        print("Filled missing values with the median.")

    elif strategy == 'mode':
        # Fill missing values with the mode of each column
        cleaned_data = data.fillna(data.mode().iloc[0])
        print("Filled missing values with the mode.")

    else:
        print("Invalid strategy. Please choose 'drop', 'mean', 'median', or 'mode'.")
        return None

    # Step 4: Optionally save the cleaned DataFrame back to a CSV file
    if output_file:
        cleaned_data.to_csv(output_file, index=False)
        print(f"Cleaned data saved to {output_file}.")

    # Return the cleaned DataFrame
    return cleaned_data

cleaned_data = check_and_handle_missing_data('path_to_your_csv_file.csv', 
strategy='mean', output_file='cleaned_data.csv')
```

The next section of codes includes the Epoch size, Early stopping mechanism, Optimization Algorithm implementation, model Architecture,

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.regularizers import l2
from keras.callbacks import EarlyStopping

# Define your model architecture with regularization
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    # Convolutional layer
    layers.MaxPooling2D(2, 2),  # Max pooling layer
    layers.Conv2D(64, (3, 3), activation='relu'),
    # Second convolutional layer
    layers.MaxPooling2D(2, 2),  # Second max pooling layer
    layers.Flatten(),  # Flatten the input for the dense layer
    layers.Dense(1, activation='sigmoid')
    # Output layer for binary classification
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Set up Early Stopping
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    verbose=1,
    restore_best_weights=True
)

# Fit the model with early stopping
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=50,
    callbacks=[early_stopping]
)
```

Dense – Fully connected to every neuron of previous layer (Flatten at this code)
Conv2D – Feature extractor.
MaxPooling2D – reduces spatial dimensions (Height and Width) of the future maps.
Flatten – converts 2D feature maps from previous MaxPooling2D into 1D vector.

The model above can be visualized using matplotlib and networkx for better understanding of how layers forward pass and backpropagate.



Hyper Parameter Tuning, in Keras, when passing hyperparameters to the function that hosts the model creation function (subfunction “tf.keras.models.sequential” for example), Keras automatically knows how to assign them into the model.

```python
import tensorflow as tf

def create_model(learning_rate, dropout_rate, optimizer):
    # Build the Sequential model
    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(128, 128, 3)),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(dropout_rate),  # Hyperparameter: Dropout rate
        tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer
    ])

    # Compile the model using the passed optimizer
    model.compile(optimizer=optimizer, loss='binary_crossentropy',
metrics=['accuracy'])

    return model

# Example usage:
X = tf.keras.optimizers.Adam(learning_rate=0.001)  # You can set any optimizer
model = create_model(learning_rate=0.001, dropout_rate=0.4, optimizer=X)

# Print the model summary
model.summary()
```

In the example above, the hyperparameters are the dropout rate, learning rate and optimization algorithm, they can all be configured as variables (Algorithm for example as X) to be changed at ease.

At the model.compile stage, it is possible to add other metrics for Evaluation, such as Recall, Precision, Accuracy etc... as they’re already pre-defined.

```python
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy', 'Precision', 'Recall'])
```

Activation Functions can be placed in every layer to add nonlinearity at the model building function like the following example –

```
tf.keras.layers.Conv2D(64, (3, 3), activation='relu')
```

where the activation can be changed to pre-defined activation functions such as Tanh or ReLU.

Similar to Activation functions, Batch Normalization can also be added after layers to reduce issues we discussed earlier such as exploding gradient and vanishing gradient.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(128, 
128, 3)),
    tf.keras.layers.BatchNormalization(),  # Add Batch Normalization here
    tf.keras.layers.MaxPooling2D(2, 2),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.BatchNormalization(),  # Add Batch Normalization here as 
well
    tf.keras.layers.MaxPooling2D(2, 2),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.BatchNormalization(),  # Add Batch Normalization before 
the output layer
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

While it's possible to alter some Batch Normalization configurations it's generally accepted that the default is good enough, The default configurations do not need to be explicitly configured as the Keras' functionality handles it.

Transfer Learning allows us to also utilize pre-trained models, we can use them with Keras to train our own models.

```python
import tensorflow as tf
from tensorflow.keras.applications import ResNet50 # Pre-trained ResNet50 
model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

# Step 1: Load the pre-trained ResNet50 model without the top layers
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(128, 
128, 3))

# Step 2: Freeze the layers of the base model
base_model.trainable = False

# Step 3: Create a new model with the pre-trained base model
model = Sequential([
    base_model,  # Pre-trained model as the base
    Flatten(),  # Flatten the output of the base model
    Dense(128, activation='relu'),  # New fully connected layer
    Dense(1, activation='sigmoid')  # Output layer for binary classification
])


# Unfreeze the last few layers of the base model for fine-tuning
for layer in base_model.layers[-10:]:  # Unfreeze the last 10 layers
    layer.trainable = True
```

in the example above, we utilize ResNet50 for training our model with a pre-trained model.

When talking about freezing layers, it refers to the concept of preserving the Pre-trained model's weights and bias, for example –

Model has 15 layers, by using a pre-trained model and freezing all my model's layers we basically use ResNet50 for teaching the model on our data, this can be useful at the start due to the nature of ResNet50 and the amount of data it already learnt.

This means our model mimicks ResNet50's Weight and Bias.

For optimization, we can unfreeze our last few layers to specifically look for patterns in our data, these last few layers will be able to adjust their weights and bias to our specific data.

This means our last few layers can be optimized better for the task we want to do.

GPU Acceleration can be done through Keras, in the example below it'll be shown how to find your GPUs.

It's important to note that Keras requires specific needs such as WSL environment for specific CUDNN & Cuda drivers for it to work.

```
import tensorflow as tf

# List all the GPUs detected by TensorFlow
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    print(f"GPUs available: {len(gpus)}")
    for gpu in gpus:
        print(gpu)
else:
    print("No GPUs detected")
```

the example above showcases how to detect GPUs in our local system while the example below shows how we can utilize GPUs in the network.

```
import tensorflow as tf

strategy = tf.distribute.MultiWorkerMirroredStrategy()

with strategy.scope():
    # Define and compile your model here
    Pass
```

Another method of finding the best hyperparameters is using Keras-tuner library which can be installed via pip, in this method we can automatically search for the best parameters without having to manually do it.

```
import tensorflow as tf
from kerastuner import HyperModel, RandomSearch
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
from tensorflow.keras.applications import ResNet50

# Define the base ResNet50 model
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224,
224, 3))
base_model.trainable = False  # Freeze the layers

# Define the HyperModel for tuning
class CNNHyperModel(HyperModel):
    def build(self, hp):
        model = tf.keras.models.Sequential([
            base_model,  # Pre-trained ResNet50 as the base
            GlobalAveragePooling2D(),  # Reduces feature maps to a vector

            # Define Dense layer hyperparameters
            Dense(hp.Int('dense_units', min_value=32, max_value=256, step=32),
activation='relu'),

            # Dropout with tunable rate
            Dropout(hp.Float('dropout_rate', min_value=0.0, max_value=0.5,
step=0.1)),

            # Output layer for binary classification
            Dense(1, activation='sigmoid’])
        # Compile the model with a tunable learning rate
        model.compile(
            optimizer=tf.keras.optimizers.Adam(
                hp.Float('learning_rate', min_value=1e-4, max_value=1e-2,
sampling='LOG')
            ),
            loss='binary_crossentropy',
            metrics=['accuracy'])
        return model

# RandomSearch tuner setup
tuner = RandomSearch(
    CNNHyperModel(),
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=1,
    directory='hyper_tuning'
)
```

```
# Start the hyperparameter search
tuner.search(train_generator, validation_data=validation_generator, epochs=10)

# Get the best hyperparameters and model
best_hps = tuner.get_best_hyperparameters(1)[0]
best_model = tuner.get_best_models(1)[0]

# Print the best hyperparameters
print(f"Best learning rate: {best_hps.get('learning_rate')}")
print(f"Best dense_units: {best_hps.get('dense_units')}")
print(f"Best dropout_rate: {best_hps.get('dropout_rate')}")
```