|  |  |
| --- | --- |
| NAME | MAHESWARAN.S |
| REG.NO. | 420121104031 |
| DEPARTMENT | CSE |
| YEAR | III |
| COLLEGE NAME | AKT MCET |
| GROUP | IBM GROUP-5 |

**MEASURE ENERGY CONSUMPTION**

PHASE-3

INNOVATION

. In this phase you need to put your design into innovation to solve the problem.

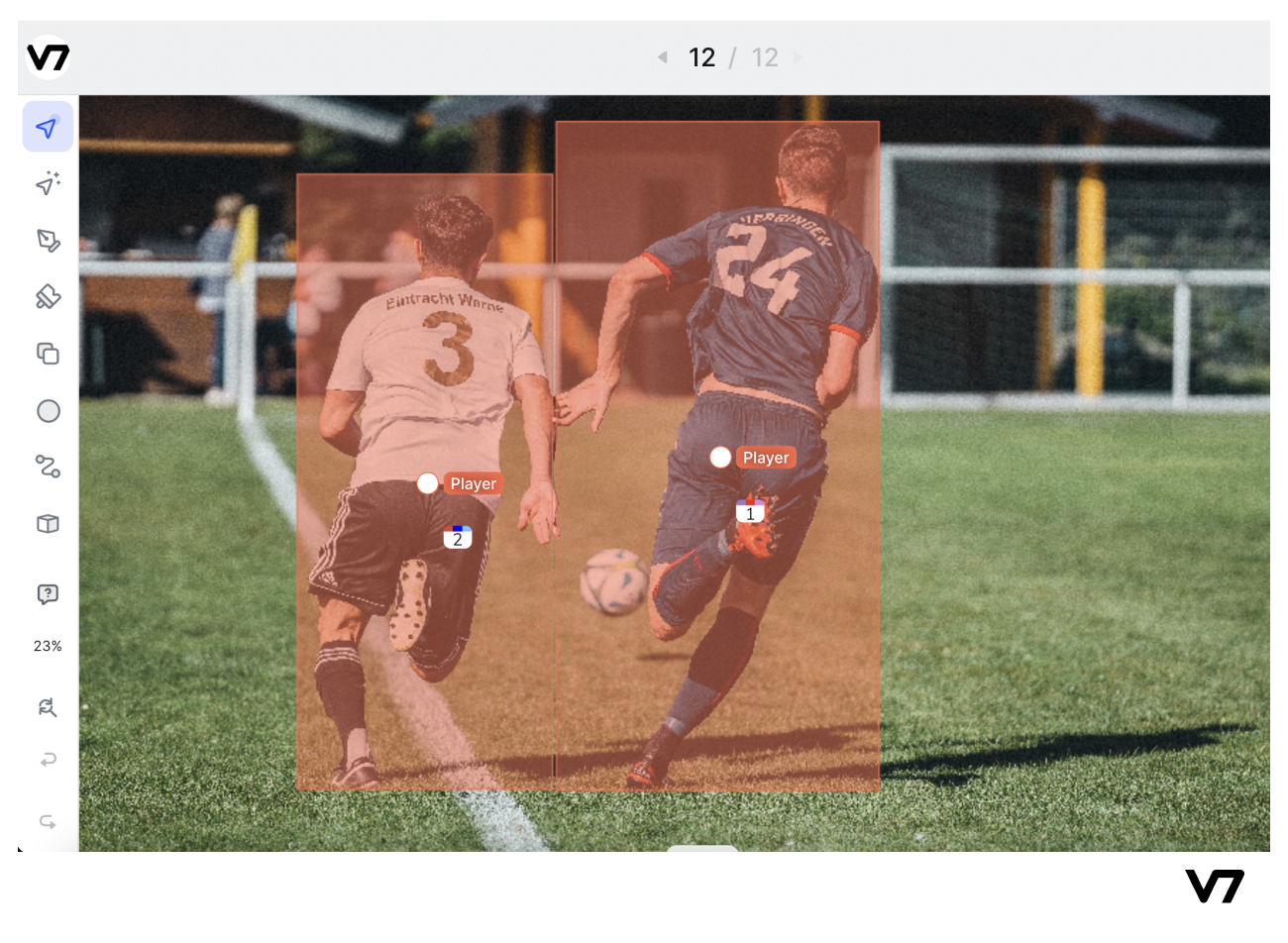
.Explain detail the complete steps that will be taken by you to put your design that you thought of in previous phase in to transformation.

.Create a document around it and share the same for assessment.

Module 8: OBJECT DETECTION WITH YOLO

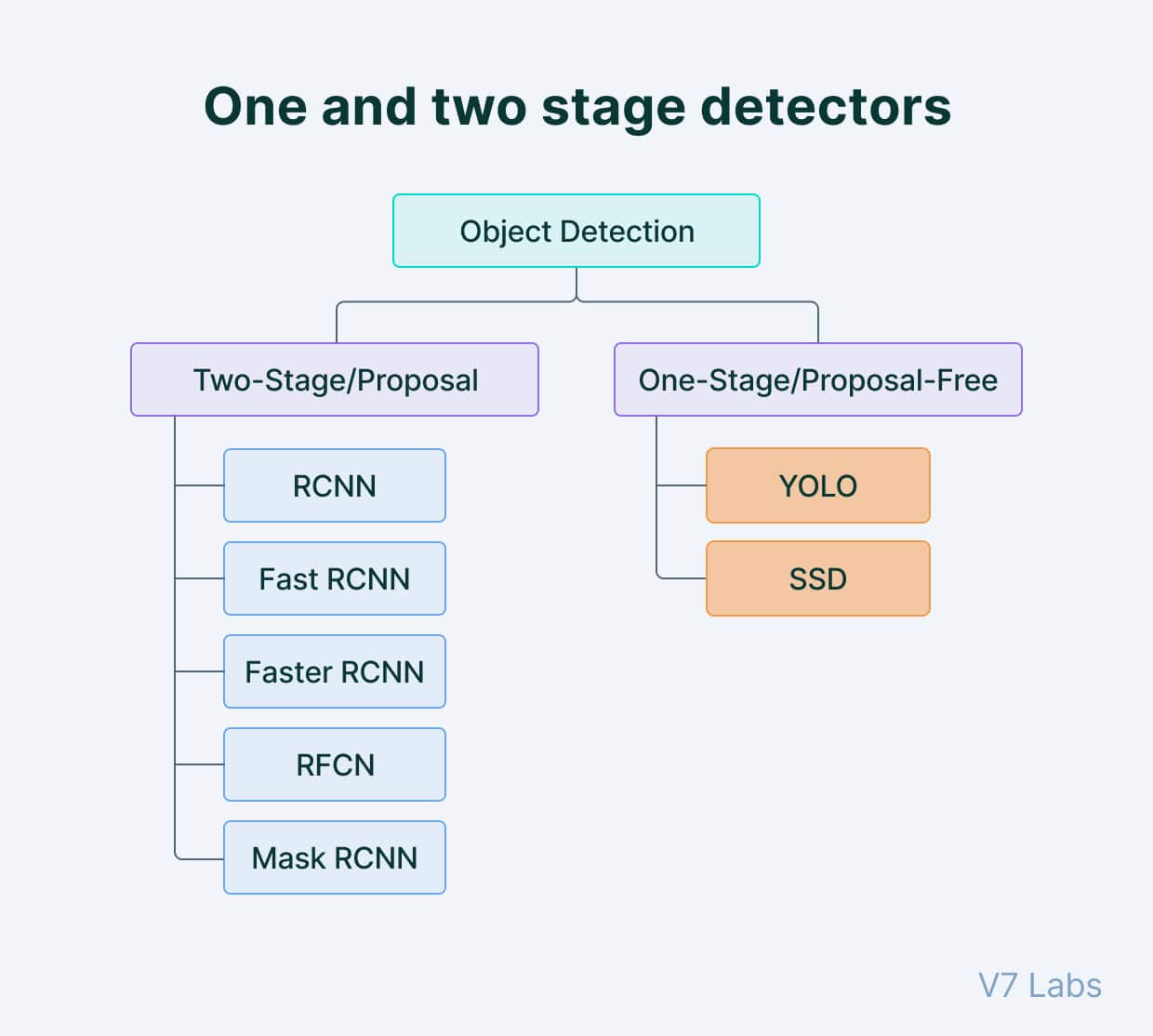
## OBJECT DETECTION

[Object detection](https://www.v7labs.com/blog/object-detection-guide) is a [computer vision](https://www.v7labs.com/blog/what-is-computer-vision) task that involves identifying and locating objects in images or videos. It is an important part of many applications, such as surveillance, self-driving cars, or robotics. Object detection algorithms can be divided into two main categories: single-shot detectors and two-stage detectors.



One of the earliest successful attempts to address the object detection problem using deep learning was the R-CNN (Regions with CNN features) model, developed by Ross Girshick and his team at Microsoft Research in 2014. This model used a combination of region proposal algorithms and [convolutional neural networks (CNNs)](https://www.v7labs.com/blog/convolutional-neural-networks-guide) to detect and localize objects in images.

Object detection algorithms are broadly classified into two categories based on how many times the same input image is passed through a network.



### ****Single-shot object detection****

Single-shot object detection uses a single pass of the input image to make predictions about the presence and location of objects in the image. It processes an entire image in a single pass, making them computationally efficient.

However, single-shot object detection is generally less accurate than other methods, and it’s less effective in detecting small objects. Such algorithms can be used to detect objects in real time in resource-constrained environments.

YOLO is a single-shot detector that uses a fully convolutional neural network (CNN) to process an image. We will dive deeper into the YOLO model in the next section.

### ****Two-shot object detection****

Two-shot object detection uses two passes of the input image to make predictions about the presence and location of objects. The first pass is used to generate a set of proposals or potential object locations, and the second pass is used to refine these proposals and make final predictions. This approach is more accurate than single-shot object detection but is also more computationally expensive.

Overall, the choice between single-shot and two-shot object detection depends on the specific requirements and constraints of the application.

Generally, single-shot object detection is better suited for real-time applications, while two-shot object detection is better for applications where accuracy is more important.

### ****Object detection models performance evaluation metrics****

To determine and compare the predictive performance of different object detection models, we need standard quantitative metrics.

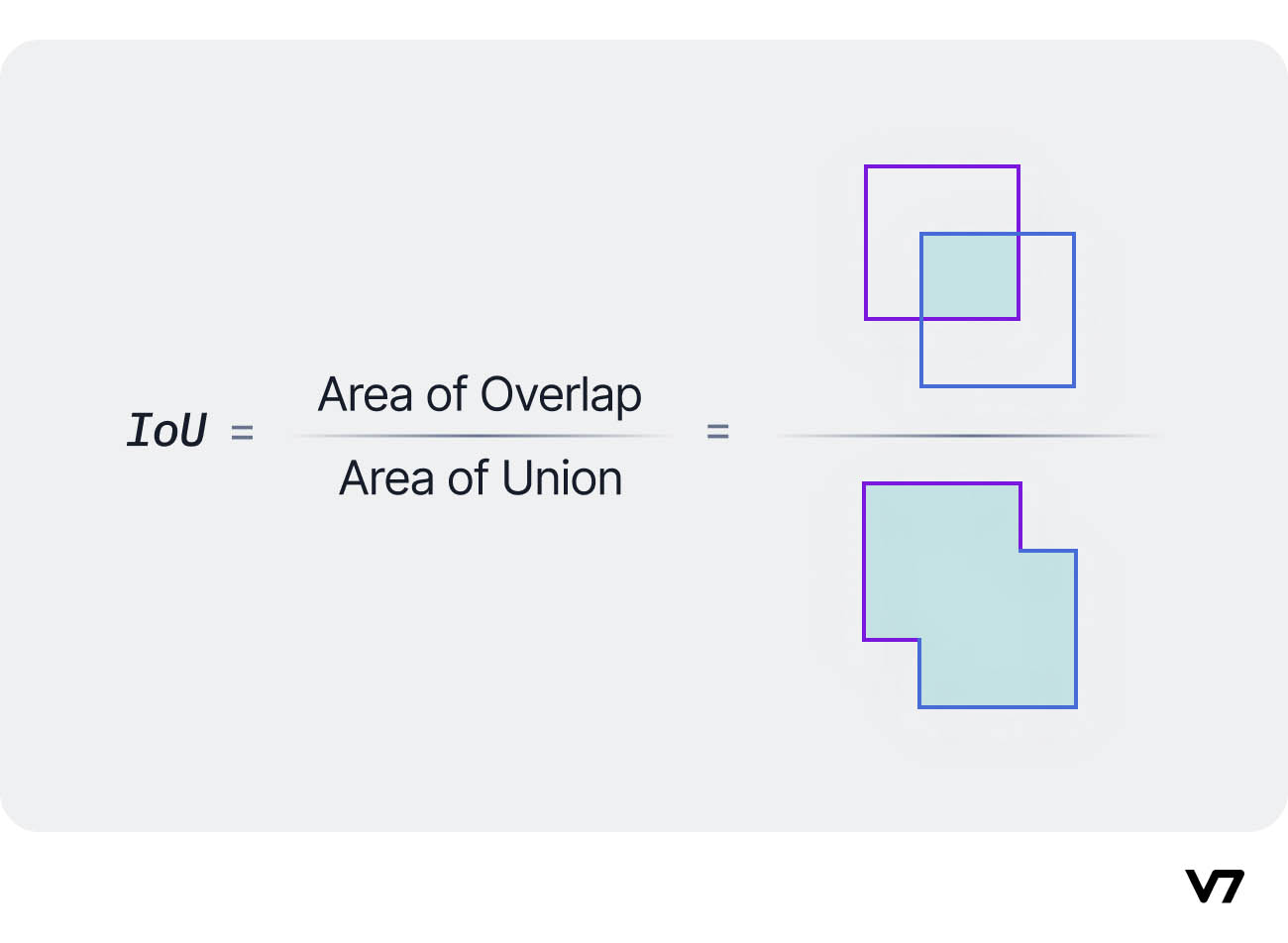
The two most common evaluation metrics are Intersection over Union (IoU) and the Average Precision (AP) metrics.

#### ****Intersection over Union (IoU)****

Intersection over Union is a popular metric to measure localization accuracy and calculate localization errors in object detection models.

To calculate the IoU between the predicted and the ground truth [bounding boxes](https://www.v7labs.com/blog/bounding-box-annotation), we first take the intersecting area between the two corresponding bounding boxes for the same object. Following this, we calculate the total area covered by the two bounding boxes— also known as the “Union” and the area of overlap between them called the “Intersection.”

The intersection divided by the Union gives us the ratio of the overlap to the total area, providing a good estimate of how close the prediction bounding box is to the original bounding box.



#### ****Average Precision (AP)****

Average Precision (AP) is calculated as the area under a precision vs. recall curve for a set of predictions.

Recall is calculated as the ratio of the total predictions made by the model under a class with a total of existing labels for the class. Precision refers to the ratio of true positives with respect to the total predictions made by the model.

[Recall and precision](https://www.v7labs.com/blog/precision-vs-recall-guide) offer a trade-off that is graphically represented into a curve by varying the classification threshold. The area under this precision vs. recall curve gives us the Average Precision per class for the model. The average of this value, taken over all classes, is called mean Average Precision (mAP).

##### ****💡 Read more:**** [Mean Average Precision (mAP) Explained: Everything You Need to Know](https://www.v7labs.com/blog/mean-average-precision)

In object detection,  precision and recall aren’t used for class predictions. Instead, they serve as predictions of boundary boxes for measuring the decision performance. An IoU value > 0.5. is taken as a positive prediction, while an IoU value < 0.5 is a negative prediction.

## What is YOLO?

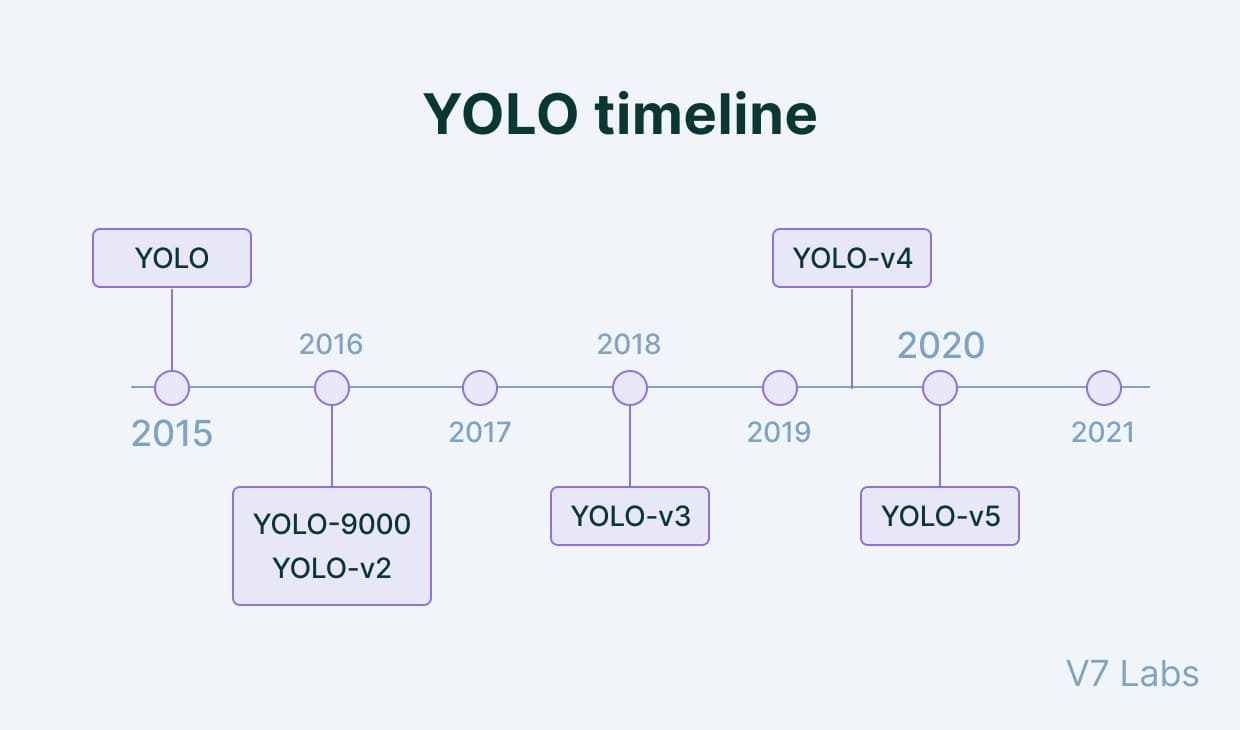
You Only Look Once (YOLO) proposes using an end-to-end [neural network](https://www.v7labs.com/blog/neural-network-architectures-guide) that makes predictions of bounding boxes and class probabilities all at once. It differs from the approach taken by previous object detection algorithms, which repurposed classifiers to perform detection.

Following a fundamentally different approach to object detection, YOLO achieved state-of-the-art results, beating other real-time object detection algorithms by a large margin.

While algorithms like Faster [RCNN](https://www.v7labs.com/blog/recurrent-neural-networks-guide) work by detecting possible regions of interest using the Region Proposal Network and then performing recognition on those regions separately, YOLO performs all of its predictions with the help of a single fully connected layer.

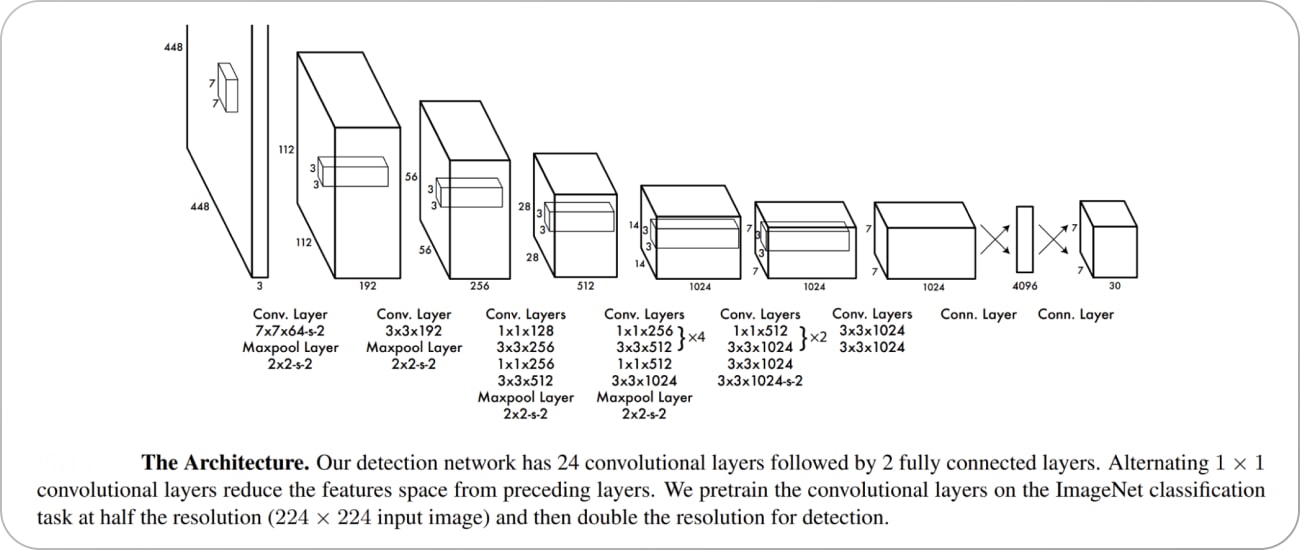
Methods that use Region Proposal Networks perform multiple iterations for the same image, while YOLO gets away with a single iteration.

Several new versions of the same model have been proposed since the initial release of YOLO in 2015, each building on and improving its predecessor. Here's a timeline showcasing YOLO's development in recent years.



## How does YOLO work? YOLO Architecture

The [YOLO algorithm](https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf) takes an image as input and then uses a simple deep convolutional neural network to detect objects in the image. The architecture of the CNN model that forms the backbone of YOLO is shown below.



The first 20 convolution layers of the model are pre-trained using ImageNet by plugging in a temporary average pooling and fully connected layer. Then, this pre-trained model is converted to perform detection since previous research showcased that adding convolution and connected layers to a pre-trained network improves performance. YOLO’s final fully connected layer predicts both class probabilities and bounding box coordinates.

YOLO divides an input image into an S × S grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and how accurate it thinks the predicted box is.

YOLO predicts multiple bounding boxes per grid cell. At training time, we only want one bounding box predictor to be responsible for each object. YOLO assigns one predictor to be “responsible” for predicting an object based on which prediction has the highest current IOU with the ground truth. This leads to specialization between the bounding box predictors. Each predictor gets better at forecasting certain sizes, aspect ratios, or classes of objects, improving the overall recall score.

One key technique used in the YOLO models is ****non-maximum suppression (NMS)****. NMS is a post-processing step that is used to improve the accuracy and efficiency of object detection. In object detection, it is common for multiple bounding boxes to be generated for a single object in an image. These bounding boxes may overlap or be located at different positions, but they all represent the same object. NMS is used to identify and remove redundant or incorrect bounding boxes and to output a single bounding box for each object in the image.

Now, let us look into the improvements that the later versions of YOLO have brought to the parent model.

##### 💡****Pro tip:****Take a look at this list of[65+ Best Free Datasets for Machine Learning](https://www.v7labs.com/blog/best-free-datasets-for-machine-learning) to find relevant data for [training your models.](https://www.v7labs.com/training)

### ****YOLO v2****

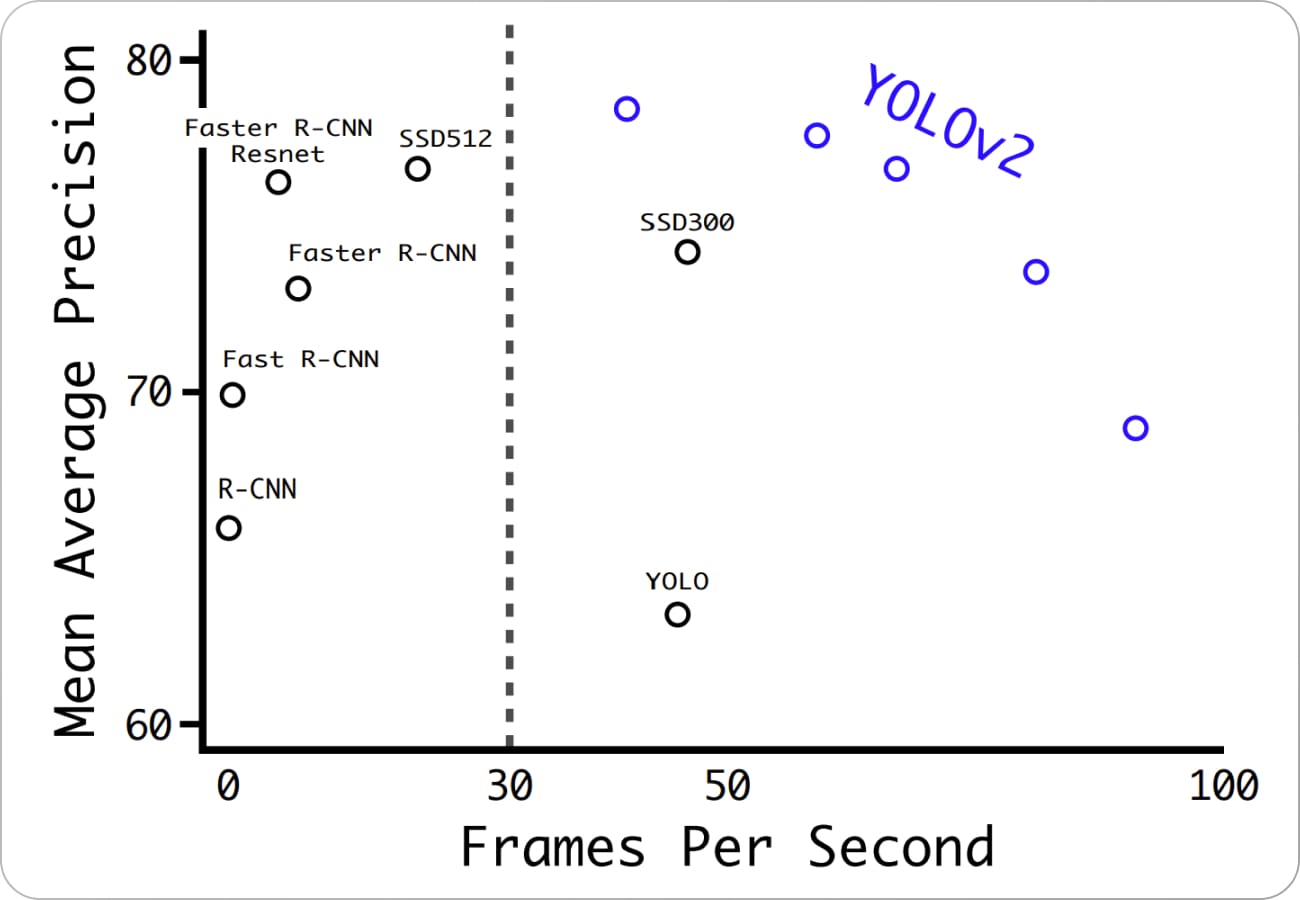
[YOLO v2](https://arxiv.org/pdf/1612.08242), also known as YOLO9000, was introduced in 2016 as an improvement over the original YOLO algorithm. It was designed to be faster and more accurate than YOLO and to be able to detect a wider range of object classes. This updated version also uses a different CNN backbone called Darknet-19, a variant of the VGGNet architecture with simple progressive convolution and pooling layers.

One of the main improvements in YOLO v2 is the use of anchor boxes. Anchor boxes are a set of predefined bounding boxes of different aspect ratios and scales. When predicting bounding boxes, YOLO v2 uses a combination of the anchor boxes and the predicted offsets to determine the final bounding box. This allows the algorithm to handle a wider range of object sizes and aspect ratios.

Another improvement in YOLO v2 is the use of batch normalization, which helps to improve the accuracy and stability of the model. YOLO v2 also uses a multi-scale training strategy, which involves training the model on images at multiple scales and then averaging the predictions. This helps to improve the detection performance of small objects.

YOLO v2 also introduces a new [loss function](https://www.v7labs.com/blog/pytorch-loss-functions) better suited to object detection tasks. The loss function is based on the sum of the squared errors between the predicted and ground truth bounding boxes and class probabilities.

The results obtained by YOLO v2 compared to the original version and other contemporary models are shown below.



Source: [Paper](https://arxiv.org/pdf/1612.08242" \t "https://www.v7labs.com/blog/_blank)

### ****YOLO v3****

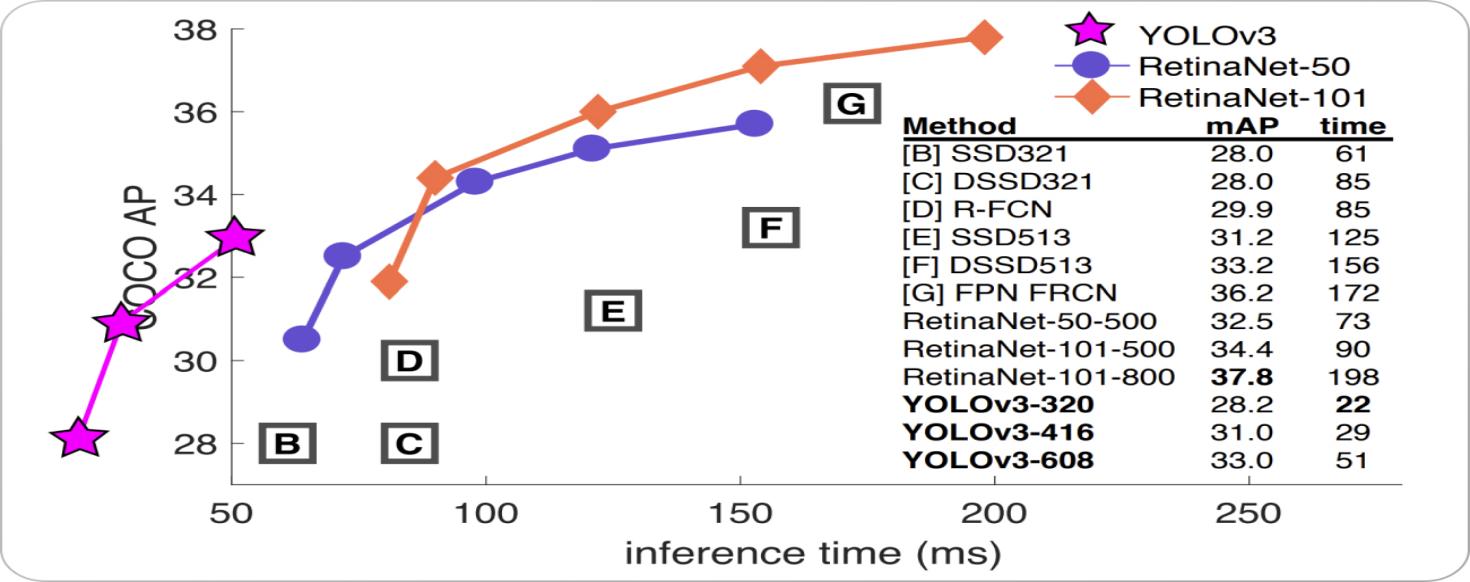
[YOLO v3](https://arxiv.org/pdf/1804.02767.pdf" \t "https://www.v7labs.com/blog/_blank) is the third version of the YOLO object detection algorithm. It was introduced in 2018 as an improvement over YOLO v2, aiming to increase the accuracy and speed of the algorithm.

One of the main improvements in YOLO v3 is the use of a new CNN architecture called Darknet-53. Darknet-53 is a variant of the ResNet architecture and is designed specifically for object detection tasks. It has 53 convolutional layers and is able to achieve state-of-the-art results on various object detection benchmarks.

Another improvement in YOLO v3 are anchor boxes with different scales and aspect ratios. In YOLO v2, the anchor boxes were all the same size, which limited the ability of the algorithm to detect objects of different sizes and shapes. In YOLO v3 the anchor boxes are scaled, and aspect ratios are varied to better match the size and shape of the objects being detected.

YOLO v3 also introduces the concept of "feature pyramid networks" (FPN). FPNs are a CNN architecture used to detect objects at multiple scales. They construct a pyramid of feature maps, with each level of the pyramid being used to detect objects at a different scale. This helps to improve the detection performance on small objects, as the model is able to see the objects at multiple scales.

In addition to these improvements, YOLO v3 can handle a wider range of object sizes and aspect ratios. It is also more accurate and stable than the previous versions of YOLO.



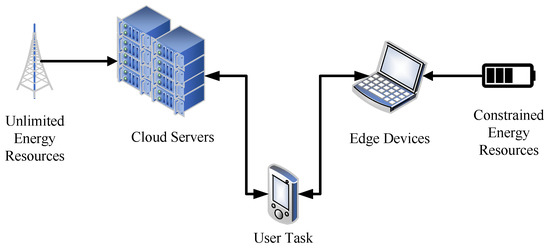
***Module: 9 RECURRENT NEURAL NETWORKS***

# 

## **1. Introduction**

With the rapid growth of big data and artificial intelligence (AI), researchers have developed artificial neural networks (ANN) for fast and effective data mining [**[1](https://www.mdpi.com/2073-8994/14/12/2524" \l "B1-symmetry-14-02524" \o ")**,**[2](https://www.mdpi.com/2073-8994/14/12/2524" \l "B2-symmetry-14-02524" \o ")**,**[3](https://www.mdpi.com/2073-8994/14/12/2524" \l "B3-symmetry-14-02524" \o ")**,**[4](https://www.mdpi.com/2073-8994/14/12/2524" \l "B4-symmetry-14-02524" \o ")**]. The recurrent neural network (RNN) is a particularly important ANN, and it is widely used to process temporal sequence tasks such as speech recognition, text generation and biometric authentication [**[5](https://www.mdpi.com/2073-8994/14/12/2524" \l "B5-symmetry-14-02524" \o ")**,**[6](https://www.mdpi.com/2073-8994/14/12/2524" \l "B6-symmetry-14-02524" \o ")**,**[7](https://www.mdpi.com/2073-8994/14/12/2524" \l "B7-symmetry-14-02524" \o ")**,**[8](https://www.mdpi.com/2073-8994/14/12/2524" \l "B8-symmetry-14-02524" \o ")**].

Edge cloud computing systems are in an asymmetric structure which consists of cloud servers and edge devices, as shown in **[Figure 1](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f001)**. RNN tasks are processed in edge cloud computing systems, where the training of RNNs is performed on symmetric cloud servers with various accelerators [**[9](https://www.mdpi.com/2073-8994/14/12/2524" \l "B9-symmetry-14-02524" \o ")**,**[10](https://www.mdpi.com/2073-8994/14/12/2524" \l "B10-symmetry-14-02524" \o ")**] and the inference of short tasks can be performed on edge nodes, taking into account the factors of latency, security, etc.



**Figure 1.** The topology of cloud edge computing systems.

However, it is worth mentioning that cloud servers and edge devices have different energy requirements. On the one hand, cloud servers have no energy limitations because they are connected to the electrical grid, which provides unlimited energy resources. On the other hand, edge devices have low energy consumption requirements because they use batteries for computing and are thus constrained in energy resources.

In order to minimize energy consumption and task latency in asymmetric structures, some inference tasks are performed on edge systems, while others are processed on remote servers [**[11](https://www.mdpi.com/2073-8994/14/12/2524" \l "B11-symmetry-14-02524" \o ")**,**[12](https://www.mdpi.com/2073-8994/14/12/2524" \l "B12-symmetry-14-02524" \o ")**,**[13](https://www.mdpi.com/2073-8994/14/12/2524" \l "B13-symmetry-14-02524" \o ")**,**[14](https://www.mdpi.com/2073-8994/14/12/2524" \l "B14-symmetry-14-02524" \o ")**,**[15](https://www.mdpi.com/2073-8994/14/12/2524" \l "B15-symmetry-14-02524" \o ")**,**[16](https://www.mdpi.com/2073-8994/14/12/2524" \l "B16-symmetry-14-02524" \o ")**]. On the one hand, edge systems that are close to users can reduce task latency by avoiding task transmission to servers. On the other hand, remote cloud servers help process large and complex inference tasks.

At present, some researchers adopt input-independent task allocation strategies between asymmetric cloud and edge computing systems, while others take advantage of data characteristics and perform input-dependent optimizations such that tasks with short estimated processing times can be executed on edge systems [**[15](https://www.mdpi.com/2073-8994/14/12/2524" \l "B15-symmetry-14-02524" \o ")**,**[16](https://www.mdpi.com/2073-8994/14/12/2524" \l "B16-symmetry-14-02524" \o ")**,**[17](https://www.mdpi.com/2073-8994/14/12/2524" \l "B17-symmetry-14-02524" \o ")**,**[18](https://www.mdpi.com/2073-8994/14/12/2524" \l "B18-symmetry-14-02524" \o ")**]. It is worth mentioning that when dealing with tasks on edge systems, various techniques can be applied to minimize energy consumption while meeting quality of service (QoS) requirements, which refer to task latency in this paper.

In this paper, we propose a low-cost runtime manager to assign tasks between asymmetric edge and cloud computing systems. Our proposed manager dynamically allocates tasks based on their predicted running times using a regression task model and QoS requirements. It then leverages dynamic voltage and frequency scaling (DVFS) techniques to perform energy optimization on edge systems. The experimental results on a real edge cloud system reveal that our method can reduce the energy by up to 45% compared with existing approaches on edge systems.

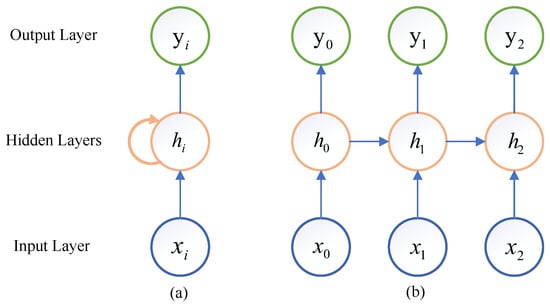
The rest of this paper is organized as follows. **[Section 2](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec2-symmetry-14-02524)** introduces the background of the work, and our motivation is demonstrated in **[Section 3](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec3-symmetry-14-02524)**. **[Section 4](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec4-symmetry-14-02524)** explains the related works, the methodology is presented in **[Section 5](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec5-symmetry-14-02524)**, the evaluation is performed in **[Section 6](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec6-symmetry-14-02524)**, and **[Section 7](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec7-symmetry-14-02524)** concludes the paper.

## **2. Background**

The RNN is a type of artificial neural network that is widely adopted to tackle sequence-related tasks such as speech recognition and natural language processing. The node connections of RNNs form a directed or undirected graph, which provides the internal states to deal with variable-length sequences of inputs.

In traditional feedforward neural networks (e.g., CNNs), the current output is independent of previous outputs. The RNN leverages feedback structures. This way, the RNN saves the output of a particular layer and feeds this back to the input in order to predict the output of the layer; that is, prior outputs can be utilized by memory units (i.e., hidden layers) for the current output.

An example of an RNN architecture is illustrated in **[Figure 2](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f002)**. **[Figure 2](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f002)**a displays a rolled RNN, where �� and �� represent the input and output at time step *i*, respectively. Hidden layers are used to model memory units using a hidden state ℎ� such that data can be persisted into the system. This shows that hidden layers exhibit feedback structures. They send prior information to the current state, which affects the output �� for a given input ��.



**Figure 2.** The architecture of an RNN (**a**) Rolled RNN, (**b**) Unrolled RNN.

**[Figure 2](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f002)**b unrolls the RNN from **[Figure 2](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f002)**a. We can see that the current hidden state of the hidden layers depends on previous information and the current input (i.e., ℎ�=�(ℎ�−1,��)). The current output �� is determined by the current input �� and state ℎ� (i.e., ��=�(ℎ�)=�(�(ℎ�−1,��))). It is clear that the prior state can affect the current output. For example, the final output �2 depends on the hidden state ℎ2, which is affected by the prior state ℎ1. Therefore, to obtain the final result of an RNN model, the computation of previous states must be accomplished. Aside form that, the unrolled architecture indicates that the computational time increases linearly with the number of hidden states.

## **3. Motivation**

When dealing with RNN inference task allocations in edge cloud computing systems, existing approaches [**[15](https://www.mdpi.com/2073-8994/14/12/2524" \l "B15-symmetry-14-02524" \o ")**,**[16](https://www.mdpi.com/2073-8994/14/12/2524" \l "B16-symmetry-14-02524" \o ")**] leverage input-dependent methods to dynamically allocate tasks among different computing devices.

It is worthwhile to mention that the running time of an RNN inference task is almost linearly proportional to that of the input length. Thus, state-of-the-art approaches allocate the tasks to edge and cloud devices based on the input length. The tasks of short lengths are executed on edge systems, while others are on cloud servers such that the overall energy consumption can be reduced by utilizing the energy-efficient task processing capabilities of edge systems.

Nonetheless, we note that existing input-dependent approaches [**[15](https://www.mdpi.com/2073-8994/14/12/2524" \l "B15-symmetry-14-02524" \o ")**,**[16](https://www.mdpi.com/2073-8994/14/12/2524" \l "B16-symmetry-14-02524" \o ")**] do not take into account task latency, which we define as the QoS requirements. Modern CPUs allow for dynamic voltage and frequency scaling, which make it feasible to save system energy using DVFS techniques without QoS violations.

We adopt an edge device (as described in **[Section 6.1](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec6dot1-symmetry-14-02524)**) and measure its energy consumption using a multimeter for a fixed input, as listed in **[Table 1](https://www.mdpi.com/2073-8994/14/12/2524" \l "table_body_display_symmetry-14-02524-t001)**. From the table, we can see that when lowering the frequency, the energy consumption decreases, and the running time increases accordingly. The energy consumption reduces by 41% when varying the frequency from 1.5 GHz to 0.6 GHz. Therefore, we conclude that DVFS can help save energy in an RNN inference task.

**Table 1.** RNN task energy consumption using DVFS techniques.



## **4. Related Work**

Numerous methods have been proposed to deal with inference tasks in edge and cloud computing systems [**[19](https://www.mdpi.com/2073-8994/14/12/2524" \l "B19-symmetry-14-02524" \o ")**,**[20](https://www.mdpi.com/2073-8994/14/12/2524" \l "B20-symmetry-14-02524" \o ")**,**[21](https://www.mdpi.com/2073-8994/14/12/2524" \l "B21-symmetry-14-02524" \o ")**,**[22](https://www.mdpi.com/2073-8994/14/12/2524" \l "B22-symmetry-14-02524" \o ")**,**[23](https://www.mdpi.com/2073-8994/14/12/2524" \l "B23-symmetry-14-02524" \o ")**]. To efficiently run deep learning (DL) tasks, researchers have proposed various hardware accelerators and interconnections in edge devices. Belabed et al. [**[24](https://www.mdpi.com/2073-8994/14/12/2524" \l "B24-symmetry-14-02524" \o ")**] developed an energy-efficient DL accelerator using FPGAs. Xia et al. [**[25](https://www.mdpi.com/2073-8994/14/12/2524" \l "B25-symmetry-14-02524" \o ")**] leverage FPGAs to process lightweight CNN models. Liu et al. [**[26](https://www.mdpi.com/2073-8994/14/12/2524" \l "B26-symmetry-14-02524" \o ")**] designed FPGA acceleration with a systolic array, matrix tiling, fixed-point precision and parallelism for compact MobileNet models. Xu et al. [**[27](https://www.mdpi.com/2073-8994/14/12/2524" \l "B27-symmetry-14-02524" \o ")**] implement FPGA acceleration for computer vision DL tasks that effectively reduced the response time and energy consumption.

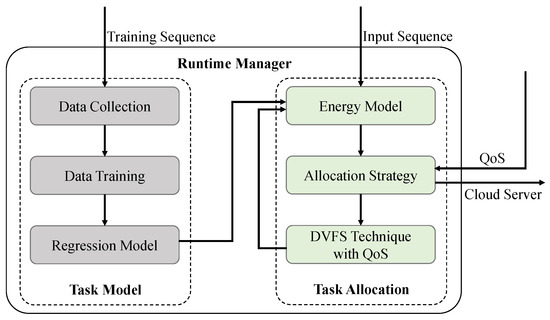
Another approach is to optimize DL models for inference acceleration. Zhou et al. [**[28](https://www.mdpi.com/2073-8994/14/12/2524" \l "B28-symmetry-14-02524" \o ")**] developed a lightweight CNN model for edge computing. Kim et al. [**[29](https://www.mdpi.com/2073-8994/14/12/2524" \l "B29-symmetry-14-02524" \o ")**] optimized DL models for hardware-specific characteristics using TVM. Li et al. [**[30](https://www.mdpi.com/2073-8994/14/12/2524" \l "B30-symmetry-14-02524" \o ")**] proposed dynamic filter pruning together with a model transformation method to reduce the computational complexity. Matsubara et al. [**[31](https://www.mdpi.com/2073-8994/14/12/2524" \l "B31-symmetry-14-02524" \o ")**] used knowledge distillation to build compressed models for edge devices. Kim and Deka [**[32](https://www.mdpi.com/2073-8994/14/12/2524" \l "B32-symmetry-14-02524" \o ")**] optimized configurations for DL models to work on tensor processing units. Li et al. [**[33](https://www.mdpi.com/2073-8994/14/12/2524" \l "B33-symmetry-14-02524" \o ")**] took advantage of a greedy-based filter pruning technique to optimize DL models.

In addition, researchers utilized collaborative edge cloud systems for DL tasks. Zhou et al. [**[34](https://www.mdpi.com/2073-8994/14/12/2524" \l "B34-symmetry-14-02524" \o ")**] explored various accelerators and guided accelerator selection for tasks. Gong et al. [**[35](https://www.mdpi.com/2073-8994/14/12/2524" \l "B35-symmetry-14-02524" \o ")**] dealt with private and public data on edge and cloud devices, respectively. Feng et al. [**[36](https://www.mdpi.com/2073-8994/14/12/2524" \l "B36-symmetry-14-02524" \o ")**] proposed a blockchain-enabled technique that optimizes offloading tasks between edge devices and servers. Liu et al. [**[37](https://www.mdpi.com/2073-8994/14/12/2524" \l "B37-symmetry-14-02524" \o ")**] presented a framework that offloads computation from unmanned aerial vehicles (UAVs) to devices based on optimal decisions and resource management policies. Kennedy et al. [**[38](https://www.mdpi.com/2073-8994/14/12/2524" \l "B38-symmetry-14-02524" \o ")**] proposed a framework to offload DL tasks to virtualized GPUs. Kuang et al. [**[39](https://www.mdpi.com/2073-8994/14/12/2524" \l "B39-symmetry-14-02524" \o ")**] minimized task latency while meeting the requirements of power and energy.

Note that current methods mainly focus on feedforward networks such as CNNs, and few collaborative methods study recurrent networks. Pagliari et al. [**[15](https://www.mdpi.com/2073-8994/14/12/2524" \l "B15-symmetry-14-02524" \o ")**] managed RNN inference task mappings between edge and cloud systems using an input-dependent method, and they extended the method to a collaborative mapping engine to support an arbitrary number of devices and interconnections [**[16](https://www.mdpi.com/2073-8994/14/12/2524" \l "B16-symmetry-14-02524" \o ")**]. Nonetheless, for RNN inference tasks, existing approaches map tasks based on the length of an input sequence. We propose a technique combining the input sequence and DVFS that minimizes the energy consumption of edge devices while meeting the QoS requirements of RNN tasks.

## **5. Methodology**

Motivated by the energy reduction observations with DVFS in **[Section 3](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec3-symmetry-14-02524)**, we propose an energy-aware runtime manager that takes advantage of DVFS techniques to consume less energy and meet the QoS requirements of RNN tasks. The workflow of the runtime manager is shown in **[Figure 3](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f003)**. In the Task Model phase, we first collect the running time at various operating frequencies for different input lengths. Then, we perform data training to build the regression model snd predict the running time in various operating conditions.



**Figure 3.** The workflow of the proposed energy-aware runtime manager.

In the Task Allocation phase, we establish the energy model using the predicted running time together with the power consumption. The allocation strategy is then applied to perform task allocations between edge and cloud devices. It takes into account the QoS requirement to make sure that tasks are properly assigned to cloud servers and edge devices. When operating on edge devices, DVFS techniques are applied to edge computing systems to minimize energy consumption while meeting QoS requirements.

#### *5.1. Task Model*

When dealing with different RNN tasks, we need to build a performance model to estimate the running time of RNN tasks. With the task model, one can predict the running time of each task, which helps determine task allocation strategies afterward.

To build a task model, we first collected data from the training sequences. We adopted *n* different lengths of input sequences, and each length of input sequences was repeated *m* times to measure the variations. Note that due to DVFS, there existed different CPU operating frequencies. Suppose that there were *p* frequency modes. Thus, we obtained �×�×� sets of input sequences, and the *j*th time repetition of the *i*th length of input sequences for the *k*th frequency is denoted by ��,�,�, which yields the running time ��,�,�. Therefore, we obtained data tuples <��,�,�,��,�,�>,1≤�≤�,1≤�≤�,1≤�≤� in the data collection.

We used the collected data for training to obtain the task model. Since the running time of an RNN inference task is almost linearly proportional to that of the the input length, we built a linear performance model as follows:

��(�)=��·�+��+��

(1)

where �� is the predicted running time of the RNN task, *d* is the length of the input sequence, �� is the proportional coefficient between the input length and running time, �� is the bias coefficient, �� is the overhead time to switch frequencies and subscript � denotes the *k*th operating frequency, as described in **[Section 6.2](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec6dot2-symmetry-14-02524)**.

To obtain the coefficients of Equation (**[1](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD1-symmetry-14-02524)**), we employed the linear least squares regression method using data collection <��,�,�,��,�,�>. Then, we obtained

��=�·�·Σ�,�(��,�,�·��,�,�)−Σ�,���,�,�·Σ�,���,�,��·�·Σ�,���,�,�2−(Σ�,���,�,�)2

(2)

��=Σ�,���,�,�−��·Σ�,���,�,��·�

(3)

With Equations (**[2](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD2-symmetry-14-02524)**) and (**[3](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD3-symmetry-14-02524)**), we obtained the regression model at the *k*th operating frequency, which was used for task allocation. We obtained a task model � of all frequencies as follows:

�={��|1≤�≤�}

(4)

#### *5.2. Task Allocation*

From the observations in **[Section 3](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec3-symmetry-14-02524)**, we found that the frequency affects the task energy. To reduce the energy consumption of edge cloud computing systems, we first built an energy model at the *k*th operating frequency:

��=��·��

(5)

where �� is the task energy, �� is the running time of the inference task and �� is the power at the *k*th operating frequency. Then, we obtained the energy model of all frequencies as follows:

�={��|1≤�≤�}

(6)

Then, we designed a task allocation strategy as follows:

�(�,�)=Edgeif∃��∈�,��(�)≤�Cloudotherwise

(7)

where *S* is the task allocation strategy function, *d* is the input data sequence and *q* is the QoS requirement (i.e., the maximum allowable task latency). If the QoS requirement can be met on edge devices, the task allocation strategy executes the task on the edge. Otherwise, it sends the task to the cloud server for execution.

When executing tasks on edge devices, we applied DVFS techniques to dynamically adjust the processor frequency to minimize energy consumption as follows:

argmin���(��)={��|��∈�,��≤�}

(8)

We selected the lowest operating frequency that met the QoS requirements to save energy. Note that when we varied the CPU frequencies from the *i*th operating frequency to the *k*th operating frequency, the overhead time in Equation (**[1](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD1-symmetry-14-02524)**) was computed as follows:

��=0if�=�Cotherwise

(9)

where *C* is a constant, as described in **[Section 6.2.2](https://www.mdpi.com/2073-8994/14/12/2524" \l "sec6dot2dot2-symmetry-14-02524)**.

After frequency variation using DVFS techniques, we updated the energy model using the latest data. We updated the coefficients in Equation (**[1](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD1-symmetry-14-02524)**) using online data such that the model always reflected the current characteristics of the edge device.

## **6. Experimental Results**

#### *6.1. Experimental Set-Up*

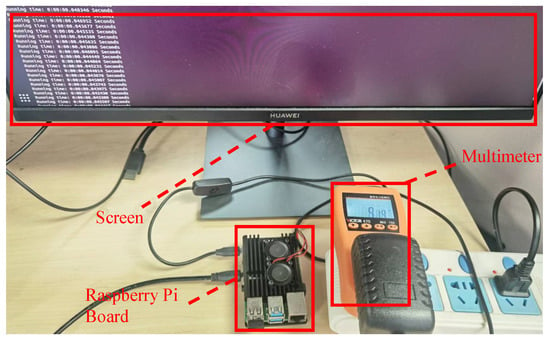
To evaluate the effectiveness of our proposed method, without loss of generality, we used a server with Intel processors and a Raspberry Pi board device to emulate the cloud server and the edge device of an edge cloud computing system, respectively. Note that our method can be applied to a system with multiple cloud servers and edge devices, where each RNN task is allocated accordingly. In this paper, our experimental system set-up was as follows:

An ARM Cortex-A72@1.5 GHz, 4 GB RAM to denote an edge computing device;

A Dual Intel Xeon E5-2630@2.40 GHz, 128 GB RAM plus an NVIDIA 3080Ti GPU to represent a cloud computing server.

We used Python 3.6 to build an RNN model and implement the proposed task allocation strategy. In addition, we applied a multimeter to measure the power throughout the experiments.

**[Figure 4](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f004)** displays the experimental set-up of the edge computing device, where the Raspberry Pi development board was used as the edge computing device, a multimeter was adopted for power measurement, and the screen was used to show the working status of the edge device, such as the task latency and system status.

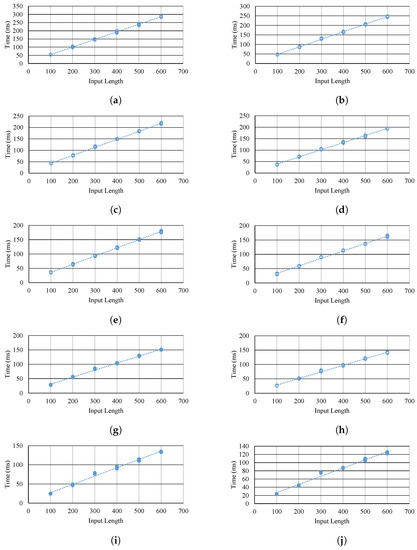


**Figure 4.** Experimental set-up of the edge computing device.

#### *6.2. Task Model*

#### 6.2.1. Linear Coefficients

We first performed experiments to determine the task model’s linear coefficients in Equation (**[1](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD1-symmetry-14-02524)**) (i.e., �� and ��). The input length and running time results at different operating frequencies are scattered in **[Figure 5](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f005)**, where each input length was measured five times to evaluate the time variation. From **[Figure 5](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f005)**, we applied the linear least squares regression method in Equations (**[2](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD2-symmetry-14-02524)**) and (**[3](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD3-symmetry-14-02524)**) to obtain the coefficients of �� and ��, respectively. The computed coefficients are displayed in **[Table 2](https://www.mdpi.com/2073-8994/14/12/2524" \l "table_body_display_symmetry-14-02524-t002)**.



**Figure 5.** Input length and running time of RNN tasks at different operating frequencies. (**a**) Input length and running time at 0.6 GHz. (**b**) Input length and running time at 0.7 GHz. (**c**) Input length and running time at 0.8 GHz. (**d**) Input length and running time at 0.9 GHz. (**e**) Input length and running time at 1.0 GHz. (**f**) Input length and running time at 1.1 GHz. (**g**) Input length and running time at 1.2 GHz. (**h**) Input length and running time at 1.3 GHz. (**i**) Input length and running time at 1.4 GHz. (**j**) Input length and running time at 1.5 GHz.

**Table 2.** RNN task energy consumption using DVFS techniques.



In **[Figure 5](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f005)**, the dotted linear line is displayed for each frequency. We observe that for a fixed frequency, the running time was linearly proportional to the input length. There were some points that exhibited offsets from the linear line, such as the case of an input length of 300 in **[Figure 5](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f005)**j, but those points are not far away from the linear line. There was a linear relationship between the input length and its running time, because for each input data point, all hidden states were calculated, which resulted in a fixed processing time. As the input length increased linearly, the running time accumulated linearly as well.

As the frequency increased from 0.6 GHz to 1.5 GHz, the running time decreased accordingly, and this makes sense since higher frequencies lead to higher computing capabilities. The line slope �� decreased from 0.4629 to 0.1998 from 0.6 GHz to 1.5 GHz, and this indicates that when the input length increased by 100, the running time increment reduced from 46.29 ms to 19.98 ms, and the system performance was improved at higher frequencies.

#### 6.2.2. Overhead Coefficient

In addition to linear coefficients, we also needed to determine the overhead coefficient �� in Equation (**[1](https://www.mdpi.com/2073-8994/14/12/2524" \l "FD1-symmetry-14-02524)**). We applied DVFS techniques and switched operating frequencies under different conditions, as shown in **[Table 3](https://www.mdpi.com/2073-8994/14/12/2524" \l "table_body_display_symmetry-14-02524-t003)**.

**Table 3.** The overhead of switching operating frequencies. The first column and the first row denote the target frequency and source frequency, respectively. The value of the table represents the switching time from the source frequency to the target frequency in ms.



The target frequency and source frequency are used as headers, and the switching from each source frequency to each target frequency is displayed. We can see that the switching overhead time was zero if the source and target frequency were the same because there was no need to change frequencies in this case. Otherwise, it took approximately 5–10 ms to finish the frequency switching. Therefore, we set up a lookup table for �� such that for every instance of frequency switching, we could find the precise overhead value.

It is worth mentioning that on the one hand, if the source frequency is higher, it takes less time to finish the switching. For example, when switching to a 0.7 GHz target frequency from a 0.6 GHz source frequency, it took 10.01 ms, while the switching time decreased to 6 ms from a 1.5 GHz source frequency. There was a 4.01 ms switching time difference in this case.

On the other hand, if the target frequency is higher, it also spends less time switching. When either the source frequency or target frequency is higher, it is faster to finish the frequency switching because the CPU operates in higher frequencies, and it accomplishes tasks more quickly.

However, when the target frequency is high, the time difference is not as large as when the source frequency is high. For example, when the source frequency was 1.4 GHz, the switching time difference was 1.11 ms between the target frequencies of 0.6 GHz and 1.5 GHz. This was much smaller than the time difference value of 4.01 ms when the source frequency was high.

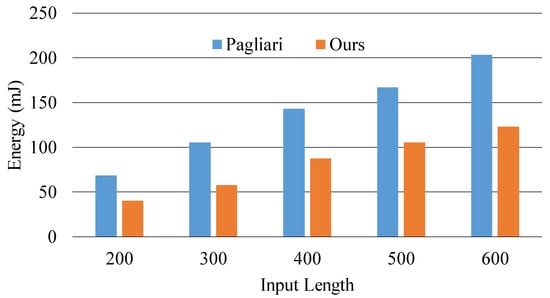
#### *6.3. Energy Consumption*

We tested different lengths of inputs and compared the edge energy consumption of our proposed method with that of the state-of-the-art approach by Pagliari et al. [**[15](https://www.mdpi.com/2073-8994/14/12/2524" \l "B15-symmetry-14-02524" \o ")**]. Similarly, the method of Pagliari [**[15](https://www.mdpi.com/2073-8994/14/12/2524" \l "B15-symmetry-14-02524" \o ")**] maps RNN tasks among cloud and edge devices such that the execution time and energy consumption requirements can be met. The method comparison is summarized in **[Table 4](https://www.mdpi.com/2073-8994/14/12/2524" \l "table_body_display_symmetry-14-02524-t004)**.

**Table 4.** Methodology comparison.



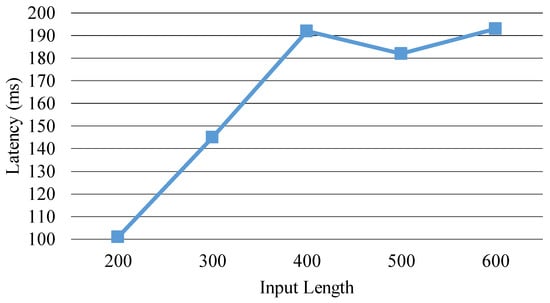
We used �=200ms as the QoS latency requirement. The results are shown in **[Figure 6](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f006)**.



**Figure 6.** Energy consumption comparison on edge devices for separate input lengths.

In **[Figure 6](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f006)**, the x-axis represents separate input lengths from 200 to 600, and the y-axis denotes the energy consumption for each input length in mJ. We can see that compared with the method of Pagliari, our proposed method could effectively reduce the energy consumption of edge devices for each input length. It reduced the energy consumption from 36% to 45% for different input lengths and by up to 45% for the input length of 300.

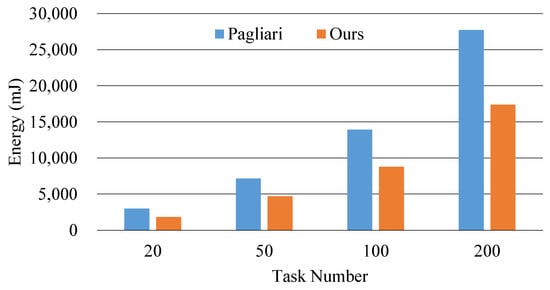
Together with energy consumption, we plotted the task latency in **[Figure 7](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f007)**. We can see that for all input sequences, the task latency met the specified QoS requirement (i.e., ��≤�). When the input length was small (i.e., 200 and 300), the latency of such a task was very small, and the QoS could be readily met using the lowest frequency. However, as the input length increased, using the lowest frequency did not meet the QoS requirement. Therefore, the operating frequency varied accordingly to meet the QoS demands.



**Figure 7.** Task latency using the proposed method on edge devices.

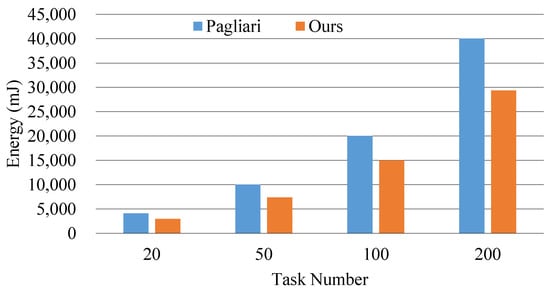
Compared with the start-of-the-art method, our method was able to significantly reduce energy consumption, because our method slowed down the task execution while making sure that the QoS latency requirement was satisfied. This way, the processor runs tasks at low frequencies, and thus a large amount of energy is saved.

Apart from fixed input lengths, we use mixed different lengths for the input (from 200 to 600 with an interval of 100) and applied our method for energy optimization, as shown in **[Figure 8](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f008)**, where the x-axis represents the mixed task number that includes random input lengths and the y-axis denotes the energy consumption in mJ. We found that for tasks with randomly mixed input lengths, our approach consumed 37% less energy on average (from 34% to 39%) compared with the method of Pagliari.

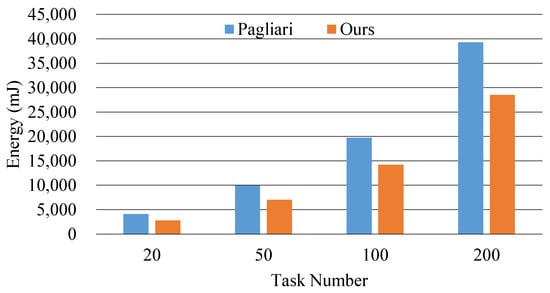


**Figure 8.** Energy consumption comparison on edge devices using mixed input lengths for the RNN model.

In addition to the normal RNN model, we also compared our approach with that of Pagliari using two specialized versions of RNN (i.e., long short-term memory (LSTM) and gated recurrent unit (GRU)). The results are displayed in **[Figure 9](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f009)** and **[Figure 10](https://www.mdpi.com/2073-8994/14/12/2524" \l "fig_body_display_symmetry-14-02524-f010)**, where the x-axis represents the mixed task number that includes random input lengths (from 100 to 250 with an interval of 50) and the y-axis denotes the energy consumption in mJ. We found that for the LSTM model, our approach consumed 26% less energy on average—from 25% to 27%—compared with the method of Pagliari, and for the GRU model, our approach consumed 28% less energy on average—from 27% to 31%—compared with the method of Pagliari.



**Figure 9.** Energy consumption comparison for edge devices using mixed input lengths for the LSTM model.



**Figure 10.** Energy consumption comparison for edge devices using mixed input lengths for the GRU model.

## **7. Conclusions**

In this paper, we presented an energy-aware runtime manager that assigns RNN inference tasks to edge cloud systems. Our approach performs energy optimizations based on QoS requirements when processing tasks in edge systems. It decreases the operating frequency for tasks with short input lengths, and the experimental results reveal that it can reduce the energy consumption by up to 45% in edge devices compared with the state-of-the-art approach. Based on the successful experience of this paper, we conclude that we can improve the energy consumption of various projects by specifying the QoS requirement and slowing down the operating frequency using DVFS techniques. In the future, we plan to extend our manager to more neural network inference tasks.

**Module 10: NATURAL LANGUAGE PROCESSING**

hat is natural language processing?

Natural language processing (NLP) refers to the branch of computer science—and more specifically, the branch of [artificial intelligence or AI](https://www.ibm.com/topics/artificial-intelligence" \o "what-is-artificial-intelligence)—concerned with giving computers the ability to understand text and spoken words in much the same way human beings can.

NLP combines computational linguistics—rule-based modeling of human language—with statistical, machine learning, and deep learning models. Together, these technologies enable computers to process human language in the form of text or voice data and to ‘understand’ its full meaning, complete with the speaker or writer’s intent and sentiment.

NLP drives computer programs that translate text from one language to another, respond to spoken commands, and summarize large volumes of text rapidly—even in real time. There’s a good chance you’ve interacted with NLP in the form of voice-operated GPS systems, digital assistants, speech-to-text dictation software, [customer service chatbots](https://www.ibm.com/products/watsonx-assistant/customer-service), and other consumer conveniences. But NLP also plays a growing role in enterprise solutions that help streamline business operations, increase employee productivity, and simplify mission-critical business processes.

How to build responsible AI at scale

[Explore the guide](https://www.ibm.com/resources/the-data-differentiator/scale-ai" \t "https://www.ibm.com/topics/_blank)

NLP tasks

Human language is filled with ambiguities that make it incredibly difficult to write software that accurately determines the intended meaning of text or voice data. Homonyms, homophones, sarcasm, idioms, metaphors, grammar and usage exceptions, variations in sentence structure—these just a few of the irregularities of human language that take humans years to learn, but that programmers must teach natural language-driven applications to recognize and understand accurately from the start, if those applications are going to be useful.

Several NLP tasks break down human text and voice data in ways that help the computer make sense of what it's ingesting. Some of these tasks include the following:

* ****Speech recognition****, also called speech-to-text, is the task of reliably converting voice data into text data. Speech recognition is required for any application that follows voice commands or answers spoken questions. What makes speech recognition especially challenging is the way people talk—quickly, slurring words together, with varying emphasis and intonation, in different accents, and often using incorrect grammar.
* ****Part of speech tagging****, also called grammatical tagging, is the process of determining the part of speech of a particular word or piece of text based on its use and context. Part of speech identifies ‘make’ as a verb in ‘I can make a paper plane,’ and as a noun in ‘What make of car do you own?’
* ****Word sense disambiguation**** is the selection of the meaning of a word with multiple meanings  through a process of semantic analysis that determine the word that makes the most sense in the given context. For example, word sense disambiguation helps distinguish the meaning of the verb 'make' in ‘make the grade’ (achieve) vs. ‘make a bet’ (place).
* ****Named entity recognition,****or NEM, identifies words or phrases as useful entities. NEM identifies ‘Kentucky’ as a location or ‘Fred’ as a man's name.
* ****Co-reference resolution**** is the task of identifying if and when two words refer to the same entity. The most common example is determining the person or object to which a certain pronoun refers (e.g., ‘she’ = ‘Mary’),  but it can also involve identifying a metaphor or an idiom in the text  (e.g., an instance in which 'bear' isn't an animal but a large hairy person).
* ****Sentiment analysis****attempts to extract subjective qualities—attitudes, emotions, sarcasm, confusion, suspicion—from text.
* ****Natural language generation****is sometimes described as the opposite of speech recognition or speech-to-text; it's the task of putting structured information into human language.

See the blog post “[NLP vs. NLU vs. NLG: the differences between three natural language processing concepts](https://www.ibm.com/blog/nlp-vs-nlu-vs-nlg-the-differences-between-three-natural-language-processing-concepts/)” for a deeper look into how these concepts relate.

Now available: watsonx.ai

The all new enterprise studio that brings together traditional machine learning along with new generative AI capabilities powered by foundation models

[Try watsonx.ai](https://www.ibm.com/products/watsonx-ai" \t "https://www.ibm.com/topics/_self)

NLP tools and approaches

### Python and the Natural Language Toolkit (NLTK)

The Python programing language provides a wide range of tools and libraries for attacking specific NLP tasks. Many of these are found in the Natural Language Toolkit, or NLTK, an open source collection of libraries, programs, and education resources for building NLP programs.

The NLTK includes libraries for many of the NLP tasks listed above, plus libraries for subtasks, such as sentence parsing, word segmentation, stemming and lemmatization (methods of trimming words down to their roots), and tokenization (for breaking phrases, sentences, paragraphs and passages into tokens that help the computer better understand the text). It also includes libraries for implementing capabilities such as semantic reasoning, the ability to reach logical conclusions based on facts extracted from text.

### Statistical NLP, machine learning, and deep learning

The earliest NLP applications were hand-coded, rules-based systems that could perform certain NLP tasks, but couldn't easily scale to accommodate a seemingly endless stream of exceptions or the increasing volumes of text and voice data.

Enter statistical NLP, which combines computer algorithms with machine learning and [deep learning](https://www.ibm.com/topics/deep-learning" \o "deep-learning) models to automatically extract, classify, and label elements of text and voice data and then assign a statistical likelihood to each possible meaning of those elements. Today, deep learning models and learning techniques based on convolutional neural networks (CNNs) and recurrent neural networks (RNNs) enable NLP systems that 'learn' as they work and extract ever more accurate meaning from huge volumes of raw, unstructured, and unlabeled text and voice data sets.

For a deeper dive into the nuances between these technologies and their learning approaches, see “[AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What’s the Difference?](https://www.ibm.com/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks/)”

NLP use cases

Natural language processing is the driving force behind machine intelligence in many modern real-world applications. Here are a few examples:

* ****Spam detection:****You may not think of spam detection as an NLP solution, but the best spam detection technologies use NLP's text classification capabilities to scan emails for language that often indicates spam or phishing. These indicators can include overuse of financial terms, characteristic bad grammar, threatening language, inappropriate urgency, misspelled company names, and more. Spam detection is one of a handful of NLP problems that experts consider 'mostly solved' (although you may argue that this doesn’t match your email experience).
* ****Machine translation:****Google Translate is an example of widely available NLP technology at work. Truly useful machine translation involves more than replacing words in one language with words of another.  Effective translation has to capture accurately the meaning and tone of the input language and translate it to text with the same meaning and desired impact in the output language. Machine translation tools are making good progress in terms of accuracy. A great way to test any machine translation tool is to translate text to one language and then back to the original. An oft-cited classic example: Not long ago, translating “*The spirit is willing but the flesh is weak”* from English to Russian and back yielded “*The vodka is good but the meat is rotten*.” Today, the result is “*The spirit desires, but the flesh is weak*,” which isn’t perfect, but inspires much more confidence in the English-to-Russian translation.
* ****Virtual agents and chatbots:**** [Virtual agents](https://www.ibm.com/products/watsonx-assistant" \o "Follow link) such as Apple's Siri and Amazon's Alexa use speech recognition to recognize patterns in voice commands and natural language generation to respond with appropriate action or helpful comments. [Chatbots](https://www.ibm.com/topics/chatbots" \o "Follow link) perform the same magic in response to typed text entries. The best of these also learn to recognize contextual clues about human requests and use them to provide even better responses or options over time. The next enhancement for these applications is question answering, the ability to respond to our questions—anticipated or not—with relevant and helpful answers in their own words.
* ****Social media sentiment analysis:****NLP has become an essential business tool for uncovering hidden data insights from social media channels. Sentiment analysis can analyze language used in social media posts, responses, reviews, and more to extract attitudes and emotions in response to products, promotions, and events–information companies can use in product designs, advertising campaigns, and more.
* ****Text summarization:****Text summarization uses NLP techniques to digest huge volumes of digital text and create summaries and synopses for indexes, research databases, or busy readers who don't have time to read full text. The best text summarization applications use semantic reasoning and natural language generation (NLG) to add useful context and conclusions to summaries.