Neural Networks & Image Classification with Convolutional Neural Networks (CNNs)

Author: Yana Jakhwal

Date: August 25th, 2024

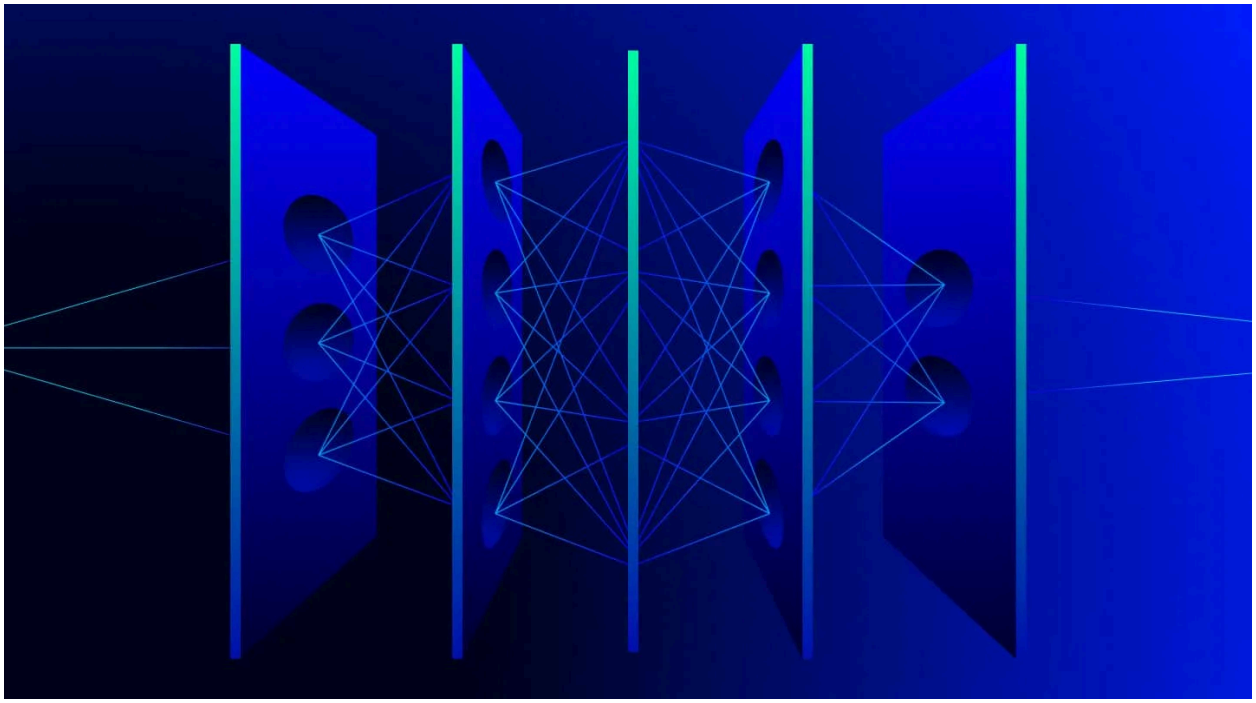


Table of Contents

1.0 INTRODUCTION 3

2.0 FOUNDATIONS 4

2.1 Neural Networks 4

2.1.1 Introduction and History 4

2.1.2 Basic Concepts and Components 6

2.1.3 Types of Neural Networks 11

2.2 Convolutional Neural Networks 15

2.2.1 Architecture/Structure of CNNs For Image Processing 15

2.2.2 Applications of CNNs 18

2.3 Comparison of Frameworks with Python 21

2.3.1 Kesar Model with Tensorflow Backend Process 21

2.3.2 Pytorch 22

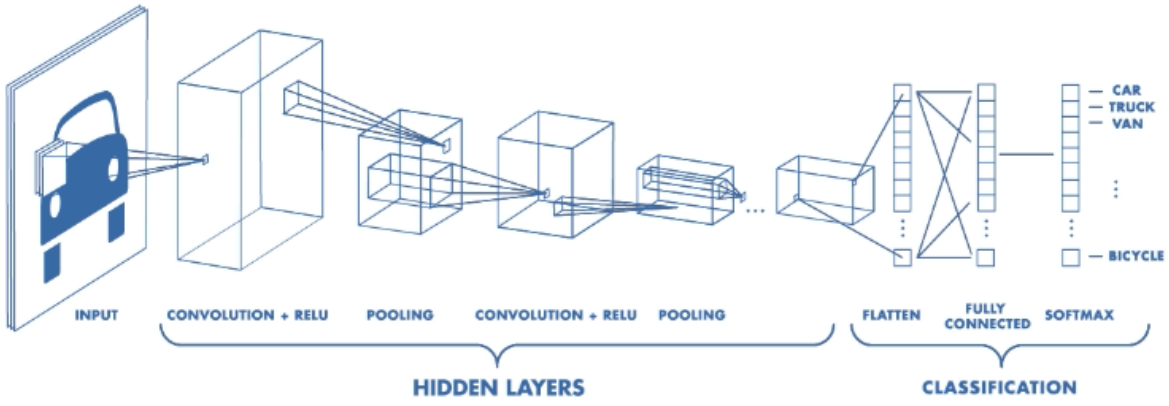
2.3.3 Comparing the Two Frameworks 23

3.0 CONCLUSIONS .....

REFERENCES .....

1.0 INTRODUCTION

Neural network models are inspired by the structure and function of biological neural networks in the human brain, with Convolutional Neural Networks (CNNs) specifically mimicking aspects of human visual processing. In recent years, CNNs have emerged as a powerful tool for image classification, revolutionising the field of computer vision. The development of LeNet by Yann LeCun in 1989 marked a significant milestone, introducing a CNN architecture (see Fig. 1) that consists of convolutional layers, pooling operations, and fully connected layers. This architecture laid the groundwork for modern CNNs.



(Figure 1)

Since the introduction of LeNet, CNNs have advanced rapidly, leading to the development of new architectures like AlexNet, VGG, and ResNet, as well as optimization techniques that have expanded their applicability across various domains. Despite these improvements, challenges such as overfitting, interpretability, and the need for large labelled datasets remain. Nevertheless, CNNs continue to be at the forefront of deep learning, powering applications from object recognition to medical image analysis.

This report aims to explore the foundational concepts of neural networks and their evolution into CNNs, providing a detailed overview of how these networks function and why they are particularly suited for image-related tasks. This paper will begin by explaining the basic principles of neural networks, setting the stage for a deeper dive into the structure and mechanics of CNNs. Followed by comparison of the implementation of CNNs using three popular Python frameworks: Keras, TensorFlow, and PyTorch.

By examining the performance, usability, and application of these frameworks, the aim is to offer insights that can guide both newcomers to machine learning and those looking to expand their knowledge of image processing techniques.

This comprehensive review seeks to contribute to a deeper understanding of CNNs and provide a valuable resource for individuals interested in the practical application of these models within the Python ecosystem.

2.0 FOUNDATIONS

2.1 Neural Networks

2.1.1 Introduction and History

To put it simply, a neural network can be defined as “a machine learning program, or model, that makes decisions in a manner similar to the human brain, by using processes that mimic the way biological neurons work together to identify phenomena, weigh options and arrive at conclusions,” (IBM, 2021). The human brain consists of interconnected nodes, or neurons, organised in layers. This framework is the closest algorithm to mimicking the human brain.

The exploration of neural networks has been an ongoing process for over 80 years, significantly influencing the evolution of machine learning. In 1943, Warren McCulloch and Walter Pitts published a seminal paper that introduced the first mathematical model of a neural network, which combined ideas from physiology, biology, mathematics, and computer science. Their model, inspired by Alan Turing’s work on computation, represented neural activity using binary logic (all-or-nothing responses). Although foundational, their work initially went unrecognised.

Not too many years after the publication of the founding fathers’ report, in 1947, Donald Hebb in his book, “The Organization of Behaviour” described how neural pathways are strengthened whenever they are used, which is quite similar to the learning process of humans. His findings specified that when two nerves are fired, the connection is enhanced, and quite simply put, “neurons that fire together, wire together”. From the view of artificial neurons and artificial neural networks, Hebb's principle is a method

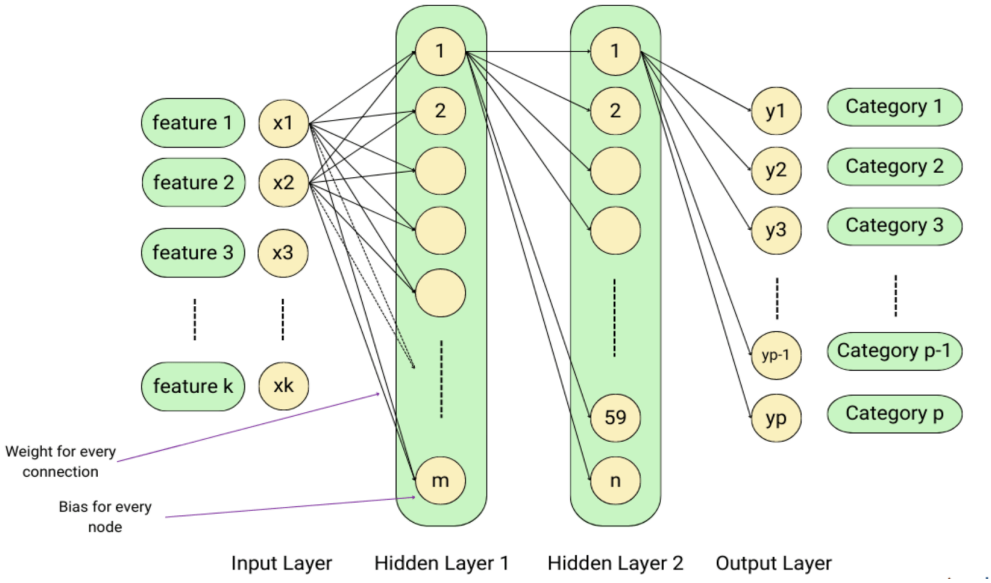
of determining how to alter weights between model neurons where two neurons increase in weight if both activate at the same time. If both are activated, the nodes tend to be more positive while nodes that are different have negative weights. These discoveries and principles were not executed as full networks by this point.

The advancement of computers in the 1950s enabled the simulation of neural networks, leading to significant developments by Bernard Widrow and Marcian Hoff. They introduced ADALINE (Adaptive Linear Neuron) and MADALINE (Multiple ADALINE), models designed to recognize binary patterns and predict sequences using the least-mean-squares algorithm. MADALINE was particularly notable for its real-world application in adaptive filtering, such as eliminating echoes on phone lines—a technology still in use today.

In 1974, Paul Werbos developed the backpropagation algorithm, a method for efficiently calculating gradients in neural networks. Although his work initially went unnoticed, backpropagation was independently rediscovered in the mid-1980s by David Rumelhart, Geoffrey Hinton, and Ronald Williams, leading to its widespread adoption in training multi-layered networks.

In 1980, Kunihiko Fukushima developed the Neocognitron, a hierarchical, multilayered neural network designed for visual pattern recognition, particularly for handwritten characters. The Neocognitron was a precursor to modern Convolutional Neural Networks (CNNs) and introduced concepts like convolutional layers and hierarchical feature extraction. The ReLU (Rectified Linear Unit) activation function, which has become standard in modern CNNs, was popularised much later, particularly with the success of AlexNet in 2012.

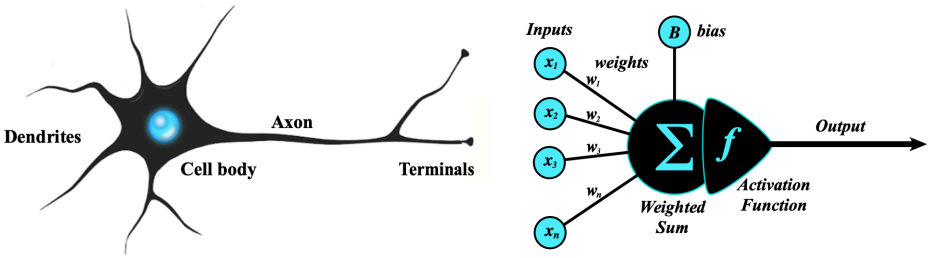
2.1.2 Basic Concepts and Components



(Figure 2)

Neurons

Neurons are vital to the functioning of a neural network as they play roles in every single part of the process. In the context of machine learning they are meant to mimic the human neurons in the brain where the nucleus of the neurons contains a bias which is typically initialised with a random number and that's when the network fine tunes them to minimise the difference between the computed and actual output (see Fig. 3).



(Figure 3)

Input Layer

The input layer in neural networks are the first layers that the raw data goes through. Each neuron represents a feature of the input data where one neuron represents one unit of a certain feature. The primary role of the input layer is to pass the data into different features to the next layer.

For example:

- Images: An image of size 4x4 being processed, means that there are 16 neurons going through the input layer.
- Humans: Features include the height, age, and weight. The input layer receives the numerical values of the data for each of the layers.

Output Layer

This layer is responsible for outputting the network's computations where the number of neurons corresponds to the type of model.

For example:

- Linear regression model: the process should output a single neuron.
- Classifications: 15 different classes, then the output layer should give 15 neurons corresponding to the number of classes.

Softmax : Is an activation function utilised in the output layer of the neural network, typically for multi-classification problems where it converts raw output scores (logits) from the network into probabilities that sum to 100%, which makes the interpretation of the output as a probability distribution over the multiple classes. Before the logits are converted, they are exponentiated to the raw score for the class. Finally, the values are normalised by dividing the sum of the exponentiated scores across all classes which ensures that the outputs of a probability distribution where the values are between 0 and 1 and they sum up to 1.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Hidden Layer(s)

Hidden layers occur after the input layer and before the output layer. They utilise the raw data and apply transformations to the data to extract meaningful features or patterns useful for making predictions. Each hidden layer has neurons for computing units of the data, which allows the learning of complex patterns like multi-regression models. A typical misconception is that more layers equate to a more accurate output, which is not correct as the more layers there are, the more vulnerable the network is to overfitting. Overfitting is when a model learns to include outliers and it works so well on the current data, but is not general enough to be applied to different data sets.

Neurons typically use different activation functions for the introduction of non-linearity into the network. ReLU (Rectified Linear Unit) is the most commonly used activation function used in hidden filters as they aid in the learning of complex patterns while being computationally efficient at the same time. ReLU: $f(x) = max(0, x)$ which sets all negative neurons to zero, which creates multiple linear regression in the network thus allowing for more complex functions. The function also tells the model which neurons are to be activated and how much information they should transmit. They are used in hidden layers because if x is greater than 0 the gradient would be either 1 or 0 when x is less than or equal to 0. The gradients are the vectors that represent direction and magnitude that are utilised to minimise the loss function and update the weights during training.

Gradients are computed using a process called backpropagation. This algorithm efficiently computes the gradient of the loss function with respect to all the weights in the network by applying the chain rule of calculus. These derivatives are calculated by differentiating one weight and treating the other(s) as a constant.

Connections

Connections in the context of neural networks are like links between neurons in different layers, where data and computations flow, the process is referred to as forward propagation where it starts from the beginning to the end (input layer to output layer). During training, weights are adjusted and biases are

utilised to make accurate predictions for the network. Backpropagation is also utilised to update the weights by minimising the error.

Weights are associated with each connection which is a numerical value determining the strength and direction of the influence that one neuron has on another. The weights are multiplied by the output of the neuron from the origin of the connection and then the weighted output is passed to the neuron in the next layer.

Every neuron has a bias, which is a value added to the sum of inputs before the use of the activation function. Biases are also learnable parameters which can be adjusted during training.

Forward propagation is the process of moving data from the input layers to the hidden layer(s), and finally the output layer via the connections. Through this process, the neurons are multiplied by their respective weights, summed and then used for the activation function.

Backpropagation is the process of calculating the gradient of the loss function with respect to each weight using chain rule. The error that is calculated by the time the outer layer is reached, the connections (weights) are updated to reduce the error.

From Input to Hidden Layer:

- Neuron 1 in the hidden layer receives inputs from both x_1 and x_2 .
- The connection between x_1 and Neuron 1 has a weight w_{11} , and the connection between x_2 and Neuron 1 has a weight w_{12} .
- Neuron 1 computes the weighted sum: $z_1 = w_{11} \times x_1 + w_{12} \times x_2 + b_1$, where b_1 is the bias for Neuron 1.
- This process is repeated for Neurons 2 and 3 in the hidden layer.

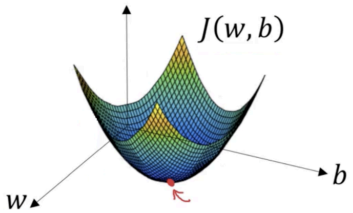
From Hidden to Output Layer:

- The output neuron receives inputs from all three neurons in the hidden layer.
- The connection between Neuron 1 in the hidden layer and the output neuron has a weight w_{01}.

- The output neuron computes the weighted sum: $z_0 = w_{01} \times h_1 + w_{02} \times h_2 + w_{03} \times h_3$, where h_i are the outputs of the hidden neurons and b_0 is the bias for the output neuron.

Lost/Cost Function

Cost functions calculate how much the network has failed to learn numerically. Machine learning techniques optimise these costs by changing the parameter values which achieves the best ones for which the cost is at a minimum. The cost functions must represent the errors between the output of the neural network and the expected output by the neural network, so the changes in the parametric function should reflect the change in the cost function. The cost function should be differentiable in order to calculate the error and it should only be used for the output layer as it utilises the comparison of the expected output and actual output. Some optimization algorithms include Gradient Descent and Adam.



(Figure 4)

Gradient Descent: The process for finding a local minimum (see Fig. 4) of a differentiable function in calculus. $\theta_{new} = \theta_{old} - \alpha G(\theta)$, where $G(\theta)$is the gradient of the loss function. The process starts with initialising with random values for the model's parameters (weights and biases). Then, the gradient is calculated where the direction and rate of the steepest increase of loss. With that, the parameters should be updated and then finally, the process should be iterated until the algorithm converges (the function stops decreasing significantly).

Adam: Combines two gradient descent methods with their best qualities, momentum and root mean square propagation. It begins with random values for the parameters and then the gradients are computed. Then it's first Moment estimation (Momentum) is complete by maintaining an exponentially decaying average of past gradients $m_t$which smooths out the updates and accelerates convergence in the relevant direction (decreasing the computational expense), $m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) \times g_t$, $g_t$is the

gradient at time t_1 and β_1(typically around 0.9) which controls the decay rate. The second Moment estimation (RMSprop) keeps an exponentially decaying average of past squared gradients, v_t which helps adapt the learning rate for each parameter yielding a more stable learning process, $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ where β_2 is typically around 0.999 to control the decay rate for the second moment estimate. Since m_t and v_t are initially 0, they are biassed to zero so Adam includes bias-correction terms: $\overline{m}_t = \frac{m_t}{1-\beta_1^t}$ and $\overline{v}_t = \frac{v_t}{1-\beta_2^t}$, $\theta_{t+1} = \theta_t - \alpha \frac{\overline{m}_t}{\sqrt{\overline{v}_t} + \epsilon}$ where ϵ is a very small constant like 10^{-10} for prevention of division by 0. This process is then iterated until convergence to the local minimum value.

Conclusion

In conclusion, the process of neural networks are heavily inspired by the structure and function of the human mind by mimicking how the brain processes information, learns from experience, and makes decisions. First, the network receives raw data through the input layer, where each neuron represents a feature of the data. Then, the data is passed through one or more hidden layers, where neurons compute weighted sums of the inputs, apply activation functions, and extract features or patterns. The network adjusts its weights based on the error between its predictions and the actual outcomes, using algorithms like gradient descent and techniques like backpropagation. In the final layer, the network produces a prediction or classification in the output layer, which is the result of all the computations performed in the previous layers. Finally, the process is repeated with multiple training examples, continuously refining the network's parameters until it generalises well to new, unseen data.

2.1.3 Types of Neural Networks

1. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are particularly effective in processing data with a grid-like topology, such as images. The architecture of a CNN typically includes convolutional layers,

pooling layers, and fully connected layers. The convolutional layers apply filters to the input images, extracting key features like edges, textures, and patterns. These filters slide over the input data, producing feature maps that represent various aspects of the images. Pooling layers follow the convolutional layers, reducing the spatial dimensions of the feature maps, which helps in minimizing the computational load and making the network more robust to slight variations in the input, such as shifts or rotations. Finally, fully connected layers aggregate the information from the feature maps to make predictions, such as identifying the class to which the input image belongs. CNNs are widely used in image classification, object detection, and medical image analysis, where they excel at identifying and differentiating between different visual elements.

2. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are designed to handle sequential data, making them ideal for tasks where the order of inputs is crucial. Unlike traditional neural networks, RNNs have loops within their architecture that allow information to persist across time steps, effectively giving the network a form of memory. This memory is stored in the hidden state, which is updated as the network processes each new element in a sequence. RNNs can therefore maintain context from earlier inputs, making them particularly useful for tasks like language translation, speech recognition, and image captioning, where understanding the sequence of words or sounds is essential. However, standard RNNs can struggle with long-term dependencies, where important information from earlier in the sequence needs to be remembered for a long time. This limitation has led to the development of more advanced architectures, such as Long Short-Term Memory networks (LSTMs).

3. Radial Basis Functions Networks

Radial Basis Function Networks (RBFNs) are a type of feedforward neural network that are distinct from other neural networks due to their unique approach to handling input data. In an RBFN, the input layer simply passes the data directly to the hidden layer without performing any computations. The hidden layer then applies radial basis functions, typically Gaussian functions, which calculate the distance between the input and certain predefined points (centres). These functions allow the network to model

complex, non-linear relationships in the data. The output layer of the RBFN then combines the outputs from the hidden layer to produce the final prediction, usually through a linear combination. RBFNs are particularly useful in applications such as time series prediction, where they can predict future data points in a sequence, and function approximation, where they are used to create smooth approximations of complex functions.

4. Long short-term memory networks

Long Short-Term Memory Networks (LSTMs) are a specialised form of Recurrent Neural Networks (RNNs) designed to overcome the limitations of standard RNNs, particularly the difficulty in capturing long-term dependencies. LSTMs achieve this by using a sophisticated mechanism involving memory cells and gates that control the flow of information. The memory cells store information over long periods, which allows LSTMs to maintain context for long sequences of data. The three types of gates—input, forget, and output gates—regulate the information that gets added to the memory cell, what remains in it, and what information is used to produce the output at each time step. This ability to selectively remember and forget information makes LSTMs highly effective for tasks such as handwriting recognition, language modelling, and video-to-text conversion, where understanding the sequence and context of data is crucial.

5. Multilayer perceptrons

Multilayer Perceptrons (MLPs) are a type of feedforward artificial neural network that consists of multiple layers of nodes (neurons) in a directed graph, where each layer is fully connected to the next one. MLPs are capable of learning complex relationships between input and output data, including non-linear relationships, thanks to their multiple layers and the use of non-linear activation functions like ReLU or Sigmoid. The learning process in MLPs involves backpropagation, where the error is propagated backward through the network to adjust the weights, thereby reducing the error over time. MLPs are versatile and can be used in a variety of applications, including face recognition, computer vision, and other tasks where pattern recognition in data is required.

6. Generative adversarial networks

Generative Adversarial Networks (GANs) consist of two neural networks, a generator and a discriminator, which are trained together in a game-theoretic framework. The generator creates synthetic data that mimics the real data, while the discriminator evaluates whether the data it receives is real or generated. Over time, the generator improves its ability to create realistic data as it tries to fool the discriminator, while the discriminator becomes better at distinguishing between real and fake data. This adversarial process leads to the generation of new data that closely resembles the training data. GANs have gained popularity for their ability to generate realistic images, videos, and other types of data, with applications ranging from creating art to simulating realistic environments for virtual reality.

7. Deep belief networks

Deep Belief Networks (DBNs) are a type of deep neural network that consists of multiple layers of stochastic, latent variables. Each layer in a DBN is a Restricted Boltzmann Machine (RBM), which is a type of neural network used to discover features in the data. The unique aspect of DBNs is that each layer can be pre-trained one at a time, with each layer learning to represent the data in terms of the features detected by the previous layer. After pre-training, the DBN is fine-tuned using backpropagation. This layered approach allows DBNs to learn complex representations of data, making them useful for tasks like image generation, motion capture data analysis, and other applications where deep hierarchical representations of data are beneficial.

8. Self-organising maps

Self-Organizing Maps (SOMs), also known as Kohonen maps, are a type of artificial neural network that is used for unsupervised learning. SOMs are unique because they map high-dimensional data into a lower-dimensional (typically two-dimensional) space, creating a grid where each neuron represents a cluster of similar data points. This mapping process allows SOMs to transform complex, high-dimensional datasets into visual maps that are easier to analyze and interpret. The neurons in a SOM compete to become the representative for each data point, and the neuron that best matches the input data wins and adjusts itself to better represent the input in future iterations. SOMs are often used for data

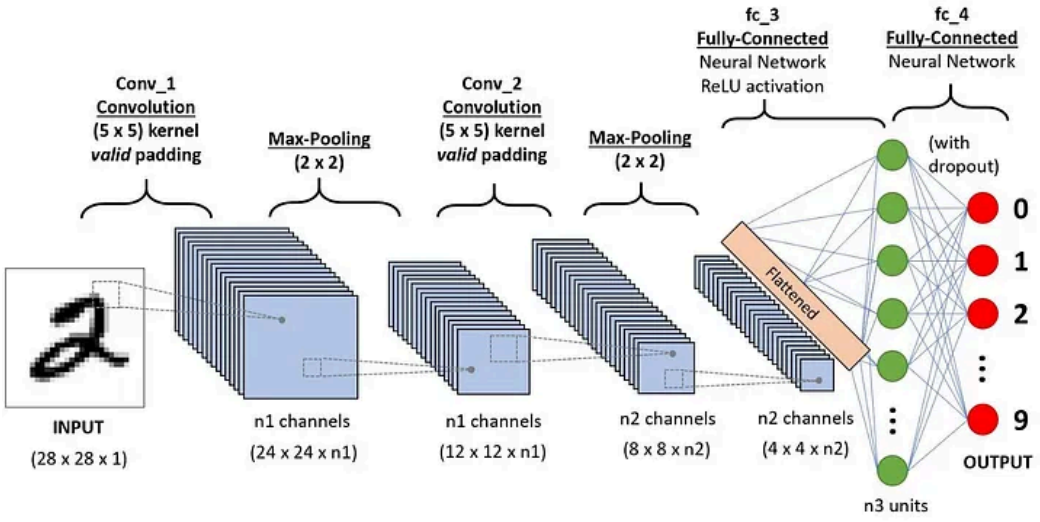
visualisation, such as displaying voting trends or organising astronomical data, where they can provide insights into the underlying structure of the data.

2.2 Convolutional Neural Networks (CNNs) For Image Processing

2.2.1 Architecture/Structure of CNNs

Introduction

Convolutional Neural Networks (CNNs) are deep learning models designed to extract features from images using convolutional layers, followed by pooling and fully connected layers, making them highly effective for tasks like image classification. These networks excel at identifying patterns in visual data, making them particularly well-suited for processing and analysing images and even other non-image data such as audio and signal data. Convolution is defined as "a function derived from two given functions by integration which expresses how the shape of one is modified by the other," which is quite relevant to CNNs as they utilise the convolution operation in its convolutional layers to generate a feature map, showing the strength of particular features at various locations in the input image.



(Figure 5)

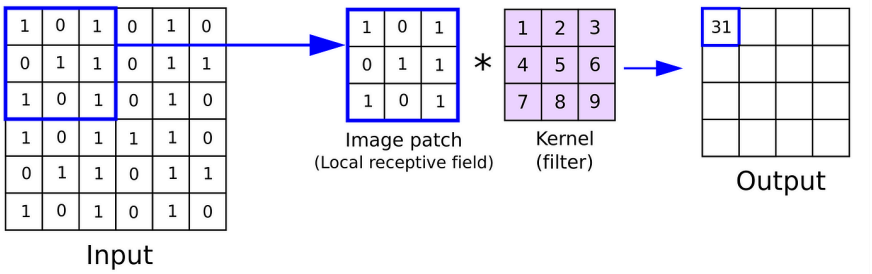
Kernel/Filter/Future Detectors

With CNNs, the kernel is a filter used to extract features for the convolutional layers after the input layer. The kernel is a small matrix, such as 3x3, 5x5, or 7x7. The size of the kernel is denoted as

$k \times k$, where k is the size of the kernel (e.g., 3 for a 3x3 kernel). The kernel's dimensions should be smaller than the dimensions of the input image. For example, if your image is 32x32 pixels, a typical kernel size might be 5x5.

The kernel slides across the input image depending on the Stride number, starting from the top-left corner, moving to the right "Stride times", and then down "Stride times", covering the entire image. At each position, the kernel is placed over a corresponding subregion of the image.

For each position of the kernel on the image, you perform an element-wise multiplication between the kernel's values and the corresponding pixel values of the image. After multiplying, you sum all the resulting products to get a single number. This number represents a pixel in the output feature map.



(Figure 6)

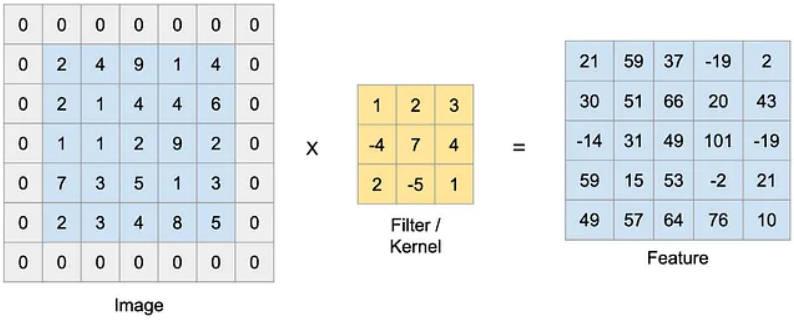
The output of the convolution operation is a feature map, which shows how strongly certain features (like edges or textures) are detected in the image at different positions. The feature map's dimensions depend on the size of the kernel and the stride (how far the kernel moves each step), and it is typically smaller than the original image.

For visualisation of the process, follow this link moving visualization:

Padding Layers

Padding refers to the number of pixels added to an image during the process by the kernel of a CNN. For example, padding set to 0 in a CNN: then every pixel value added is of value 0.

In order to avoid the image going small and preserve the original size, pixels will be added to the outside the image.



(Figure 7)

Pooling Layers

Pooling is a crucial operation to CNNs. The process includes reducing the spacial dimensions (width and height) of the feature maps generated by the convolutional layers. The goal of this is to downsample the feature maps while retaining the important pixels, which reduces the computational complexity. The different types include max pooling, average pooling, and global pooling.

Max pooling: uses the maximum value as the output for that region.

Average pooling: uses the average value as the output for that region.

Global pooling: reduces the entire feature map by taking the average or maximum of all values in the feature map as the output. Typically used in the final stages of a CNN for the preparation of data to the fully connected layers.

Flattening Layer

The process of converting the rows of pooled feature maps into long linear vectors.

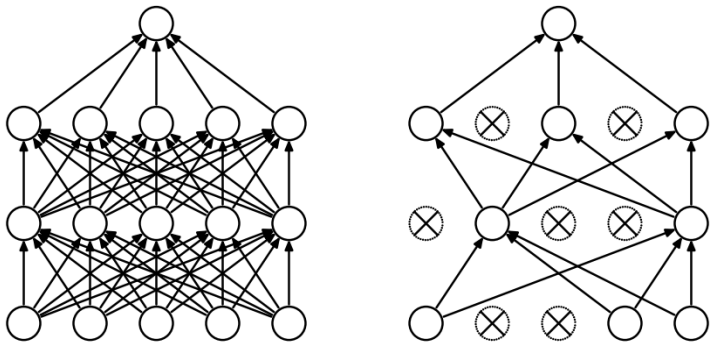
Fully-Connected Layer

Consists of the weights, biases, and neurons and is used to connect the neurons between two different layers which are usually placed before the output layer and then form the last few layers of a CNN architecture.

Dropout Layer

This layer is a preventive measure for overfitting, reducing redundancy, and encouraging co-adaptation. It randomly “drops out” or sets to zero a certain percentage of neurons in a layer during

each training iteration which prevents the reliance on a single neuron by focusing on other features as well. The process includes randomly selecting a fraction of the neurons and then scaling the outputs where the remaining “un-dropped-out” active neurons are scaled up in order to compensate for the neurons being inactivated (dropped out). This is only completed during training to “diversify” the data. This randomization can be demonstrated through the comparison of the before and after of dropout filters.



(a) Standard Neural Net (b) After applying dropout.

(Figure 8)

Activation Function

These functions are utilised as means of deciding whether a neuron should be activated or not. Typical activation functions include: ReLU, Softmax, tanH and the Sigmoid functions.

2.2.2 Applications of CNNs

Image Classification: CNNs are widely used in image classification tasks, where the goal is to categorise images into predefined classes.

- For example: recognizing objects in images, such as determining whether an image contains a cat, dog, car, etc.
- Use cases: ImageNet competition, Google Photos, Facebook's automatic tagging system.

Object Detection: Object detection involves not only classifying objects in an image but also locating them by drawing bounding boxes around them.

- For example: Detecting multiple objects like pedestrians, cars, and traffic signs in a single image.
- Use Cases: Autonomous vehicles, security systems, facial recognition.

Image Segmentation: Image segmentation is the process of partitioning an image into segments or regions, each corresponding to different objects or areas.

- For example: Identifying and colouring different parts of an image, such as separating foreground objects from the background.
- Use Cases: Medical imaging (e.g., segmenting tumours in MRI scans), satellite imagery analysis.

Facial Recognition: CNNs are used to identify and verify faces in images or videos.

- For example: Unlocking smartphones using facial recognition, identifying individuals in a crowd.
- Use Cases: Security systems, biometric authentication, social media tagging.

Handwriting Recognition: CNNs can be trained to recognize and interpret handwritten characters or digits.

- For example: Recognizing handwritten digits in the MNIST dataset.
- Use Cases: Automated processing of handwritten documents, postal address recognition, digitising historical records.

Medical Image Analysis: CNNs are used to analyse medical images for diagnosing diseases or identifying abnormalities.

- For example: Detecting tumours in X-rays, classifying skin lesions, analysing MRI scans.
- Use Cases: Computer-aided diagnosis, personalised medicine, radiology.

Autonomous Vehicles: CNNs play a crucial role in enabling autonomous vehicles to perceive their environment by processing visual data.

- For example: Lane detection, obstacle recognition, traffic sign recognition.
- Use Cases: Self-driving cars, drones, advanced driver-assistance systems (ADAS).

Video Analysis: CNNs can be extended to analyse video data, enabling the recognition of actions, activities, or events over time.

- For example: Detecting and classifying actions in security camera footage, recognizing gestures in video streams.
- Use Cases: Video surveillance, sports analytics, human-computer interaction.

Natural Language Processing (NLP): While CNNs are traditionally used for visual data, they can also be applied to NLP tasks by treating text as a sequence of data.

- For example: Text classification, sentiment analysis, language modelling.
- Use Cases: Spam detection, sentiment analysis on social media, document classification.

Art and Creativity: CNNs can be used in creative applications to generate new images, transform styles, or even compose music.

- For example: Style transfer, where the style of one image is applied to another, or generating new artwork using GANs (Generative Adversarial Networks) based on CNNs.
- Use Cases: Digital art, content creation, entertainment.

Robotics: CNNs help robots understand and navigate their environment by processing visual inputs from cameras and other sensors.

- For example: Object recognition and manipulation, scene understanding for navigation.
- Use Cases: Industrial automation, autonomous drones, home robots.

Speech and Audio Processing: CNNs can be applied to audio signals, especially in combination with spectrograms, to perform tasks like speech recognition and music genre classification.

- For example: Converting speech to text, classifying music genres based on audio features.
- Use Cases: Voice assistants, music recommendation systems, audio classification.

Gaming and Virtual Reality: CNNs enhance the gaming experience by processing visual data to generate realistic graphics or by recognizing player actions.

- For example: Real-time character recognition in video games, dynamic scene rendering in VR.
- Use Cases: Immersive gaming, VR simulations, AI-driven game characters.

Conclusion

CNNs are powerful tools for a wide array of applications, particularly those involving image and video data. Their ability to automatically extract and learn hierarchical features from raw data makes them ideal for tasks such as image classification, object detection, facial recognition, and medical image

analysis. Beyond visual data, CNNs have also found applications in natural language processing, audio processing, robotics, and even creative fields like art and music generation.

2.3 Comparison of Frameworks with Python

Access to my GitHub repository which uses the CIFAR-10 data: repository

2.3.1 Kesar Model with Tensorflow Backend

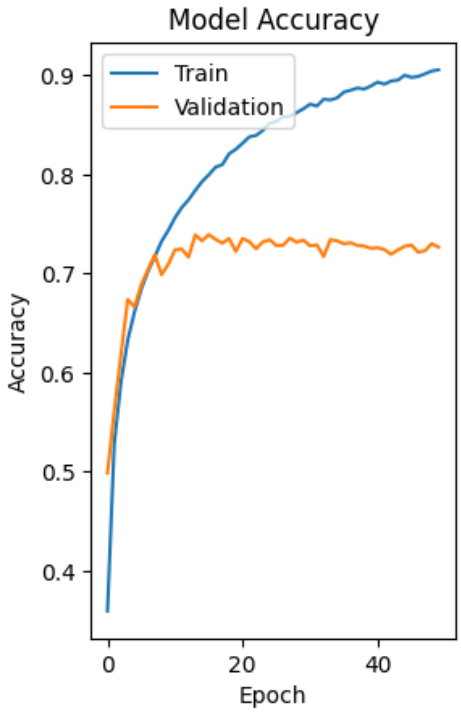
In order to create and train a CNN for image classification, one method involves the API Kesar which runs on TensorFlow via Python.

The Process

1. Import the Kesar, TensorFlow, and Matplot library packages.
2. Load the data, CIFAR-10 is a dataset consisting of 32x32 colour images in 10 different classes, with 6000 images per class (60000 total), where the dataset is separated into 50000 training images and 10000 test images.
3. Split the data into input training sets (x_train), output training sets (y_train), output test sets (x_test), and output test sets (y_test).
4. Normalise the pixel values to avoid domination.
5. One-hot-encode the labels, since the output is a probability distribution over multiple classes in order to utilise categorical cross entropy which must be used with Neural Networks.
6. Data encode the input training data, which creates diversity within the set by creating modified versions of the original images.
7. Visualise the augmented data using matplotlib.pyplot.
8. Build the CNN model using a convolutional layer and a pooling layer three times with the ReLU activation functions, flatten, densen (with ReLU), dropout half of the data, and then densen again (with softmax). Finally, compile using the Adam optimization function.

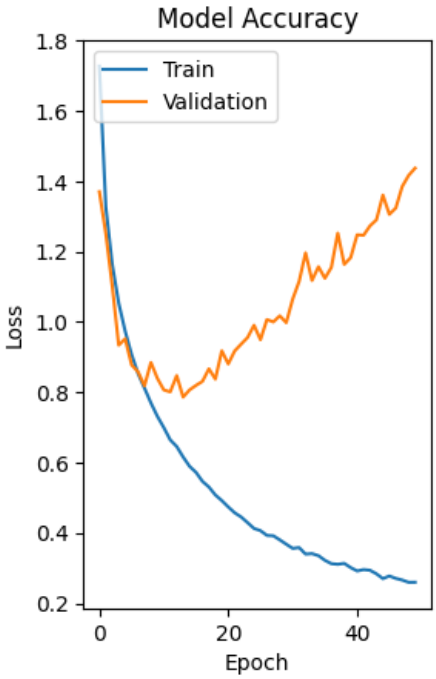
9. Train the model by fitting the training inputs and outputs with fifty epochs (iterations) and batch sizes of 64, where the validation set are the test data splits.
10. Visualise the training and validation accuracy and the training and validation loss.
11. Save the model.

Epoch vs. Accuracy and Epoch vs. Loss



The blue line indicates the accuracy on the training dataset over time. As expected, the training accuracy increases steadily with each epoch, reaching over 90% by the end of training. The orange line represents the accuracy on the validation dataset. Initially, the validation accuracy increases along with the training accuracy, indicating that the model is learning generalizable patterns. However, after a certain point (around epoch 10), the validation accuracy plateaus and even starts to show some fluctuations. The model's training accuracy continues to increase throughout the training process, indicating that the model is successfully learning from the training data. The validation accuracy plateaus fluctuates after around 10-15 epochs. This suggests that the model may be starting to overfit the training data—learning details and noise that do not generalise well to unseen data. Since the training accuracy continues to increase while the validation accuracy plateaus or slightly decreases, it's likely that overfitting is occurring.

The blue line represents the loss on the training dataset over time. As expected, the training loss decreases steadily as the model learns from the training data. This indicates that the model is improving its performance on the training data. The orange line shows the loss on the validation dataset. Initially, the validation loss decreases, which is good, but then it starts to increase again as training progresses. This suggests that the model is beginning to overfit the training data. The steadily decreasing training loss



indicates that the model is learning from the training data and improving over time. This is expected behaviour. The validation loss initially decreases but then starts to increase after a certain number of epochs. This is a classic sign of overfitting, where the model is starting to memorise the training data instead of learning to generalise to unseen data. The divergence between the training loss and the validation loss is a key indicator of overfitting. As the training loss continues to decrease, the model's ability to generalise to the validation set diminishes, leading to an increase in validation loss.

2.3.2 PyTorch

In order to create and train a CNN for image classification, one method involves the API Pytorch via Python.

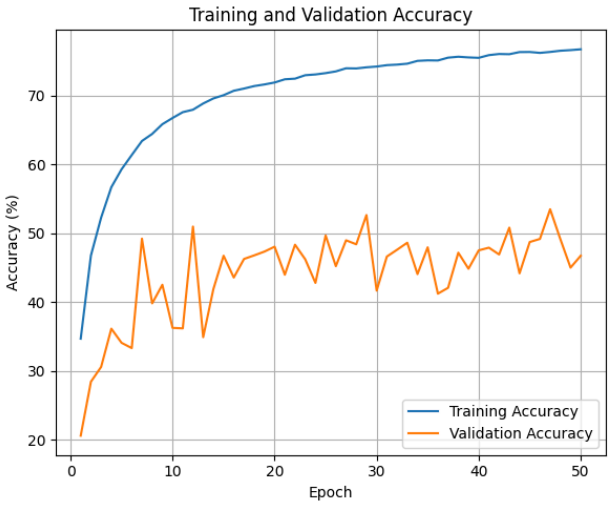
The Process

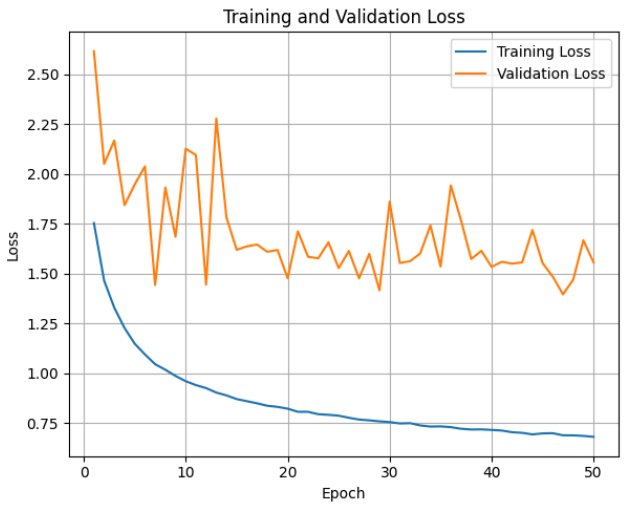
1. Import the PyTorch and Matplot library packages.
2. Load the data, CIFAR-10 is a dataset consisting of 32x32 colour images in 10 different classes, with 6000 images per class (60000 total), where the dataset is separated into 50000 training images and 10000 test images.
3. Split the data into input training sets (x_train), output training sets (y_train), output test sets (x_test), and output test sets (y_test).
4. Normalise the pixel values to avoid domination.
5. Create tensors by permuting the parameters of the input data splits.
6. Create a function for the data using matplotlib.pyplot.

7. Build the CNN model using a convolutional layer and a pooling layer three times, flatten, apply the first connected layer with ReLU, dropout half of the data with ReLU, and then apply the output layer by specifying 10 classes. Finally, create a net instance of the model.
8. Define a loss function as a cross entropy loss and optimizer function (Adam).
9. Initialise the tensorboard writer and log the model.
10. Define the data augmentation transformations to include padding, rotation, cropping, and normalisation.
11. Define a function that trains one epoch by iterating through the batch, applying data augmentation, zero-ing the gradients, creating train outputs with the batch of test inputs, and keeping track of overall and epoch loss and accuracy.
12. Define a function that evaluates the performance of the neural network on a validation set over one epoch.
13. Create a function to visualise the training and validation accuracy and the training and validation loss.
14. Train the model by iterating 50 times (50 epochs) and then visualise the data.
15. Save the model.

Epoch vs. Accuracy and Epoch vs. Loss

The training accuracy steadily increases over time, which is expected as the model learns from the training data. It eventually stabilises around 70-75%. The validation accuracy, however, fluctuates significantly and does not follow the same upward trend as the training accuracy. This indicates that while the model is improving on the training data, it is struggling to generalise to the validation data, which could be a sign of overfitting.





The training loss decreases consistently, which is expected as the model becomes better at minimising errors on the training set. The validation loss, similar to the accuracy, fluctuates significantly and does not show a clear trend of improvement. This further suggests that the model may be overfitting to the training data. The gap between the training accuracy and validation accuracy, along with the fluctuation in validation loss, suggests that the model is likely overfitting to the training data. While the model continues to improve on the training data, it does not generalise well to unseen data, which is reflected in the erratic validation metrics. The training process appears effective in terms of reducing loss and improving accuracy on the training data. However, the lack of consistent improvement in the validation metrics indicates that the model may be learning noise or specific patterns in the training data that do not apply to the validation set.

2.3.3 Comparison

Ease: Keras with TensorFlow was a lot simpler to program as a beginner. There are fewer steps and simpler ways to code processes with Keras and TensorFlow, whereas with PyTorch certain steps required separate functions to be created and run for each of the iterations of the epochs – each step had to be specified and controlled. Another aspect was plotting, both utilised MatplotLib, however, PyTorch required the accumulation of the variables to be plotted and required accumulations of loss and accumulation through the iterations of the epochs. Finally, researching the process and understanding the code of the Kesar TensorFlow duo was a lot simpler as each part of the process was simply done with single lines and no iterations with for loops. Thus, Keras with TensorFlow were a lot more simple and easy to use and apply to the CIFAR-10 dataset.

Performance:

In terms of speed and efficiency, the Keras TensorFlow combination was a lot faster on average, taking about 15 minutes to run through the fifty epochs, whereas PyTorch would take 110 minutes to run through the fifty epochs. Thus, Keras' performance with TensorFlow was clearly the victor as it ran 7 times faster than its counterpart.

The first pair of graphs shows better training performance, with a higher final training accuracy (~90%) compared to the second pair (~70-75%). The consistent decrease in training loss in both cases is good, but the higher training accuracy in the first graph indicates more effective learning. Both pairs of graphs show signs of overfitting, but the first pair demonstrates a more controlled and steady pattern in validation accuracy and loss. The second pair shows erratic behaviour in the validation metrics, suggesting more severe overfitting or other issues. The first pair of graphs is generally better because it shows higher training accuracy and a more consistent, although still slightly overfitting, validation performance. The second pair of graphs indicates significant fluctuation in validation performance, which suggests that the model is struggling more to generalise to unseen data.

3.0 CONCLUSION

In this project, the foundational concepts of neural networks and their evolution into Convolutional Neural Networks (CNNs) are explored, applying these insights to a practical image classification task using the CIFAR-10 dataset. Through a detailed analysis of neural network architectures, we examined various types of neural networks, including CNNs, Recurrent Neural Networks (RNNs), Long Short-Term Memory Networks (LSTMs), and others, providing a comprehensive understanding of their structures, functions, and applications. The core of this project involved implementing CNNs using two popular deep learning frameworks: Keras with TensorFlow and PyTorch. Each framework was utilised to build and train a CNN model, with the objective of comparing their performance, usability, and effectiveness in handling image classification tasks. By closely

monitoring the training and validation accuracy and loss over multiple epochs, the aim was to assess how well each model learned and generalised unseen data.

Findings

Model Performance of Keras/TensorFlow: The model implemented with Keras and TensorFlow demonstrated a strong training performance, achieving over 90% accuracy. However, the validation accuracy plateaued after a certain number of epochs, indicating potential overfitting. The validation loss also began to increase, reinforcing the need for regularisation techniques to improve generalisation. Keras, with its high-level API and integration with TensorFlow, provided a user-friendly and efficient environment for building and training the CNN model. It abstracted many of the complexities, making it an excellent choice for rapid prototyping and beginners.

PyTorch: The PyTorch model, while achieving good training accuracy, showed significant fluctuation in validation accuracy and loss, suggesting more pronounced overfitting or difficulties in generalising to the validation data. The training accuracy stabilised around 70-75%, which is lower than the Keras/TensorFlow model. PyTorch offered greater flexibility and control, allowing for more in-depth experimentation with the model architecture and training process. However, this came at the cost of a steeper learning curve and more complex and computationally expensive code.

Recommendations

Both models displayed signs of overfitting, but the PyTorch model was particularly prone to fluctuating validation performance. This highlighted the importance of implementing regularisation techniques, such as dropout or L2 regularisation, and possibly employing early stopping to prevent excessive overfitting.

1. Future iterations of this project should incorporate more robust regularisation methods to prevent overfitting and improve the model's ability to generalise to new data.
2. Hyperparameter Tuning: Experimenting with different hyperparameters, such as learning rate, batch size, and number of layers, could yield better results and more stable validation performance.

3. Cross-Validation: To ensure the robustness of the models, cross-validation could be used to assess performance across multiple subsets of the data, providing a more comprehensive evaluation of the model's generalisation capabilities.
4. Further Exploration: Expanding the project to include other architectures, such as ResNet or VGG, could provide additional insights into the strengths and weaknesses of different CNN architectures when applied to image classification tasks.

Final Thoughts

This project has successfully demonstrated the application of CNNs in image classification using two of the most popular deep learning frameworks: Keras with TensorFlow and PyTorch. By writing the report and developing the code, a deeper understanding of both the theoretical and practical aspects of neural networks was achieved. The insights gained from comparing the performance and usability of these frameworks provide valuable guidance for future projects, whether in research, development, or deployment in real-world applications. As deep learning continues to evolve, the tools and techniques explored in this project will remain central to the advancement of image processing and other machine learning tasks.

References

Chen, J. (2024, July). *What is a neural network?*. Investopedia. https://www.investopedia.com/terms/n/neuralnetwork.asp

Hardesty, L. (2017, April). *Explained: Neural networks*. MIT News | Massachusetts Institute of Technology. https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

Staff. (2024). 8 common types of neural networks. Coursera. https://www.coursera.org/in/articles/types-of-neural-networks

What is a neural network? - artificial neural network explained. AWS - Amazon Web Services. (2023). https://aws.amazon.com/what-is/neural-network/

What is a neural network?. IBM. (2021, October 6). https://www.ibm.com/topics/neural-networks

Figure 1: Chatterjee, C. C. (2019). Classic CNN Structure. Basics of the Classic CNN. How does a classic CNN (Convolutional Neural Network) work? Medium. Retrieved from https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add.

Figure 2: Raj, R. (2023). Schematic Diagram of a Neural Network. Components of an Artificial Neural Network. Code Algorithms Pvt. Ltd. Retrieved from https://www.enjoyalgorithms.com/blog/components-of-ann.

Figure 3: AD Elster, ELSTER LLC. (2024). Biological and Artificial Neurons. Neural Network Definition and Components. Retrieved from https://s.mriquestions.com/what-is-a-neural-network.html.

Figure 4: Dongas, N. (2025). Minimum Cost Visual. Gradient Descent in Machine Learning: A Basic Introduction. Built In. Retrieved from https://builtin.com/data-science/gradient-descent.

Figure 5: Dharmaraj (2022). The CNN Process. Convolutional Neural Networks (CNN) — Architecture Explained. Medium. Retrieved from https://medium.com/@draj0718/convolutional-neural-networks-cnn-architectures-explained-716fb197b243.

Figure 6: Sankhe, S. (2021). Kernel Process. Convolutional Neural Network. Medium. Retrieved from https://siddharthsankhe.medium.com/convolutional-neural-network-dc942931bff8.

Figure 7: Stephen, A. (2021). Padding in Kernaling. Convolutional Neural Networks: Leveraging the Power of Convolution for Machine Learning Tasks. Retrieved from https://medium.com/@acheampongstephen392024/the-role-of-convolution-in-image-processing-and-computer-vision-f96cf052a0b8

Figure 8: Budhiraja, S. (2021). Dropout Process . Dropout in (Deep) Machine learning Medium. Retrieved from https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5.