Comparative Analysis of Machine Learning Models for Image Classification: From Traditional Methods to Deep Learning Architectures

Qian J Oscar[1]

[1]University of Oslo, Problemveien 11, 0313 Oslo, Norway

Abstract

This research report compares the performance of various machine learning models on image classification tasks, evaluating their accuracy, computational efficiency, and scalability. The models studied include Feed-Forward Neural Networks (FFNN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Support Vector Machines (SVM), and Logistic Regression. Some of these models are implemented from scratch, while others leverage advanced machine learning libraries like TensorFlow to provide a balance of control and computational optimization. This study provides a comprehensive analysis of each model's strengths and limitations in handling image classification, with a focus on identifying suitable models for real-world applications. The findings aim to inform choices for researchers and practitioners in selecting models based on performance trade-offs in accuracy, speed, and interpretability.

Keywords

Image Classification, Neural Networks, Machine Learning, Hyperparameter Selection

1. Introduction

In recent years, image classification has emerged as a central task in machine learning, with applications spanning healthcare, autonomous driving, surveillance, and numerous other fields. As the complexity of image datasets have grown, selecting the most appropriate model for image classification has become a critical challenge. The vast landscape of machine learning algorithms offers a variety of architectures—ranging from classical statistical methods to deep learning architectures—each with unique advantages for certain data characteristics.

This paper aims to evaluate and compare the performance of six machine learning models for image classification: FFNN, CNN, RNN, SVM, and Logistic Regression. These models were chosen for their distinct structural properties and their varying approaches to handling spatial information in images. Specifically, CNNs and RNNs represent advanced deep learning architectures well-suited for complex feature extraction and sequence modeling, while traditional methods such as SVM, Logistic Regression, and Random Forests provide interpretable baselines.

By implementing some models from scratch and utilizing libraries such as TensorFlow for others, this study examines each model's practical strengths and challenges. The analysis will focus on evaluating classification accuracy, computational efficiency, and interpretability to identify models best suited for image classification tasks across diverse applications.

jianhenq@uio.no (Q. J. Oscar)
https://github.com/oscarqjh (Q. J. Oscar)

2. Theory

2.1. Logistic Regression

Logistic Regression is a classification algorithm used when the target variable is binary. Unlike linear regression, logistic regression models the probability that a given instance belongs to a particular class. The logistic sigmoid function constrains the output between 0 and 1 and is defined as:

$$logit(p) = log(\frac{p}{1-p})$$

This can be rewritten as:

$$\phi(z) = \frac{1}{1+e^{-z}}$$

where z is the linear combination of weights and the inputs, defined as:

$$z = \boldsymbol{w^T x}$$

We then aim to maximise the probability of seeing the observed data. For this purpose, we define the cost function, also known as cross entropy as:

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right) \tag{1}$$

In order for us to use this cost function for gradient methods for optimisation, we have to add a regularisation term. This is to prevent the weights from exploding to infinity [1]. For this project, we will use the L2 regularisation where we add the L2 regularisation term to equation (2), hence the new cost function is defined as:

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right) + \lambda \sum_{j=1}^{m} (\beta_1)^2$$

To minimise the cost function using gradient methods, we will find its first derivative:

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{p}) + \lambda \cdot \beta$$

We can also generalise this to a multiclass setting, which is known as multinomial logistic regression [2].

2.2. Feed Forward Neural Network (FFNN)

2.2.1. Perceptron Model

The perception model is considered one of the best type of ANN and has since influenced the development of many other NNs. The perceptron model starts by multiplying each input by its corresponding weight, then sums these weighted inputs to produce a total weighted sum. This

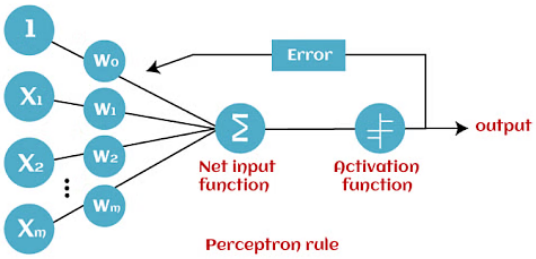

Figure 1: Illustration of a single perception model

(https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron#:~:text=Single%20Layer%20Perceptron%20model%3A%20One,separable%20objects%20with%20binary%20outcomes.)

sum is then passed through an activation function, $\boldsymbol{f}$, often referred to as the step function, to generate the final output.

The figure above shows the architecture of a single perception model. The FFNN developed in this project is a much higher level derivation of the perception model, where instead of a single node, FFNN utilise multiple layers, each with multiple nodes.

2.2.2. Activation Functions (AFs)

Activation functions have to be non-constant, bounded, monotonically-increasing and continuous to fulfill the universal approximation theorem. It is indeed a very important area of research for NNs and has been studied extensively [3]. In this project, we will only explore a subset of activation functions.

Sigmoid

The sigmoid function $\sigma : \mathbb{R} \to [0,1]$ maps any real-valued number to a value between 0 and 1. It is often used as an activation function in the output layer of a binary classification neural network and is defined as:

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

The first derivative of sigmoid is defined as:

$$\sigma^{'}(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Softmax

The softmax function converts a vector of K real values into another vector of K real values that add up to 1. Regardless of whether the input values are positive, negative, zero, or greater than one, the softmax function transforms them into values ranging between 0 and 1, making them interpretable as probabilities [4]. It is defined as:

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Table 1
Advantage and disadvantage of primary AFs

AFs	Diminishing gradients	Computational inefficiency
sigmoid	Yes	Yes
softmax	Yes	Yes
ReLU	Partial	No
LReLU	No	No

where, z_i represents the input to the softmax function for class i, and the denominator is the sum of the exponentials of all the raw class scores in the output layer. We almost always uses the Softmax function together with the Cross Entropy Loss function in the output layer of the NN for classification task. Hence, the derivative of Softmax is often calculated together with the cross entropy loss function. Hence, we simply pass the derivative of the cost function down instead [5].

Rectified Linear Units (ReLU)
The Rectified Linear Unit (ReLU) function maps any negative input to 0 and any positive input to itself. It is often used as an activation function in the hidden layers of a neural network. It is defined as:

$$ReLU(x) = \begin{cases} x & : x \geq 0 \\ 0 & : x < 0 \end{cases}$$

The first derivative of ReLU is defined as:

$$ReLU^{'}(x) = \begin{cases} 1 & : x \geq 0 \\ 0 & : x < 0 \end{cases}$$

Leaky ReLU
Vanishing gradient is the main problem with ReLU AF which is caused due to the non-utilization of negative values. A Leaky Rectified Linear Unit (LReLU) is the extension of ReLU by utilizing the negative values [6]. The LReLU is defined as:

$$LReLU(x) = \begin{cases} x & : x \geq 0 \\ 0.01x & : x < 0 \end{cases}$$

The first derivative of LReLU is defined as:

$$LReLU^{'}(x) = \begin{cases} x & : x \geq 0 \\ 0.01x & : x < 0 \end{cases}$$

The selection of AF is crucial to the performance of the NN, in some cases, a unsuitable AF can lead to non-convergence of the model. Table 1 shows some comparison of the different AFs explored in this project.

2.2.3. Feed-forward

In the forward pass, each layer is defined by number of nodes and its AF. The input, which can be either a vector or matrix depending on whether batch input is implemented, is passed to the AF. Then for subsequent layers, the output of the previous layer will be used as the input of that layer. Output of a layer can be defined as:

$$z^l = (W^l)^T a^{l-1} + b^l$$

where W is the weights of layer l, a is the output from AF of layer $l-1$ and b is the bias of layer l.

2.2.4. Back-propagation

The back-propagation is the main mechanism to how the NN "learn". It is basically a gradient method to optimize the cost function based on the derivative of the cost function given a certain learning rate.

The cost function which we want to optimise in this project is given by equation (1) - MSE - for regression tasks, and equation (2) - Cross Entropy Loss - for classification tasks.

For each layer, we want to optimise the cost function with respect to the weights, I will then utilise the chain rule have to find

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w}$$

where $\frac{\partial C}{\partial a}$ is given by $\frac{\partial C}{\partial z} \cdot W_{l-1}^T$ for non-output layers, and $\frac{\partial z}{\partial w}$ is just the layer input. For simplicity, we will define δ as,

$$\delta = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}$$

After obtaining the gradient of cost function for a layer, I will then update the weights and biases according to the gradient method that was chosen. The simplest example for how the weight and biases are updated is

$$W \leftarrow W - lr \cdot \frac{\partial C}{\partial w}$$

$$b \leftarrow b - lr \cdot \delta$$

where W and b is the weights and bias respectively.

2.3. Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) excel in image classification by automatically learning features directly from raw data, bypassing manual feature engineering. Through convolutional layers, CNNs progressively capture feature hierarchies: early layers detect basic patterns like edges, while deeper layers recognize complex shapes. Key principles like sparse connectivity (each output connects to only a small image patch) and parameter sharing (the same weights

are applied across patches) make CNNs efficient and highly effective at extracting and utilizing relevant features for classification tasks [7].

To put simply, in CNN we introduced the Convolutional Layer and Pooling Layer before feeding the output into a Dense Layer for classification. The new layers essentially act as feature extractor, which I will be discussing below.

2.3.1. Convolutional Layer

In mathematics, **convolution** represents the integral on the product of two functions. And as far as machine learning is concerned, we are only interested in the **discretised** convlution defined as,

$$\mathbf{Y}(I,j) = (\mathbf{X} * \mathbf{W})(I,j) = \sum_m \sum_n \mathbf{X}(i-m, j-n)\mathbf{W}(m,n)$$

where $\mathbf{X}$ is a two-dimensional mxn input image, while $\mathbf{W}$ is a two-dimensional kernel/filter. Below is a simple visualisation of the convolution step with a kernel size of (2, 2) and stride size of (1, 1),

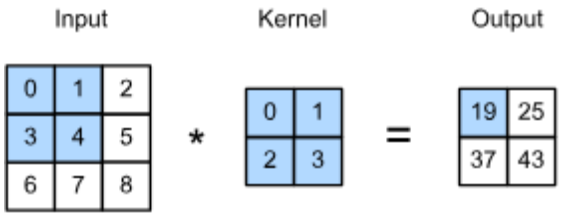

Figure 2: Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$ https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html

2.3.2. Padding

From Figure 2, we can see some issues in applying convolution in this way, that is the corner and edge elements will be less involved in the calculation of the final output as compared to the middle element. This can result in a bias toward the middle elements, i.e. information of middle elements will have a greater impact on the output. One way to solve this issue is to add paddings to the input matrix, effectively moving the edge and corner elements towards the center, hence allowing them to have a greater impact on the final output [7]. This ultimately leads to better performance of the model[8]. We will discuss the different ways to pad the input matrix below.

Full Padding

The idea of full padding is to allow for every position of the kernel to be applied, and the calculation of the padding size, p is defined as,

$$p = m - 1$$

where m is the kernel size

Same Padding
The idea of same padding is add padding such that the output has the same dimensions as the input given that the stride size is (1, 1). The padding size, p can be calculated as,

$$p = \frac{m-1}{2}$$

where m is the kernel size. This is the most common method of padding used in CNN since it preserves the dimentionality.

Valid Padding
The idea of valid padding is for the kernel to only convolve over valid input positions without extending beyond the edges, and this basically means no padding is applied. The padding size, p can be calculated as,

$$p = 0$$

The output size of all the above padding methods can be calculated as:

$$OutputSize = \left\lfloor \frac{n + 2p - m}{s} + 1 \right\rfloor$$

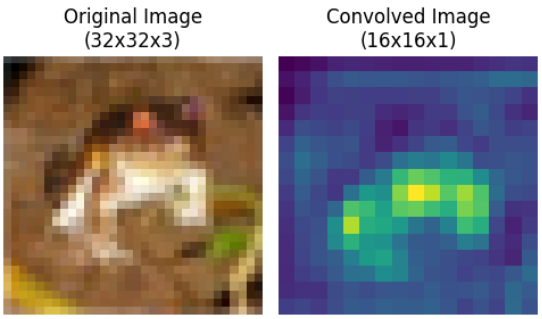

Figure 3: Example of convolution applied to a real 32x32x3 image with a 4x4 kernel, stride=2, and "same" padding with own implementation.

2.3.3. Pooling Layer

Convolutional layers in CNNs apply filters to input images to create feature maps, allowing early layers to detect simple patterns and deeper layers to identify complex shapes or objects. However, they record precise feature locations, making models sensitive to minor shifts or rotations in the input.

To address this, CNNs use down-sampling—often with pooling layers—to reduce feature map size and retain key structures. Pooling, especially max pooling, summarizes regions by selecting maximum values, making models more robust to small translations and improving generalization. Figure 4 shows the example of max pooling and average pooling.

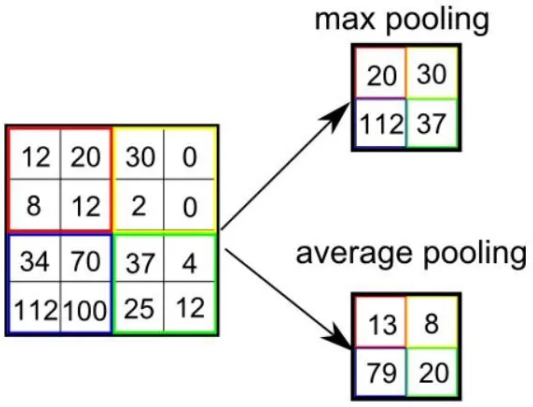

Figure 4: Example of max-pooling and average-pooling with a filter size of (2, 2) https://medium.com/analytics-vidhya/convolutional-neural-networks-an-intuitive-approach-part-2-729bfb5e4d87

Figure 5: Example of max and average pooling applied on a real 32x32x3 image with a 2x2 filter, stride=2, and "same" padding with own implementation.

2.3.4. Dropout Layer

Dropout is a regularization technique used in neural networks to prevent overfitting. During training, a fraction of the neurons in a layer is randomly "dropped" (set to zero) at each iteration with a probability, p_drop. This forces the model to learn redundant representations, making it more robust and generalizable. During inference, dropout is not applied, and all neurons are active. To maintain consistent activation levels between training and inference, the activations are scaled during training (commonly using inverse dropout). Dropout can be seen as a form of ensemble learning, where the model trains multiple sub-models and averages their predictions during inference [9].

2.4. Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNNs) extend feedforward networks by including backward connections, allowing them to process sequential data with dependencies, such as time series, text, and audio. Unlike models assuming independent data, RNNs are ideal for tasks like stock price prediction, autonomous driving, and natural language processing, as they can handle sequences of varying lengths with interdependent elements [10].

2.4.1. Long Short-Term Memory (LSTM)

LSTM networks were designed to overcome the vanishing gradient problem in traditional Recurrent Neural Networks (RNNs) by addressing long-term dependencies in sequential data. They consist of specialized cells in the hidden layers that use three gates—input, output, and forget gates—to regulate the flow of information. This structure allows LSTMs to remember or forget important information, making them capable of learning from long sequences where distant context is necessary. For example, in language tasks, LSTMs can retain context from earlier sentences to improve predictions, even when the relevant information is far back in the sequence.

2.4.2. Gated Recurrent Units (GRU)

GRUs are a simplified variant of LSTMs, designed to tackle the same issue of short-term memory in RNNs. Instead of using separate cell states, GRUs rely on hidden states and have two gates: a reset gate and an update gate. These gates control the amount of information to reset or retain at each time step. The GRU's architecture is less complex than that of LSTMs, making it computationally more efficient and requiring fewer parameters, which results in faster training. Due to these advantages, GRUs are often preferred for applications with limited resources or in real-time systems.

2.5. Support Vector Machines (SVMs)

A Support Vector Machine (SVM) is a powerful and versatile machine learning technique used for classification, regression, and even outlier detection, especially effective on complex but smaller datasets. SVMs work by finding a hyperplane that best separates classes (in classification) or fits data (in regression), creating a margin around this hyperplane to improve generalization [11].

2.5.1. Hard Margin

SVM aims to find a hyperplane that will obtain the maximum margin between the classes of data points. In order to achieve this, we will look at the dual optimisation problem:

$$\max_{\alpha \geq 0} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}(\sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m x_n^T x_m), where \sum_{n=1}^{N} \alpha_n y_n = 0$$

where the solution to this yields a Lagrange multiplier for each point in the dataset.

This form of SVM forces all points to be classified correctly, which leads to issues when the hyperplane does not exist for a certain dataset.

2.5.2. Soft Margin

To generalise hard margin SVM, we introduce a new constant, C, which adds the constraint to α for each point:

$$\max_{\alpha} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}(\sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m x_n^T x_m), where \sum_{n=1}^{N} \alpha_n y_n = 0 \& 0 \leq \alpha_n \leq C$$

This basically means now we only consider support vectors that are not violating (where $0 < \alpha < C$).

2.5.3. Kernel Soft Margin

However, this approach alone does not resolve issues when data is not linearly separable. To address this, we can map each data point x to a higher-dimensional space using a function $z = \Phi(x)$, which makes the data more linear in this transformed space. This is equivalent of replacing x with z in the optimisation problem:

$$\max_{\alpha} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}(\sum_{n=1}^{N}\sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m z_n^T z_m), where \sum_{n=1}^{N} \alpha_n y_n = 0 \& 0 \leq \alpha_n \leq C$$

This approach, however, makes solving the optimization problem computationally intensive. The kernel trick offers a solution by replacing the dot product in the higher-dimensional space with a kernel function, which achieves the same effect more efficiently [12].

Table 2
This table shows the common kernel used for Kernel Soft Margin SVMs

kernel	$K(x, x') =$
Linear	$x^T x'$
Polynomial	$(1 + x^T x')^Q$
Radial Basis Function	$e^{-\gamma \|\|x - x'\|\|^2}$

2.6. Gradient Methods

Gradient methods are a class of optimisation algorithms that is used to minimize the loss function by iteratively adjusting model parameters in the direction of the negative gradient [13]. In this project, I explored multiple different gradient methods in the for both linear regression and NN.

2.6.1. Gradient Descent (GD)

GD is the simplest gradient method. It is a first-order iterative optimisation algorithm where each step updates parameter in the direction of the steepest descent. The update rule is simply:

$$\theta_{i+1} = \theta_i - \eta \cdot \nabla C(\theta_i)$$

where θ is the parameter you want to optimise, η is the learning rate.

2.6.2. Stochastic Gradient Descent (SGD)

An easy and effective improvement is to approximate the gradient using a subset of the training data, known as a minibatch, rather than computing the gradient over the entire dataset.

In SGD, instead of using a fixed learning rate for every iteration, we add a time decay rate to the learning rate,

$$\gamma(t;,t_0,t_1) = \frac{t_0}{t+t_1}$$

where t_0 and t_1 are new parameter to tune, and t is the current epoch. We can then define the update rule for SGD as:

$$\theta_{t+1} = \theta_t - \gamma(t) \cdot \nabla C(\theta_t)$$

where t is the current epoch.

This adds stochasticity which decreases the chance of the algorithm getting stuck in a local minima. In this project, SGD will refer to stochastic gradient descent with mini-batches.

2.6.3. Adaptive Gradient Descent (AdaGrad)

The goal of AdaGrad is to minimize the expected value of a stochastic objective function concerning a set of parameters, based on a sequence of function realizations. Similar to other methods that rely on sub-gradients, AdaGrad updates the parameters in the opposite direction of the sub-gradients. However, unlike standard sub-gradient methods that apply fixed step sizes without considering past observations, AdaGrad customizes the learning rate for each parameter individually by utilizing the sequence of gradient estimates [14]. In AdaGrad, we introduce G_t, the outer product of all previous subgradients in addition to the learning rate, given by:

$$G_{t+1} = G_t + \nabla C(\theta_t)^2$$

We can then define the update rule of AdaGrad as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t}} \cdot \nabla C(\theta_t)$$

2.6.4. Root Mean Squared Propagation (RMSProp)

RMSprop is an adaptive learning rate algorithms. Adagrad adjusts the learning rate by scaling each gradient element-wise based on a running sum of squared gradients for each parameter, helping it adapt to directions with different gradient magnitudes. When dealing with high condition numbers, this scaling helps accelerate movement in directions with small gradients and slows down in directions with large gradients by dividing by a larger value.

As training progresses, Adagrad's steps continuously shrink due to the growing accumulation of squared gradients, making it suitable for convex optimization where slowing down near a minimum is beneficial. However, in non-convex optimization, this can lead to issues like getting stuck at saddle points. RMSprop addresses this by maintaining an estimate of squared gradients but using a moving average instead of an accumulating sum, allowing for more consistent updates throughout training [15]. For RMSProp, we define G_t as:

$$G_{t+1} = \rho \cdot G_t + (1-\rho)(\nabla C(\theta_t))^2$$

Here, we see that ρ is a new tunable parameter. We can then define the update rule of RMSProp as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t}} \cdot \nabla C(\theta_t)$$

2.6.5. Adaptive Moment Estimation (Adam)

Adam is a first-order optimization method specifically designed for stochastic objectives in high-dimensional parameter spaces, where higher-order methods are impractical. Adam calculates adaptive learning rates for each parameter using estimates of the gradients' first and second moments, drawing from AdaGrad's strength in handling sparse gradients and RMSProp's suitability for non-stationary and online settings. Adam's advantages include gradient-rescaling invariance, approximate control over step sizes, compatibility with non-stationary objectives, support for sparse gradients, and built-in step size annealing [16]. For Adam, we will have to define the first moment vector, m_t, and the second moment vector, v_t,

$$m_{t+1} = \beta_1 \cdot m_t + (1 - \beta_1) \cdot \nabla C(\theta_t)$$

$$v_{t+1} = \beta_2 \cdot v_t + (1 - \beta_2) \cdot \nabla C(\theta_t)$$

where β_1 and β_2 are new tunable parameters, with a recommended default value to 0.9 and 0.999 respectively. We will also need to compute the bias-corrected first moment estimate, $\hat{m}_t$, and the bias-corrected second raw moment estimate, $\hat{v}_t$,

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

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

With this, we can then define the Adam update rule as:

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

where ϵ is a very small value to prevent zero-division error.

2.6.6. Momentum

The problem with gradient descent is that the weight update at moment t is governed by the learning rate and gradient at that moment only. It does not take into account the past steps taken while traversing the cost space. By applying momentum to gradient descent methods, we can allow the algorithm to prevent itself getting stuck in a saddle point [17]. We can define the velocity, v_t, as:

$$v_t = \gamma \cdot v_{t-1} + updaterule$$

where γ is the momentum parameter. With this, we can define the update rule as:

$$\theta_{t+1} = \theta_t - v_t$$

We note that momentum can be added to any of the gradient methods above.

3. Method

3.1. The CIFAR-10 dataset

For this project, I will be using the CIFAR-10 dataset across all models for multi-class image classification task. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images [18].

Figure 6: Sample images of each class of CIFAR-10 dataset

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

For this project, due to limited computing resources on my personal device, I will only be using the first **10000** images from the training set for training of all machine learning models. I will still use the whole testing set for evaluation of the trained models. This also means that the performance of the models in this project will be lower than other benchmarks due to the lower number of training images.

3.2. Normalisation

In CNN-based image classification, normalizing images is essential for improving both model performance and training stability. By standardizing pixel values, typically scaling them to a 0-1 range or adjusting for zero mean and unit variance, normalization facilitates faster training and stabilizes gradient flow, which is especially important in deep networks to prevent issues like vanishing or exploding gradients. Furthermore, normalization helps the model generalize better by reducing its sensitivity to lighting and contrast variations in the data. This preprocessing step enables the CNN to focus on identifying relevant patterns in images, ultimately leading to more accurate and robust classification results. Moreover, normalisation can also improve the effectiveness of activation functions like the rectified linear unit [19].

For this dataset, each pixel is a value between 0 and 255, hence I simply scaled it to between 0 and 1 by dividing each pixel value by 255.

3.3. Data Augmentation

Data augmentation is a technique used to artificially expand the size of a training dataset by applying label-preserving transformations to the original data. This approach helps address the challenge of limited training data, which is often costly and time-consuming to collect. In the context of Convolutional Neural Networks (CNNs), data augmentation is widely used to improve model performance by generating new, diverse samples from the existing data. Common augmentation methods include geometric transformations (such as cropping, flipping, and rotation) and photometric transformations (such as adjusting brightness or contrast). Studies have shown that applying data augmentation, such as cropping, can significantly enhance the performance of CNNs, especially when evaluated on tasks like image classification [20].

In this project, I will be mainly applying 2 types of augmentation: flipping and rotating. Effectiveness of this technique will be discussed in the results section.

Figure 7: Example of applying augmentation on the first sample in the dataset

3.4. Evaluation Metrics

3.4.1. Accuracy Score

The accuracy score is a straight forward metric that evaluate the model's performance on classification tasks, defined as,

$$Accuracy = \frac{NumberofCorrectPredictions}{TotalNumberofPredictions}$$

3.4.2. Macro-Average Precision and Recall

Precision and recall are another common metric used in classification tasks. Precision measures the proportion of correctly predicted positive observations out of all predictions made for a particular class, while recall measures the proportion of actual positives that are correctly identified by the model.

However, precision and recall cannot be directly since they are meant for binary classification tasks. In order to use these metrics for multi-class classification, we have to use their variants, such as macro-average, micro-average, or weighted precision and recall which have various pros and cons [21].

In this project, we will be using the macro-average precision, given by,

$$Precision_i = \frac{TP_i}{TP_i + FP_i}, MacroAveragePrecision = \frac{1}{N}\sum_{i=1}^{N} Precision_i$$

where TP_i is the number of true positives and FP_i is the number of false positives for class i, and N is the total number of classes. While macro-average recall, given by,

$$Recall_i = \frac{TP_i}{TP_i + FN_i}, MacroAveragePrecision = \frac{1}{N}\sum_{i=1}^{N} Recall_i$$

4. Implementation

To improve reusability of code, all own code are implemented as Object Oriented Programming (OOP) classes. For this project, I implemented my own CNN and SVM. All code can be found in the Github repository for this project.

4.1. Implementation of Optimisers

Inspired by the lecture notes, I enhanced my implementation of the optimizers to be more modular, enabling them to be applied to any machine learning model that uses gradient-based methods. This marks an improvement over the approach in Project 2, where each optimizer was tailored to a specific model. The optimizers are now designed to be flexible and reusable across models, streamlining the process of experimenting with different optimization techniques. The code for these optimizers is located in the 'optimisers.py' file.

The following optimizers are implemented for this project:

- BasicOptimiser
- AdagradOptimiser
- RMSPropOptimiser
- AdamOptimiser
- Momentum variant for all the above optimisers

4.2. Implementation of Logistic Regression

See Project 2 for my own implementation of Logistic Regression. In this project, I will be mainly using Sklearn's LogisticRegression model for evaluation.

4.3. Implementation of Convolutional Neural Network (CNN)

The CNN implementation for this project is primarily based on the lecture notes, with several enhancements to improve code quality and align with best practices in Python and TensorFlow. Key improvements include adopting uppercase names for constants, modifying class parameters to use tuples for stride and kernel sizes (in line with TensorFlow conventions), and replacing individual layer-adding methods with a unified method that supports adding any layer type to the model. These adjustments contribute to more readable, maintainable, and flexible code.

```python
from utils import CostCrossEntropy, softmax
from cnn import Conv2DLayer, FlattenLayer, FullyConnectedLayer, OutputLayer, CNN,
    Conv2DLayerOPT
from optimisers import AdamOptimiser

optimiser = AdamOptimiser(learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8)
cnn = CNN(
    cost_func=CostCrossEntropy,
    optimiser=optimiser,
    seed=42,
)
cnn.add(Conv2DLayerOPT(
    input_channels=3,
    feature_maps=1,
    kernel_size=(3, 3),
    stride=(1, 1),
    reset_weights_independently=False
))
cnn.add(FlattenLayer())
cnn.add(FullyConnectedLayer(32))
cnn.add(FullyConnectedLayer(10))
cnn.add(OutputLayer(10, output_func=softmax))
```

Listing 1: Example of setting up CNN model with own code

As you can see, this is very similar to TensorFlow's convention. The code can be found in 'cnn.py'.

4.4. Implementation of Support Vector Machines (SVMs)

I developed an SVM implementation with support for kernelized soft-margin classification, offering a choice of linear, polynomial, or RBF kernels. While I utilized the CVXOPT library [22]to handle the dual optimization problem, the core structure and kernel functionality are my own. This implementation draws on the approach by Essam Wisam [12] for guidance and inspiration.

```python
from svm import SVM

# create the model
svm = SVM(kernel='rbf')

# fit the model
svm.fit(X, y)

# predict
y_pred = svm.predict(X)
```

Listing 2: Example of setting up SVM with own code

It is simple to use and the code can be found in 'svm.py'.

4.5. Benchmarking

In this section, I benchmark the runtime of my custom implementation against external libraries, including TensorFlow's implementation. Finally, I provide a comprehensive comparison of all models, incorporating both custom and library-based implementations. Also, see Project 2 for comparison of runtime for own implementation of Logistic Regression and FFNN.

4.5.1. CNN

For CNN, I will be comparing the runtime performance of my own implementation with optimised convolution layer, and non-optimized convolution layer, against TensorFlow implementation. I ran all tests for 3 runs, 4 loops each. The architecture I used for testing can be seen in the following Table 3.

Table 3
This table shows the architecture of CNN used for testing the runtime performance.

Layer	Kernel	Stride	Output Shape	Activation
Conv2D	(3,3)	(1,1)	(None, 30, 30, 32)	ReLU
Flatten	-	-	(None, 28800)	-
Dense	-	-	(None, 32)	LReLU
Dense	-	-	(None, 10)	LReLU
Dense	-	-	(None, 10)	LReLU

Optimiser: Adam

Table 4
This table shows the run time to fit the CNN model with different implementations. All tests are ran with 100 samples at 10 epochs.

Implementation	runtime
Own CNN (unoptimised)	32.7 s ± 609 ms
Own CNN (optimised)	780 ms ± 19.1 ms
Tensorflow	478 ms ± 82.5 ms

As shown, there are significant performance differences among the implementations, particularly with the unoptimized version. The unoptimized convolution layer follows the "by-the-book" approach discussed in the Theory section above, but its low efficiency makes it impractical for real-world use. Therefore, for further tests, I will only use the optimized version.

The test results show that my implementation performs significantly worse than TensorFlow's, which is expected given that I did not incorporate advanced optimization techniques such as separable kernels or the Fast Fourier Transform (FFT) method. This comparison highlights the importance of ongoing research in developing more efficient techniques for performing convolution operations.

Table 5
This table shows the run time to fit the CNN model at different epochs for different implementations. All test is run with 100 samples.

Epochs ↓	Implementation→	Own	Tensorflow
10		738 ms ± 11.9 ms	413 ms ± 45.1 ms
50		3.29 s ± 12.5 ms	1.65 s ± 9.54 ms
100		6.54 s ± 54.4 ms	3.27 s ± 36.4 ms
200		13.2 s ± 64.3 ms	6.63 s ± 7.31 ms

Table 6
This table shows the run time to fit the CNN model at different number of samples for different implementations. All test is run for 10 epochs.

No. samples ↓	Implementation→	Own	Tensorflow
100		703 ms ± 17.2 ms	435 ms ± 91.5 ms
1000		6.75 s ± 12 ms	2.91 s ± 6.54 ms
10000		1min 23s ± 123 ms	29 s ± 417 ms

4.5.2. SVMs

As with the CNN tests, I will evaluate the runtime performance of my SVM implementation against Scikit-Learn's version. Each test will be conducted over 3 runs with 4 loops each. Since SVMs can be solved analytically, there's no need to compare time evolution across different numbers of epochs.

Table 7
This table shows the run time to fit the SVM model at different number of samples for different implementations. All models uses the rbf kernel.

No. samples ↓	Implementation→	Own	Scikit
100		412 ms ± 10.1 ms	16.2 ms ± 1.32 ms
1000		23.6 s ± 39.8 ms	1.28 s ± 20.7 ms
10000		2min 27s ± 8.02 s	5.37 s ± 80.5 ms

The results clearly show that my SVM implementation struggles to scale with the sample size, requiring 2.5 minutes to train on 10,000 samples, compared to just 5 seconds with Scikit-Learn's implementation. This highlights significant room for optimization in my implementation.

4.5.3. All Models

I also compared the run time of all machine learning models used in this project. 10 epochs are used for models that uses gradient method.

Table 8
This table shows the run time to fit various models. All tests are run with 100 samples for 10 epochs(models using gradient method)

Implementation	runtime
Own CNN (unoptimised)	32.7 s ± 609 ms
Own CNN (optimised)	780 ms ± 19.1 ms
Tensorflow CNN	478 ms ± 82.5 ms
Own SVM	32.7 s ± 609 ms
Tensorflow SVM	780 ms ± 19.1 ms
Tensorflow FFNN	283 ms ± 50.8 ms
Scikit Logistic Regression	1.17 s ± 65.5 ms
Tensorflow RNN	17.9 s ± 603 ms
Tensorflow CNN+RNN	3.12 s ± 331 ms

5. Results & Discussion

5.1. Overview of Algorithms and Their Applications

In this project, I implemented several machine learning algorithms for multiclass image classification tasks, allowing for a thorough evaluation of model performance and efficiency. The primary algorithms tested include Feed-Forward Neural Networks (FFNN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Support Vector Machines (SVM), and Logistic Regression.

Feed-Forward Neural Network (FFNN): The FFNN was applied to the multiclass image classification task, using optimizers such as Adam. While capable of learning complex non-linear patterns, FFNNs were computationally demanding, especially when compared to traditional methods.

Convolutional Neural Network (CNN): CNNs were utilized for their superior performance in image classification tasks. Their architecture, designed to capture spatial hierarchies, proved to be highly effective at extracting features from images, though it came with significant computational overhead.

Recurrent Neural Network (RNN): The RNN model was tested for its capability to handle sequential dependencies within image data by using time-distributed layers across image segments. Although RNNs are typically used for sequence data, their addition allowed for improved handling of dependencies within the CNN+RNN hybrid model, capturing temporal or spatial correlations. However, RNNs alone were limited in capturing detailed spatial hierarchies and required significant memory, posing challenges in scalability.

Support Vector Machines (SVM): SVMs were used for multiclass classification with both linear and non-linear kernels. The key issue with SVMs is that their training time and computational cost increased as the dataset size grew, limiting scalability in larger image classification tasks.

Logistic Regression: Logistic Regression was employed as a baseline for multiclass image classification. Although simpler and more interpretable, it provided competitive results for smaller datasets but was less efficient compared to more complex models like CNNs and FFNNs.

Each algorithm presented unique strengths and limitations, contributing to a broader understanding of model performance across varying levels of complexity and computational requirements.

5.2. Comparison of Performance Metrics

In this section, we compare the models' performance based on key metrics relevant to multiclass image classification task. These metrics provide a quantitative view of each algorithm's strengths and limitations, particularly in accuracy, precision, recall.

5.2.1. Logistic Regression

Hyperparameter Search

To ensure a fair comparison, I conducted a hyperparameter search on a subset of the training data to identify the optimal parameters for the final model. This process determined the best values for "C" as 0.032 and "penalty" as "l2, which were then used in the final evaluation to achieve the highest model performance.

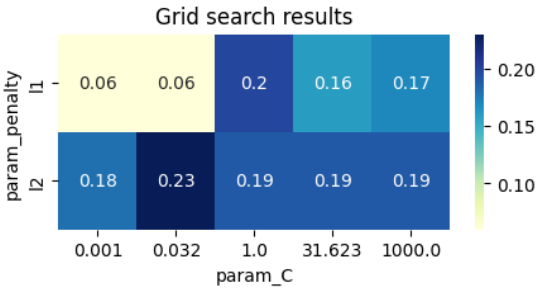

Figure 8: Hyperparameter search for logistic regression model

Table 9
This table shows the metric scores for Logistic regression. The test is done with the 10000 samples and an addition of 10000 augmented samples.

	Best train accuracy	Best test accuracy	Best test precision	Best test recall
Logistic regression	0.449	0.385	0.377	0.385

As expected, from Table 9, we see that the classification performance of logistic regression is lower than that of the other models, serving effectively as a baseline for comparison.

5.2.2. SVMs

Hyperparameter Search

To ensure a fair comparison, I conducted a hyperparameter search on a subset of the training data to identify the optimal parameters for the final model. This process determined the best values for "C" as 10 and "gamma" as 0.001, which were then used in the final evaluation to achieve the highest model performance.

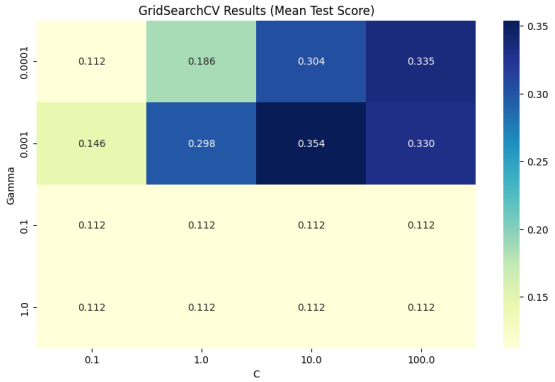

Figure 9: Result of hyperparameter search on SVM

Table 10
This table shows the metric scores for SVM. The test is done with the 10000 samples and an addition of 10000 augmented samples.

	Best train accuracy	Best test accuracy	Best test precision	Best test recall
SVM	0.654	0.4673	0.463	0.4673

Interestingly, the SVM achieved a comparable classification accuracy to that of a basic FFNN, highlighting the robustness and effectiveness of SVMs in handling complex classification tasks.

5.2.3. FFNN

The FFNN architecture used in testing is relatively straightforward, as detailed in Table 11. Unlike the CNN model, the FFNN lacks convolutional layers, which serve as feature extractors, limiting its ability to automatically capture spatial patterns in image data.

The results show that the FFNN's overall performance is noticeably lower than that of the CNN, as expected, since it lacks the specialized feature extraction layers that effectively capture spatial information in images. Nevertheless, the FFNN still achieves a reasonable test accuracy of 45%, demonstrating its capability to handle the task, albeit less effectively than the CNN.

Table 11
This table shows the architecture of CNN used for testing classification performance on CIFAR-10 dataset.

Layer	Kernel	Stride	Output Shape	Activation
Flatten	-	-	(None, 28800)	-
Dense	-	-	(None, 256)	ReLU
Dense	-	-	(None, 128)	ReLU
Dense	-	-	(None, 10)	Softmax

Optimiser: Adam

Table 12
This table shows the metric scores for FFNN. The test is done with the 10000 samples and an addition of 10000 augmented samples.

	Best train accuracy	Best test accuracy	Best test precision	Best test recall
FFNN	0.466	0.442	0.688	0.256

5.2.4. CNN

The CNN architecture I employed for testing is a VGG-like [23] architecture as described in Table 13.

Table 13
This table shows the architecture of CNN used for testing classification performance on CIFAR-10 dataset.

Layer	Kernel	Stride	Output Shape	Activation
Conv2D	(3,3)	(1,1)	(None, 30, 30, 32)	ReLU
Conv2D	(3,3)	(1,1)	(None, 28, 28, 32)	ReLU
MaxPooling2D	(2,2)	-	(None, 14, 14, 32)	-
Conv2D	(3,3)	(1,1)	(None, 12, 12, 64)	ReLU
Conv2D	(3,3)	(1,1)	(None, 10, 10, 64)	ReLU
MaxPooling2D	(2,2)	-	(None, 5, 5, 64)	-
Flatten	-	-	(None, 1600)	-
Dense	-	-	(None, 128)	ReLU
Dense	-	-	(None, 10)	Softmax

Optimiser: Adam

Effect of Data Augmentation
Next, we will assess the effectiveness of data augmentation, a technique that expands the dataset using transformed versions of existing data. This approach not only increases the dataset size but also enhances the model's robustness by exposing it to a wider variety of input patterns.

Table 14
This table shows the metric scores for CNN model with and without data augmentation. The test is done with the 10000 samples and an addition of 10000 augmented samples.

Data augmentation	Best train accuracy	Best test accuracy	Best test precision	Best test recall
Yes	0.802	0.640	0.724	0.627
No	0.977	0.617	0.717	0.591

The results indicate that data augmentation effectively reduces overfitting and improves test accuracy by approximately 2.3%. Given these benefits, the augmented dataset will be used in all subsequent tests.

5.2.5. RNN

The RNN architecture I employed for testing is a simple architecture with a single Gated Recurrent Unit (GRU) as described in Table 15.

Table 15
This table shows the architecture of RNN used for testing classification performance on CIFAR-10 dataset.

Layer	Kernel	Stride	Output Shape	Activation
Input	-	-	(None, 1024, 3)	-
GRU	-	-	(None, 128)	ReLU
Dense	-	-	(None, 128)	ReLU
Dense	-	-	(None, 10)	Softmax

Optimiser: Adam

Table 16
This table shows the metric scores for RNN. The test is done with the 10000 samples and an addition of 10000 augmented samples.

	Best train accuracy	Best test accuracy	Best test precision	Best test recall
RNN	0.507	0.495	0.738	0.341

From the results in Table 16, we observe that while the RNN outperformed other models such as logistic regression, SVM, and even FFNN, the improvement was marginal. This aligns with expectations, as RNNs are not inherently designed for image classification tasks. Their sequential processing capabilities are better suited for handling temporal or ordered data, which limits their effectiveness in capturing the spatial features necessary for robust image classification.

5.2.6. CNN+RNN

The combination of CNN and RNN has seen some success in improving the overall performance of image classification tasks [24]. CNNs filter non-essential information from images, retaining key features, while RNNs capture dependencies and continuity in the intermediate CNN layer outputs. These features are then passed to a fully connected network for final classification. To address input sequence length restrictions and prevent gradient issues, the model incorporates wavelet transform (WT) for data preprocessing. The proposed CNN-RNN model is tested on the CIFAR-10 dataset, showing improved classification accuracy over the standard CNN model, though further research is suggested. Hence, in this project, I will also explore the combination of these two deep learning models.

Table 17
This table shows the architecture of CNN+RNN used for testing classification performance on CIFAR-10 dataset.

Layer	Kernel	Stride	Output Shape	Activation
Conv2D	(3,3)	(1,1)	(None, 30, 30, 32)	ReLU
Conv2D	(3,3)	(1,1)	(None, 28, 28, 32)	ReLU
MaxPooling2D	(2,2)	-	(None, 14, 14, 32)	-
Dropout(0.2)	-	-	(None, 14, 14, 32)	-
Conv2D	(3,3)	(1,1)	(None, 12, 12, 64)	ReLU
Conv2D	(3,3)	(1,1)	(None, 10, 10, 64)	ReLU
MaxPooling2D	(2,2)	-	(None, 5, 5, 64)	-
Dropout(0.2)	-	-	(None, 5, 5, 64)	-
Reshape	-	-	(None, 25, 64)	-
GRU	-	-	(None, 128)	-
Dense	-	-	(None, 128)	ReLU
Dense	-	-	(None, 10)	Softmax

Optimiser: Adam

Table 18
This table shows the metric scores for CNN+RNN. The test is done with the 10000 samples and an addition of 10000 augmented samples.

	Best train accuracy	Best test accuracy	Best test precision	Best test recall
CNN+RNN	0.903	0.669	0.737	0.645

The results of the CNN+RNN architecture demonstrate its effectiveness, achieving higher test accuracy compared to a standalone CNN, though it does show some tendency toward overfitting. This suggests that dependencies exist between individual pixels or data points, which the RNN component effectively captures to enhance classification accuracy.

5.2.7. Overall Comparison

With all the results gathered, we can now provide a comprehensive comparison of model performance across key metrics, including accuracy, precision, and recall. Additionally, a ROC AUC curve will be plotted to visualize and compare the classification performance of each model.

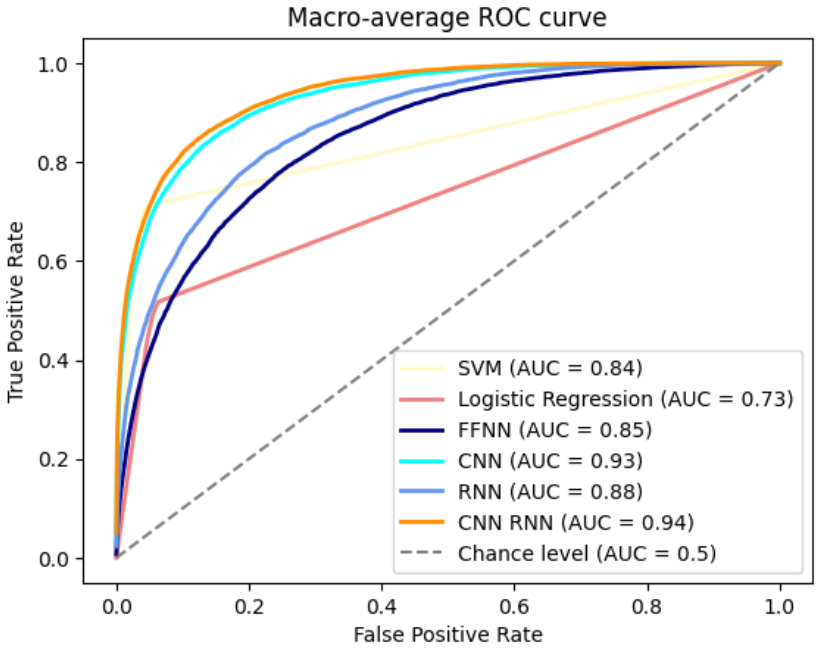

Figure 10: Macro-average ROC curve comparison for every model

Table 19
This table shows the metric scores for all models deployed in this project. The test is done with the 10000 samples and an addition of 10000 augmented samples. Also, all models train only up to 20 epochs due to limited computing resources on my device.

Models	Best train accuracy	Best test accuracy	Best test precision	Best test recall
Logistic regression	0.449	0.385	0.377	0.385
SVM	0.654	0.4673	0.463	0.4673
FFNN	0.466	0.442	0.688	0.256
RNN	0.507	0.495	0.738	0.341
CNN	0.802	0.640	0.724	0.627
CNN+RNN	**0.903**	**0.669**	0.737	0.645

The table reveals clear distinctions in the performance of different models, reflecting the impact of model complexity and architecture on accuracy, precision, and recall. Logistic Regression and SVM have the lowest test accuracy and precision, suggesting their limitations in

handling complex image data. The FFNN and RNN models show improved precision but still fall short in terms of overall accuracy, indicating that, while they can capture some nonlinear relationships, they lack the feature extraction capability needed for high-dimensional image data.

The CNN, however, significantly outperforms these simpler models, achieving 63.7% test accuracy, leveraging its convolutional layers to capture spatial dependencies within the image data effectively. The CNN+RNN model further improves on this by achieving the highest scores across metrics, particularly excelling in train and test accuracy (90.3% and 66.9%, respectively). This suggests that adding an RNN layer enhances the model's ability to capture sequential dependencies, providing a more holistic understanding of the image structure. While this combination increases the risk of overfitting, it also demonstrates the strength of combining CNNs for spatial feature extraction with RNNs for sequential learning, showing that these dependencies are useful for image classification.

5.3. Loading of Models

For anyone interested in using the trained models from this project, the model files are available in the GitHub repository, each saved with the .keras extension. The included model files are: *vggcnn_model.keras, rnn_model.keras, cnn_rnn_model.keras.* These files can be easily loaded and deployed for further testing or integration into other projects.

```
# load the model
model = tf.keras.models.load_model('cnn_rnn_model.keras')
```

Listing 3: Example of loading trained Tensorflow models from .keras file. See Tensorflow documentation for more information about how to use the model.

5.4. Pros and Cons of Various Models

This section discusses the advantages and limitations of the algorithms applied in this study: Logistic Regression, Support Vector Machines (SVMs), Feed-Forward Neural Networks (FFNN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and a hybrid CNN-RNN model. Each model offers unique strengths and weaknesses that influence its suitability for image classification tasks, especially those requiring a balance between accuracy, efficiency, and interpretability.

5.4.1. Logistic Regression

Pros

Simplicity and Interpretability: Logistic regression is straightforward to implement and interpret, making it useful as a baseline for classification tasks. It provides insight into the contribution of each feature, which can be helpful for simpler datasets or problems requiring transparency in decision-making.

Computational Efficiency: Logistic regression is relatively fast to train and does not demand significant computational resources, making it suitable for smaller datasets or as an initial model.

Cons
Limited to Linear Boundaries: Logistic regression assumes a linear decision boundary, which limits its effectiveness in more complex image classification tasks where decision boundaries are often highly non-linear.
Sensitive to Feature Scaling: The performance of logistic regression can be impacted by discrepancies in feature scales, making preprocessing crucial to achieving reliable results.

5.4.2. Support Vector Machines (SVMs)

Pros
Effective in High-Dimensional Spaces: SVMs perform well with high-dimensional feature spaces, offering strong classification performance even with fewer samples per class. This makes SVMs a good choice for image classification with rich feature sets.
Robust to Overfitting: The use of regularization (via the "C" parameter) in SVMs helps prevent overfitting, especially on complex datasets.

Cons
High Computational Cost with Large Datasets: SVMs can be computationally intensive, especially for larger datasets, as training time scales with the number of samples.
Limited Scalability: SVMs are less efficient on datasets with a large number of samples and may struggle with complex image data where more flexible models (e.g., deep learning) might perform better.

5.4.3. Feed-Forward Neural Network (FFNN)

Pros
Flexible for Non-linear Patterns: FFNNs can model non-linear relationships between inputs and outputs, making them suitable for image classification tasks that require more complexity than traditional algorithms can provide.
Customizable Architectures: FFNNs allow for various customizations in terms of layers, neurons, and activation functions, enabling the model to be tailored to specific tasks or constraints.

Cons
Limited Feature Extraction: Unlike CNNs, FFNNs do not have a feature extraction component, which limits their ability to capture spatial information in image data. This restricts their effectiveness in image classification.
High Computational Cost: FFNNs, especially deep ones, can require significant computational resources and time to train, even for simpler image classification tasks.

5.4.4. Convolutional Neural Network (CNN)

Pros
Effective Feature Extraction: CNNs are specifically designed to capture spatial and hierarchical features in image data through convolutional layers, making them highly effective for image

classification tasks.
Strong Performance in Image Classification: With the ability to learn from patterns within images, CNNs typically outperform simpler models like logistic regression in terms of classification accuracy.

Cons
Prone to Overfitting: CNNs may overfit on small datasets if not carefully regularized or augmented.
Higher Computational Demands: CNNs are computationally intensive due to the large number of parameters and layers, often requiring specialized hardware such as GPUs for efficient training.

5.4.5. Recurrent Neural Network (RNN)

Pros
Ability to Capture Sequential Dependencies: RNNs are well-suited to capture temporal or sequential dependencies within data, which can help in image classification by integrating dependencies across image pixels.
Useful for Sequence-Based Applications: RNNs can add value to image classification tasks that might benefit from a focus on sequences, such as video frames or structured image patterns.

Cons
Less Effective for Pure Image Tasks: RNNs were not originally designed for image data, and their sequential approach may not be as effective for general image classification.
Susceptible to Vanishing Gradient: RNNs can suffer from the vanishing gradient problem, which can make training difficult on longer sequences or deep architectures.

5.4.6. CNN-RNN Hybrid Model

Pros
Combines Feature Extraction and Sequence Learning: The CNN-RNN model leverages CNN layers for feature extraction and RNN layers for capturing sequential dependencies, making it effective for complex image classification tasks where spatial and temporal features are important.
Improved Classification Accuracy: By combining CNN and RNN capabilities, the hybrid model can often achieve higher accuracy than standalone CNNs or RNNs.

Cons
Increased Complexity and Risk of Overfitting: The hybrid model's complexity can lead to overfitting if not properly regularized, especially on smaller datasets.
Higher Computational Requirements: With both CNN and RNN layers, the CNN-RNN model demands significant computational power, making it resource-intensive and challenging to train without specialized hardware.

6. Conclusion

In this project, we implemented and evaluated a variety of machine learning models, ultimately finding that the CNN+RNN hybrid model offered the highest performance for image classification tasks, particularly when combined with regularization techniques and dropout layers to manage overfitting. This architecture demonstrated the ability to capture both spatial and sequential dependencies within the image data, providing a balanced solution with superior classification accuracy.

Each model used in this project has its own set of strengths and limitations. For simpler classification tasks, logistic regression and SVMs served as valuable baselines, offering insights into linear separability within the data. Feed-Forward Neural Networks (FFNN) showed an improvement over traditional methods by capturing non-linear patterns but lacked spatial awareness. Convolutional Neural Networks (CNN) excelled in feature extraction, capturing intricate image patterns, while the RNN component in the CNN+RNN hybrid model added sequential insight, boosting performance further.

Ultimately, choosing the best model requires a strategic balance of model complexity, regularization, and optimization methods to ensure robust performance and effective generalization. Through careful tuning, these methods can be tailored to different data complexities, achieving reliable accuracy and model efficiency for real-world applications. This project highlights the importance of model selection and architecture choice in tackling image classification tasks with varying degrees of complexity.

Acknowledgments

Many references are taken from the course lecture notes https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html

References

[1] N. Bhatt, Logistic regression with l2 regularization from scratch, 2023. URL: https://medium.com/@bneeraj026/logistic-regression-with-l2-regularization-from-scratch-1bbb078f1e88.

[2] S. Raschka, V. Mirjalili, Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2, Packt publishing ltd, 2019.

[3] S. R. Dubey, S. K. Singh, B. B. Chaudhuri, Activation functions in deep learning: A comprehensive survey and benchmark, Neurocomputing 503 (2022) 92–108. URL: https://www.sciencedirect.com/science/article/pii/S0925231222008426. doi:https://doi.org/10.1016/j.neucom.2022.06.111.

[4] P. Belagatti, Understanding the softmax activation function: A comprehensive guide, 2024. URL: https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/.

[5] Niru, Understanding backpropagation with softmax, 2023. URL: https://stackoverflow.com/questions/58461808/understanding-backpropagation-with-softmax.

[6] A. L. Maas, Rectifier nonlinearities improve neural network acoustic models, in: Rectifier Nonlinearities Improve Neural Network Acoustic Models, 2013. URL: https://api.semanticscholar.org/CorpusID:16489696.

[7] S. Raschka, Y. H. Liu, V. Mirjalili, Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python, Packt Publishing Limited, Birmingham, UK, 2022.

[8] N. T. Nam, P. D. Hung, Padding methods in convolutional sequence model: An application in japanese handwriting recognition, in: Proceedings of the 3rd International Conference on Machine Learning and Soft Computing, ICMLSC '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 138–142. URL: https://doi.org/10.1145/3310986.3310998. doi:10.1145/3310986.3310998.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting, Journal of Machine Learning Research 15 (2014) 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[10] C. Stryker, What is a recurrent neural network?, 2024. URL: https://www.ibm.com/topics/recurrent-neural-networks.

[11] R. Gandhi, Support vector machine — introduction to machine learning algorithms, 2018. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

[12] E. Wisam, Easily implement multiclass svm from scratch in python, 2023. URL: https://towardsdatascience.com/implement-multiclass-svm-from-scratch-in-python-b141e43dc084.

[13] L. Jin, S. Li, B. Hu, M. Liu, A survey on projection neural networks and their applications, Applied Soft Computing 76 (2019) 533–544. URL: https://www.sciencedirect.com/science/article/pii/S1568494619300067. doi:https://doi.org/10.1016/j.asoc.2019.01.002.

[14] D. Villarraga, Adagrad, 2021. URL: https://optimization.cbe.cornell.edu/index.php?title=AdaGrad.

[15] V. Bushaev, Understanding rmsprop — faster neural network learning, 2018. URL: https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a.

[16] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2017. URL: https://arxiv.org/abs/1412.6980. arXiv:1412.6980.

[17] R. Bhat, Gradient descent with momentum, 2020. URL: https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f.

[18] A. Krizhevsky, Learning multiple layers of features from tiny images, University of Toronto (2012).

[19] Z. Liao, G. Carneiro, On the importance of normalisation layers in deep learning with piecewise linear activation units, in: 2016 IEEE Winter Conference on Applications of Computer Vision (WACV), 2016, pp. 1–8. doi:10.1109/WACV.2016.7477624.

[20] L. Taylor, G. Nitschke, Improving deep learning with generic data augmentation, in: 2018 IEEE Symposium Series on Computational Intelligence (SSCI), 2018, pp. 1542–1547. doi:10.1109/SSCI.2018.8628742.

[21] Evidently.AI, Accuracy, precision, and recall in multi-class classification, 2024. URL: https:

//www.evidentlyai.com/classification-metrics/multi-class-metrics.
[22] M. S. Andersen, J. Dahl, L. Vandenberghe, CVXOPT: A python package for convex optimization, 2004–2023. URL: https://cvxopt.org, last updated on Aug 09, 2023. Built with Sphinx using a theme provided by Read the Docs.
[23] G. Boesch, Very deep convolutional networks (vgg) essential guide, 2021. URL: https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/.
[24] Yin, Qiwei, Zhang, Ruixun, Shao, XiuLi, Cnn and rnn mixed model for image classification, MATEC Web Conf. 277 (2019) 02001. URL: https://doi.org/10.1051/matecconf/201927702001. doi:10.1051/matecconf/201927702001.

A. Appendix: Various Results

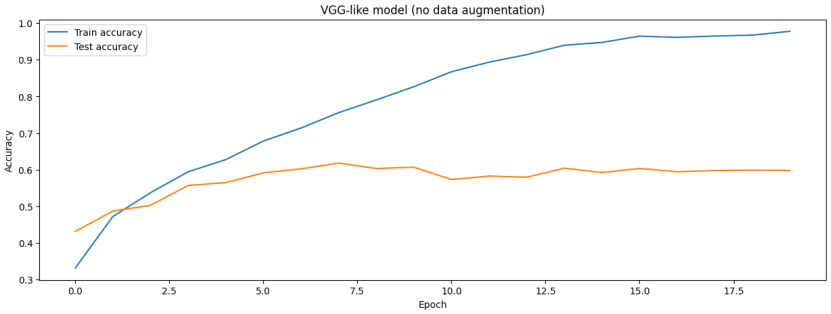

Figure 11: Train Test accuracy of VGG-like model without data augmentation

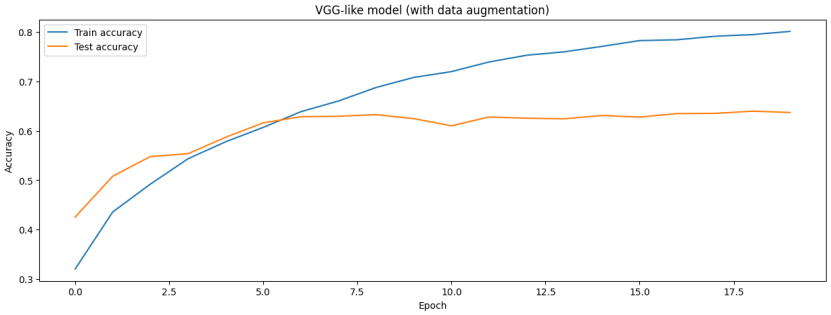

Figure 12: Train Test accuracy of VGG-like model with data augmentation

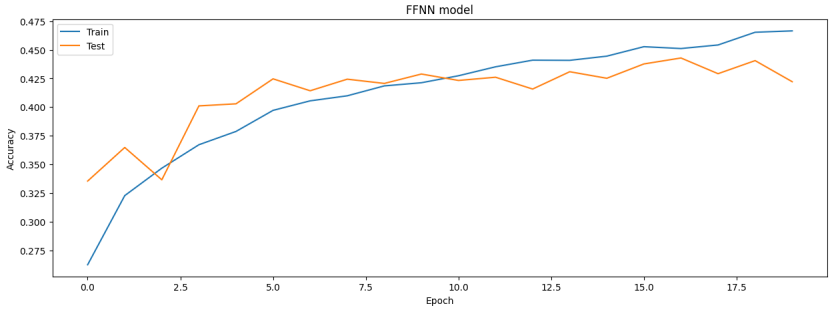

Figure 13: Train test accuracy of simple FFNN model

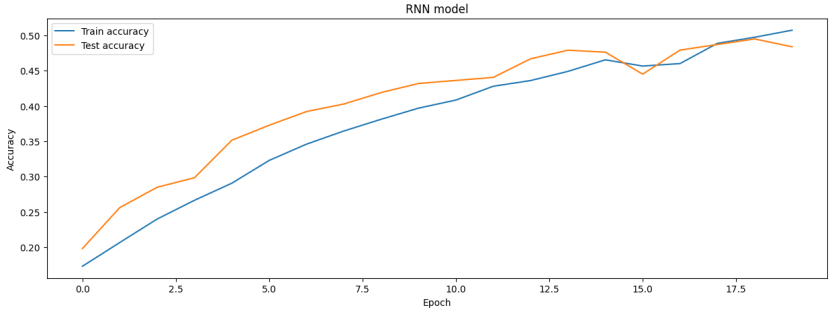

Figure 14: Train test accuracy of simple RNN model

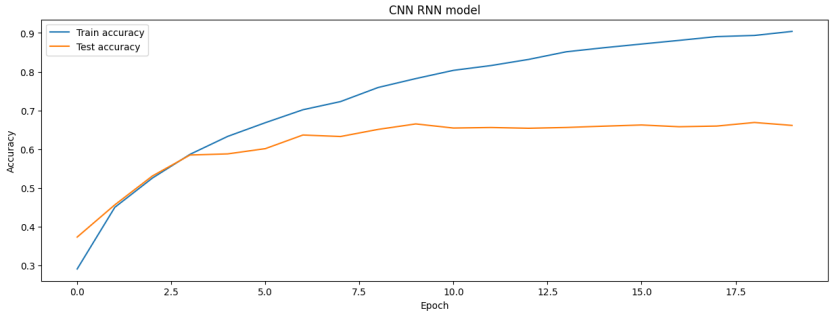

Figure 15: Train test accuracy of CNN+RNN hybrid model