Convolutional Neural Networks (CNNs) for Asset Price Prediction

Amjad Saidam

February 2026

Contents

1 Introduction

We discuss the application of convolutional neural networks (CNNs) for asset price prediction. We begin by discussing the multi-layer perceptron (chapter 2) and other fundamental concepts that allow us to understand the CNN architecture (chapter 3). Chapter 4 follows, covering our research problem and methods. Data and model pipelines are defined in chapters 5. Results and results discussion are presented in chapters 6 and chapters 7, respectively. We conclude with research limitations (chapter 8) and a conclusion on our papers findings in chapter 9.

2 Fully Connected Neural-Networks (FCNs) and the Multi-layer Perceptron (MLP)

A neural network is a susperivsed learning algorithm that be used to solve regression or classification tasks. We use $f_\theta(X)$ to denote a family of neural networks.

2.1 Feedforward Pass

Given an independent and identically distributed, *iid*, feature vector, $X^{(i)} \in \mathbb{R}^d$, we can define a linear layer as

$$z_l = W_l^\top A_{l-1} + b_l, \, ^\forall l \geq 1 \tag{1}$$

Where, $W \in \mathbb{R}^{n_{l-1} \times n_l}$ is our weight matrix, $A_l \in \mathbb{R}^{n_{l-1}}$ is the prior layer activation equal to $X^{(i)}$ for $l = 1$ and b is our bias. The result of this operation, z_l, is the l layer pre-activation. Every element of this column vector is a pre-activated neuron of layer l.

We can decompose the matrix notation and define $i'th$ pre-activation neuron of layer l.

$$z_i^{(l)} = \sum_{j=1}^{n_{l-1}} W_{ij}^{(l)\top} A_j^{(l-1)} + b_i^{(l)} \tag{2}$$

We apply an activation function to introduce non-linearity in the model.

$$A_l = \sigma_l(z_l) \tag{3}$$

Where A_l is the current layer with n_l neurons. Equations 1 and 3 hint towards the recursive nature with which these models are defined. Because of the recursion, we impose a base case, here that $z_1 = W_1^T X^{(i)} + b_1$.

Obviously, this recursion is finite and ends when $l = L$, At layer L our model returns an output $\hat{y}$.

$$\hat{y}^{(i)} = \sigma_l(W_L^\top A_{L-1} + b_L) \tag{4}$$

The superscript, (i), here denotes this is the model prediction corresponding to the $i'th$ data input. The model computation process from equation 1 through to equation 4 is called a forward pass.

2.2 Loss Function

Model parameters $\theta = \{\boldsymbol{W}, \boldsymbol{b}\}$ (bold font denotes sets, here sets of all weights and biases) are at first randomly initialised, and then "learned" via a model training process called **back-propagation**. Before defining the gradient-based learning process, we must first quantify how good our model's guess, $\hat{y}^{(i)}$ is. To do this we define a **loss-function**.

$$\mathcal{L}(\hat{y}^{(i)}, y); \, \hat{y} = f_\theta(X^{(i)}), \, \mathcal{L} : \mathbb{R}^n \to \mathbb{R} \tag{5}$$

Equation 5 is often not chosen arbitrarily and derived by finding the **maximum likelihood estimator (MLE)** of the data output distribution $p_\theta(\hat{y}^{(i)}|X^{(i)})$. To help see this, we will make an assumption our output data is Gaussian distributed. This can be decomposed as additive noise.

$$y_i = f_\theta(X^{(i)}) + \varepsilon;\ \varepsilon \sim \mathcal{N}(0,1) \tag{6}$$

Equation 6 contains the stochastic part of the output in ε. Using properties of the expectation, we can derive the conditional density.

$$y_i|X^{(i)} \sim \mathcal{N}(f_\theta(X^{(i)}),1) \tag{7}$$

Equation 7 has Guassian probability density function.

$$p(y_i|X^{(i)},\theta) = \mathcal{N}(y_i|f_\theta(X^{(i)}),1) = \frac{1}{\sqrt{2\pi}}\exp\{\frac{1}{2}(\hat{y}_i^{(i)} - y_i)^2\} \tag{8}$$

A good model will have parameters that maximise the likelihood of observing our data under. This is exactly a maximum likelihood estimation (MLE) problem. We can solve this problem by defining the following optimisation problem.

$$\arg\max_\theta \left\{ \log\left(\prod_{i=1}^{n} \mathcal{N}(y_i|f_\theta(X^{(i)}),1) \right) \right\} = \arg\min_\theta \left\{ -\ell(y_i|f_\theta(X^{(i)}),1) \right\} \tag{9}$$

Equation 9 says that maximising the MLE is an equivalent optimisation problem to minimising the negative log likelihood function. Simplifying the objective function we get the **mean-square error (MSE)**.

$$\arg\min_\theta \left\{ \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i^{(i)} - y_i)^2 \right\} = \frac{1}{2n}\arg\min_\theta \left\| f_\theta(X^{(i)}) - Y \right\|_2^2 \tag{10}$$

Equation 10 is the famous mean square error optimisation problem, and is analogous to the ordinary least squares optimisation problem if $f_\theta(X)$ denotes a linear regression model. We have substituted the neural network model for $\hat{y}_i^{(i)}$, this is to emphasise that the model output is a function of parameters θ.

2.3 Backpropagation

Because of the non-linearity introduced in the model, 2.1, no closed-form solution for 10 exists. We therefore must use gradient-based methods to solve the problem. Solving this problem is precisely the "Model-training" step in machine learning.

2.3.1 Gradient Descent

Gradient Descent (GD) is an optimisation method used to find the "optimal model" in the training set. GD works by starting with a base case guess of the optimal parameter and iteratively updating the parameter based on the difference of the current guess and the loss gradient with respect to the parameter evaluated at this guess (current parameter). This process continuous until some convergence is reached (often determined by some update threshold).

In the context of our MLP, we have two parameters wish we wish to optimize, $\theta = \{\boldsymbol{W}, \boldsymbol{b}\}$, To implement standard GD we can run the following algorithm.

$$\theta_{i,n+1}^{(l)} \leftarrow \theta_{i,n}^{(l)} - \eta\, \nabla_{\theta_{i,n}^{(l)}} \mathcal{L}(f_\theta(X^{(i)}), Y) \tag{11}$$

It is important to note that to compute the grad with respect to some parameter for some layer, we must calculate vector-valued derivatives recursively using the chain rule.

$$\frac{\partial \mathcal{L}}{\partial \theta^{(l)}} = \frac{\partial \mathcal{L}}{\partial f_\theta} \frac{\partial f_\theta}{\partial z_L} \frac{\partial z_L}{\partial A_{L-1}} \cdots \frac{\partial z_l}{\partial \theta_l} \tag{12}$$

Evaluating 12 requires computing vector-valued derivatives. To compute these, we note the following useful rules that can be used to derive the value of gradient descent for any layer and parameter.

1 Derivatives of Vector Valued Functions. For vector valued functions $f(y)$ and $y = g(x)$ (functions with codomain > 1). To compute the derivative $f(g(x)) = h(x)$

$$\frac{\partial h}{\partial x}(x) = \frac{\partial f}{\partial y}(g(x)) \frac{\partial g}{\partial x}(x) \tag{13}$$

Importantly if $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^l$ then $h : \mathbb{R}^m \to \mathbb{R}^l$ and $\frac{\partial h}{\partial x} \in \mathbb{R}^{l \times m}$

2. Gradients and Jacobian's The gradient operator is exactly the transpose of the Jacobian, that is $\nabla f(x) = Jf(x)^\top$. This is important as gradient descent uses gradients (although only a notation choice this will change the dimensions of our solutions).

3. No High Dimensional Tensor Computations in Backpropagation. Upstream gradients eliminate the need to compute any high-dimensional matrix multiplications. Specifically the the scaler valued loss function ensures that not rank three tensors or greater are ever computed.

2.3.2 Stochastic Gradient Descent

Standard GD is often not implemented in practical applications but rather used as a theoretical framework to better understand more complex GD variations which attempt to fix its limitations. Stochastic gradient decent (SGD) is one such application.

SGD is a stochastic approximation of the standard GD method and replaces the grad of the loss with respect to the parameter over the entire data set with the batch, $|B_t|$, sum of differentiable loss functions averaged over the batch (i.e. the batch loss mean).

SGD has many advantages over standard GD such as ...

- **Multiple parameter updates per epoch**: Dependent on the batch size, SGD will yield $(N/|B_i|)$ parameter updates per epoch in contrast to standard GD where $|B_i| = N \implies$ 1 parameter update per epoch. Therefore parameter convergence is faster under GD.
- **No exploding gradients**: Due to stochastic uniform sampling for each batch, SGD is unlikely to get stuck in local minima. Therefore leading to better generalisation and model performance.

Mini-Batch SGD, is a SGD with $1 < B_i \ll N$, and is the specific variation of SGD we will be implementing in our backpropagation algorithm.

$$\theta^{(l)}_{i,n+1} \leftarrow \theta^{(l)}_{i,n} - \eta \left(\frac{1}{|B_j|} \right) \sum_{k \in B_j} \nabla_{\theta^{(l)}_{i,n}} \mathcal{L}(f_\theta(X^{(k)}), y_k) \tag{14}$$

Importantly, the expectation of the average mini-batch gradient over the batch sampling distribution is equal to the average loss gradient or the empirical risk gradient.

$$\mathbb{E}[\frac{1}{|B_i|} \sum_j^n \nabla \mathcal{L}(\hat{y_j}, y_j)] = \frac{1}{N} \sum_i^n \nabla \mathcal{L}(\hat{y_i}, y_i) \tag{15}$$

2.3.3 Loss Gradient Vectors

To optimise the model and estimate the solution of the problem 10, we typically calculate the gradients for all layers, for all parameters, and update all parameters for all layers simultaneously. Applying the rules discussed in 2.3.1.

$$\frac{\partial \mathcal{L}}{\partial \theta_{i,l}} = \frac{\partial \mathcal{L}}{\partial z_l}\frac{\partial z_l}{\partial \theta_{i,l}} = \delta_l \frac{\partial z_l}{\partial \theta_{i,l}} \tag{16}$$

$$\frac{\partial \mathcal{L}}{\partial z_l} = \begin{cases} \delta_{l+1} W_{l+1} \odot \sigma'(z_l) = \delta_l & \text{, for } l < L \\ \nabla_{f_\theta} \mathcal{L} \odot \sigma'_L(z_L) = \delta_L & \text{, for } l = L \end{cases} \tag{17}$$

2.4 FCN Architecture

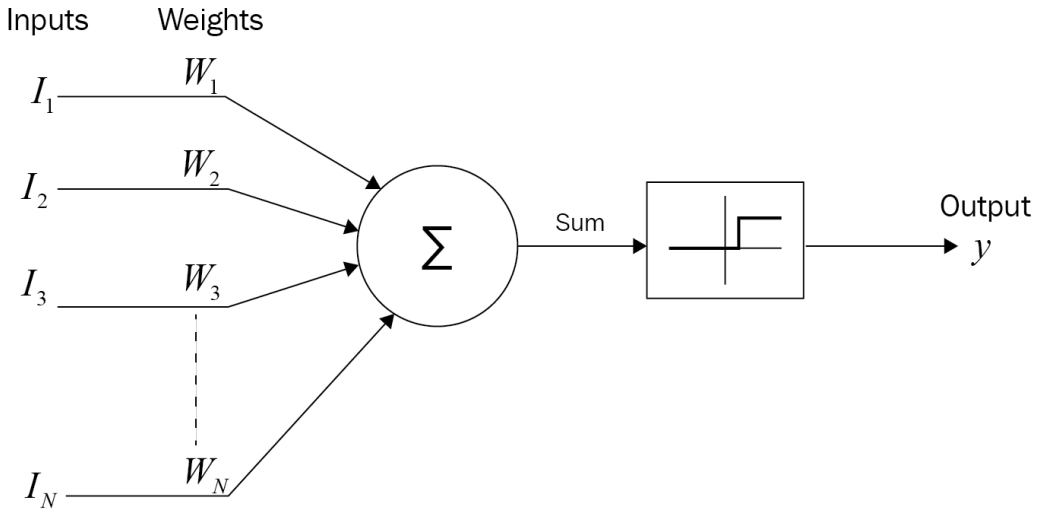

Figure 1: A single-layer fully connected feed-forward neural network. The model contains N inputs and one hidden layer neuron and associated activation [1].

Figure 1 shows a fully connected one layer MLP. The MLP takes in N inputs and applies equation 1 denoted by the summation and and equation 3 (using a ReLU activation function) denoted by the function plot to produce a single output, $\hat{y}^{(i)}$.

2.5 Training Algorithm

Algorithm 1: Fully-Connected MLP Forward-pass

Function $MLPForwardPass(X^{(i)}; \theta)$**:**
  for $l = 1, \dots L$ **do**
    // $l'th$ layer pre-activation and activation
    $z_l = W_l^\top A_{l-1} + b_l$ ▷ $A_{l-1} = X^{(i)}$ for $l = 1$
    $A_l = \sigma_l(z_l)$
    // model output (function of all prior layer parameters
    if $l = L$ **then**
      $z_L = W_L^\top A_{L-1} + b_l$
      $f_\theta(X^{(i)}) = \sigma_L(z_L)$ ▷ $A_L = f_\theta(X^{(i)})$
  end
  return $f_\theta(X^{(i)})$

Algorithm 2: Fully-connected MLP Backward-pass using SGD

Function *MLPBackwardPassSGD*():
- $\mathbf{B} = \{B_1, \dots B_n\}; B_i \subset \mathbf{X}$ ▷ $k = \frac{N}{n}$, n = number of mini-batches
- `// Loop over entire dataset epochs times, for n grad updates per epoch`
- **for** $epoch = 1$ ***to*** $epochs$ **do**
  - **for** $i = 1$ ***to*** n **do**
    - $f_{\theta_{i-1}}(B_i) = \text{MLPForwardPass}(B_i; \theta_{i-1})$ ▷ $i'th$ model
    - `// All gradients taken wrt current model parameters`
    - **for** $l = L$ ***to*** 1 **do**
      - **if** $l = L$ **then**
        - $\delta_L = \nabla_{f_\theta} \mathcal{L} \odot \sigma'_L(z_L)$ ▷ base condition
      - **else**
        - $\delta_l = (W_{l+1}^\top \delta_{l+1}) \odot \sigma'(z_l)$
      - **end**
      - $\nabla_{W_l} \mathcal{L} = \frac{1}{|B_i|} \delta_l A_l^T$
      - $\nabla_{b_l} \mathcal{L} = \frac{1}{|B_i|} \sum_{j \in B_i}^{n=|B_i|} \delta_l^{(j)}$
      - `// Apply SGD update to` $i'th$ `model parameters`
      - $W_i \leftarrow W_{i-1} - \eta \nabla_{W_{i-1}} \mathcal{L}_{B_i}$
      - $b_i \leftarrow b_{i-1} - \eta \nabla_{b_{i-1}} \mathcal{L}_{B_i}$
    - **end**
  - **end**
- **end**
- **return** $f_\theta(X)$ ▷ trained model

3 Convolutional Neural-Networks (CNNs)

A CNN is a family of neural networks that uses filters (kernels) and pooling layers to learn patterns in image data. CNNs are commonly used for image classification tasks.

3.1 Convolution via Kernel Filters

In mathematics, a convolution is an operation on a pair of functions or sets (in the discrete case). In deeplearning, the operation is similar, although we deal with matrices. Specifically, we construct a **feature map** by taking an input (prior image or feature map) and convolving it with a kernel. The operation is defined as follows.

$$(I * K)(i, j) = \sum_{m=0}^{H_k - 1} \sum_{n=0}^{W_k - 1} I(i + m, j + n) \cdot K(m, n) \tag{18}$$

Where $I \in \mathbb{R}^{N \times C_{in} \times H_{in} \times W_{in}}$ is the input, which can be an image or feature map. $K \in \mathbb{R}^{H_k \times W_k}$ is the kernel. N here denotes the batch size and C the number of channels; we will assume both equal 1 and therefore I simplifies to dimension $\dim(H_{in} \times W_{in})$. The convolution operation will output the following dimensional matrix.

$$\dim(H_{out}, W_{out}) = \left[\frac{H_{in} - H_k + 2P}{S} + 1\right] \times \left[\frac{W_{in} - W_k + 2P}{S} + 1\right] \tag{19}$$

Where P is the padding operation [3.1.1] and S is the stride operation [3.1.2]. We can use equation 19 to define the dimension of the convolution operation $(I * K)$. We will further assume that padding and stride are equal to zero (as we do for our model) 4.

$$\dim(H_{out}, W_{out}) = (H_{in} - H_k + 1) \times (W_{in} - W_k + 1) \tag{20}$$

The output of a convolution operation is referred to as a feature map, and we can have many feature maps per convolution layer. This is done by defining many kernels with different weight initialisations and convolving them with the input. Figure 2 shows a simple convolution operation.

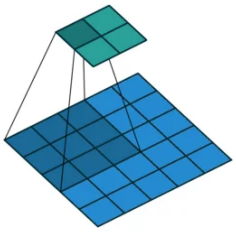

Figure 2: 3×3 filter (dark blue) operation (dark green) applied to 5×5 image (light blue) with 2 pixel stride. Filter operation produces a 2×2 feature map latent representation of image (light green) [2].

Initially, one may think that we can process image data without a convolution operation by flattening the image to 1D vector where every pixel in the image is represented by input neuron in the MLP. This approach suffers from unstable gradients and poorly scalable models. For example imagine a high-resolution image of dimension 100×100, this would require 10000 weights for each neuron in the second layer. Using a 5×5 kernel with shared weights allows us to represent the same image using only 25 parameters [2]. Another major drawback of the MLP choice is that it is unable to learn hierarchical hidden features from the image.

3.1.1 Padding

Padding is defined as the number of pixels we add to the border of the image. Padding provided control on the output size of a convolution operation. This can be used to preserve the input dimension across convolution and max-pooling operations. For odd-valued kernels, $S = 1$ and $H_{in} = H_{out}$ such that our kernel preserves the input dimension, the padding formula is as follows.

$$P = \frac{H_k - 1}{2} \tag{21}$$

Equation 21 is referred to as the zero padding formula and tells us how much we pad our input image to preserve the input dimension.

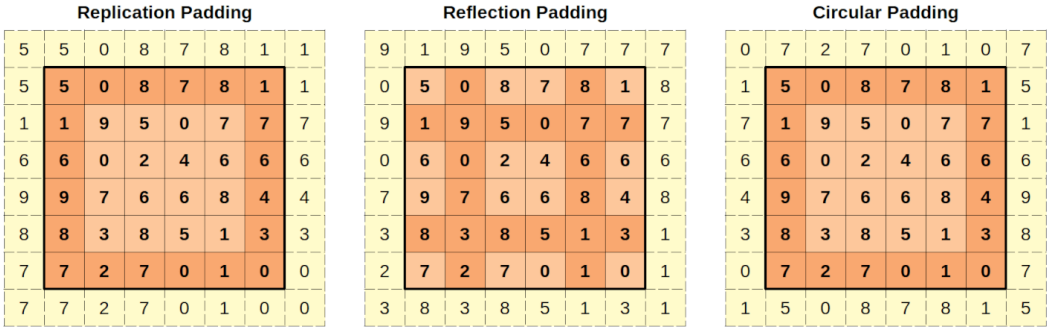

Figure 3: Three different image padding types. Reflection padding (left), where image edges are extended from the border. Reflection padding (middle) Where we pad edges using reflected inner values, and circular Padding where we extend edges using the opposite edge of the image [3].

3.1.2 Stride

This is the number of pixels we offset the kernel by, or simply the sliding window. Increasing the stride reduces the feature map dimension, which also cause their to be less overlap of pixels used to construct the feature map. Reduced overlap also decreases the density (pixel overlap in the original image) of the **receptive field**.

3.1.3 Pooling

Pooling is a downscaling operation used to reduce the input feature map dimensions while maintaining the most useful information.

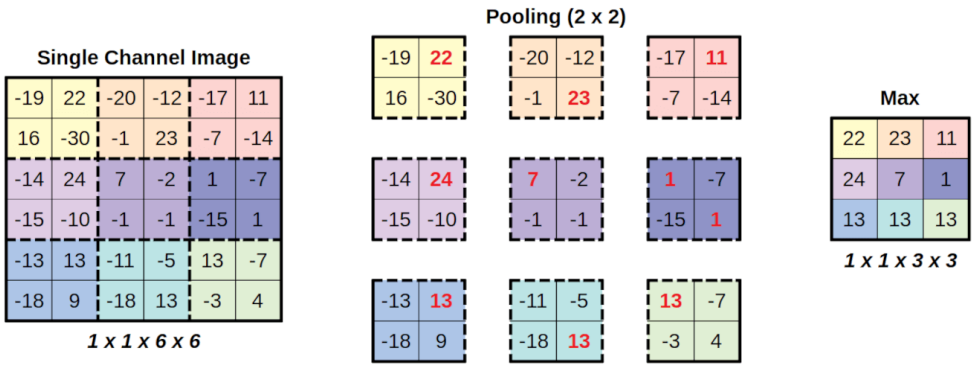

Figure 4: Max pooling applied to 3×3 feature map (convolution layer) [3].

Pooling operations are not parametrised, meaning their operations are not updated during training of the CNN.

3.2 Receptive Field

A receptive field describes the area accessible to the current layer from the image layer. For a fully connected layer e.g., in a MLP, this would contain the entire prior layer, but because of convolution operations, each neuron in the feature map has access to a limited area of the prior. The size of the field grows with the number of convolution operations (scaling of

receptive field depends on dimension of layer filter). Ideally we want the last convolution operation to have a receptive field equal to the entire input image, so that we can detect any global objects in the image.

3.3 CNN Architecture

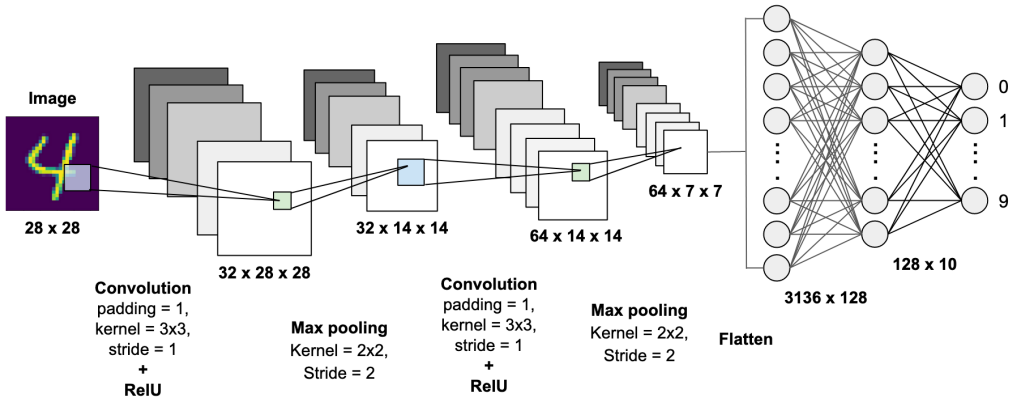

Figure 5: A simple CNN model with two convolutional layers and two pooling layers, for digit classification [4].

Figure 5 shows the architecture of a digit classification CNN with 4 feature extraction layers and two fully connected MLP layers. The model outputs a probability for all 9 classes (numbers). The output neuron with the highest probability is the models classification of the input (which we would expect to be 4).

4 Methods

This chapter discusses our financial time series based research problem, feature engineering techniques, loss function derivation and trading signal.

4.1 Research Problem

Using a CNN we must derive a predictive signal from an asset's historic price data. We begin with the asset's daily adjusted-close, high, and low prices. To construct feature and class labels, we first split our data into batches. The label is the immediately preceding asset price direction after the batch, up or down. We intend our model to draw similarities between latent representations of price patterns, classifying them as one of two directional signals, up or down. We can then bet on model predictions using market orders.

An immediate problem that arises is that our image data (the time series) is not rich enough for a CNN to learn meaningful latent features. Problems include

1. No Hierarchical Structure. We essentially have 1D data indexed by time. This makes learning features such as edges, textures, motifs or objects impossible; therefore, even early convolution layers will struggle to construct meaningful feature maps.

2. Little Diversity. Our time series images are lacking diversity and symmetry, making it hard for the CNN to learn latent features that represent variation.

4.2 Gramian Angular Field (GAF)

The GAF process transforms time-indexed one-dimensional data to a symmetric and time-variant matrix representation. The GAF operation performs a **polar embedding** of the time and price as distance and angle along the unit semi-circle.

We first pre-process our data by calculating the average price for each feature vector and then applying a min-max scaling normalisation to bound our data from -1 to 1.

$$x_i = 2\left(\frac{x - \min(x_i),}{\max(x_i) - \min(x_i)}\right) - 1 \tag{22}$$

So our feature vector is now $x_i \in [-1, 1]$. Using trigonometric identities, we can redefine our feature vector.

$$\cos(\theta_i) = x_i; \cos(\phi_j) = y_j \tag{23}$$

Where y_j defines a sample element from the feature vector and i need not equal j. Given $\theta_i, \phi_j \in [0, \pi]$ we can can define the y components using Pythagoras theorem.

$$\sin(\theta_i) = \sqrt{1 - x_i^2}; \sin(\theta_j) = \sqrt{1 - y_j^2} \tag{24}$$

Given y_j is an element from the shifted x vector, we can calculate the **cosine phase** of the current and shifted price by summing the angles. We can also decompose this polar form in Cartesian coordinates to uncover the penalised dot product.

$$\begin{aligned}
\cos(\theta_i + \phi_j) &= \cos(\cos^{-1}(x_i) + \cos^{-1}(y_j)) \\
&= \cos(\cos^{-1}(x_i))\cos(\cos^{-1}(y_j)) - \sin(\cos^{-1}(x_i))\sin(\cos^{-1}(y_j)) \\
&= x_i \cdot y_j - \sqrt{1 - x_i^2} \cdot \sqrt{1 - y_j^2} \\
&= G_{ij}
\end{aligned} \tag{25}$$

Equation 25 denotes an element of the gram like GAF matrix. We can rewrite this operation using properties of the dot product.

$$\begin{aligned}
G_{:j} &= \cos(\theta + \phi_k) \\
&= y_j \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} - \sqrt{1 - y_j^2} \cdot \begin{pmatrix} \sqrt{1 - x_1^2} \\ \vdots \\ \sqrt{1 - x_n^2} \end{pmatrix}
\end{aligned} \tag{26}$$

Equation 26 denotes a column of the GAF matrix. Using Equations 25 or 26 we can define the entire GAF matrix. Importantky no unique feature map that represents equation 25 exists that satisfies the axioms of the inner product. However it is possible to represent the GAF matrix using vectors $v_i = (\cos(\theta_i), -\sin(\theta_i))$ and $w_i = (\cos(\phi_j), \sin(\phi_j))$. Vectors $v_i, w_i \in \mathbb{R}^2$ do form a **bilinear mapping** (linear in the first/second argument).

$$G = \begin{pmatrix} \cos(\theta_1 + \phi_1) & \dots & \cos(\theta_1 + \phi_n) \\ \vdots & \ddots & \vdots \\ \cos(\theta_n + \phi_1) & \dots & \cos(\theta_n + \phi_n) \end{pmatrix} = \begin{pmatrix} \langle v_1, w_1 \rangle & \dots & \langle v_1, w_n \rangle \\ \vdots & \ddots & \vdots \\ \langle v_n, w_1 \rangle & \dots & \langle v_n, w_n \rangle \end{pmatrix} \tag{27}$$

Given no feature map exists that represents 25, no equivalent **PSD** kernel exists, which implies the GAF matrix not a **gram matrix** that factorises as the matrix product of two feature maps with identical embeddings.

For a time series $x \in \mathbb{R}^n$, we can define the polar embedding using equation 28.

$$(r_i, \theta_i) = \begin{cases} \frac{i}{n} \\ \cos^{-1}(x_i) \end{cases} , i = \{1, \ldots, n\} \tag{28}$$

The radius encodes the time element of our series, and is strictly increasing.

$$(x_i, y_i) = \begin{cases} r_i x_i \\ r_i \sin(\theta_i) \end{cases} , i = \{1, \ldots, n\} \tag{29}$$

We can apply the inverse polar transform, equation 29, to get the polar embedding in Cartesian coordinates, which will show the geometry of our transformation. Importantly, y_i here denotes the y coordinate axis, not to be confused with a lagged version of the x vector.

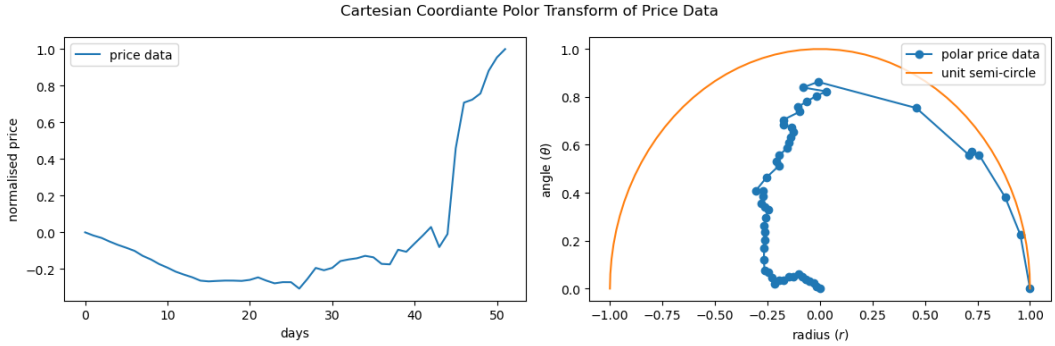

Figure 6: HLC3 (average of high, low and close) for each of 52 days (left) and corresponding polar transform in Cartesian coordinates (right).

Applying equation 29, we get the Cartesian coordinate embedding of the polar transform [Figure 6]. Importantly the GAF does not use the radius (r). This is just a visual representation of the complete polar embedding.

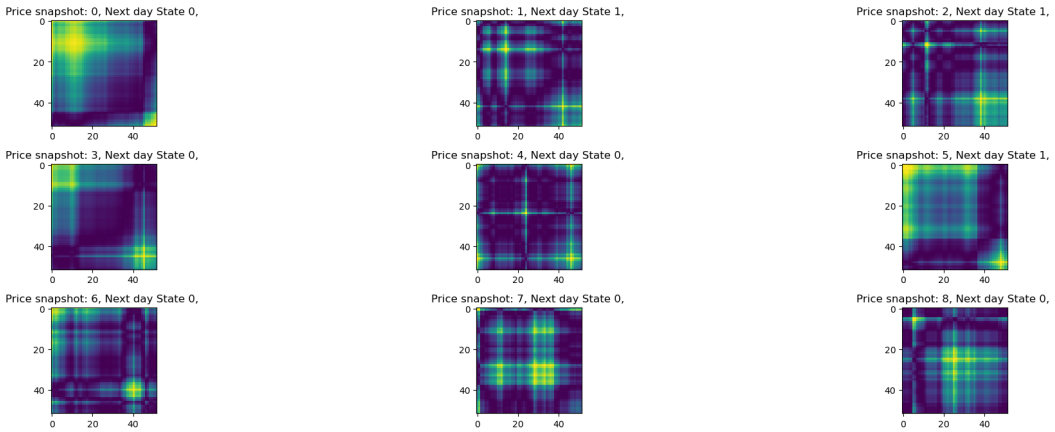

Figure 7: 52×52 GAF matrix for first 52 HLC3 data batches. GAF matrix values range between -1 and 1 and are coloured by intensity.

Figure 7 shows the GAF matrix for the first nine image in our training set, with their corresponding class label.

4.3 Probabilistic Loss Function

Our model outputs two logits, which we can transform to probabilities using the softmax activation function.

$$y_i|X^{(i)} \sim \text{Bernoulli}(\hat{p}_i); \hat{p}_i = \text{softmax}(f_\theta(X^{(i)})) \tag{30}$$

$$\begin{aligned} p_\theta(y_i|X^{(i)}) &= \begin{cases} \hat{p} & \text{, if } y_i = 1 \\ 1 - \hat{p} & \text{, otherwise} \end{cases} \\ &= \hat{p}^{y_i}(1-\hat{p})^{1-y_i} \end{aligned} \tag{31}$$

Following from the definitions in chapter 2.2 our loss function is derived from the log-likelihood of our true class label distribution conditional on the input data that generated it. Here, we do not make an additive Gaussian noise assumption; we assume the distribution of our observations, given our input features follow a Bernoulli distribution.

$$\begin{aligned} \ell(\hat{p}; y_i) &= \sum_{i=1}^{n} \log(\hat{p}^{y_i}(1-\hat{p})^{1-y_i}) \\ &= \sum_{i=1}^{n} [y_i \log(\hat{p}_i) - (1-y_i)\log(1-\hat{p}_i)] \end{aligned} \tag{32}$$

Equation 32 is the Bernoulli log-likelihood. The likelihood is increasing for $\hat{p}_i \to y_i$, leading to a model with higher accuracy. For $y_i = 1$ (asset price increase after HLC3 batch) the likelihood is increasing for values of $\hat{p}_i$ that tend to 1, as for $y_i = 0$ the likelihood is increasing for values of $\hat{p}_i$ that tend to 0. In both cases, the opposite class term evaluates to 0.

We will optimise our model by maximising 32 with respect to the parameter $\hat{p}$. As shown in chapter 2.2 this is identical to minimising the negative log-likelihood with respect to the parameter of interest.

$$\begin{aligned} \theta^* &= \arg\min_\theta \{-\ell(\hat{p}; y_i)\} \\ &= \arg\min_\theta \left\{ -\sum_{i=1}^{n} [y_i \log(\hat{p}_i) - (1-y_i)\log(1-\hat{p}_i)] \right\} \end{aligned} \tag{33}$$

The cross-entropy $H(p, q) = -\mathbb{E}_{x\sim p(x)}[\log(q(x))]$ is defined as the expectation of the log model distribution over the distribution of the data-generating process. Although $p(x)$ is nearly always intractable (not computable), we define the empirical cross entropy.

$$\begin{aligned} \hat{H}(p^*, p_\theta(y|X^{(i)})) &= -\frac{1}{n} \sum_{y\in\{0,1\}} \log(p_\theta(y|X^{(i)})) \\ &= -\frac{1}{n}[\log(p_\theta(y=0|X^{(i)})) + \log(p_\theta(y=1|X^{(i)}))] \\ &= -\frac{1}{n}[y_i \log(\hat{p}_i) - (1-y_i)\log(1-\hat{p}_i)] \end{aligned} \tag{34}$$

Summing over all *iid* feature vectors we get

$$\hat{H}(p^*, p_\theta(y|X)) = -\sum_{i=1}^{n} \hat{H}(p^*, p_\theta(y|X^{(i)}))$$
$$= -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{p}_i) - (1-y_i)\log(1-\sigma(\hat{p}_i)] \tag{35}$$

Equation 35 is the Binary-cross entropy and is proportional to the negative Bernoulli log likelihood loss function defined by 32. In practice, we use the Binary cross-entropy, redefining our loss function we have.

$$\theta^* = \arg\min_{\theta} \left\{ \hat{H}(p^*, p_\theta(y|X)) \right\} \tag{36}$$

4.3.1 Probabilistic Signal

We can derive actionable signals from our model prediction.

$$\text{signal} = \begin{cases} \text{long} & \text{, if } \hat{p}_i \geq 0.5 \\ \text{short} & \text{, otherwise} \end{cases} \tag{37}$$

5 Data and ML Model Pipeline

Our dataset contains Open, High, Low, Close, Adjusted Close, and Volume time series data for Alphabet Inc (GOOG), ranging from August 2004 to August 2025. We use the Adjusted Close, High and Low series to construct our feature vector and derive our class label from a lagged version of our feature series.

We subset our entire dataset so that it is divisible by the number of data batches, (100). We intern have 100, 52×52 images after applying the GAF transformation [section **??**], similar to MINST data our label is a scaler. Our label takes values $\{0, 1\}$ depending on if asset price increased the next day after our batch. We split out features and labels into training, validation and test set, using a $49\% - 21\% - 30\%$ train-validation-test split.

We train our model by passing 10 epochs of data through the Adam optimisation algorithm [], defined using a binary cross-entropy loss function [Equation 35] and learning rate of 0.0001. We define convolution filter dimensions and max pooling dimensions as hyperparameters, and run a grid search on a combination of $n \cdot m$ dimensions, where n and m denote the number of convolution dimensions and pooling dimensions. We choose $n = m = 3$ for a total of 9 different pair combinations. The model that achieves the lowest validation set loss is chosen as the optimal model and used for prediction in the tets set.

6 Backtesting Results

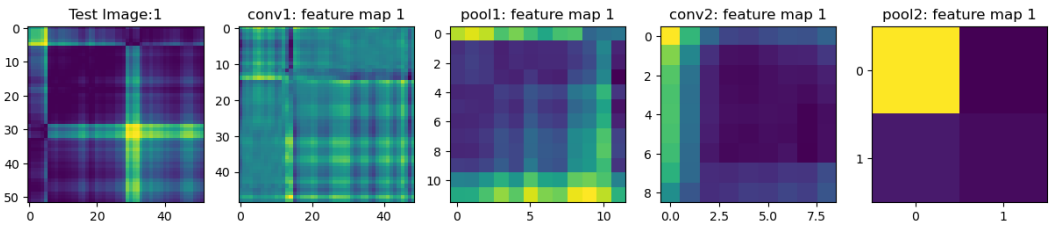

Figure 8: Trained CNN model convolutional and pooling layers.

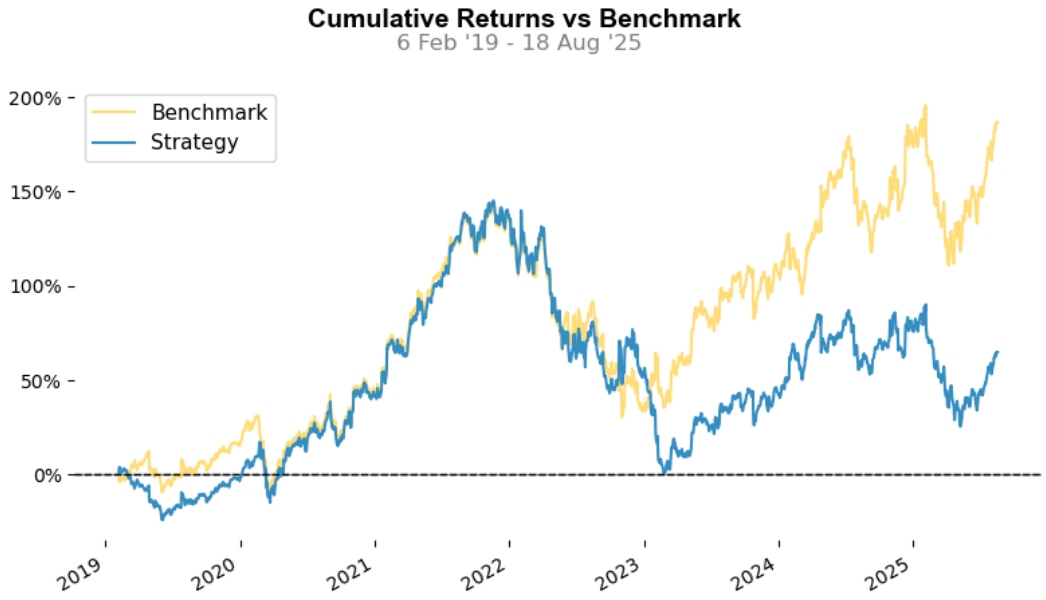

Figure 9: Benchmark (GOOG) vs CNN trading strategy cumulative returns.

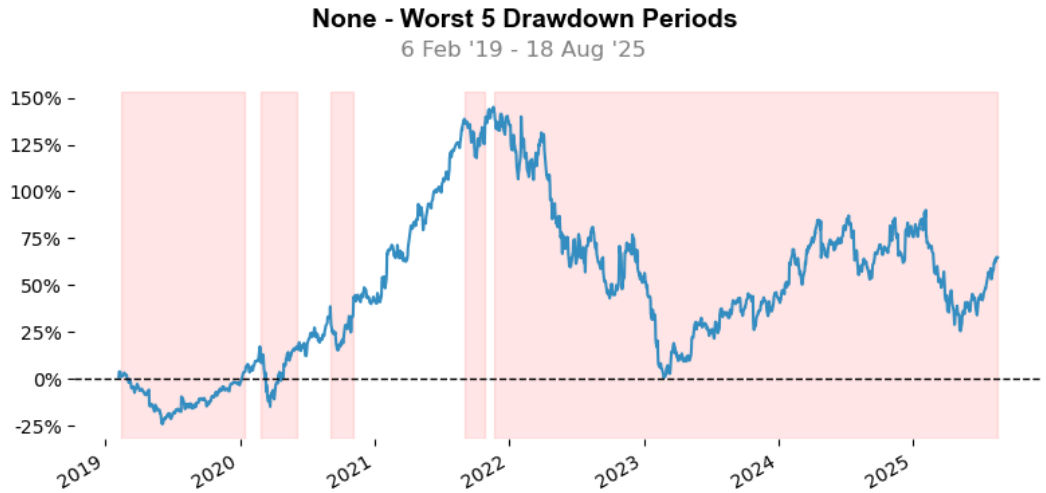

Figure 10: CNN trading strategy cumulative return (blue) and drawdown until recovery (red).

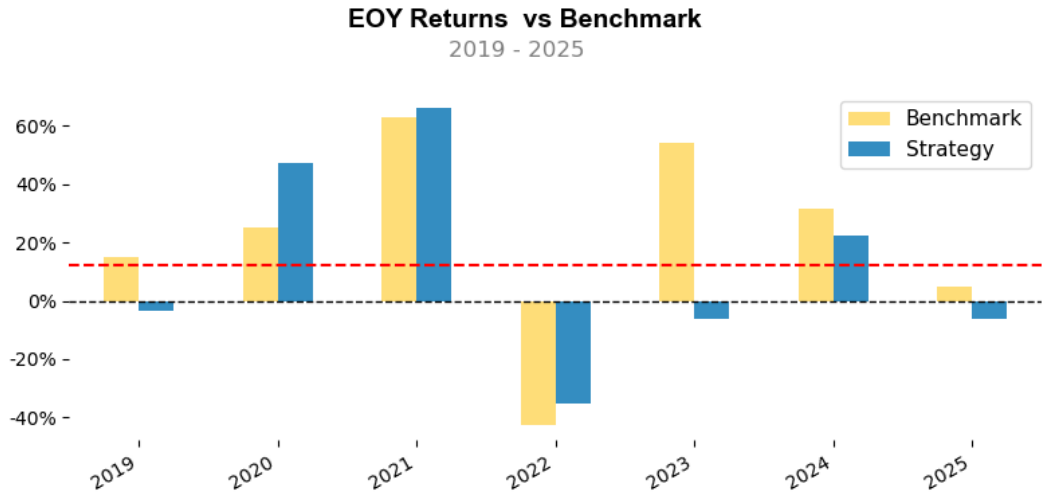

Figure 11: End of year cumulative return of benchmark (blue) vs CNN strategy (yellow).

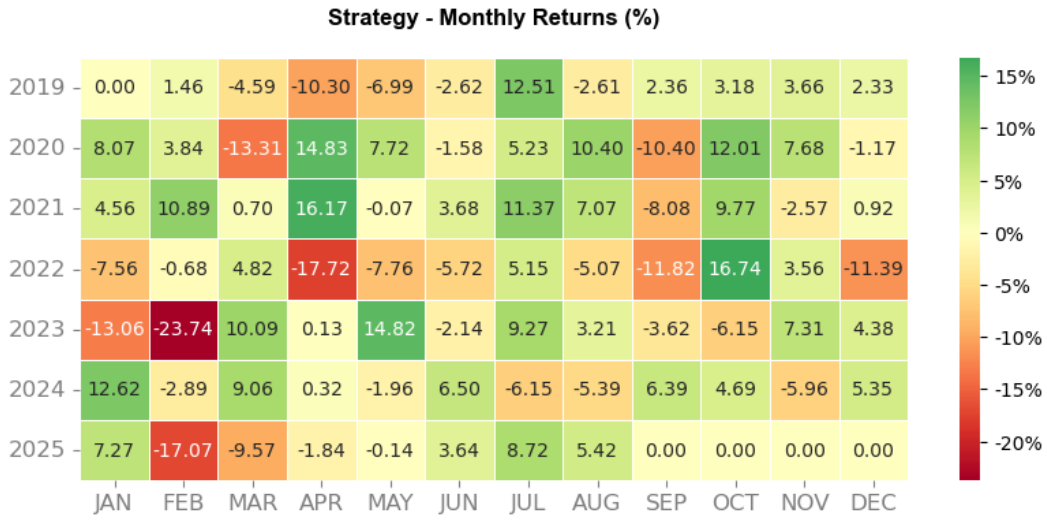

Figure 12: CNN strategy monthly cumulative return.

7 Discussion

7.1 Feature Extraction

We determine a convolution matrix and maximum pooling matrix dimension of 4×4 to be optimal. Figure 8 shows the feature extraction section of our optimised CNN using GAF test image 1 of test price batch 1. The feature maps learn a hierarchical latent representation of the image. Convolutional layer 1 detects lower-level features, specifically the edges of the right side cross in GAF test image 1. Proceeding pooling and convolutional layers grow the receptive field and downscale the test image into a 2×2 final latent representation. The last layer in a CNN is the richest embedding of the input, in theory making the classification task for the CNN simpler [Figure 8].

7.2 Backtesting Results

We pass test set data through our optimised and trained model and extract a trading signal from the model output using equation **??**. Our benchmark grows our initial wealth by a greater factor than our strategy [Figure 9].

Our strategy matches benchmark performance starting from the first quarter of 2020 to the first quarter of 2022 [Figure 9] suggesting that our model correctly classified many batches as price patterns that suggest a price increase given their GAF form. Strategy cumulative return performance again follows a near identical path to the benchmark from periods 2023 to 2025, but not before taking a briefly sustained opposite market position in the last quarter of 2022 to the first quarter of 2023 [Figure 9]. This indicates that our models class predictions where not in line with the true class label.

The strategy highest cumulative return of $\approx 145\%$, was achieved in Q4 2022, in line with the benchmark at the time [Figure 9]. Our strategy would remain in drawdown from this period, although did recover from its maximum drawdown achieved in Q1 2023 [Figure 10].

Our strategy has greater end-of-year (EOY) returns than the benchmark, three years (2020, 2021, 2022) out of the seven years (2019-2025) in the test set [Figure 11]. Our strategy has volatile monthly returns, with our best month being in October 2022 with a monthly cumulative return of 16.74%, and our worst monthly cumulative return was only four months after in February 2023 with a cumulative monthly return of -23.74% [Figure 12].

8 Research Limitations

Limitations of our research methodology include but are not limited to.

Too Large Batch Size. Input data quality and data quantity are inversely related and finding the balance is difficult. The higher the number of batches the more data we have to pass to our CNN and the more signals can be produced, although this is at the expense of image resolution. For example increasing the batch size from 100 will result in images with dimension $\leq 52 \times 52$, this reduces the amount of information that can be learned from every image that could impact the CNNs accuracy, although we will have more signals and possibly better out-of-sample performance.

Noisy Class Label. We are deriving our class label by lagging our feature vector and creating a class label of 1 if the price of the first entry in the current batch is greater than the last price in the prior batch, setting the class label to 0 otherwise. The intuition is the next day market state is dependent on the prior batch information, although this may be true the signal is likely to noisy, if we where to consider a longer post batch period and compare the price difference we may be able to derive more robust and accurate directional predictions. The idea is the batch pattern itself produces a pattern where price increases or decreases rather than a single price move.

9 Conclusion

We started by defining a the Multi Layer Percepatron (MLP) and its use cases before moving to mathematical definitions of the pre-activation and activation neruon. Chapters 2.2 to chapters **??** discussed loss functions and how they can be derived from a probabilistic perspective given an assumption on the conditional distribution of our class label given input data (our model output). Chapter **??** defined gradient descent methods, specifically mini-batch stochastic gradient descent (SGD) and how it is used in the backward pass algorithm, backpropagation, to train deep neural networks. Chapter 2 finished with defining

the forward pass and backward pass algorithms.

Chapter 3, built on the fundamental definition and ideas in chapter 2. We learned CNNs are fundamentally MLPs with an added layer of complexity called feature extraction that allows them to classify image data. Chapter 3.1 discussed key operations within the feature extraction layers, such as convolution, padding, pooling and stride.

We motivated the research problem of asset price prediction using CNN's in chapter 4. We proposed the use of the Gramian Angular Field (GAF) to transform our time series data to a rich image representation and improve the CNN's ability to classify latent representations of price patterns. The GAF is defined as a matrix of n^2 ordered pairs of $\cos(\theta_i + \phi_j)$ where $\theta_i = \cos^{-1}(x_i)$ and $\phi_i = \cos^{-1}(y_i)$, and y_j is a lagged version of $x_i \in [-1, 1]$ given $x \in \mathbb{R}^n$.

Chapter 4.3.1 defined how we would transform our mode output to a trading signal, using a likelihood approach. Chapter 5 discussed our dataset and model training pipeline. Chapter 6 presented our research findings, which we discussed and evaluated in chapter 7. Research limitations where identified in chapter 8 and included batch size and incorrectly defined class labels.

We have developed a solid understanding of simple neural networks, the model training process and distinct architectures such as MLPs and CNNs. We chose CNNs as a solution to our asset price pattern classification and prediction task. The GAF was used as a feature engineering technique to improve models identification ability. Defining a trading signal from model output and using our trained and optimised model on unseen data we found our CNN to substantially underperform the benchmark. Upon critical evaluation of model output we suggested model improvements leave the research task open to improvement.

10 References

[1] - Chakraborty, S. and Sarddar, D., Optimized Routing Approach using McCulloch-Pitts Neuron model.

[2] - Harsh Yadav 07/05/2022, Towards Data Science. Computer Vision: Convolution Basics, accessed 02/12/2026, https://towardsdatascience.com/computer-vision-convolution-basics-2d0ae3b79346/

[3] - Wikipedia, accessed 01/12/2026, https://en.wikipedia.org/wiki/Convolutional_neural_network

[4] - Krut Patel 09/08/2019, Medium. MNIST Handwritten Digits Classification using a Convolutional Neural Network (CNN). https://medium.com/data-science/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9