



**ADVANCED DATA ANALYTICS ALGORITHMS,
MACHINE LEARNING**

Assignment 2: Algorithm Implementation Journal
Implementing Convolutional Neural Network
in image classification

Table of Contents

1.	Introduction	3
2.	Motivation	3
3.	Theoretical background	4
4.	Implementation	14
5.	Result evaluation	18
6.	Conclusion and reflection	28
7.	Reference and AI assistance	28

1. Introduction

This journal is composed with the purpose of demonstrating my understanding and implementation of convolutional neural network (CNN), a neural network–based model, which is specific for tasks relating to image classification. I decided to opt for option 1 for this journal, meaning diving into the task to the technical level, scrutinising all the aspects of the model and the task, and then implementing the model from scratch. This includes:

- Motivation
- Problem addressing
- Theoretical background
- Implementation
- Result evaluation
- Conclusion and reflection
- Reference.

Further model elaborations, which focus on specific facets, are going to be presented through each part accordingly.

2. Motivation

In the era of digitalisation, it is imperative to recognize that all data should be regarded as pivotal resources that can be harnessed by humans to gain significant advantages for various courses of action, such as obtaining valuable insights and facilitating informed decision-making. Images, in particular, constitute a category of data that contains a wealth of information encompassing attributes like shape, colours, textures, and more, which can be extracted and leveraged for a multitude of purposes.

It is crucial to acknowledge that computers perceive and process images in a fundamentally different manner compared to humans. Human perception involves the transmission of light waves through the lens of the eye, which are then converted into electrical signals by photoreceptors in the retina. These signals are subsequently transmitted to the brain, where they are reconstructed into a coherent image. In contrast, the computational environment operates exclusively with numerical values, affording no exceptions for visual data like images.

It is essential to credit the pioneering work of computer scientist Russell Kirsch, who introduced the concept of the "pixel" (short for "picture element"). Pixels serve as the closest computational analog to photoreceptors (Bigun, 2006), allowing computers to represent images as a grid of pixels arranged in rows and columns. Each of these pixels is associated with numerical values ranging from 0 to 255. Consequently, an image is encoded as an array of such numerical values, delineated by its height and width dimensions.

Moreover, depending on the chosen colour model, an image may possess one, three, or four color channels, each of which represents the primary colour components of the image. These color models typically encompass grayscale, RGB (Red, Green, Blue), or CMYK (Cyan, Yellow, Magenta, Black), with each channel contributing to the overall colour composition of the image and further augmenting the complexity of its numerical representation.

Image classification has been being utilised for many purposes in practice. It could be detecting explicit images on social network platforms, car plate recognition, medical imaging diagnosis,....And this demand trend is still witnessing exponential growth. To meet this demand, top-notch models have been invented, one of those is convolutional neural network. Convolutional neural network (CNN) has proved its capability in image classification via all image classification champion models for the last ten years, for instance, AlexNet (2012),

VGGNet (2014), etc., mostly are predicated upon convolutional neural network architecture (. Convolutional neural network overwhelms other traditional image processing due to its self-learning characteristic without any pre-feature engineering, meaning utilising learning algorithm to update its weights and biases. Moreover, CNN outweighs the ANN models with the capability to extract image features maps , soothing the computing complexity with which ANN models usually struggle when it comes to large image(O' Shea, 2015).

3. Theoretical background

Convolution, in terms of mathematics, is an operation on two functions to produce the third function. The third function expresses how the first function's shape is modified by the second function. In practice, an image, in form of the pixels grid, could be expressed as an function expressed as:

$$I(x,y)$$

Where:

$I(x,y)$ is the value of the image at the coordinate (x,y) .

For the justification to apply kernel/filter, which is another function:

$$K(i,j)$$

Where:

$K(i,j)$ is the value of the kernel at the coordinate (i,j) ,

In order to let CNNs operate properly in image classification task, it is necessary to have following assumptions based on the nature of the image:

- Spatial information: each image contains specific information about location, shape, size of objects and patterns
- Statistical properties: this relate to the consistency of image statistical characteristics, such as mean and variance, in different regions.
- Pixel dependencies: Pixels have a tendency to be related to other nearby pixels, indicating spatial correlation or locality.
- Locality of features: Features in an image are often localized, and can be hierarchically.
- Translation invariance: the ability to recognize features and patterns in image regardless of their exact position or translation.

In fact , a couple of them are proved correctly. Krizhevsky et al. (2017) has proved the assumptions of image's stationary of statistics and locality of pixel dependencies with winning the ImageNet LSVRC-2010 contest.

The concept of CNN architecture has been introduced for the first time in 1980 by Kunihiro Fukushima, a Japanese computer scientist, with the 'neocognitron', a hierarchical, multilayer artificial neural network inspired by the discovery of Hubel & Wiesel in the 1950s and 1960s about two cell types of visual primary cortex, namely simple and complex cell, and a proposed cascading model for use in pattern recognition.

Based on the model 'Neocognition', CNN architect is comprised of three types of layers (O'Shea, 2015), a convolutional layer, a pooling layer, and a fully connected layer.

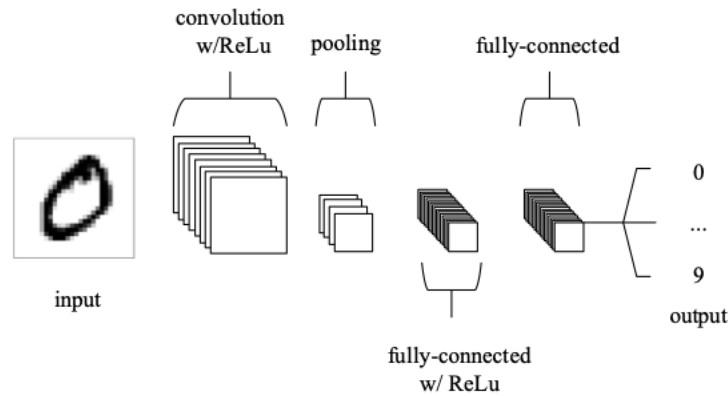


Fig.1: The basic CNN model architecture (O'Shea, 2015)

The convolutional layer is the layer in which the convolution operation of the image and the kernels to extract feature maps from the input data. In this layer, input is the input image. The convolution operation involves sliding a small filter (or kernel) over the input image, and at each position, element-wise multiplication occurs between the values in the receptive field (a local region of the input data) and the values in the kernel. These element-wise multiplications are summed up to produce a single value for the corresponding position in the output feature map. Below is the equation of the convolution:

$$x_{ij} = \sum_{a=-\infty}^{+\infty} \sum_{b=-\infty}^{+\infty} w_{ab} \cdot y_{(i-s-a)(j-s-b)} + b \quad \forall i \in (0, \dots, N) \quad \forall j \in (0, \dots, M)$$

Where:

x: the output feature map

i,j,a,b: the indexes of the elements in the matrices

s: is the value of the convolution step(stride)

w: convolutional kernel

b: bias

y: is the input image

The size of the output feature maps depends on the stride and the padding. Stride is a hyperparameter that defines how many pixels the kernel moves when sliding over the input data for the convolution. Another hyperparameter that should be considered in the convolution is padding. Padding is the hyperparameter that is utilized to extract features at the edge of the input data. To do the padding, columns and rows of zeros are added to the input image. The output of the convolution layer is feature maps with dimensions that are calculated as below:

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

Where:

W_{out} : the width of the feature map

W_{in} : the width of the input image

padding[1]: the column number of the zeros added on each side of the input image

kernel_size[1]: is the column number of the kernel

stride: the step size of the convolution

H_{out} : the height of the feature map

H_{in} : the height of the input image

padding[0]: the row number of the zeros added on each side of the input image

kernel_size[0]: the row number of the kernel.

To have a better understanding, below is the image demonstrating the convolutional operation in two cases: Case 1 is the unpadded image with kernel size 2x2, and stride of 1; and Case 2 is padded image with zero padding of 1, kernel size 2x2, and stride of 2

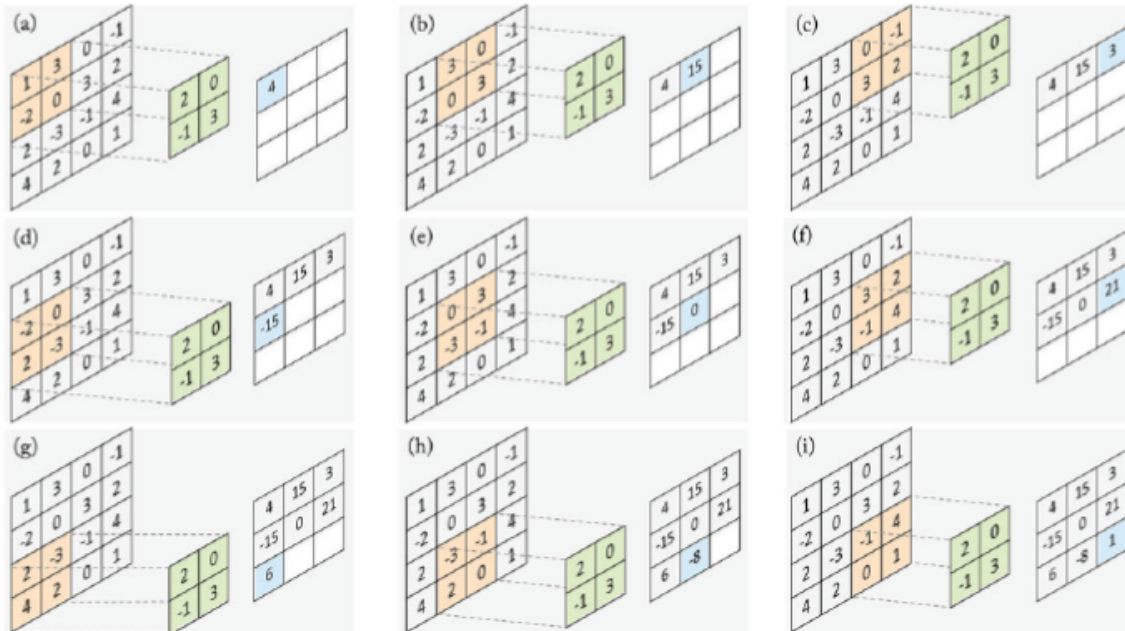


Fig.2: Case 1 convolution operation (Khan, 2018)

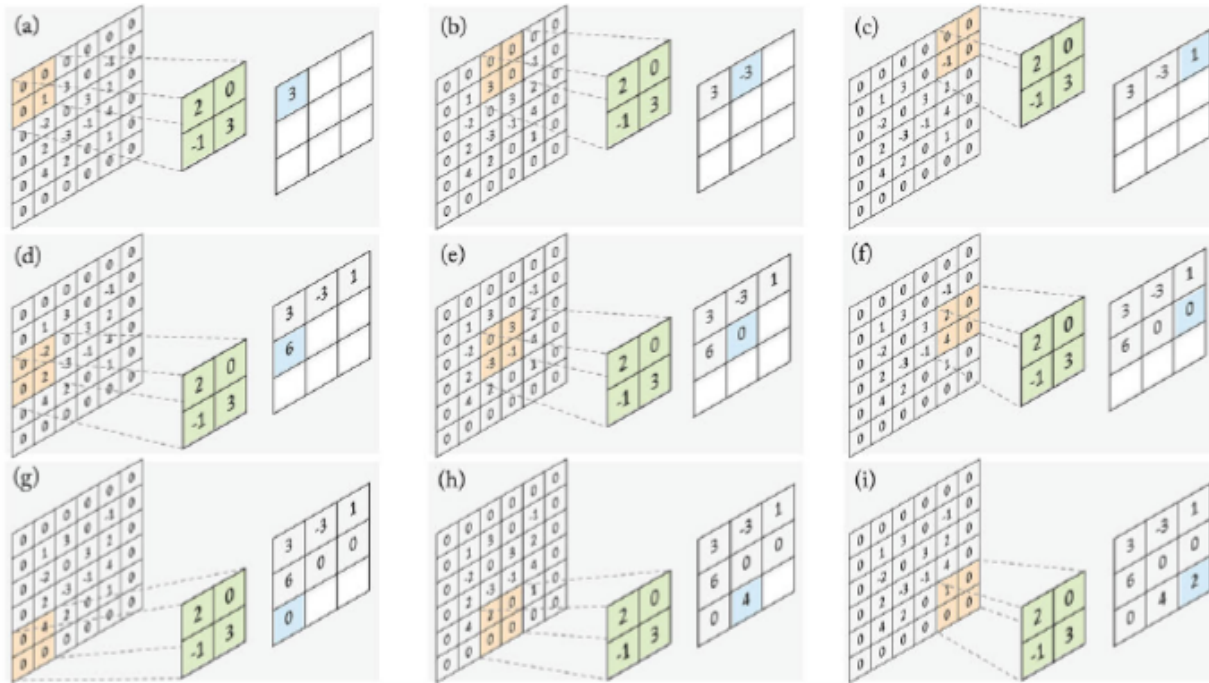


Fig.3: Case 2 convolution operation (Khan, 2018)

After the convolution, a non-linear activation function, Rectified Linear Unit (ReLU) is applied in order to facilitate the computation. Image is highly non-linear, meaning pixels in the image are not linearly related to visual information. Without activation function, the number of layers in the neuron network doesn't matter at all, those layers could be replaced by a single layer with a linear activation function. After applying convolution, there are relevant and irrelevant pixel values in the feature map. Acting like a switch, the ReLU activation function switches off all irrelevant values (negative values in the feature map), turning them to 0, and leaves all positive values unchanged. This manner helps the model detect meaningful patterns in the image more feasibly, ignoring all noises and irrelevant details. Furthermore, the ReLU leaves the gradient of positive input values to be 1, This means that the gradient doesn't decrease during backpropagation, ensuring lower layers can be learnable and avoiding "vanishing" gradient problem. The equation of ReLU activation function is simple:

$$f_{ReLU} = \max(0, x)$$

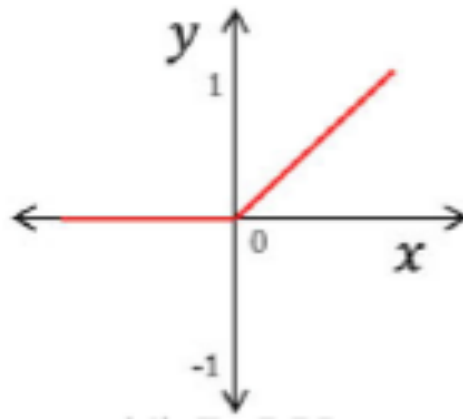


Fig.4: ReLU activation function (Khan, 2018)

The next layer is pooling, where the downsampling is conducted in a specific local region of the input feature map to reduce the dimensionality of the input, while maintaining the crucial patterns and values. The dimensionality reduction helps the model to lower the number of learnable parameters and the computational cost. Accordingly. In CNNs, the pooling operation is conducted in two common types: max and average pooling. These types poses two different approaches; while max pooling highlights the highest value in the local region, emphasizing the prominent features, average pooling computes the mean value among the local region, smoothening the information summary. The number of the local region of the input feature map depends on the size of pooling window and pooling window step size (stride). For the the size of the output feature map after pooling, we have equations:

$$W_{poolout} = \left\lfloor \frac{W_{in} - Pooling_{width}}{Stride_{horizontal}} \right\rfloor + 1$$

$$H_{poolout} = \left\lfloor \frac{H_{in} - Pooling_{height}}{Stride_{vertical}} \right\rfloor + 1$$

Where:

$w_{poolout}$ and $H_{poolout}$: output width and height

w_{in} and H_{in} : input width and height

$Pooling_{width}$ and $Pooling_{height}$: pooling window width and height

$Stride_{horizontal}$ and $Stride_{vertical}$: stride used in horizontal and vertical direction

To compute the value of each position in the output feature map, below equation is applied:

For max pooling:

$$B_{ij} = \max(A_{(i:(i+1) \cdot k, j:k:(j+1) \cdot k)}) \quad \forall i \in (0, \dots, N) \quad \forall j \in (0, \dots, M)$$

Where:

B_{ij} : the value at the i-th row and j-th of the output feature map

$A_{(i:k:(i+1) \cdot k, j:k:(j+1) \cdot k)}$: the submatrix of the input feature map A within the pooling window with size $k \times k$ at position (i,j)

For average pooling:

$$B_{ij} = \frac{1}{Pooling_{height} \cdot Pooling_{width}} \sum_{p=i \cdot Pooling_{height}}^{(i+1) \cdot Pooling_{height} - 1} \sum_{q=j \cdot Pooling_{width}}^{(j+1) \cdot Pooling_{width} - 1} A_{pq}$$

$$\forall i \in (0, \dots, N) \quad \forall j \in (0, \dots, M)$$

Where:

B_{ij} : the value at the i-th row and j-th of the output feature map

$Pooling_{width}$ and $Pooling_{height}$: pooling window width and height

A_{pq} : the value at the p-th row and q-th column of the input feature map.

The last layer in CNN is the fully connected layer, where a set of operations are conducted. The first operation is flattening, and concatenating all output feature map into a one-dimensional vector. Then values in this vector are fed to a fully connected layer with added biases. The neuron number of the fully connected layer is identical to the class number of the target. The fully connected layer is basically an ANN model activated by a function. In a fully connected layer, each neuron computes a weighted sum of the inputs it receives from the previous layer and then applies an activation function. With the classification task, the activation function is sigmoid for binary classification and softmax for multiclass classification. Below is the graph demonstrating that the fully connected layer with input is the vector from flattening and the output is a vector containing numerical values that represent the activations.

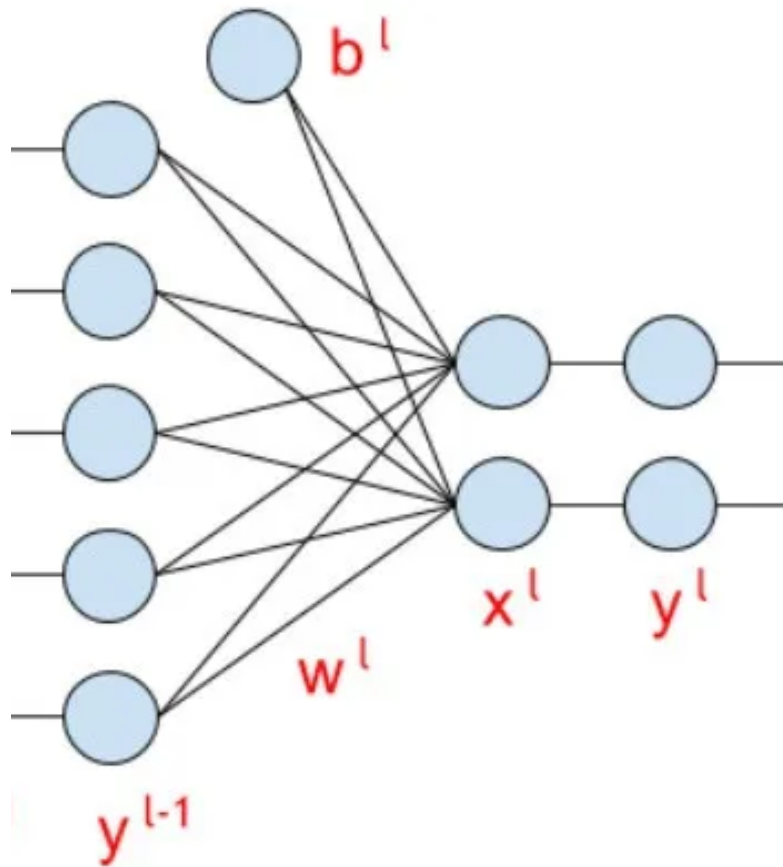


Fig.5: Fully connected layer for binary classification with two neurons (Kalinin, 2021)

The equation of the fully connected layer is :

$$x_i = \sum_{k=0}^m w_{ki} y_k + b_i \quad \forall i \in (0, \dots, n)$$

Where:

x_i : the numerical value of i-th output neuron

w : the weight matrix

k : the position k-th of the input vector

i: the neuron number i-th of the output layer

m: the number of values in the input vector

n: the number of neurons in the output layer

In the scope of this journal, the task is multiclass classification, thus softmax is utilised to produce probability distribution, which are measures of the likelihood of an input belonging to each class, with the following equation:

$$y_i = \frac{e^{x_i}}{\sum_{k=0}^n e^{x_k}}$$

Where:

y_i : the predicted probability of the i-th class

n: the number of classes.

The softmax activation function is utilised for multiclass classification due to following reasons:

- The output of softmax is a probability distribution, meaning softmax transforms values of fully connected output layer raw values into probability scores ranging from 0 to 1, and sum of these scores equals 1.
- The class prediction in classification is straightforward, by choosing the class with the highest probability as the prediction class, facilitating the decision making
- The softmax is smooth and differentiable, thus suitable for gradient-based optimization with which cross-entropy is combined during the training process

To measure the dissimilarity, between the predicted probability distribution, and true (target) probability distribution, cross-entropy is used as loss function., the lower the loss is, the better the model is. With the categorical target for the classification task, another step should be taken prior to calculating the loss, one-hot encoding the target. One-hot encoding transforms categorical targets into binary vectors. The binary vector has the number of values equal to the number of classes, with the value at the position corresponding to the category is "1", and all other positions are "0". This format fits to the probability distribution of the target. Apart from measuring the dissimilarity, the cross-entropy applies logarithmic transformation to the predicted probabilities. This transformation has the following benefits:

- Penalising strongly confident incorrect predictions more than slightly confident incorrect predictions.
- Encouraging the model to be well-calibrated, meaning that predicted probabilities represent the actual likelihood of class membership

The equation of cross-entropy is:

$$E = - \sum_{i=0}^n y_i^{truth} \ln(y_i)$$

Where:

E: the loss value of the predicted probabilities distribution comparing to ground-truth one

y_i^{truth} : the true probability of i-th class

y_i : the predicted probability of the i-th class

n: the number of classes.

In general, a basic CNN for multiclass classification should have these layers and operations:

- Convolution layer (convolution, ReLU)
- Maxpooling layer
- Fully connected layer (Flattening, Fully connecting, softmax activation function, cross-entropy loss function)

The following process in the CNN model is the training process in which all learnable parameters are updated accordingly. The concept of updating these variables is how the loss changes when the parameters and biases change. Firstly, we need to calculate the gradient, which the partial derivative, of the loss with respect to weights and biases in convolutional layers and fully connected layers. Then gradient values are utilised in optimization algorithms, like gradient descent. In gradient descent, parameters are updated by subtracting a fraction of the gradient from the current parameters:

Parameter Update: $\text{parameter} \leftarrow \text{parameter} - \text{learning rate} \times \nabla L$.

Learning rate is one of the crucial hyperparameters, scaling the magnitudes of the parameter updates. With small values, the learning process requires lots of updates to reach the convergence point (the point with the minimum loss). Otherwise, too large a learning rate causes drastic updates, leading to divergent behaviours.

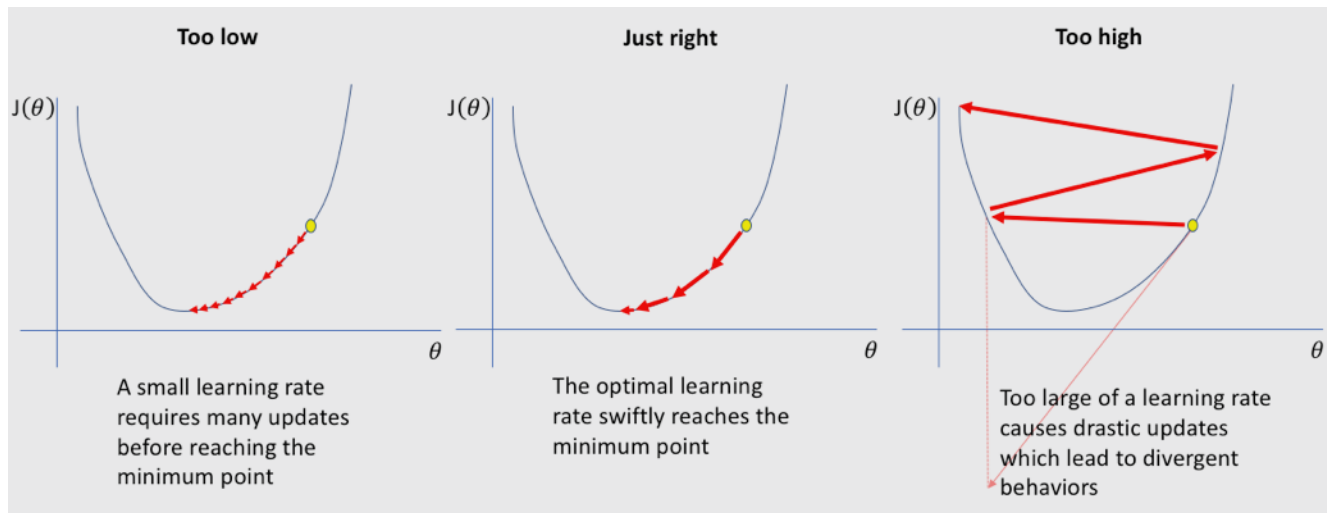


Fig.6: The effect of the learning rate to the convergence (Jordan, 2018)

By applying the chain rule, we could calculate the derivatives of loss with respect to parameters indirectly via layers and operations.

With fully connected layer, we have the derivatives of loss with respect to weights and biases:

$$\frac{\partial E}{\partial w_{ki}^l} = \frac{\partial E}{\partial y_i^l} \frac{\partial y_i^l}{\partial x_i^l} \frac{\partial x_i^l}{\partial w_{ki}^l}$$

$$\frac{\partial E}{\partial b_i^l} = \frac{\partial E}{\partial y_i^l} \frac{\partial y_i^l}{\partial x_i^l} \frac{\partial x_i^l}{\partial b_i^l}$$

With the convolutional layer, we have the derivatives of loss with respect to the weights and biases

$$\frac{\partial E}{\partial w_{ab}^l} = \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \frac{\partial x_{ij}^l}{\partial w_{ab}^l}$$

$$\frac{\partial E}{\partial b^l} = \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \frac{\partial x_{ij}^l}{\partial b^l}$$

To train the model, we need to do the backpropagation for each layer and operation.

With the derivative of loss through the loss function, we have below equation:

$$\begin{aligned} \frac{\partial E}{\partial y_i^l} &= \frac{\partial(-\sum_{i=0}^n (y_i^{truth} \cdot \ln(y_i^l))}{\partial y_i^l} \\ &= \frac{\partial(-(y_0^{truth} \ln(y_0^l) + \dots + y_i^{truth} \ln(y_i^l) + \dots + y_n^{truth} \ln(y_n^l))}{\partial y_i^l} \\ &= \frac{\partial(-y_i^{truth} \ln(y_i^l))}{\partial y_i^l} = -\frac{y_i^{truth}}{y_i^l} \end{aligned}$$

Before the loss function operation is the softmax operation, with the derivative of predicted probability through the softmax operation function. For this operation, we have the derivative of predicted probabilities with respect to the output logics of fully connected layer, and the equation is split into two, one with x_i^l and x_j^l :

$$\begin{aligned} \frac{\partial y_i^l}{\partial x_i^l} &= \frac{\partial}{\partial x_i^l} \left(\frac{e^{x_i^l}}{\sum_{k=0}^n e^{x_k^l}} \right) = \frac{e^{x_i^l} \cdot \sum_{k=0}^n e^{x_k^l} - e^{x_i^l} \cdot e^{x_i^l}}{\left(\sum_{k=0}^n e^{x_k^l} \right)^2} \\ &= \frac{e^{x_i^l} \cdot \left(\sum_{k=0}^n e^{x_k^l} - e^{x_i^l} \right)}{\sum_{k=0}^n e^{x_k^l} \cdot \sum_{k=0}^n e^{x_k^l}} = y_i^l \cdot \frac{\sum_{k=0}^n e^{x_k^l} - e^{x_i^l}}{\sum_{k=0}^n e^{x_k^l}} \\ &= y_i^l \cdot \left(\frac{\sum_{k=0}^n e^{x_k^l}}{\sum_{k=0}^n e^{x_k^l}} - \frac{e^{x_i^l}}{\sum_{k=0}^n e^{x_k^l}} \right) = y_i^l \cdot (1 - y_i^l) \end{aligned}$$

$$\begin{aligned} \frac{\partial y_i^l}{\partial x_j^l} &= \frac{\partial}{\partial x_j^l} \left(\frac{e^{x_i^l}}{\sum_{k=0}^n e^{x_k^l}} \right) = \frac{0 \cdot \sum_{k=0}^n e^{x_k^l} - e^{x_i^l} \cdot e^{x_j^l}}{\left(\sum_{k=0}^n e^{x_k^l} \right)^2} \\ &= \frac{e^{x_i^l} \cdot e^{x_j^l}}{\sum_{k=0}^n e^{x_k^l} \cdot \sum_{k=0}^n e^{x_k^l}} = -y_i^l \cdot y_j^l \end{aligned}$$

Then we have sum of the backpropagation for the softmax:

$$\frac{\partial E}{\partial x_i^l} = \sum_{k=0}^n \frac{\partial E}{\partial y_k^l} \frac{\partial y_k^l}{\partial x_i^l} = \frac{\partial E}{\partial y_0^l} \frac{\partial y_0^l}{\partial x_i^l} + \dots + \frac{\partial E}{\partial y_1^l} \frac{\partial y_1^l}{\partial x_i^l} + \dots + \frac{\partial E}{\partial y_n^l} \frac{\partial y_n^l}{\partial x_i^l} \quad \forall i \in (0, \dots, n)$$

For the fully connected layer, there are backpropagation for parameters updates and through fully connected layer:

$$\begin{aligned} \frac{\partial E}{\partial w_{ki}^l} &= \frac{\partial E}{\partial y_i^l} \frac{\partial y_i^l}{\partial x_i^l} \frac{\partial x_i^l}{\partial w_{ki}^l} = \delta_i^l \cdot \frac{\partial x_i^l}{\partial w_{ki}^l} = \delta_i^l \cdot \frac{\partial (\sum_{k'=0}^m w_{k'i}^l y_{k'}^{l-1} + b_i^l)}{\partial w_{ki}^l} \\ &= \delta_i^l \cdot \frac{\partial (w_{0i}^l y_0^{l-1} + \dots + w_{ki}^l y_k^{l-1} + \dots + w_{mi}^l y_m^{l-1} + b_i^l)}{\partial w_{ki}^l} = \delta_i^l \cdot y_k^{l-1} \\ &\quad \forall i \in (0, \dots, n) \quad \forall k \in (0, \dots, m) \end{aligned}$$

$$\begin{aligned} \frac{\partial E}{\partial b_i^l} &= \frac{\partial E}{\partial y_i^l} \frac{\partial y_i^l}{\partial x_i^l} \frac{\partial x_i^l}{\partial b_i^l} = \delta_i^l \cdot \frac{\partial x_i^l}{\partial b_i^l} \\ &= \delta_i^l \cdot \frac{\partial (\sum_{k'=0}^m w_{k'i}^l y_{k'}^{l-1} + b_i^l)}{\partial b_i^l} = \delta_i^l \\ &\quad \forall i \in (0, \dots, n) \end{aligned}$$

$$\begin{aligned} \frac{\partial E}{\partial y_k^{l-1}} &= \sum_{i=0}^n \delta_i^l \cdot \frac{\partial x_i^l}{\partial y_k^{l-1}} = \sum_{i=0}^n \delta_i^l \cdot \frac{\partial (\sum_{k'=0}^m w_{k'i}^l y_{k'}^{l-1} + b_i^l)}{\partial y_k^{l-1}} \\ &= \sum_{i=0}^n \delta_i^l \cdot \frac{\partial (w_{0i}^l y_0^{l-1} + \dots + w_{ki}^l y_k^{l-1} + \dots + w_{mi}^l y_m^{l-1} + b_i^l)}{\partial y_k^{l-1}} \\ &= \sum_{i=0}^n \delta_i^l \cdot w_{ki}^l \\ &\quad \forall i \in (0, \dots, n) \quad \forall k \in (0, \dots, m) \end{aligned}$$

For the flattening layer, below is the equation:

$$\text{dL_d_inp.reshape(self.orig_im_shape)}$$

For the maxpooling, the error passes through values selected as maximum values in the forward maxpooling. The other values will be zero.

$$\frac{\partial L}{\partial I_{m,n}} = \begin{cases} \frac{\partial L}{\partial O_{i,j}}, & \text{if the } I_{m,n} \text{ was the maximum in the window} \\ 0, & \text{otherwise} \end{cases}$$

For the ReLU, the error passes through the positive values in the forward ReLU step, while the other will be set to zero.

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} \cdot \begin{cases} 1 & \text{if } X > 0 \\ 0 & \text{if } X \leq 0 \end{cases}$$

Where:

X is the input of ReLU layer

Y is the output of ReLU layer

For the convolution, the backpropagation is for parameters update and through the input

$$\begin{aligned}
 \frac{\partial E}{\partial b^l} &= \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \frac{\partial x_{ij}^l}{\partial b^l} \\
 &= \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \cdot \frac{\partial \left(\sum_{a=-\infty}^{+\infty} \sum_{b=-\infty}^{+\infty} w_{ab}^l \cdot y_{(is-a)(js-b)}^{l-1} + b^l \right)}{\partial b^l} \\
 &= \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l}
 \end{aligned}$$

$$\begin{aligned}
 \frac{\partial E}{\partial w_{ab}^l} &= \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \frac{\partial x_{ij}^l}{\partial w_{ab}^l} \\
 &= {}^{(1)} \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \cdot \frac{\partial \left(\sum_{a'=-\infty}^{+\infty} \sum_{b'=-\infty}^{+\infty} w_{a'b'}^l \cdot y_{(is-a')(js-b')}^{l-1} + b^l \right)}{\partial w_{ab}^l} \\
 &= {}^{(2)} \sum_i \sum_j \frac{\partial E}{\partial y_{ij}^l} \frac{\partial y_{ij}^l}{\partial x_{ij}^l} \cdot y_{(is-a)(js-b)}^{l-1}
 \end{aligned}$$

$$\forall a \in (-\infty, \dots, +\infty) \quad \forall b \in (-\infty, \dots, +\infty)$$

$$\begin{aligned}
 \frac{\partial E}{\partial y_{ij}^{l-1}} &= \sum_{i'} \sum_{j'} \frac{\partial E}{\partial y_{i'j'}^l} \frac{\partial y_{i'j'}^l}{\partial x_{i'j'}^l} \frac{\partial x_{i'j'}^l}{\partial y_{ij}^{l-1}} \\
 &= \sum_{i'} \sum_{j'} \frac{\partial E}{\partial y_{i'j'}^l} \frac{\partial y_{i'j'}^l}{\partial x_{i'j'}^l} \cdot \frac{\partial \left(\sum_{a=-\infty}^{+\infty} \sum_{b=-\infty}^{+\infty} w_{ab}^l \cdot y_{(i's-a)(j's-b)}^{l-1} + b^l \right)}{\partial y_{ij}^{l-1}} \\
 &= \sum_{i'} \sum_{j'} \frac{\partial E}{\partial y_{i'j'}^l} \frac{\partial y_{i'j'}^l}{\partial x_{i'j'}^l} \cdot w_{(i's-i)(j's-j)}^l
 \end{aligned}$$

4. Implementation

To implement the model, Python and its libraries are applied for specific function. The input of the model is two arrays, the image array and the label array. The image array includes pixel values of the image in grid-like format, and the label array contains the category of the image. The data is 3000 images from MNIST database, showing hand written digit images from number 0 to number 9. In order to avoid the imbalance, which could lead to bias in prediction, all categories are evenly collected. In this scope of this model, due to the complexity of RGB images, all input images are convert to one greyscale layer. Then the image array and the label array are splitted into training part, test part and validation part with the ratio 0.85, 0.075, 0.075. The label distribution and label count are conducted to explore the input data

```

1
2
3 # Plot a histogram of the labels
4 plt.hist(y, bins=10, rwidth=0.8, alpha=0.33, color='blue', edgecolor='blue')
5 plt.xlabel('Label')
6 plt.ylabel('Count')
7 plt.title('Label Distribution')
8 plt.xticks(range(10)) # Assuming you have 10 classes (0-9)
9 plt.show()

```

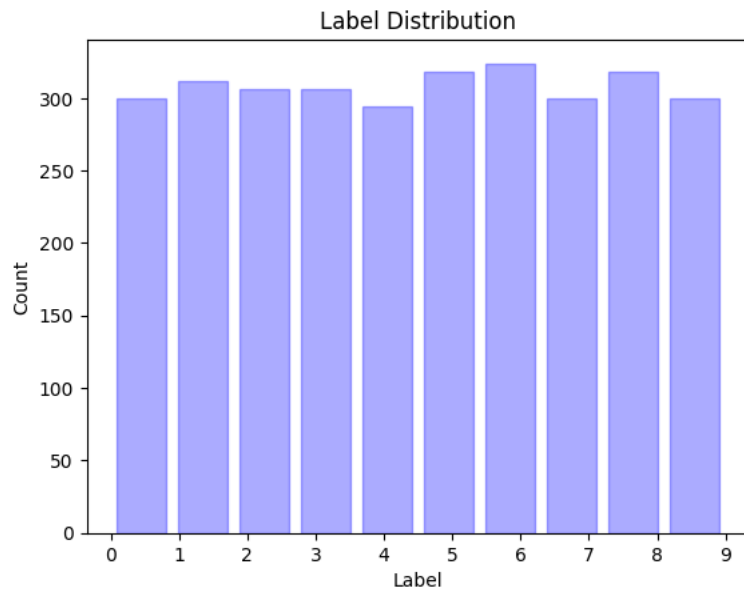


Fig.7: Label distribution

```
1
2
3 label_counts = np.bincount(y)
4 print(label_counts)
5 print(y.shape)
```

```
[300 312 306 306 294 318 324 300 318 300]
(3078,)
```

Fig.8: Label count

```
1 import os
2 import numpy as np
3 from tensorflow.keras.preprocessing.image import load_img, img_to_array
4 from sklearn.model_selection import train_test_split
5 import cv2
6 import matplotlib.pyplot as plt
7 from sklearn.preprocessing import OneHotEncoder
8 encoder=OneHotEncoder(sparse=False)
9 from sklearn.metrics import confusion_matrix
10 import seaborn as sns
11
12
13 # Define your main folder path
14 main_folder = '/content/drive/MyDrive/Hnd_3000/'
15
16 # Initialize empty lists to store images and labels
17 images = []
18 labels = []
19
20 # Loop through subfolders (0-9) and load images
21 for label in range(10):
22     subfolder_path = os.path.join(main_folder, str(label))
23     for filename in os.listdir(subfolder_path):
24         img_path = os.path.join(subfolder_path, filename)
25         img = load_img(img_path, target_size=(28, 28)) # Adjust target_size as needed
26         img_array = img_to_array(img)
27         images.append(img_array)
28         labels.append(label)
29
30 # Convert lists to NumPy arrays
31 X = np.array(images)
32 y = np.array(labels)
33
34 # Split the dataset into training and testing sets
35
36
37 X_train, X_test_val, y_train, y_test_val = train_test_split(X, y, test_size=0.15, random_state=42)
38 X_test, X_val, y_test, y_val=train_test_split(X_test_val, y_test_val, test_size=0.5, random_state=42)
39 # Convert X_train to grayscale
40 X_train_grayscale = np.array([cv2.cvtColor(img, cv2.COLOR_RGB2GRAY) for img in X_train])
41
42 # Convert X_test to grayscale
43 X_test_grayscale = np.array([cv2.cvtColor(img, cv2.COLOR_RGB2GRAY) for img in X_test])
44 X_val_grayscale= np.array([cv2.cvtColor(img, cv2.COLOR_RGB2GRAY) for img in X_val])
45
```

Then we define layer and its operations. In the following implementation, in order to prevent vanishing and exploding gradient, He initialization method has been applied when generating initial kernels and biases in the convolution layer, weights and biases in softmax.

```
#define convolution
class ConvLayer:
    def __init__(self, num_filters, filter_size, padding=0, stride=1):
        self.num_filters = num_filters
        self.filter_size = filter_size
        self.conv_filter = np.random.randn(num_filters, filter_size, filter_size)*(np.sqrt(2.0 / (filter_size * filter_size)))
        self.bias = np.zeros(num_filters) # Initialize bias terms
        self.padding = padding # Padding size
        self.stride = stride # Stride size

    def image_region(self, image):
        height, width = image.shape
        for j in range(0, height - self.filter_size + 1, self.stride):
            for k in range(0, width - self.filter_size + 1, self.stride):
                image_patch = image[j : (j + self.filter_size), k : (k + self.filter_size)]
                yield image_patch, j, k

    def forward(self, image):
        height, width = image.shape
        padded_image = np.pad(image, ((self.padding, self.padding), (self.padding, self.padding)), mode="constant")
        self.padded_image = padded_image
        # Calculate the dimensions of the conv_out array
        output_height = (height + 2 * self.padding - self.filter_size) // self.stride + 1
        output_width = (width + 2 * self.padding - self.filter_size) // self.stride + 1

        conv_out = np.zeros((output_height, output_width, self.num_filters))

        for image_patch, i, j in self.image_region(padded_image):
            i_out = i // self.stride
            j_out = j // self.stride
            conv_out[i_out, j_out] = np.sum(image_patch * self.conv_filter, axis=(1, 2)) + self.bias # Include bias

        return conv_out

    def backward(self, dL_dout, learning_rate):
        dL_dF_params = np.zeros(self.conv_filter.shape)
        dL_dbias = np.zeros(self.bias.shape) # Initialize gradient for bias

        for image_patch, i, j in self.image_region(self.padded_image):
            for k in range(self.num_filters):
                dL_dF_params[k] += image_patch * dL_dout[i, j, k]
                dL_dbias[k] += dL_dout[i, j, k]

        self.conv_filter -= learning_rate * dL_dF_params
        self.bias -= learning_rate * dL_dbias # Update bias

        return dL_dF_params, dL_dbias
```

```
#define ReLU:
class ReLUlayer:
    def __init__(self):
        self.input = None

    def forward(self, input_data):
        self.input = input_data
        return np.maximum(0, input_data)

    def backward(self, dL_dout):
        dL_din = dL_dout * (self.input > 0)
        return dL_din
```

```
#define Maxpool:
class Maxpool:
    def __init__(self, filter_size=2):
        self.filter_size = filter_size

    def image_region(self, image):
        new_height = image.shape[0] // self.filter_size
        new_width = image.shape[1] // self.filter_size
        self.image = image
        for i in range(new_height):
            for j in range(new_width):
                image_patch = image[(i*self.filter_size):(i*self.filter_size+ self.filter_size), (j*self.filter_size):(j*self.filter_size+self.filter_size)]
                yield image_patch, i, j

    def forward(self, image):
        height, width, num_filters = image.shape
        output = np.zeros((height // self.filter_size, width // self.filter_size, num_filters))
        for image_patch, i, j in self.image_region(image):
            output[i, j] = np.amax(image_patch, axis=(0, 1))
        return output

    def backward(self, dL_dout):
        dL_dmaxpool = np.zeros(self.image.shape)
        for image_patch, i, j in self.image_region(self.image):
            height, width, num_filters = image_patch.shape
            maximum_val = np.amax(image_patch, axis=(0, 1))
            for il in range(height):
                for jl in range(width):
                    for kl in range(num_filters):
                        if image_patch[il, jl, kl] == maximum_val[kl]:
                            dL_dmaxpool[i*self.filter_size + il, j*self.filter_size + jl, kl] = dL_dout[i, j, kl]
        return dL_dmaxpool
```



```

#define Softmax
class Softmax:
    def __init__(self, input_node, softmax_node):
        self.weights=np.random.randn(input_node, softmax_node)*(np.sqrt(2.0 /input_node))
        self.biases=np.zeros(softmax_node)
    def forward(self, image):
        self.orig_im_shape=image.shape
        flattened_value= image.flatten()
        self.modified_input=flattened_value
        dense_val=np.dot(flattened_value, self.weights)+ self.biases
        self.out=dense_val
        exp_out=np.exp(dense_val)
        return exp_out/np.sum(exp_out, axis=0)
    def backward(self, dL_dout, learning_rate):
        for i, grad in enumerate(dL_dout):
            if grad ==0:
                continue
            transformation_eq=np.exp(self.out)
            S_total=np.sum(transformation_eq)
            dy_dx=transformation_eq[i]*transformation_eq/(S_total**2)
            dy_dx[i]= transformation_eq[i]*(S_total-transformation_eq[i])/(S_total**2)
            dx_dw=self.modified_input
            dx_db=1
            dx_d_inp=self.weights
            dL_dx=grad*dy_dx
            dL_dw=dx_dw[np.newaxis].T @ dL_dx[np.newaxis]
            dL_db=dL_dx*dx_db
            dL_d_inp=dx_d_inp @ dL_dx
            self.weights-=dL_dw*learning_rate
            self.biases-=dL_db*learning_rate
            return dL_d_inp.reshape(self.orig_im_shape)

```

Then layers in the convolution process are set with hyperparameters

```

conv=ConvLayer(8,3, padding=0)
pool=Maxpool(2)
softmax=Softmax(1352, 10)
relu=ReluLayer()

```

One of the problem during model training is overfitting condition, where the model learns the train data too much that captures all noises and random fluctuations in the data than patterns. To prevent this problem, data augmentation has been applied as following:

```

def apply_random_data_augmentation(image):
    # Randomly select one augmentation technique to apply
    augmentation_functions = [flip_horizontal, rotate_image, translate_image, add_noise]
    chosen_augmentation = np.random.choice(augmentation_functions)

    # Apply the selected augmentation technique
    image = chosen_augmentation(image)

    # Ensure the output image size is 28x28
    image = resize_to_28x28(image)

    return image

# Function to flip the image horizontally
def flip_horizontal(image):
    return cv2.flip(image, 1)

# Function to rotate the image by a random angle between -10 and 10 degrees
def rotate_image(image):
    angle = np.random.uniform(-10, 10)
    return cv2.warpAffine(image, cv2.getRotationMatrix2D((image.shape[1] / 2, image.shape[0] / 2), angle, 1.0), (image.shape[1], image.shape[0]))

# Function to translate the image by random amounts between -5 and 5 pixels in both directions
def translate_image(image):
    dx, dy = np.random.randint(-5, 5, size=2)
    return cv2.warpAffine(image, np.float32([1, 0, dx], [0, 1, dy]), (image.shape[1], image.shape[0]))

# Function to resize the image to 28x28 pixels
def resize_to_28x28(image):
    return cv2.resize(image, (28, 28))

# Function to add random Gaussian noise
def add_noise(image):
    noise_level = np.random.uniform(0, 10)
    noise = np.random.normal(0, noise_level, image.shape).astype(image.dtype) # Match data type
    noisy_image = cv2.add(image, noise)
    return noisy_image

```

Then CNN forwarding and backpropagation are defined

```
def cnn_forward(image, label):
    out_p=conv.forward((image/225))
    out_p = relu.forward(out_p)
    out_p=pool.forward(out_p)
    out_p= softmax.forward(out_p)
    epsilon = 1e-7
    out_p=out_p + epsilon#adding epsilon to prevent
    cross_ent_loss=-np.log(out_p[label])
    accuracy_eval=1 if np.argmax(out_p)==label else 0
```

```
def training_cnn(image, label, learning_rate=.005):
    out, loss, acc = cnn_forward(image, label)
    gradient=np.zeros(10)
    gradient[label]=-1/out[label]
    grad_back=softmax.backward(gradient, learning_rate)
    grad_back=pool.backward(grad_back)
    grad_back = relu.backward(grad_back)
    grad_back=conv.backward(grad_back, learning_rate)
    return loss, acc
```

Then the training loop is implemented, in this loop, training data is shuffled, and then is splitted into batches of 100 images. For each epoch, the train loss is calculated after every batch. After one epoch is completed, the model is tested with the test data. When the training is finished, the model is validated with the validation data. The final output of the model is the predicted label, which is the class with the highest probability in the predicted probability distribution after the softmax. These values are added to a list for further evaluation.

```
from sklearn.metrics import confusion_matrix, accuracy_score

# Lists to store train and test losses after each epoch
train_losses_epoch = []
test_losses = []
test_accuracies = []
true_labels = []
predicted_labels = []

# Training loop
for epoch in range(20):
    print('Epoch %d-->' % (epoch + 1))
    shuffle_data = np.random.permutation(len(X_train_grayscale))
    X_train_grayscale = X_train_grayscale[shuffle_data]
    y_train = y_train[shuffle_data]
    train_loss = 0
    num_correct = 0
    train_losses_batch=[]
    for i, (im, label) in enumerate(zip(X_train_grayscale, y_train)):
        augmented_image = apply_random_data_augmentation(im)
        loss, accu = training_cnn(augmented_image, label)
        train_loss += loss
        num_correct += accu
        if i % 100 == 0:
            train_loss /= 100 #calculate average train loss each batch
            train_losses_batch.append(train_loss) # Collect train loss after each batch
            print('%d steps out of 100 steps: Average train loss %.3f and Accuracy: %d%%' % (i + 1, train_loss, num_correct))
            train_loss = 0
            num_correct = 0
    train_losses_epoch.append(sum(train_losses_batch)/len(train_losses_batch))

    print('Epoch %d: Train Loss %.3f' % (epoch + 1, train_losses_epoch[-1])) # Print train loss after each epoch

    # Test loop (calculate test loss and accuracy)
    test_loss = 0
    num_test_correct = 0

    for i, (im, label) in enumerate(zip(X_test_grayscale, y_test)):
        _, loss, accu = cnn_forward(im, label)
        test_loss += loss
        num_test_correct += accu
    test_loss.append(test_loss / len(X_test_grayscale)) # Collect test loss after each epoch
    test_accuracy = num_test_correct / len(X_test_grayscale)
    test_accuracies.append(test_accuracy) # Collect test accuracy after each epoch
    print('Epoch %d: Test Loss %.3f, Test Accuracy %.2f%%' % (epoch + 1, test_losses[-1], test_accuracy * 100))

# Plot train and test losses for each epoch
for i, (im, label) in enumerate(zip(X_val_grayscale, y_val)):
    _, loss, accu = cnn_forward(im, label)
    true_labels.append(label)
    predicted_labels.append(np.argmax(_))

# Compute and plot the confusion matrix
```

In the model, there are hyperparameters that could be tuned for better model's performance, a combination of kernels number and learning rate have been tested to illustrate how the hyperparameters affect the result. Further analysis in the next part.

5. Result evaluation

The model has been tested for 20 epochs with following hyperparameters:

- 8 kernels and learning rate 0.001
- 16 kernels and learning rate 0.001
- 24 kernels and learning rate 0.001
- 24 kernels and learning rate 0.005
- 24 kernels and learning rate 0.01

To plot the graph for each combination, following codes are applied:

```

1 plt.figure(figsize=(10, 5))
2 plt.plot(range(20), test accuracies, label='Test Accuracy', marker='o')
3 plt.xlabel('Epoch')
4 plt.ylabel('Accuracy')
5 plt.legend()
6 plt.title('Accuracy for Each Epoch with 24 kernels and lr=0.005')
7 plt.xticks(range(1,20))
8 plt.show()
9
10 plt.figure(figsize=(10, 5))
11 plt.plot(range(20), train_losses_epoch, label='Train Loss', marker='o')
12 plt.plot(range(20), test_losses, label='Test Loss', marker='x')
13 plt.xlabel('Epoch')
14 plt.ylabel('Loss')
15 plt.legend()
16 plt.title('Train and Test Losses for Each Epoch with 24 kernels and lr=0.005')
17 plt.xticks(range(1,20))
18 plt.show()
19
20
21 # Compute and plot the confusion matrix
22 confusion = confusion_matrix(true_labels, predicted_labels)
23 plt.figure(figsize=(8, 6))
24 sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues", xticklabels=True, yticklabels=True)
25 plt.xlabel('Predicted Labels')
26 plt.ylabel('True Labels')
27 plt.title('Confusion Matrix with 24 kernels and lr=0.005')
28 plt.show()

```

```

1 from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
2
3 accuracy = accuracy_score(true_labels, predicted_labels)
4 report = classification_report(true_labels, predicted_labels)
5 cm = confusion_matrix(true_labels, predicted_labels)
6
7 print(f"Accuracy: {accuracy}")
8 print(f"Classification Report:\n{report}")
9 print(f"Confusion Matrix:\n{cm}")

```

To evaluate the result, precision, recall, and F1-score are applied.

The precision indicates how many predicted positive instances are actually positive instances with following equation:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

High precision means that when model predicts a positive class, it is likely to be correct.

The recall, or the sensitivity, measures how many of the true positive instances are correctly predicted with following equation:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False negatives}}$$

High recall means that the model is effective at capturing most of the positive instances.

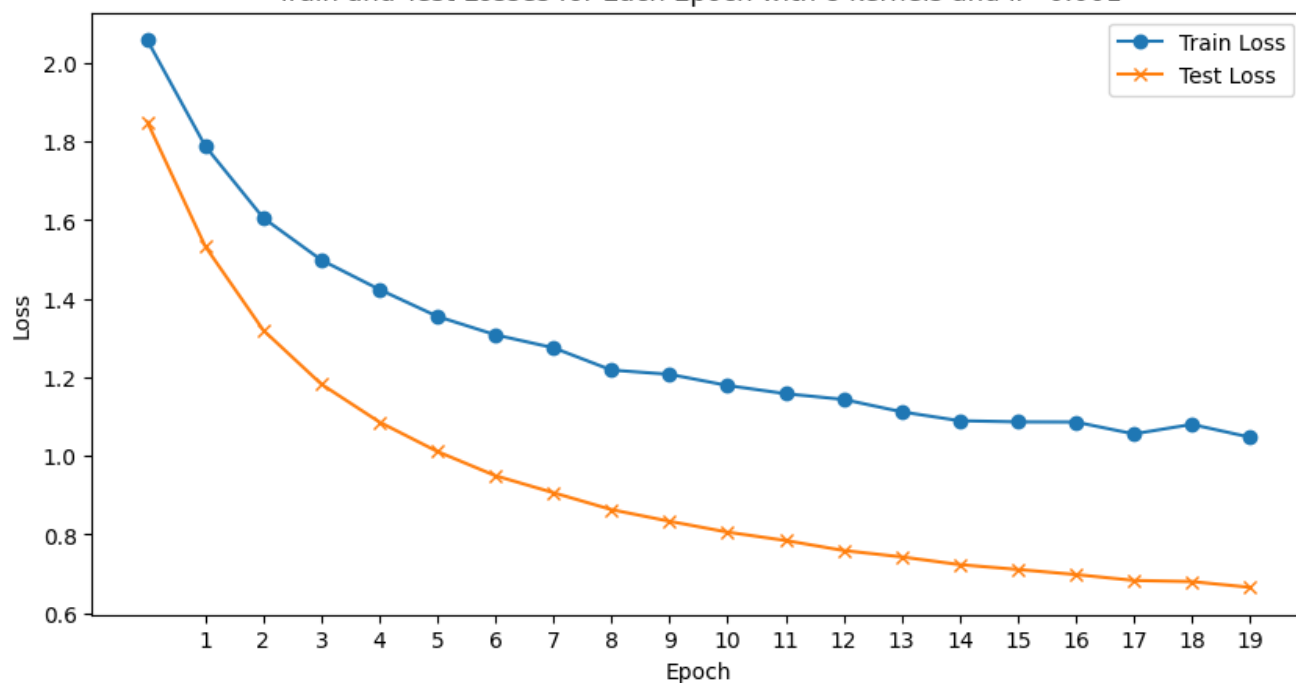
The F1-Score is the harmonic mean of precision and recall and provides a balance between the two. Normally, the F1-Score is used for imbalanced dataset. Following is the equation of F1-Score:

$$\text{F1-Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

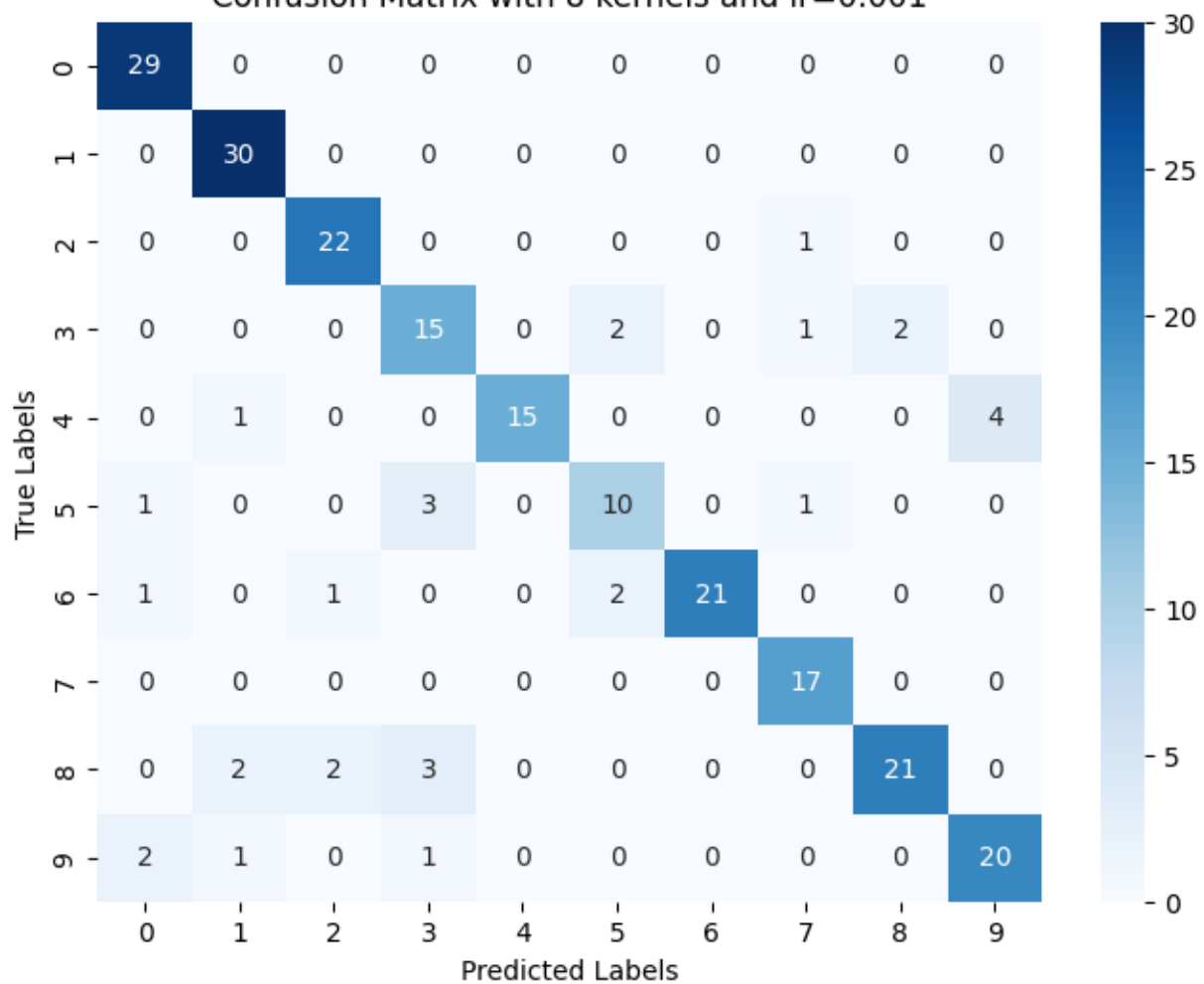
Despite these metrics are used in binary classification problems, they can be adapted for multi-class classification problems by considering each class separately.

The results are shown as below:

Train and Test Losses for Each Epoch with 8 kernels and lr=0.001



Confusion Matrix with 8 kernels and lr=0.001



```

Validation Accuracy 8 kernels lr 0.001: |.8658008658008658
Classification Report:

```

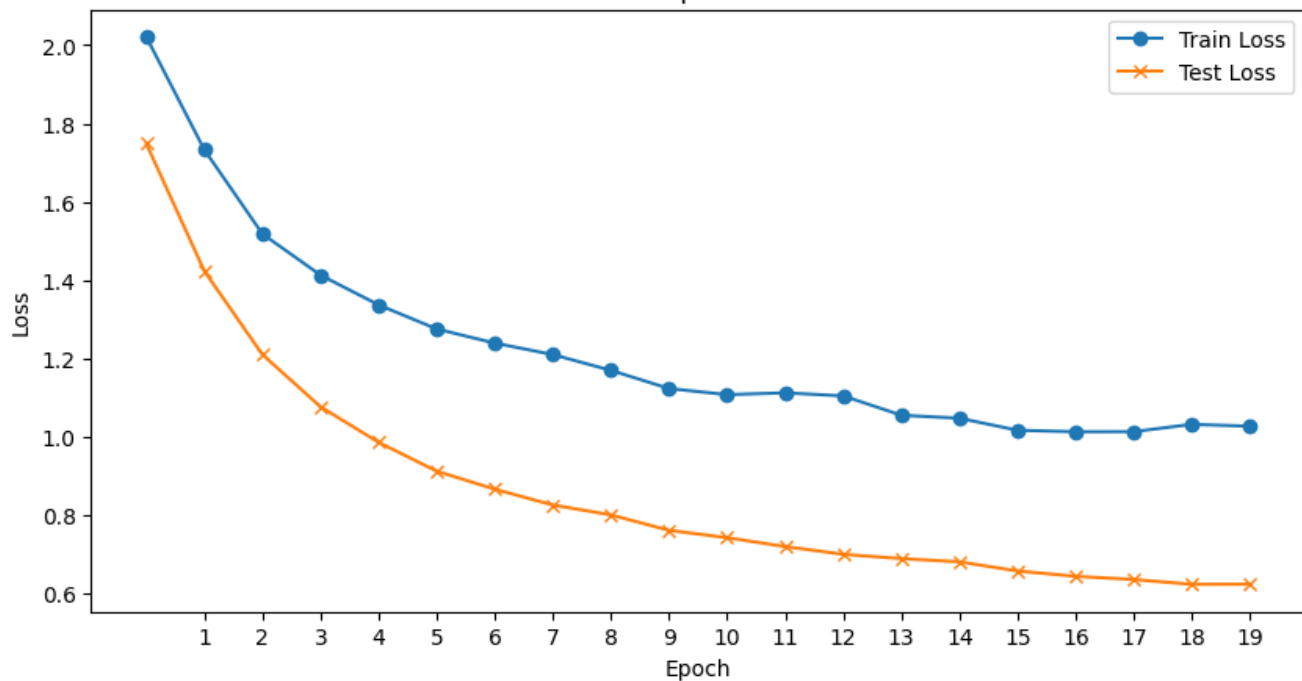
	precision	recall	f1-score	support
0	0.88	1.00	0.94	29
1	0.88	1.00	0.94	30
2	0.88	0.96	0.92	23
3	0.68	0.75	0.71	20
4	1.00	0.75	0.86	20
5	0.71	0.67	0.69	15
6	1.00	0.84	0.91	25
7	0.85	1.00	0.92	17
8	0.91	0.75	0.82	28
9	0.83	0.83	0.83	24
accuracy			0.87	231
macro avg	0.86	0.85	0.85	231
weighted avg	0.87	0.87	0.86	231

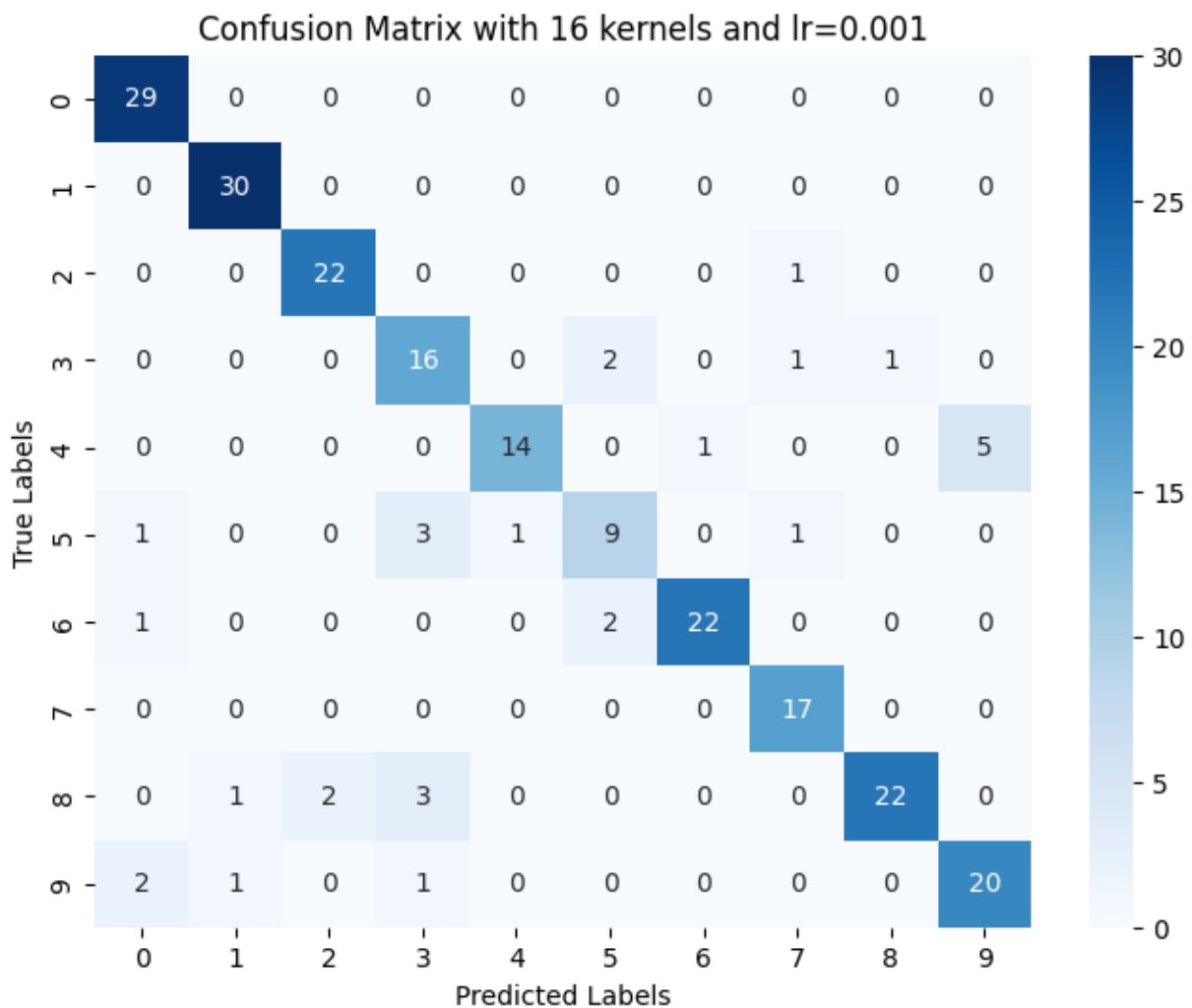
```

Confusion Matrix:
[[29  0  0  0  0  0  0  0  0  0]
 [ 0 30  0  0  0  0  0  0  0  0]
 [ 0  0 22  0  0  0  0  1  0  0]
 [ 0  0  0 15  0  2  0  1  2  0]
 [ 0  1  0  0 15  0  0  0  0  4]
 [ 1  0  0  3  0 10  0  1  0  0]
 [ 1  0  1  0  0  2 21  0  0  0]
 [ 0  0  0  0  0  0  0 17  0  0]
 [ 0  2  2  3  0  0  0  0 21  0]
 [ 2  1  0  1  0  0  0  0  0 20]]

```

Train and Test Losses for Each Epoch with 16 kernels and lr=0.001





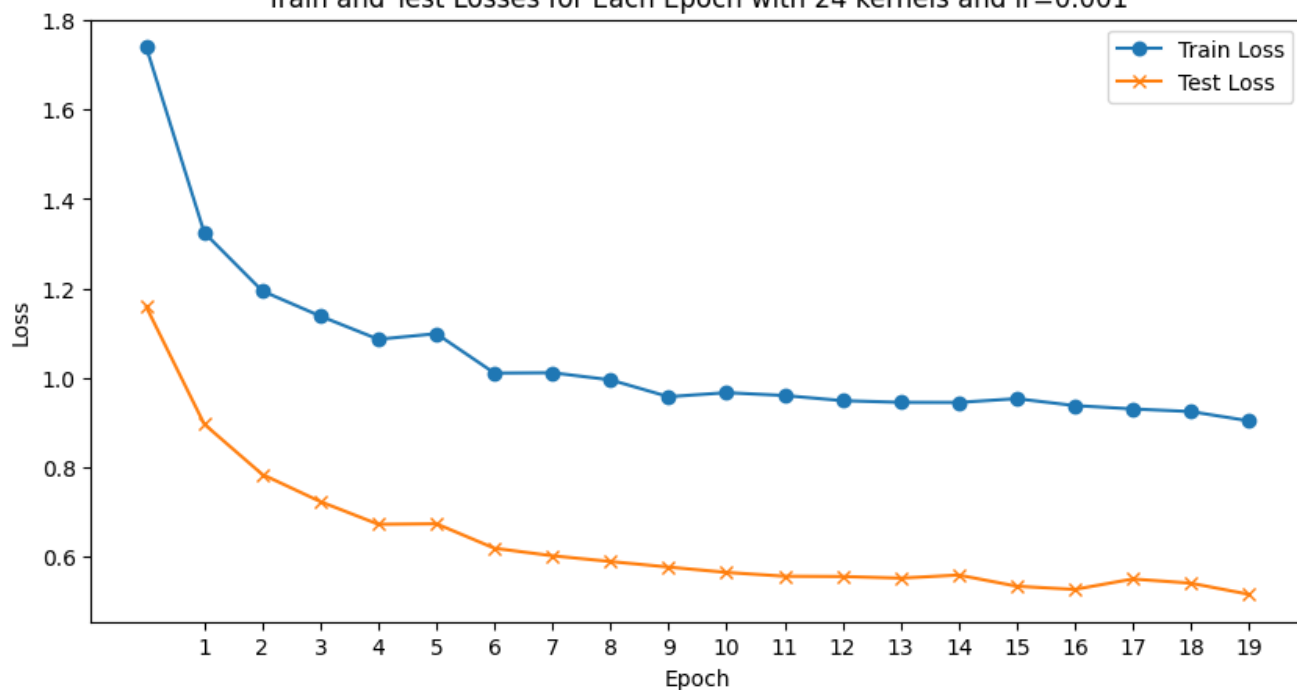
```
Validation Accuracy: 0.8701298701298701
Classification Report:
              precision    recall  f1-score   support

     0:       0.88         1.00         0.94         29
     1:       0.94         1.00         0.97         30
     2:       0.92         0.96         0.94         23
     3:       0.70         0.80         0.74         20
     4:       0.93         0.70         0.80         20
     5:       0.69         0.60         0.64         15
     6:       0.96         0.88         0.92         25
     7:       0.85         1.00         0.92         17
     8:       0.96         0.79         0.86         28
     9:       0.80         0.83         0.82         24

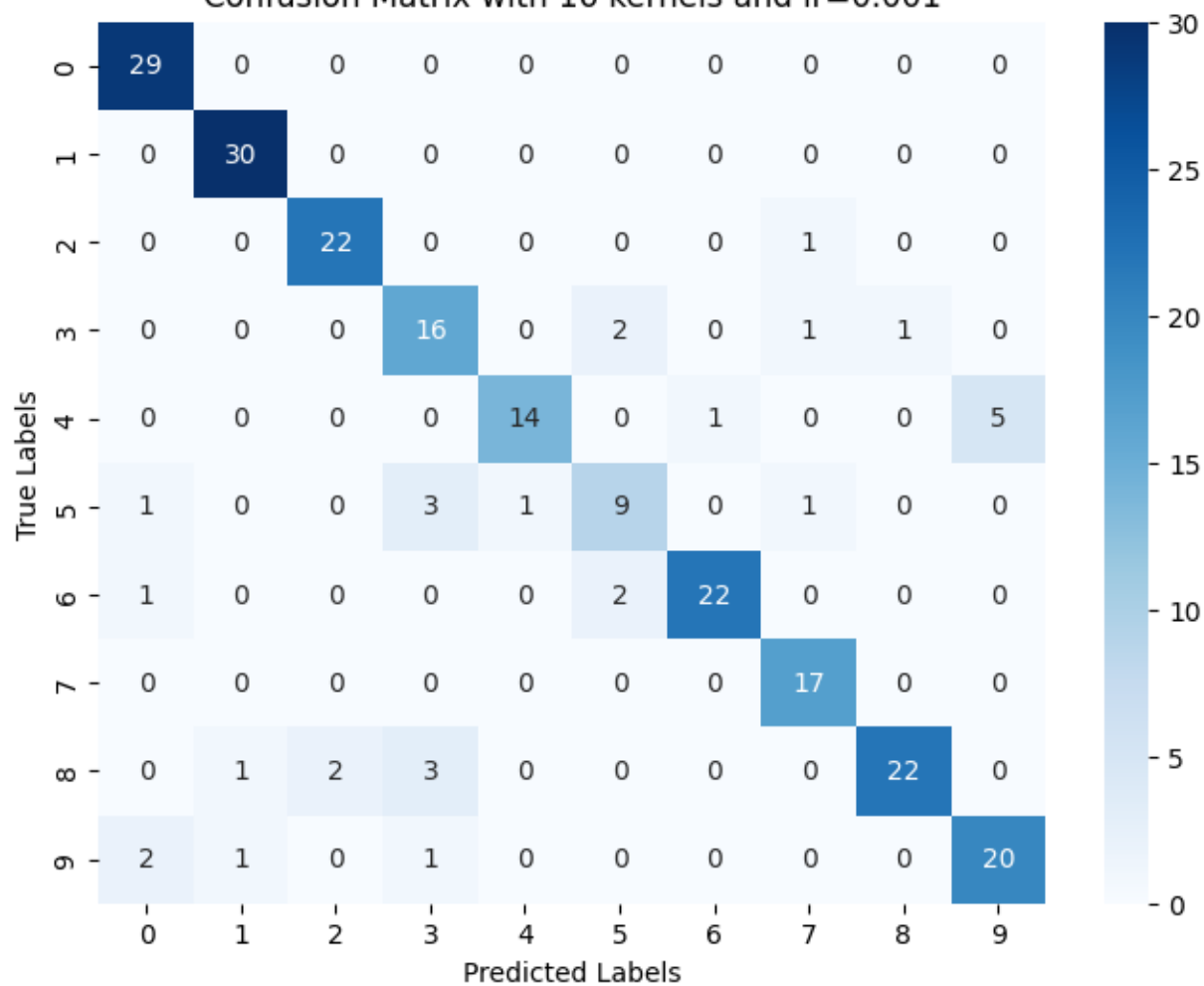
 accuracy          0.87
 macro avg         0.86         0.86         0.85         231
 weighted avg      0.87         0.87         0.87         231
```

```
Confusion Matrix:
[[29  0  0  0  0  0  0  0  0  0]
 [ 0 30  0  0  0  0  0  0  0  0]
 [ 0  0 22  0  0  0  0  1  0  0]
 [ 0  0  0 16  0  2  0  1  1  0]
 [ 0  0  0  0 14  0  1  0  0  5]
 [ 1  0  0  3  1  9  0  1  0  0]
 [ 1  0  0  0  0  2 22  0  0  0]
 [ 0  0  0  0  0  0  0 17  0  0]
 [ 0  1  2  3  0  0  0  0 22  0]
 [ 2  1  0  1  0  0  0  0  0 20]]
```

Train and Test Losses for Each Epoch with 24 kernels and lr=0.001



Confusion Matrix with 16 kernels and lr=0.001



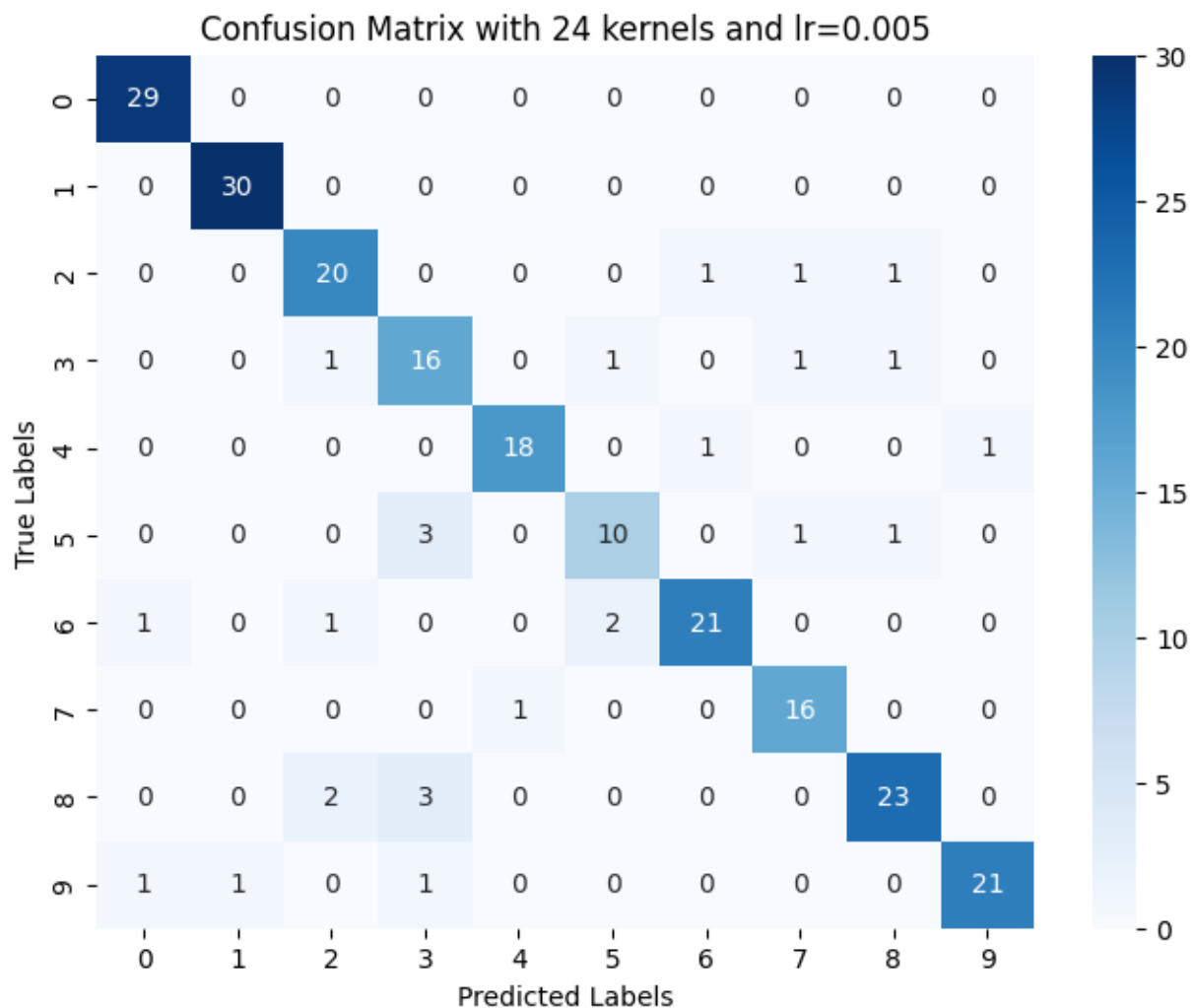
Accuracy: 0.8831168831168831

Classification Report:

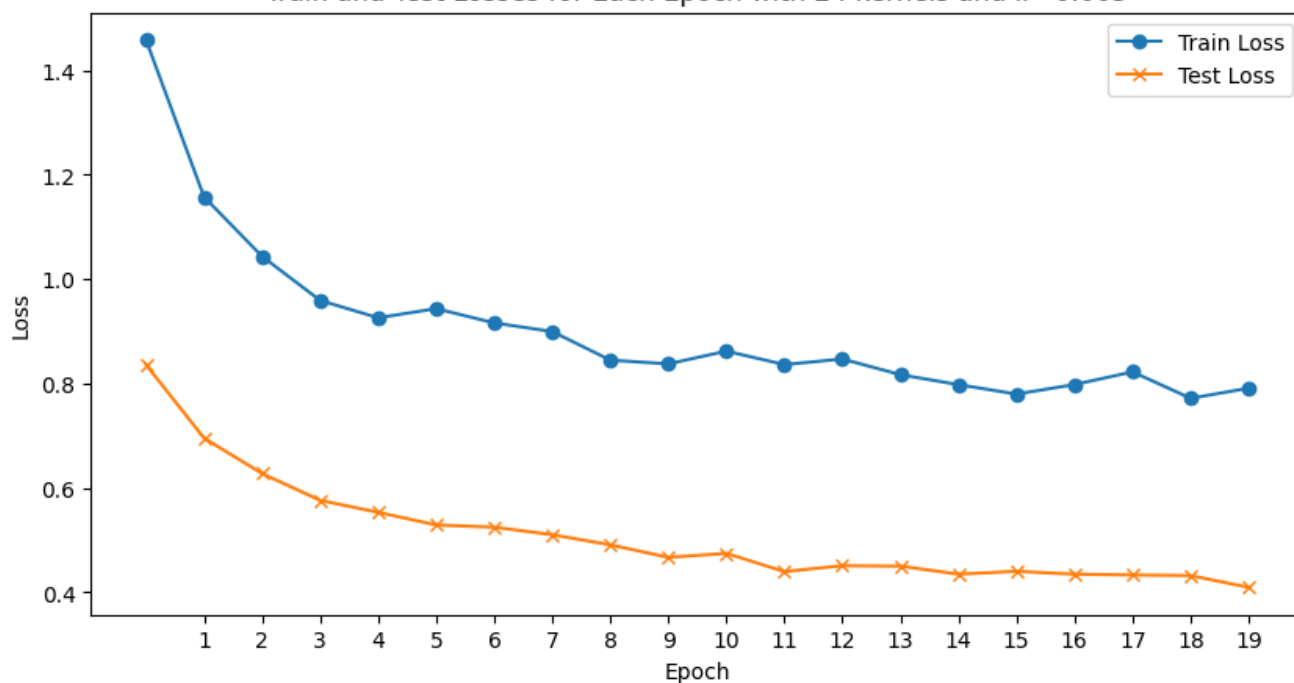
	precision	recall	f1-score	support
0	0.94	1.00	0.97	29
1	0.91	1.00	0.95	30
2	0.85	0.96	0.90	23
3	0.75	0.75	0.75	20
4	0.84	0.80	0.82	20
5	0.73	0.73	0.73	15
6	1.00	0.84	0.91	25
7	0.84	0.94	0.89	17
8	0.96	0.86	0.91	28
9	0.91	0.83	0.87	24
accuracy			0.88	231
macro avg	0.87	0.87	0.87	231
weighted avg	0.89	0.88	0.88	231

Confusion Matrix:

```
[[29  0  0  0  0  0  0  0  0  0]
 [ 0 30  0  0  0  0  0  0  0  0]
 [ 0  0 22  0  0  0  0  1  0  0]
 [ 0  0  1 15  0  2  0  1  1  0]
 [ 0  2  0  0 16  0  0  0  0  2]
 [ 0  0  0  2  1 11  0  1  0  0]
 [ 1  0  1  0  0  2 21  0  0  0]
 [ 0  0  0  0  1  0  0 16  0  0]
 [ 0  0  2  2  0  0  0  0 24  0]
 [ 1  1  0  1  1  0  0  0  0 20]]
```



Train and Test Losses for Each Epoch with 24 kernels and lr=0.005



```

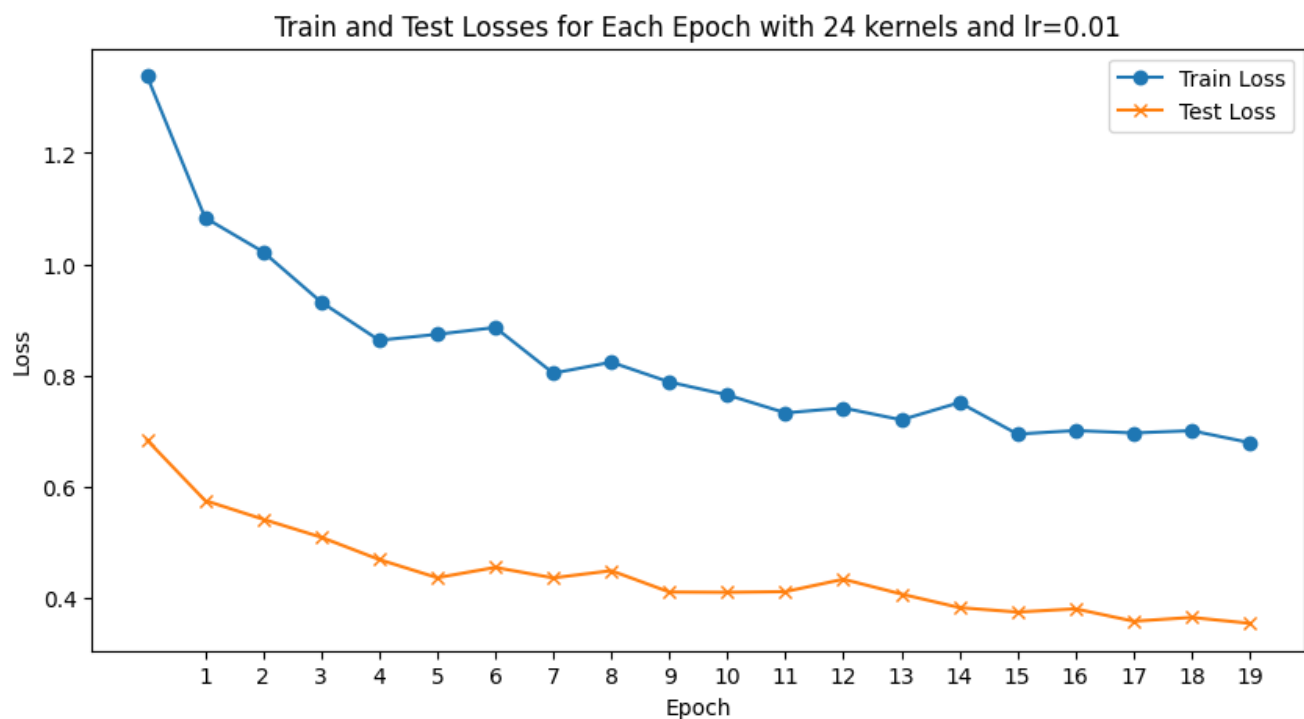
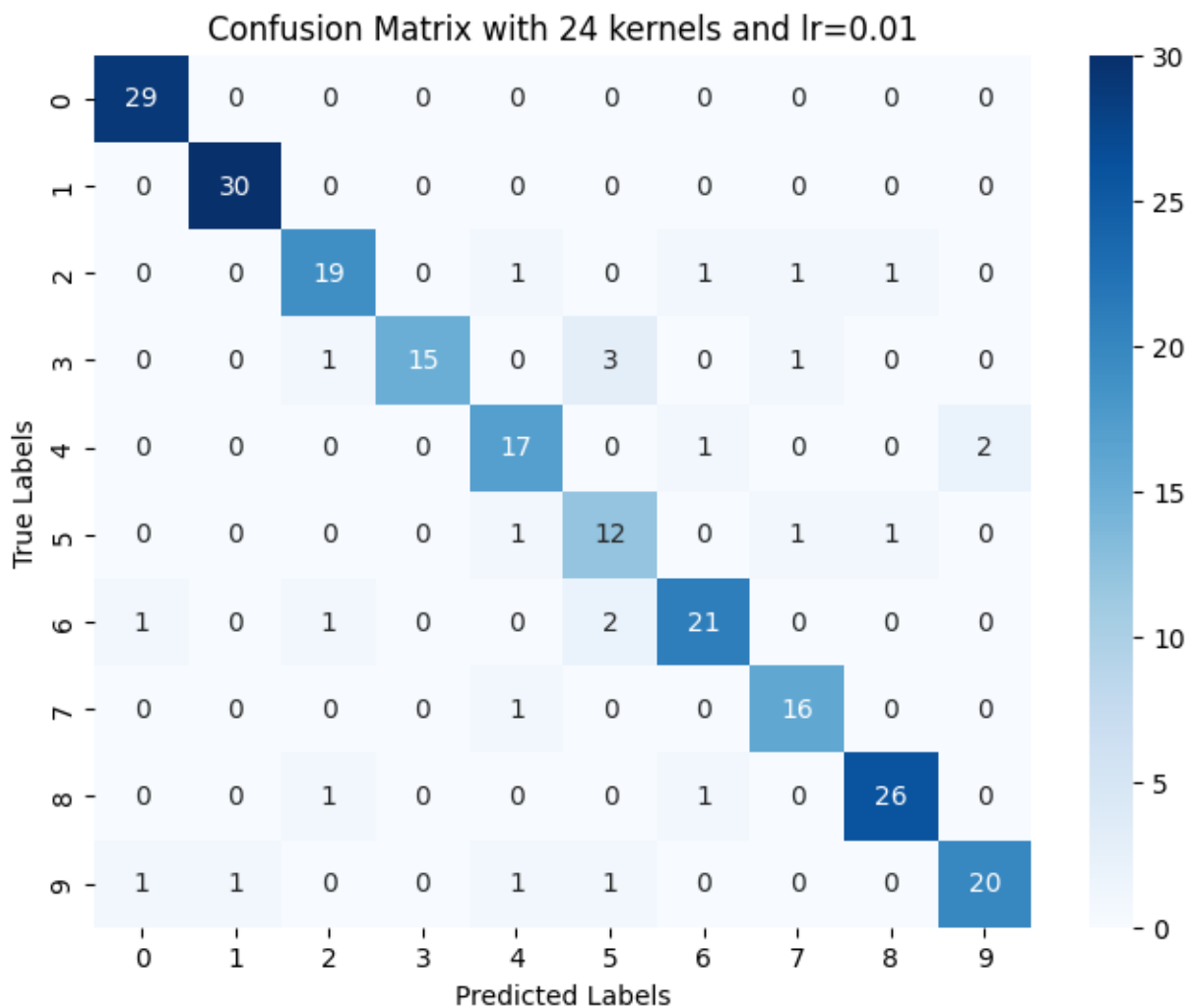
Accuracy: 0.8831168831168831
Classification Report:
              precision    recall  f1-score   support

     0       0.94         1.00         0.97         29
     1       0.97         1.00         0.98         30
     2       0.83         0.87         0.85         23
     3       0.70         0.80         0.74         20
     4       0.95         0.90         0.92         20
     5       0.77         0.67         0.71         15
     6       0.91         0.84         0.87         25
     7       0.84         0.94         0.89         17
     8       0.88         0.82         0.85         28
     9       0.95         0.88         0.91         24

   accuracy          0.88         0.88         0.88         231
  macro avg          0.87         0.87         0.87         231
 weighted avg          0.89         0.88         0.88         231

Confusion Matrix:
[[29  0  0  0  0  0  0  0  0]
 [ 0 30  0  0  0  0  0  0  0]
 [ 0  0 20  0  0  0  1  1  1]
 [ 0  0  1 16  0  1  0  1  1]
 [ 0  0  0  0 18  0  1  0  1]
 [ 0  0  0  3  0 10  0  1  1]
 [ 1  0  1  0  0  2 21  0  0]
 [ 0  0  0  0  1  0  0 16  0]
 [ 0  0  2  3  0  0  0  0 23]
 [ 1  1  0  1  0  0  0  0  21]]

```



```

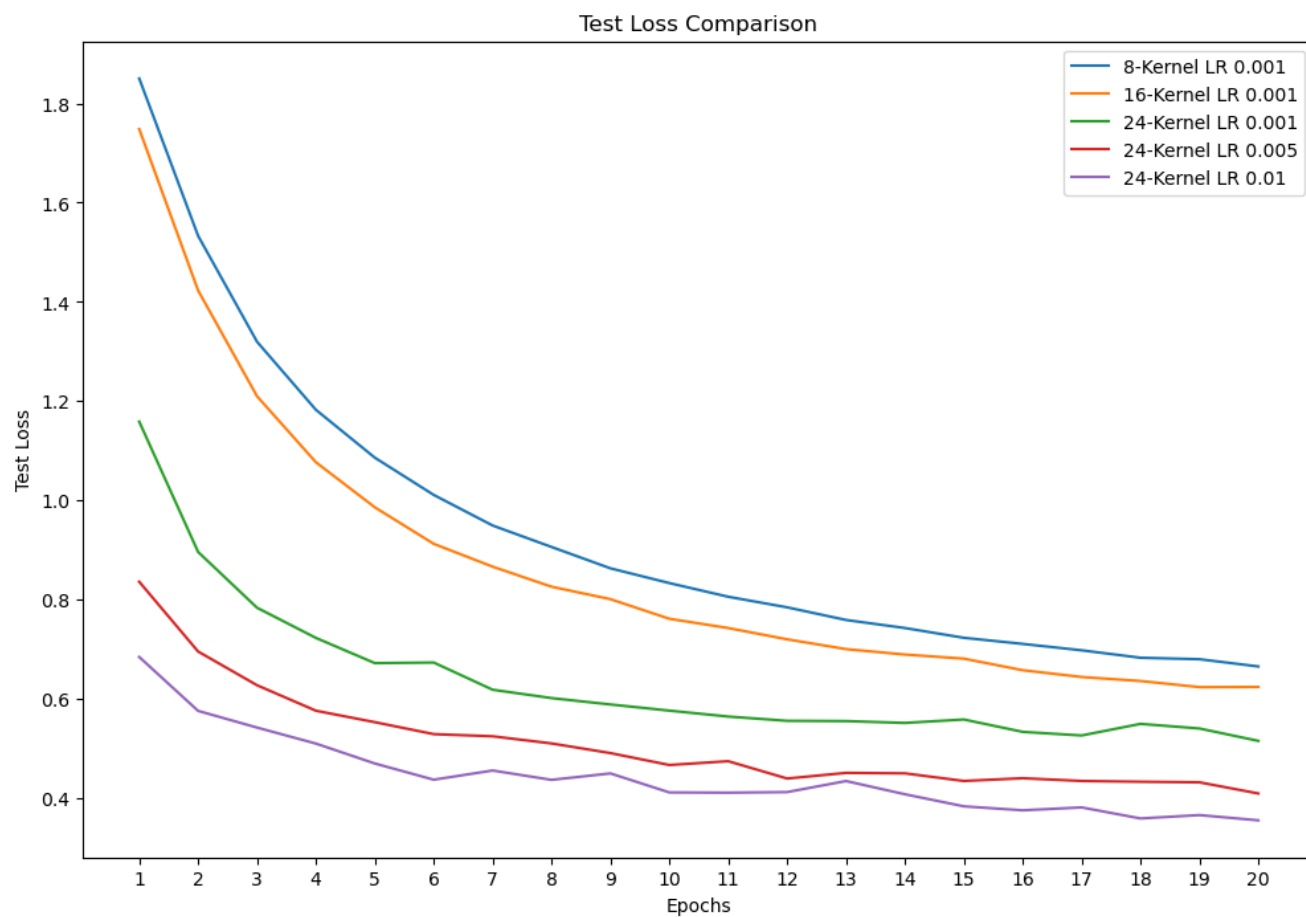
Val_Accuracy: 0.8874458874458875
Classification Report:
              precision    recall  f1-score   support

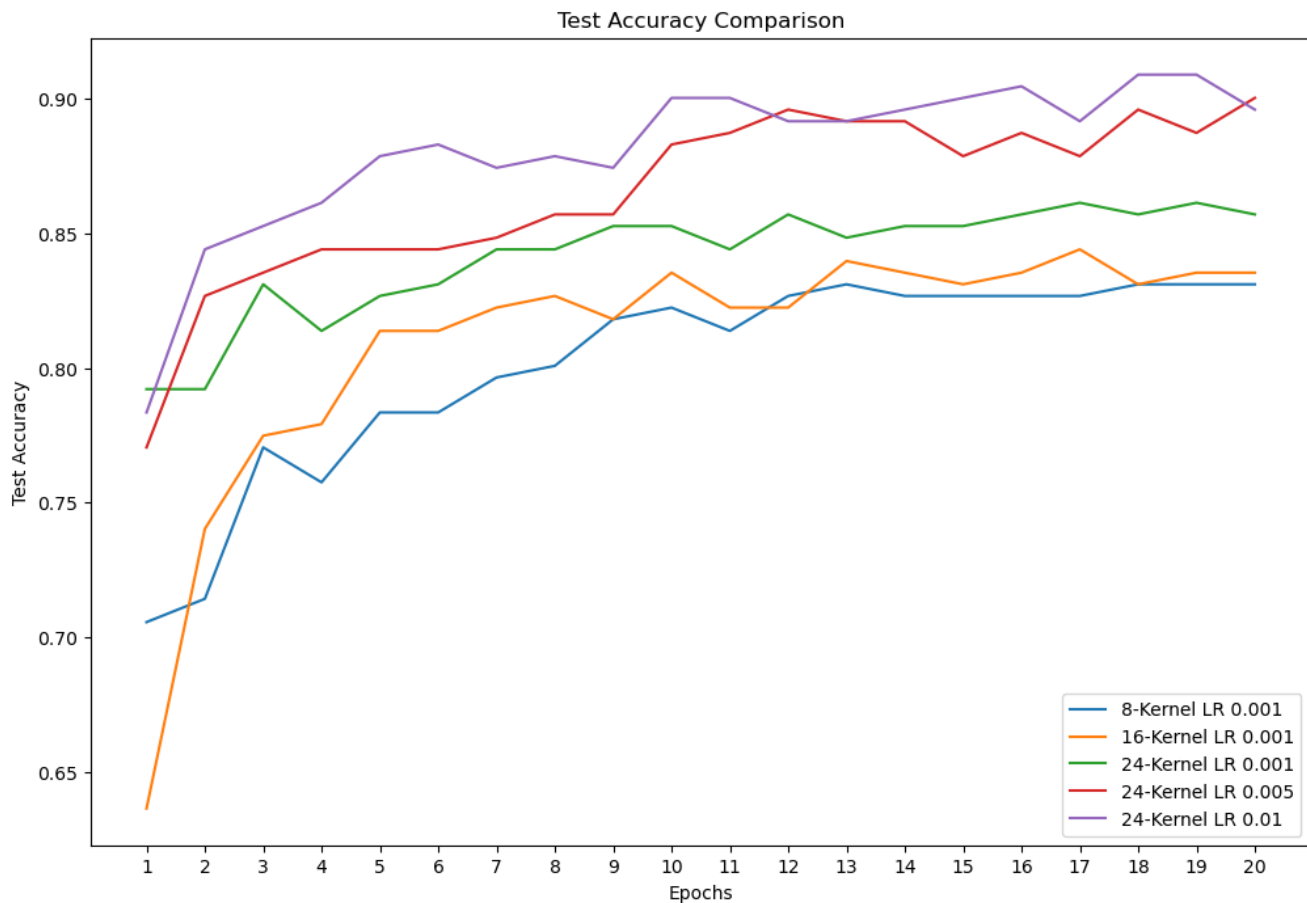
     0           0.94         1.00         0.97         29
     1           0.97         1.00         0.98         30
     2           0.86         0.83         0.84         23
     3           1.00         0.75         0.86         20
     4           0.81         0.85         0.83         20
     5           0.67         0.80         0.73         15
     6           0.88         0.84         0.86         25
     7           0.84         0.94         0.89         17
     8           0.93         0.93         0.93         28
     9           0.91         0.83         0.87         24

 accuracy          0.89
 macro avg         0.88         0.88         0.88         231
 weighted avg      0.89         0.89         0.89         231

Confusion Matrix:
[[29  0  0  0  0  0  0  0  0  0]
 [ 0 30  0  0  0  0  0  0  0  0]
 [ 0  0 19  0  1  0  1  1  1  0]
 [ 0  0  1 15  0  3  0  1  0  0]
 [ 0  0  0  0 17  0  1  0  0  2]
 [ 0  0  0  0  1 12  0  1  1  0]
 [ 1  0  1  0  0  2 21  0  0  0]
 [ 0  0  0  0  1  0  0 16  0  0]
 [ 0  0  1  0  0  0  1  0 26  0]
 [ 1  1  0  0  1  1  0  0  0 20]]

```





With the comparison, it can be seen that with different hyperparameters, the model outcomes vary accordingly. Higher kernel number and larger learning rate reduces the loss and increase the accuracy of the model. The hyperparameters combination is kernel number 24 and learning rate 0.01, making the model converges faster.

6. Conclusion and reflection

With this journal and the corresponding framework, CNN model is proven to be efficient in multiclass image classification by mimicking the way image is transferred and projected in human being's brain. This journal also confirms the ability of deep-learning neural networks with activation functions in tasks relating to high dimensional, non-linear data like images. The CNN model in this journal is still quite simple when the input data is only one channel image, meaning it hasn't reflected all the aspects of a practical RGB image with up to 3 channels. Further works and implementations are required to handle more complex data with optimised learning process.

7. Reference and AI assistance

Bigun, J. (2006). Vision with Direction: A Systematic Introduction to Image Processing and Computer Vision. Springer Nature. <https://doi.org/10.1007/b138918>.

Kalinin, S. (2021, September 20). CNN in Python, part 1. Defining the main parameters of the model. *Medium*. <https://medium.com/@skalinin/cnn-in-python-part-1-defining-the-main-parameters-of-the-model-288e52485290>.

Khan, S. (2018). *A Guide to Convolutional Neural Networks for Computer Vision*(Vol. 15). Springer Nature.

Krizhevsky, A., Sutskever, I., & Hinton, G. (2017). ImageNet classification with deep convolutional neural networks. In Communications of the ACM (Vol. 60, Issue 6, pp. 84–90). ACM. <https://doi.org/10.1145/3065386>.

O'Shea, K., & Nash, R. (2015). *An Introduction to Convolutional Neural Networks*. <https://doi.org/10.48550/arxiv.1511.08458>.



the nature of image for convolution operation



In the context of convolutional neural networks (CNNs), the "Image" refers to the input data that is processed by the network. This input data can be an actual image, but it can also be any two-dimensional data, such as a matrix, where the convolution operation is applied. The nature of the image for a convolution operation is characterized by the following aspects:



1. **Spatial Structure:** The input data is usually organized in a two-dimensional grid or matrix. In the case of images, this grid corresponds to the rows and columns of pixels, and for other data, it represents a spatial arrangement of elements.
2. **Channels:** Images often have multiple color channels (e.g., red, green, blue in RGB images), or in the case of CNNs for different tasks, they may have multiple feature channels. The data may be organized as a three-dimensional tensor, with one dimension for height, one for width, and one for the number of channels.
3. **Pixel Values:** Each element in the input grid represents a pixel or a feature value. In the case of images, pixel values are typically intensity values, and they can be grayscale or have different color channels with varying intensities.
4. **Local Patterns:** Convolution operations work on local patterns within the input data. A convolutional kernel (filter) is used to detect specific features or patterns within a small receptive field of the input.
5. **Translation Invariance:** CNNs leverage convolution operations to detect features in a translation-invariant manner. This means that a feature can be detected regardless of its exact location in the input.
6. **Feature Hierarchy:** In deep CNNs, convolutional layers are often stacked, allowing the network to learn increasingly complex features from simple edges and textures to more abstract high-level features, such as object parts and objects themselves.

The convolution operation involves sliding a convolutional kernel over the input data and computing a weighted sum of the elements in the receptive field, producing an output feature map. This process is essential for tasks like image recognition, where the network learns to recognize patterns and objects within images. However, CNNs can also be applied to various other data types beyond images, such as speech, text, and more, as long as the data has a structured spatial nature that can benefit from the convolution operation.

AN

why softmax is use for multiclass classification



Softmax is commonly used for multiclass classification because it provides a way to convert a vector of raw scores or logits into a probability distribution over multiple classes. This is particularly useful in scenarios where you want to assign an input to one of several possible classes, which is a fundamental task in machine learning and deep learning. Here are some key reasons why the softmax function is used for multiclass classification:

1. **Probability Interpretation:** Softmax converts raw scores into probabilities. Each element in the output vector represents the probability that the input belongs to a particular class. These probabilities sum to 1, which is a fundamental requirement for multiclass classification.
2. **Classification Decision:** Once the probabilities are obtained using softmax, you can make a classification decision by selecting the class with the highest probability as the predicted class for the input.
3. **Gradient Optimization:** Softmax is differentiable, making it suitable for training with gradient-based optimization algorithms like stochastic gradient descent (SGD). This enables the use of backpropagation to update the model's parameters during training.
4. **Multi-Class Problems:** Softmax is well-suited for problems with more than two classes. It can handle tasks where you have to classify inputs into three or more categories, making it a versatile choice for a wide range of classification problems.
5. **Comparison of Scores:** Softmax normalizes the raw scores by exponentiating them and dividing by the sum of exponentiated scores. This amplifies the difference between high-scoring and low-scoring classes, making it easier for the model to distinguish between different classes.

The softmax function is defined as follows for a vector of raw scores (logits) z :

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

Where:

- z_i is the raw score for class i .
- K is the total number of classes.
- e is the base of the natural logarithm.

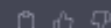
The softmax function transforms the raw scores into a probability distribution where each element represents the likelihood of the input belonging to a specific class. This makes it an essential component in the output layer of neural networks designed for multiclass classification tasks, such as image classification, natural language processing, and more.

AN

why cross entropy is used as loss for multiclass classification



Cross-entropy is commonly used as a loss function for multiclass classification tasks because it measures the dissimilarity between the predicted class probabilities and the actual class labels, making it a suitable choice for training and evaluating classification models. Here are several reasons why cross-entropy is used for multiclass classification:



1. **Measure of Discrepancy:** Cross-entropy quantifies the dissimilarity between the predicted class probabilities and the true class labels. When the predicted probabilities match the actual labels, the cross-entropy loss is low. As the predictions deviate from the true labels, the loss increases. This provides a clear and interpretable measure of how well the model is performing.
2. **Gradient Descent Optimization:** Cross-entropy is differentiable, which is crucial for gradient-based optimization methods like stochastic gradient descent (SGD). The gradients of the cross-entropy loss with respect to model parameters can be computed efficiently using the chain rule, allowing for backpropagation to update the model during training.
3. **Enforces Confidence:** Cross-entropy encourages the model to be confident in its predictions. The more incorrect a prediction is, the larger the loss, which discourages the model from making ambiguous predictions.
4. **Logarithmic Transformation:** Cross-entropy involves a logarithmic term, which helps amplify the difference between predicted probabilities. When the predicted probability for the correct class is high, the loss is low, and when it is low, the loss increases significantly. This sharpens the model's focus on the correct class during training.
5. **Mathematical Suitability:** Cross-entropy has a sound mathematical foundation and is derived from information theory. It has properties that make it well-suited for probabilistic models and is a natural choice for models that predict class probabilities.

The cross-entropy loss function for multiclass classification is typically defined as follows:

$$L(y, \hat{y}) = - \sum_{i=1}^N \sum_{j=1}^C y_{ij} \log(\hat{y}_{ij})$$

Where:

- L is the cross-entropy loss.
- N is the number of data samples.
- C is the number of classes.
- y_{ij} is an indicator function that equals 1 if the true label for data sample i is class j , and 0 otherwise.
- \hat{y}_{ij} is the predicted probability that data sample i belongs to class j .

The goal of training a multiclass classification model is to minimize this cross-entropy loss, which effectively encourages the model to produce predicted probabilities that align with the true class labels. This loss function is widely used in a variety of applications, including image classification, text categorization, and more.

AN

how to avoid overfitting in image classification



Overfitting is a common challenge in image classification, as it can lead to a model that performs well on the training data but poorly on unseen data (validation or test data). To avoid overfitting in image classification, you can employ several strategies:



1. **Increase Data Quantity:**

- **Data Augmentation:** Augment your training dataset by creating variations of existing images through techniques like rotation, flipping, cropping, and changing brightness and contrast. This increases the diversity of the training data and helps the model generalize better.

2. **Regularization Techniques:**

- **Dropout:** Add dropout layers to your neural network architecture. Dropout randomly deactivates a fraction of neurons during training, preventing the network from relying too heavily on a specific set of features.
- **L1 and L2 Regularization:** Apply L1 or L2 regularization to the model's weights to prevent them from becoming excessively large. This encourages the model to learn simpler patterns.

3. **Architectural Choices:**

- **Simpler Models:** Consider using simpler model architectures with fewer layers or neurons. Complex models are more prone to overfitting, especially when training data is limited.
- **Early Stopping:** Monitor the model's performance on a validation dataset during training and stop when the validation loss starts increasing. This helps prevent the model from over-optimizing the training data.

4. **Validation Dataset:**

- Use a separate validation dataset to monitor the model's performance during training. Adjust hyperparameters, such as learning rate and batch size, based on the validation performance.

5. **Cross-Validation:** If you have a limited amount of data, consider using techniques like k-fold cross-validation to train and evaluate the model on different subsets of the data. This can provide a more reliable estimate of the model's performance.

6. **Data Preprocessing:**

- **Normalization:** Ensure that your input data is properly normalized to have a mean of 0 and a standard deviation of 1. This helps the model converge faster and prevents overfitting.
- **Data Cleaning:** Carefully preprocess your data to remove noise, outliers, and irrelevant features.

7. **Model Ensembling:**

- Combine multiple models (e.g., using techniques like bagging or boosting) to improve overall performance and reduce overfitting.

8. **Hyperparameter Tuning:**

- Experiment with various hyperparameters, such as learning rate, batch size, and the number of layers or neurons in your model. Use techniques like grid search or random search to find optimal hyperparameters.

AN

what metrics to evaluate the model



When evaluating a machine learning or deep learning model for image classification, there are several metrics you can use to assess its performance. The choice of metrics depends on the specific characteristics of your classification task and what you consider most important. Here are some commonly used metrics for evaluating image classification models:

1. **Accuracy:** Accuracy is the most straightforward metric and represents the ratio of correctly predicted instances to the total number of instances. While it provides a general sense of overall model performance, it may not be suitable for imbalanced datasets.
2. **Precision:** Precision measures the ratio of true positive predictions to the total number of positive predictions (both true positives and false positives). It is particularly important in cases where false positives are costly or need to be minimized.
3. **Recall (Sensitivity):** Recall, also known as sensitivity or true positive rate, measures the ratio of true positive predictions to the total number of actual positive instances. It is important when false negatives are costly or when you want to identify as many positive instances as possible.
4. **F1 Score:** The F1 score is the harmonic mean of precision and recall and provides a balance between the two. It is useful when you want to consider both false positives and false negatives equally.
5. **Specificity (True Negative Rate):** Specificity measures the ratio of true negative predictions to the total number of actual negative instances. It is important when minimizing false positives is a priority.
6. **Area Under the Receiver Operating Characteristic (ROC-AUC):** ROC-AUC measures the model's ability to distinguish between positive and negative instances. It's particularly useful for binary classification problems, but it can also be adapted for multiclass problems.
7. **Area Under the Precision-Recall Curve (PR-AUC):** PR-AUC measures the precision-recall trade-off, which is especially important when working with imbalanced datasets.
8. **Confusion Matrix:** A confusion matrix provides a detailed breakdown of the model's predictions, showing true positives, true negatives, false positives, and false negatives. It is a valuable tool for assessing classification performance.
9. **Cohen's Kappa:** Cohen's Kappa accounts for the agreement between the model's predictions and the actual class labels while correcting for chance agreement. It's useful for multiclass classification and can handle class imbalance.
10. **Mean Average Precision (mAP):** mAP is commonly used in object detection and image retrieval tasks. It assesses the precision-recall curve across multiple classes and calculates the mean.
11. **Top-k Accuracy:** In cases where multiple classes can be correct, you can measure the model's accuracy in the top-k predictions, such as top-1 accuracy (standard accuracy) and top-5 accuracy.
12. **Cross-Entropy Loss:** The cross-entropy loss on a validation dataset is often used as a primary metric for training deep learning models. Lower cross-entropy indicates a better model.

Implementation link:

<https://colab.research.google.com/drive/138RhrA8ySWjmM7zhfo2nXSfBC0FMDO8Q#scrollTo=atMNGLboEURD>.