Technical University of Crete
SCHOOL OF ELECTRICAL ENGINEERING & COMPUTER ENGINEERING

THΛ 311 – Statistical Modeling and Pattern Recognition – 2024

Instructor: Vasilis Digalakis **2st Set of Exercises** Resolution by: 27/06/2024

Report

Topic 1: Perceptron Algorithm

In this exercise, we will focus on implementing the Perceptron algorithm, in its batch form, for classification between two classes. To begin with, in order to implement the perceptron algorithm, we need to implement:

- A linear function that aggregates the input signals
- A threshold function that determines whether or not the response neuron is fired
- A learning process for adjusting the weighting coefficients of connections

Let us consider a threshold function that compares the weighted sum of the inputs with a threshold θ, that is,

$$z = \sum_i w_i x_i \geq \theta$$

We remove both members of the inequality according to θ and get:

$$z = \sum_i w_i x_i - \theta \geq -\theta + \theta = 0$$

If we replace $-\theta$ with b then:

$$x = b + \sum_i w_i x_i \geq 0 \tag{1}$$

The Eξ.1 will be called the aggregation function.

We proceed by defining the threshold function, which is defined as follows:

$$\hat{y} = f(z) = \begin{cases} +1, \text{ if } z > 0 \\ -1, \text{ else} \end{cases}$$

Note that z is the output of the aggregation function we calculated above.

Finally, we need to develop a learning process for adjusting the weights.

We use the error-correction learning rule, which is defined as follows:

$$w_{k+1} = w_k + \Delta w_k$$

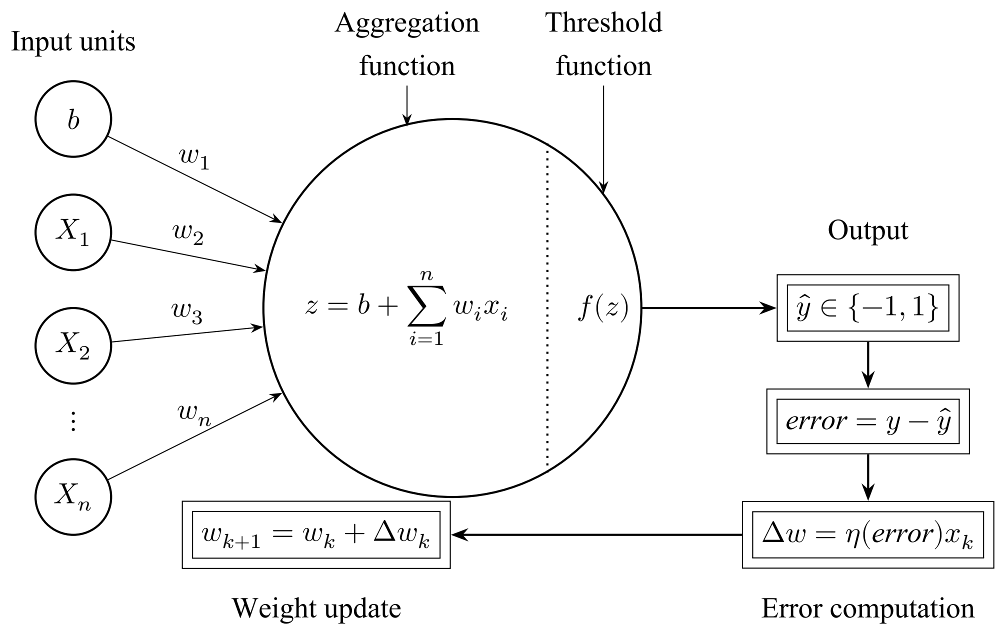
Δw_k is calculated as follows:

$$\Delta w_k = \eta(y - \hat{y})x_k$$

Where:

- w_k : is the weight coefficient for the k case
- η : is the learning rate[1] $(\eta \in (0,1))$
- y : is the true value (”true class”)
- $\hat{y}$: is the predicted value (”predicted class”)
- x_κ : is the vector of inputs for case k

The following image shows a schematic representation of the perceptron with the learning process.



Schematic representation of the Perceptron algorithm

[1]The learning rate serves to facilitate the training process by weighting the Δ used to update the weights. This essentially means that instead of completely replacing the previous weight with the sum of the weight + Δ, we incorporate a percentage of the error into the update process, making the learning process more stable.

We begin our implementation by reading the following data:

	ω_1		ω_2		ω_3		ω_4	
Sample	x_1	x_2	x_1	x_2	x_1	x_2	x_1	x_2
1	0.1	1.1	7.1	4.2	-3	-2.9	2	-8.4
2	6.8	7.1	-1.4	-4.3	0.5	8.7	-8.9	0.2
3	-3.5	-4.1	4.5	0	2.9	2.1	-4.2	-7.7
4	2	2.7	6.3	1.6	-0.1	5.2	-8.5	-3.2
5	4.1	2.8	4.2	1.9	-4	2.2	-6.7	-4
6	3.1	5	1.4	-3.2	-1.3	3.7	-0.5	-9.2
7	-0.8	-1.3	2.4	-4	-3.4	6.2	-5.3	-6.7
8	0.9	1.2	2.5	-6.1	-4.1	3.4	-8.7	-6.4
9	5	6.4	8.4	3.7	-5.1	1.6	-7.1	-9.7
10	3.9	4	4.1	-2.2	1.9	5.1	-8	-6.3

Table 1: Data for the Perceptron algorithm

Sample Display

We plot our samples and obtain the following image:

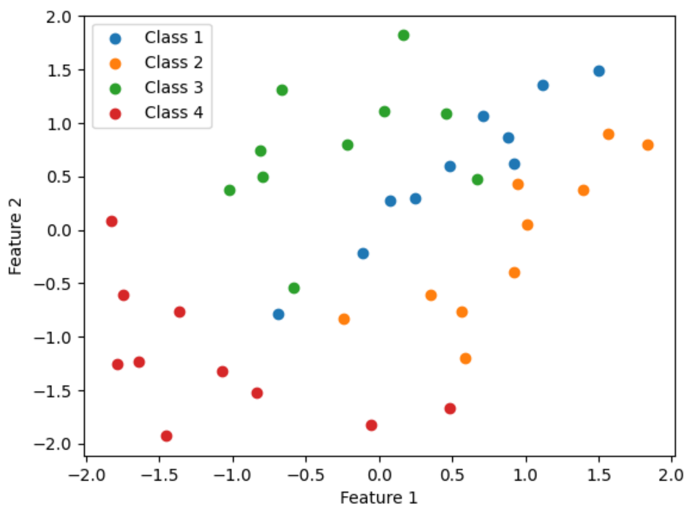


Figure 1: Illustration of the samples in Table 1

Implementation of the Perceptron algorithm and display of results

At the same time, we implement the functions we analyzed earlier in Python and we have:

normalize_features

```python
def normalize_features(X):
'''Normalize features to zero mean and unit variance'''
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
return (X - mean) / std
```

zero_weights

```python
def zero_weights(n_features):
    '''Create vector of zero weights'''
    return np.zeros(1 + n_features)  # Add 1 for the bias term
```

net_input

```python
def net_input(X, w):
'''Compute net input as dot product'''
return np.dot(X, w[1:]) + w[0]
```

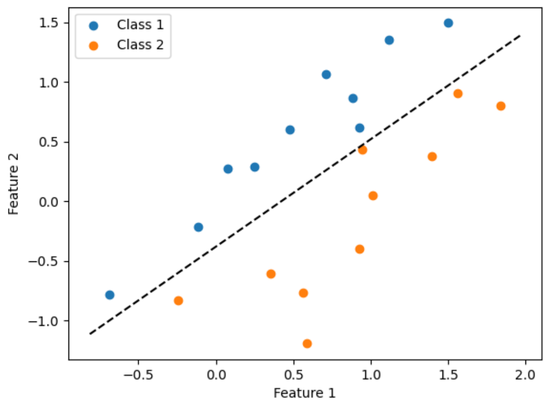
predict

```python
def predict(X, w):
'''Return class label after unit step'''
return np.where(net_input(X, w) >= 0.0, 1, -1)
```

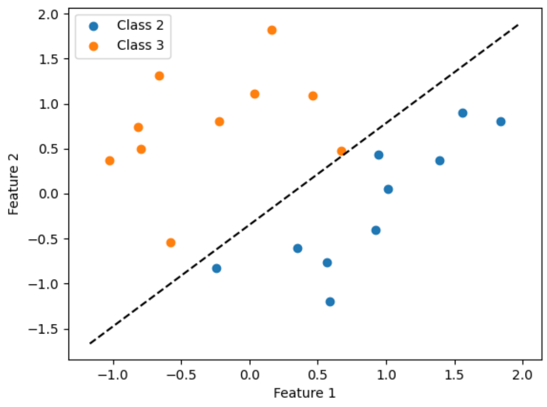
fit_batch

```python
def fit_batch(X, y, eta=0.05, n_iter=100):
    '''Batch form of the Perceptron algorithm'''
    iterations = 0
    errors = []
    w = zero_weights(X.shape[1])
    for _ in range(n_iter):
        output = net_input(X, w)
        errors_vector = y - np.where(output >= 0.0, 1, -1)
        if np.all(errors_vector == 0):
            break
        delta_w = eta * np.dot(errors_vector, X)
        w[1:] += delta_w
        w[0] += eta * errors_vector.sum()
        error = np.count_nonzero(errors_vector)
        errors.append(error)
        iterations += 1
    return w, errors, iterations
```

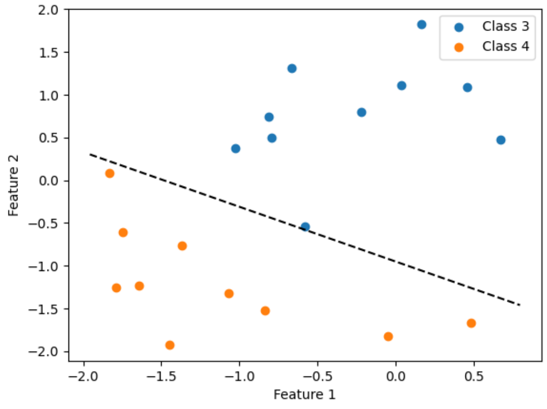
Having implemented the algorithm, we test its effectiveness sequentially on pairs of classes $<\omega_1, \omega_2>$, $<\omega_2, \omega_3>$ and $<\omega_3, \omega_4>$, obtaining the following images:



(a) Classification of class samples ω_1, ω_2



(b) Classification of class samples ω_2, ω_3



(c) Classification of class samples ω_3, ω_4

Figure 2: Application of Perceptron on successive pairs of classes

At the same time, in order to better evaluate the efficiency of the algorithm, we plot the number of errors as a function of the number of time steps, and we obtain the following figures:

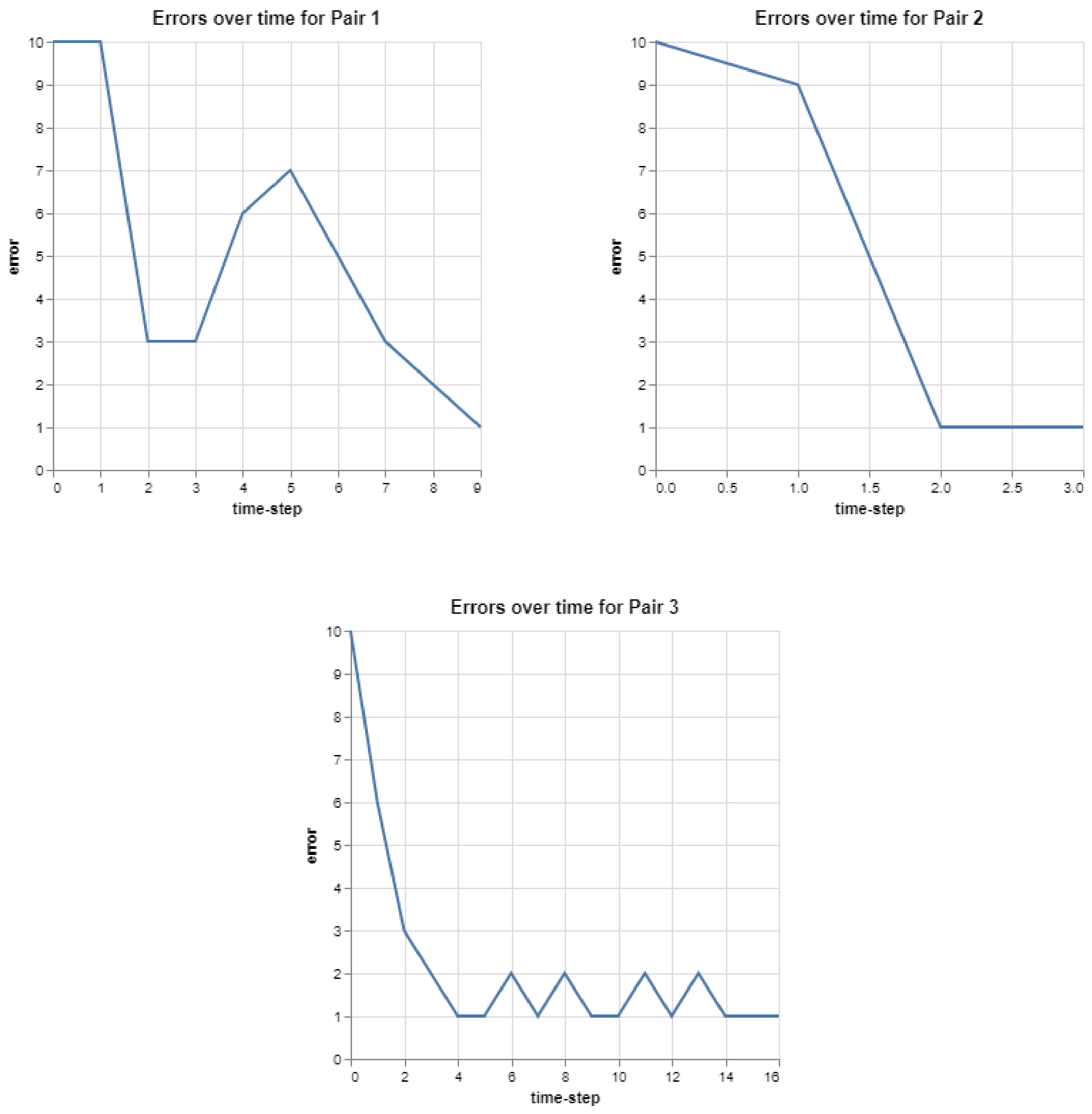


Figure 3: Graphical representation of the error of the Perceptron algorithm in relation to iterations

Comparison and Evaluation of Results

We conclude by presenting our results in the following table:

Pair of classes	Accuracy	Num. iterations	Learning rate
ω_1, ω_2	100%	10	0.05
ω_2, ω_3	100%	4	0.05
ω_3, ω_4	100%	17	0.05

Table 2: Perceptron Aggregate Results

We therefore observe variations in the number of iterations required for each pair of classes. These variations are mainly due to the separability of the samples and the learning rate. Specifically, if a pair of classes is linearly separable with a clear margin, the algorithm converges quickly. A notable example is the pair <ω_2, ω_3>, which requires only 4 iterations for the algorithm to converge.

In addition to class separability, the learning rate (η) also plays a critical role. A relatively high learning rate can lead to faster convergence when classes are well separated. However, with overlapping classes, a high learning rate can cause oscillations or slower convergence due to overly large updates.

Topic 2: Logistic Regression - Analytical slope estimation

Finding the error slope for the j-element

The logistic regression function is defined as follows:

$$h_\theta(\mathbf{x}) = f(\theta^T \mathbf{x}) \tag{2}$$

The logistic function is defined as follows:

$$f(z) = \frac{1}{1 + 2e^{-z}} \tag{3}$$

Therefore, the logistic regression function is also defined as:

$$h_\theta(\mathbf{x}) = \frac{1}{1 + 2e^{-\theta^T \mathbf{x}}} \tag{4}$$

The cost/error function called cross-entropy is also defined as:

$$\begin{aligned}
J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \left(-y^{(i)} \ln(\hat{y}^{(i)}) - (1 - y^{(i)}) \ln(1 - \hat{y}^{(i)})\right) \iff \\
J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \left(-y^{(i)} \ln(\widehat{h_\theta(\mathbf{x})}^{(i)}) - (1 - y^{(i)}) \ln(1 - \widehat{h_\theta(\mathbf{x})}^{(i)})\right) \iff \\
J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} \ln\left(\frac{1}{1 + 2e^{-\theta^T \mathbf{x}^{(i)}}}\right) - (1 - y^{(i)}) \ln\left(1 - \frac{1}{1 + 2e^{-\theta^T \mathbf{x}^{(i)}}}\right)\right] \iff \\
J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} \ln\left(\frac{1}{1 + 2e^{-\theta^T \mathbf{x}^{(i)}}}\right) - (1 - y^{(i)}) \ln\left(\frac{2e^{-\theta^T x^{(i)}}}{1 + 2e^{-\theta^T x^{(i)}}}\right)\right]
\end{aligned} \tag{5}$$

From the properties of logarithms, we have the following equivalences:

1. $\ln\left(\frac{1}{1+2e^{-\theta^T \mathbf{x}^{(i)}}}\right) = -\ln(1+2e^{-\theta^T \mathbf{x}^{(i)}})$

2. $\ln\left(\frac{2e^{-\theta^T x^{(i)}}}{1+2e^{-\theta^T x^{(i)}}}\right) = \ln(2e^{-\theta^T x^{(i)}}) - \ln(1+2e^{-\theta^T x^{(i)}}) = -\theta^T x^{(i)} + \ln(2) - \ln(1+2e^{-\theta^T x^{(i)}})$

Therefore, Eξ.5 can also be written as:

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ln(1+2e^{-\theta^T\mathbf{x}^{(i)}}) + (1-y^{(i)})\left(\theta^T\mathbf{x}^{(i)} + \ln(2) - \ln(1+2e^{-\theta^T\mathbf{x}^{(i)}})\right)\right] \\ &= \frac{1}{m}\sum_{i=1}^{m}\left[\ln(1+2e^{-\theta^T\mathbf{x}^{(i)}}) + (1-y^{(i)})(\theta^T\mathbf{x}^{(i)} + \ln(2))\right] \end{aligned} \tag{6}$$

However, since $\ln(2) \approxeq 0$, the above expression simplifies as follows:

$$\frac{1}{m}\sum_{i=1}^{m}\left[\ln(1+2e^{-\theta^T\mathbf{x}^{(i)}}) + (1-y^{(i)})(\theta^T\mathbf{x}^{(i)})\right] \tag{7}$$

We calculate the partial derivative, $\frac{\partial J(\theta)}{\partial \theta_j}$, and we have:

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_j} &= \frac{1}{m}\sum_{i=1}^{m}\left[\frac{\partial}{\partial \theta_j}\left(\ln(1+2e^{-\theta^T\mathbf{x}^{(i)}}) + (1-y^{(i)})(\theta^T\mathbf{x}^{(i)}\right)\right] \\ &= \frac{1}{m}\sum_{i=1}^{m}\left[\frac{\partial}{\partial \theta_j}\left(\ln(1+2e^{-\theta^T\mathbf{x}^{(i)}})\right) + \frac{\partial}{\partial \theta_j}\left((1-y^{(i)})(\theta^T\mathbf{x}^{(i)})\right)\right] \end{aligned} \tag{8}$$

We calculate the partial derivatives and we have:

- $\frac{\partial}{\partial \theta_j}\ln(1+2e^{-\theta^T x^{(i)}}) = \frac{1}{1+2e^{-\theta^T x^{(i)}}}\cdot(-2x_j^{(i)}\cdot e^{-\theta^T x^{(i)}}) = \frac{-2x_j^{(i)}\cdot e^{-\theta^T x^{(i)}}}{1+2e^{-\theta^T x^{(i)}}}$
- $\frac{\partial}{\partial \theta_j}((1-y^{(i)})(\theta^T\mathbf{x}^{(i)})) = (1-y^{(i)}x_j^{(i)})$

Finally, we have:

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_j} &= \frac{1}{m}\sum_{i=1}^{m}\left[\frac{-2x_j^{(i)}\cdot e^{-\theta^T x^{(i)}}}{1+2e^{-\theta^T x^{(i)}}} + (1-y^{(i)})x_j^{(i)}\right] \iff \\ &= \frac{1}{m}\sum_{i=1}^{m}\left[x_j^{(i)}\cdot\frac{-2e^{-\theta^T x^{(i)}}}{1+2e^{-\theta^T x^{(i)}}} + (1-y^{(i)})x_j^{(i)}\right] \end{aligned} \tag{9}$$

However, $h_\theta(\mathbf{x}) = \frac{1}{1+2e^{-\theta^T\mathbf{x}}}$.

Therefore, it follows that:

$$\boxed{\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}\right]} \tag{10}$$

In matrix form, we have:

$$\boxed{\text{grad} = \frac{1}{m}X^T(h-y)} \tag{11}$$

where X is the following matrix:

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}$$

Reading and Displaying Samples

Next, we use logistic regression to calculate the decision threshold and classify our samples into two distinct categories (admitted and not admitted). First, we design our samples and obtain the following image:

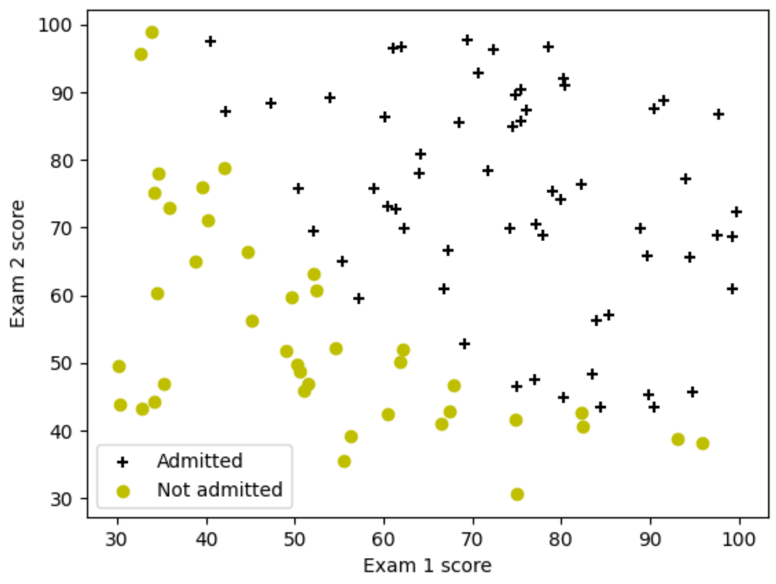


Figure 4: Sample display

Sigmoid function

Then, we complete the sigmoid function so that it returns the sigmoid function $f(z)$ of the exercise.

sigmoid

```
def sigmoid(z):
    sigmoid_function = 1 / (1 + 2*np.exp(-z))
    return sigmoid_function
```

CostFunction and gradient functions

We continue by filling in the code in the costFunction and gradient functions so that they return the cost and gradient, respectively.

costFunction

```
def costFunction(theta, X, y):
    ### ADD YOUR CODE HERE
    m = len(y)
    h = sigmoid(X @ theta)
    J = (1/m) * np.sum(-y * np.log(h) - (1 - y) * np.log(1 - h))
    return J
```

gradient

```
def gradient(theta, X, y):
    ### ADD YOUR CODE HERE
    m = len(y)
    h = sigmoid(X @ theta)
    error = h - y
    grad = (1/m) * (X.T @ error)
    return grad
```

Decision Boundary and Evaluation of Results

We conclude by calculating the decision boundary and projecting it together with our initial samples, obtaining the following figure:

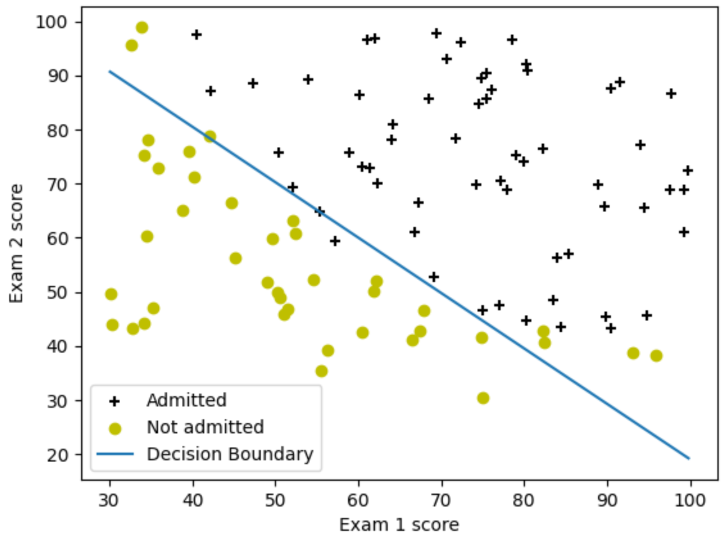


Figure 5: Decision boundary

Therefore, it can be observed that this algorithm achieves a relatively high degree of accuracy (accuracy = 89%) in classifying between the two classes, with only a few errors being noted in the image.

At the same time, we calculate the optimization of our model and extract the following table:

Table 3: Optimization of Regression Accounting

Parameter	**Value**
Initial cost	0.821
Initial slope	$[-0.267, -22.950, -22.300]$
Cost after optimisation	0.203
Slope after optimization	$[-24.468, 0.206, 0.201]$

It is clear that the initial model is not optimal and requires further improvement. The initial cost is close to unity, while the gradient components are particularly high. In order to achieve a significant improvement in the model, the Truncated Newton algorithm was used. The cost has been significantly reduced, tending towards zero, while the slope components are now close to zero.

Finally, we complete the predict function to find the probabilities that a student will be accepted with specific scores. So we have:

predict

```python
def predict(theta, X):
    ### ADD YOUR CODE HERE
    prob = sigmoid(X @ theta)
    return (prob >= 0.5).astype(int)
```

Therefore, the probability of a student with scores of 45 and 85 being accepted is: 0.7763 or 77.63%.

Topic 3: Parameter Estimation with Maximum Likelihood

Maximum Likelihood for each feature of class ω_1

We calculate the mean value, $\hat{\mu}$, as well as the variance $\hat{\sigma}^2$ for each feature x_i of class ω_1 using Maximum Likelihood Estimation (MLE). The desired result is implemented in Python as follows:

calculate_mle

```python
def calculate_mle(samples):
    # Mean ( )
    mu_hat = np.mean(samples, axis=0)

    # Variance ( ^2)
    # axis=0 because we want to calculate the variance of each column
    # ddof=0 because we want to calculate the maximum likelihood
     ↪ estimation -- Delta Degrees of Freedom
    sigma_hat_sq = np.var(samples, axis=0, ddof=0)
    return mu_hat, sigma_hat_sq

mu_hat, sigma_hat_sq = calculate_mle(samples_omega1)
print("Mean (ˆ):", mu_hat)
print("Variance (ˆ^2):", sigma_hat_sq)
```

And we obtain the following results:

- **Mean value ($\hat{\mu}$)**: $[-0.0709, -0.6047, -0.911]$
- **Variance ($\hat{\sigma}^2$)**: $[0.9062, 4.2007, 4.5419]$

Calculation of Parameters for 2D Distribution

For each pair of characteristics of class ω_1, we calculate the vector of mean values $\hat{\mu}$ as well as the covariance matrix Σ assuming that every two characteristics of class ω_1 follow a normal distribution. We implement the requested in Python and we have:

calculate_2d_mle

```
def calculate_2d_mle(samples):
    # Mean ( )
    mu_hat = np.mean(samples, axis=0)

    # Covariance matrix (Σ)
    # rowvar=False because each column represents a variable
    # bias=True for maximum likelihood estimation
    sigma_hat = np.cov(samples, rowvar=False, bias=True)
    return mu_hat, sigma_hat
```

And we obtain the following results:

- **Pair** (x_1, x_2)
  - **Mean value (μ)**: $[-0.0709, -0.6047]$
  - **Covariance matrix (Σ)**:
$$\begin{bmatrix} 0.9062 & 0.5678 \\ 0.5678 & 4.2007 \end{bmatrix}$$
- **Pair** (x_1, x_3)
  - **Mean value (μ)**: $[-0.0709, -0.911]$
  - **Covariance matrix (Σ)**:
$$\begin{bmatrix} 0.9062 & 0.3941 \\ 0.3941 & 4.5419 \end{bmatrix}$$

- **Pair** (x_2, x_3)
  - **Mean value** (μ): $[-0.6047, -0.911]$
  - **Covariance matrix** (Σ):
$$\begin{bmatrix} 4.2007 & 0.7337 \\ 0.7337 & 4.5419 \end{bmatrix}$$

Calculation of Parameters for 3D Distribution

For the entire feature vector of class ω_1, we calculate the mean vector $\hat{\mu}$ and the covariance matrix $\hat{\Sigma}$ assuming that the total feature vector follows a 3D distribution. We implement the requested in Python and we have:

Code for part 3

```
mu_hat_3d, sigma_hat_3d = calculate_2d_mle(samples_omega1)
print("Mean (^) for 3D:", mu_hat_3d)
print("Covariance (Σ) for 3D:", sigma_hat_3d)
```

And we obtain the following results:

- **Mean value** ($\hat{\mu}$): $[-0.0709, -0.6047, -0.911]$
- **Covariance matrix** ($\hat{\Sigma}$):
$$\begin{bmatrix} 0.9062 & 0.5678 & 0.3941 \\ 0.5678 & 4.2007 & 0.7337 \\ 0.3941 & 0.7337 & 4.5419 \end{bmatrix}$$

Calculation of Parameters for Diagonal Covariance Matrix

For class ω_2, assuming that the covariance matrix is diagonal, we calculate the mean vector $\hat{\mu}$ and the diagonal elements of $\hat{\Sigma}$. We implement the requested in Python and we have:

Code for part 4

```
mu_hat_diag, sigma_hat_diag_sq = calculate_mle(samples_omega2)
print("Mean (^) for diagonal Σ:", mu_hat_diag)
print("Variances (^^2) for diagonal Σ:", sigma_hat_diag_sq)
```

And we obtain the following results:

- **Mean value** ($\hat{\mu}$): $[-0.1126, 0.4299, 0.00372]$
- **Diagonal Variances** ($\hat{\sigma}^2$): $[0.0539, 0.0460, 0.0073]$

Comparison and Evaluation of Results

At this point, we compare the different mean values and variances as calculated by the different models.

- **Mean value vectors ($\hat{\mu}$)**:
  - **1D**: $[-0.0709, -0.6047, -0.911]$
  - **2D (x_1, x_2)**: $[-0.0709, -0.6047]$
  - **2D (x_1, x_3)**: $[-0.0709, -0.911]$
  - **2D (x_2, x_3)**: $[-0.6047, -0.911]$
  - **3D**: $[-0.0709, -0.6047, -0.911]$
  - **Diagonal** Σ: $[-0.1126, 0.4299, 0.00372]$
- **Variance ($\hat{\sigma}^2$)**:
  - **1D**: $[0.9062, 4.2007, 4.5419]$
  - **Diagonal** Σ: $[0.0539, 0.0460, 0.0073]$

Commentary - Explanation

The differences in results are due to the assumptions made by each model. Specifically:

- The 1D model treats each feature independently, resulting in different variance values compared to models that take feature correlations into account.
- Two-dimensional models take into account the correlations between pairs of characteristics, resulting in covariance values that reflect these relationships.
- The three-dimensional model takes into account the entire set of characteristics and their interdependencies, providing a comprehensive covariance matrix.
- The diagonal covariance model assumes that there is no correlation between the characteristics, leading to different estimates for the variances and means.

These variations demonstrate how the underlying assumptions of each model affect the estimated parameters.

Topic 4: Clustering with K-means and GMM

In this exercise, we will implement the K-means clustering algorithm in order to compress an image.

Function findClosestCentroids

We start by filling in the findClosestCentroids function, which finds the $c^{(i)} := j$ that minimize the distance $\|x^{(i)} - \mu_j\|^2$ where $c^{(i)}$ is the index of the closest center to $x^{(i)}$,and μ_j is the vector of values of center j. In Python, we have:

find_closest_centroids

```
def find_closest_centroids(X, centroids):
    idx = np.zeros(X.shape[0], dtype=int)
    for i in range(X.shape[0]):
        distances = np.linalg.norm(X[i] - centroids, axis=1)
        idx[i] = np.argmin(distances)
    return idx
```

ComputeCentroids function

Next, we complete the computeCentroids function, which calculates the centroids of the classes and is defined as follows:

$$\mu_k = \frac{1}{|C_k|} \sum_{i \in C_k} x^{(i)}$$

where C_k is the set of examples assigned to class k At the Python level, we have:

compute_centroids

```
def compute_centroids(X, idx, K):
    centroids = np.zeros((K, X.shape[1]))
    for k in range(K):
        points = X[idx == k]
        if points.shape[0] > 0:
            centroids[k] = np.mean(points, axis=0)
    return centroids
```

Model parameterization

At the same time, we are expanding our program so that it can read images that are in grayscale as well as images that have an alpha channel.[2](i.e png)At the code level, this is implemented as follows:

Initialaziton Part

```
# Load the image
image = io.imread('Fruit.png')

# Size of the image
img_size = image.shape

# Normalize image values in the range 0 - 1
image = image / 255.0
```

[2]The alpha channel, often referred to as the transparency channel, is an element of digital images that determines the level of transparency or opacity of each pixel in an image.

```
# Check the number of channels in the image
if image.ndim == 2:
    # If the image is grayscale, duplicate the single channel to make it
    ↪ 3-channel
    image = np.stack((image,)*3, axis=-1)
elif image.shape[2] == 4:
    # If the image has an alpha channel, remove it
    image = image[:, :, :3]

# Reshape the image to be a Nx3 matrix (N = num of pixels)
X = image.reshape(img_size[0] * img_size[1], 3)

# Reshape the image to be a Nx3 matrix (N = num of pixels)
X = image.reshape(img_size[0] * img_size[1], 3)
```

Algorithm Efficiency

We examine the efficiency of our algorithm with an image and obtain the following results:

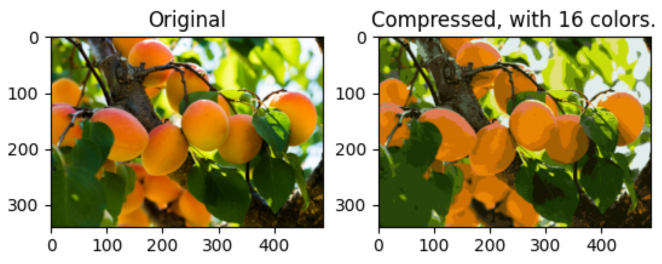
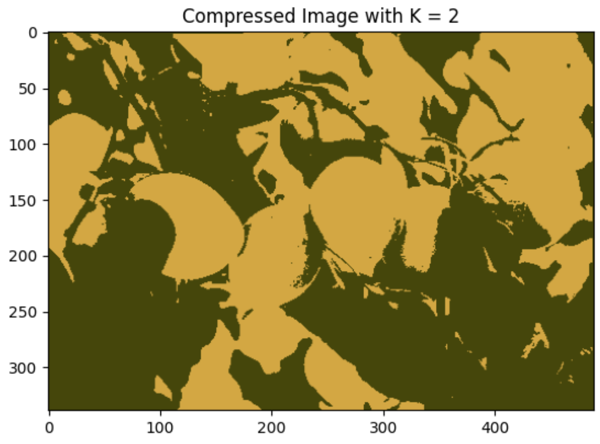
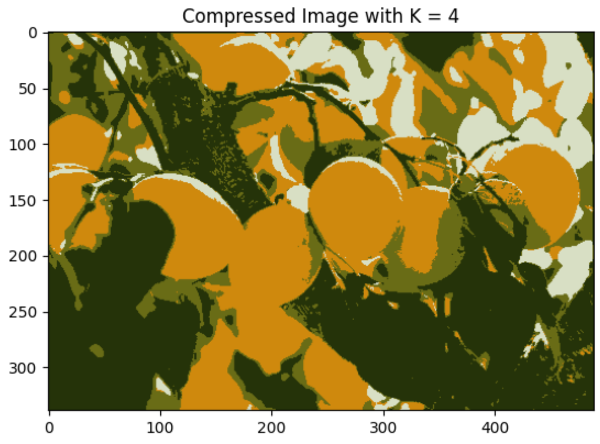
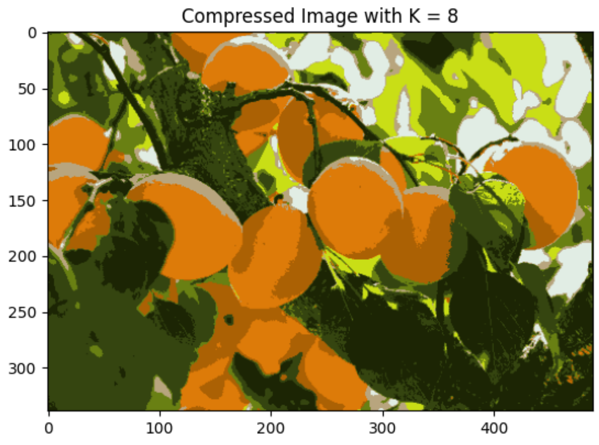


Figure 6: Efficiency of the KMeans algorithm

We extend the testing of our model by examining various numbers of features, which leads to the following cases:


Compressed Image with K = 2
0
50
100
150
200
250
300
0
100
200
300
400


Compressed Image with K = 4
0
50
100
150
200
250
300
0
100
200
300
400


Compressed Image with K = 8
0
50
100
150
200
250
300
0
100
200
300
400

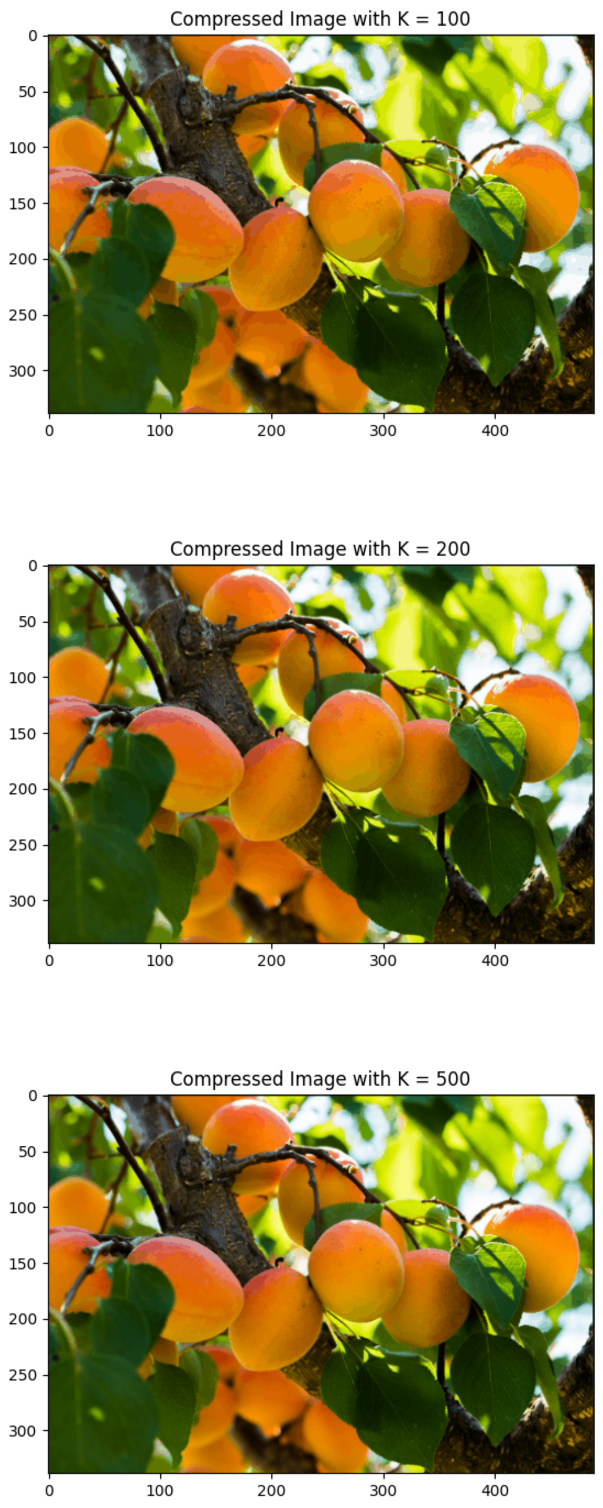

Compressed Image with K = 100
Compressed Image with K = 200
Compressed Image with K = 500
0
50
100
150
200
250
300
0
100
200
300
400

Figure 7: In top-down order, compressed images using different numbers of features

We observe that increasing the characteristics/colors of the image improves the efficiency of the algorithm and, by extension, we obtain a better result. At the same time, we also provide a summary table showing the degree of compression in relation to the original image.

Compressed with K =	Compression Rate:
2	87%
4	77%
8	68%
16	66%
100	42%
200	37%
500	33%

Table 4: Compression ratio summary table

Neural networks

A.YImplementation of a simple neural network

In this section, we implement a simple neural network without using ready-made programs.

Transformation of the Cross-Entropy Loss Function

We know that:

$$z^{(i)} = x^{(i)}W + b \tag{12}$$

As well as:

$$\hat{y}^{(i)} = f(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}} \tag{13}$$

The cross entropy for a set of B samples is defined as follows:

$$J(Y,\hat{Y};W,b) = \frac{1}{B}\sum_i \left(-y^{(i)}\ln(\hat{y}^{(i)}) - (1-y^{(i)})\ln(1-\hat{y}^{(i)})\right) \tag{14}$$

However,

$$1-\hat{y}^{(i)} = 1 - \frac{1}{1+e^{-z^{(i)}}} = \frac{e^{-z^{(i)}}}{1+e^{-z^{(i)}}} \tag{15}$$

$$\ln(\hat{y}^{(i)}) = \ln\left(\frac{1}{1+e^{-z^{(i)}}}\right) = -\ln(1+e^{-z^{(i)}}) \tag{16}$$

$$\ln(1-\hat{y}^{(i)}) = \ln\left(\frac{e^{-z^{(i)}}}{1+e^{-z^{(i)}}}\right) = -z^{(i)} - \ln(1+e^{-z^{(i)}}) \tag{17}$$

Based on Eξ.15, Eξ.16, Eξ.17, Eξ.14 can be written equivalently as follows:

$$J(Y,\hat{Y};W,b) = \frac{1}{B}\sum_i \left(-y^{(i)}(-\ln(1+e^{-z^{(i)}}) - (1-y^{(i)})(-z^{(i)} - \ln(1+e^{-z^{(i)}})\right) \iff$$
$$= \sum_i \left(y^{(i)}\ln(1+e^{-z^{(i)}}) + (1-y^{(i)})(z^{(i)} + \ln(1+e^{-z^{(i)}})\right) \iff$$
$$= \boxed{\sum_i \left(z^{(i)} - y^{(i)}z^{(i)} + \ln(1+e^{-z^{(i)}})\right) \text{ o.ε.δ}}$$

Partial derivative of the cross-entropy loss function with respect to $\mathbf{z^{(i)}}$

$$\begin{aligned}\frac{\partial J}{\partial z^{(i)}} &= \frac{\partial}{\partial z^{(i)}} \sum_i \left(z^{(i)} - y^{(i)} z^{(i)} + \ln(1 + e^{-z^{(i)}})\right) \\ &= \sum_i \left(\frac{\partial}{\partial z^{(i)}}(z^{(i)} - y^{(i)} z^{(i)})) + \frac{\partial}{\partial z^{(i)}} \ln(1 + e^{-z^{(i)}})\right)\end{aligned} \tag{18}$$

We calculate the partial derivatives separately and obtain:

- $\frac{\partial}{\partial z^{(i)}}(z^{(i)} - y^{(i)} z^{(i)})) = 1 - y^{(i)}$
- $\frac{\partial}{\partial z^{(i)}} \ln(1 + e^{-z^{(i)}}) = \frac{1}{1 + e^{-z^{(i)}}} \cdot \frac{\partial}{\partial z^{(i)}}(1 + e^{-z^{(i)}}) = \frac{-e^{-z^{(i)}}}{1 + e^{-z^{(i)}}} = \hat{y}^{(i)} - 1$

Finally, Eξ.18 is transformed as follows:

$$\boxed{\frac{\partial J}{\partial z^{(i)}} = \hat{y}^{(i)} - 1 + 1 - y^{(i)} = -y^{(i)} + \hat{y}^{(i)}, i \in batch} \tag{19}$$

Application of the chain rule

According to the chain rule, the following applies:

$$\frac{dx}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{20}$$

So, applying the chain rule to Eξ.14 to find the partial derivative $\frac{\partial J}{\partial W}$, we have:

$$\frac{\partial J}{\partial W} = \sum_{i=1} \frac{\partial J}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial z^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial W} \tag{21}$$

From the rules of differentiation, we obtain the following:

- $\frac{\partial J}{\partial \hat{y}^{(i)}} = -\frac{y^{(i)}}{\hat{y}^{(i)}} + \frac{1 - y^{(i)}}{1 - \hat{y}^{(i)}}$
- $\frac{\partial \hat{y}^(i)}{\partial z^{(i)}} = \hat{y}^{(i)}(1 - \hat{y}^{(i)})$
- $\frac{\partial z^{(i)}}{\partial W} = x^{(i)}$

Finally, Eξ.21 is written as:

$$\boxed{\frac{\partial J}{\partial W} = \sum_{i=1} \left(\left(-\frac{y^{(i)}}{\hat{y}^{(i)}} + \frac{1 - y^{(i)}}{1 - \hat{y}^{(i)}}\right) \cdot \hat{y}^{(i)}(1 - \hat{y}^{(i)}) \cdot x^{(i)}\right)} \tag{22}$$

Similarly, for the partial derivative $\frac{\partial J}{\partial b}$ έχουμε:

$$\frac{\partial J}{\partial b} = \sum_{i=1} \frac{\partial J}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial z^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial b} \tag{23}$$

From the rules of differentiation, we obtain the following results:

- $\frac{\partial J}{\partial \hat{y}^{(i)}} = -\frac{y^{(i)}}{\hat{y}^{(i)}} + \frac{1-y^{(i)}}{1-\hat{y}^{(i)}}$
- $\frac{\partial \hat{y}^(i)}{\partial z^{(i)}} = \hat{y}^{(i)}(1-\hat{y}^{(i)})$
- $\frac{\partial z^{(i)}}{\partial b} = 1$

Finally, Eξ.23 is written as:

$$\boxed{\frac{\partial J}{\partial b} = \sum_{i=1}\left(\left(-\frac{y^{(i)}}{\hat{y}^{(i)}} + \frac{1-y^{(i)}}{1-\hat{y}^{(i)}}\right)\cdot \hat{y}^{(i)}(1-\hat{y}^{(i)})\right)} \tag{24}$$

Description of backpropagation for a multi-layer neural network

Let us consider a layer of the neural network, say l, such that:

- H_{l-1} defines the input vector
- W_l defines the weighting matrix
- b_l defines the vector of bias terms
- H_l defines the output vector

A.Forward Pass:

- The layer takes as input the vector X of the neural network
- We calculate Z^l where $Z^l = XW^l + b^l$
- We apply the activation function and we have: $H^{(l)} = \sigma(Z^{(l)})$ where $\sigma = \frac{1}{1+e^{-z}}$, i.e. the logistic function.
- We now set $H^{(l)}$ as X.

B.Backward computations:

- We calculate the error term $\delta^{(l)} = \left(\delta^{(l+1)}(W^{(l+1)})^T\right) \odot \sigma'(Z^{(l)})$ where $\sigma'(Z^{(l)}) = \sigma(Z^{(l)})(1-\sigma(Z^{(l)}))$
- We “update” the weights and we have: $W^{(l)} := W^{(l)} - \alpha\left(H^{(l-1)}\right)^T \delta^{(l)}$
- We “update” the bias terms and we have: $b^{(l)} := b^{(l)} - \alpha\sum_{i=1}^{m}\delta_i^{(l)}$

Neural network training

In this section, we deal with training a neural network using samples from the MNIST sample database. Our samples are 28×28 images depicting handwritten digits from 0 to 9. The goal of the network is to recognize the class to which each sample belongs individually.

We fill in the requested code appropriately and conduct experiments in order to arrive at an optimal network.

We summarize our experiments in the following table:

Experiment	Learning Rate	Num of Epochs	Batch Size	Activation Function	MSE	Ratio
1	0.1	100	128	Sigmoid	0.0361	0.75
2	0.1	100	128	Tanh	0.041	0.75
3	0.1	50	128	Sigmoid	0.0285	0.80
4	0.01	100	64	Sigmoid	0.0277	0.85
5	0.01	100	64	Tanh	0.0508	0.75
6	0.001	100	64	Sigmoid	0.0447	0.70
7	0.001	200	64	Tanh	0.0294	0.80
8	0.001	200	32	Tanh	0.0458	0.65
9	0.001	50	64	Sigmoid	0.0578	0.55
10	0.001	50	64	Tanh	0.0842	0.40
11	0.001	100	64	Sigmoid	0.0540	0.60
12	0.001	100	64	Tanh	0.0763	0.50
13	0.01	100	64	Sigmoid	0.0287	0.80
14	0.01	100	64	Tanh	0.0620	0.50

Table 5: Experimental combinations and results

Commentary on the results

The experiments, presented in tabular form, provide important information about the effect of various hyperparameters and network architectures on performance metrics such as mean square error (MSE) and accuracy ratio. The experiments can be divided into two groups: those with a simpler two-layer network architecture (experiments 1-8) and those with a more complex five-layer network architecture[3] (experiments 9-14).

With regard to the simpler two-level architecture, experiment 4 is of particular interest. This experiment, using a learning rate of 0.01, 100 epochs, and a batch size of 64, presented the lowest MSE (0.0277) and the highest ratio (0.85), indicating a well-balanced model. In contrast, experiment 5, which used a different activation function, showed a lower ratio and a higher MSE value.

The five-layer network architecture, which was more complex than the previous one, showed a wider range of results. This indicates that deeper networks are more sensitive to hyperparameter tuning. Experiment 7, with a learning rate of 0.001, 200 epochs, and a batch size of 64 using the Tanh activation function, showed satisfactory performance with an MSE of 0.0294 and a ratio of 0.80. However, Experiment 10, which had a similar setting but only 50 epochs, performed poorly with an MSE of 0.0842 and a ratio of 0.40. This demonstrates the necessity of sufficient training duration for deeper networks.

[3]A detailed description of both architectures is provided at the end of the report

Neural networks for image recognition

In this section, we will look at various classifiers from the tensorflow/keras library using the Fashion-MNIST dataset. In order to implement the neural network, we fill in the code appropriately and we have:

Code for Convolutional Neural Networks

```
# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

from plots import plot_some_data, plot_some_predictions



fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) =
 ↪  fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
 ↪  'Coat','Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Scale these values to a range of 0 to 1 before feeding them to the
 ↪  neural network model.
    ### YOUR CODE HERE
train_images = train_images / 255.0
test_images = test_images / 255.0

plot_some_data(train_images, train_labels, class_names)

# Build the model of dense neural network
# Building the neural network requires configuring the layers of the model,
 ↪  then compiling the model.
# Define the input layer based on the shape of the images
# Then define two dense layers.
# The hidden layer with 128 neurons and RELU activation
# The output layer with 10 neurons and linear activation.

model = keras.Sequential([
    layers.Flatten(input_shape=(28, 28)),  # Assuming the images are 28x28
     ↪  pixels
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')  # Assuming there are 10
     ↪  classes
])

optimizer_list = ['adam', 'sgd', 'rmsprop', 'adamax', 'nadam', 'ftrl']


# Compile the model
# Before the model is ready for training, it needs a few more settings.
 ↪  These are added during the model's compile step:
# Loss function -This measures how accurate the model is during training.
 ↪  You want to minimize this function to "steer" the model in the right
 ↪  direction.
# Optimizer -This is how the model is updated based on the data it sees
 ↪  and its loss function.


# Loop over each optimizer
for optimizer in optimizer_list:
#Compile the model with the current optimizer
model.compile(optimizer=optimizer,
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])

    # Train the model
    model.fit(train_images, train_labels, epochs=400,
                        validation_data=(test_images, test_labels))

    # Evaluate accuracy
    test_loss, test_acc = model.evaluate(test_images,  test_labels,
     ↪  verbose=2)

    # Print an appropriate message
    print(f"Training with {optimizer} finished. Test loss: {test_loss},
     ↪  Test accuracy: {test_acc}")
    print("###############################")

probability_model = tf.keras.Sequential([model,
                                tf.keras.layers.Softmax()])

predictions = probability_model.predict(test_images)
```

```
plot_some_predictions(test_images, test_labels, predictions, class_names,
↪  num_rows=5, num_cols=3)
```

For each optimizer we test, we create a graph showing the accuracy and validation accuracy function of the epochs, where the former refers to the accuracy of the training data and the latter refers to the accuracy of the validation data, i.e., data unknown to the neural network.

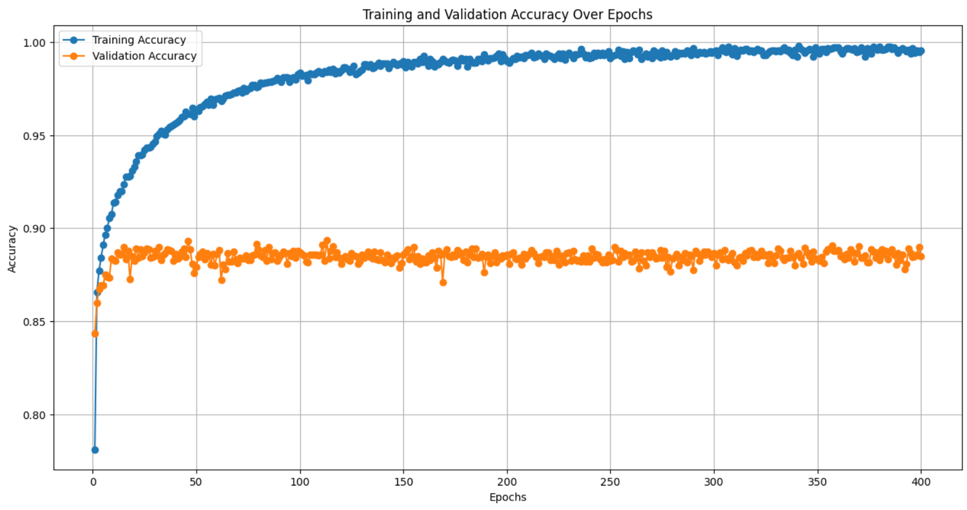


Figure 8: Comparison of accuracy values in training data with accuracy values in validation data in the Adam algorithm

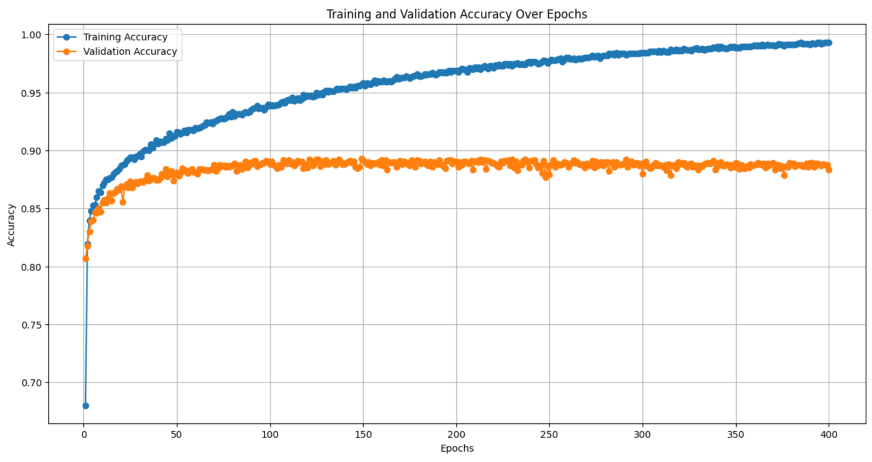


Figure 9: Comparison of accuracy values in training data with accuracy values in validation data in the sgd algorithm

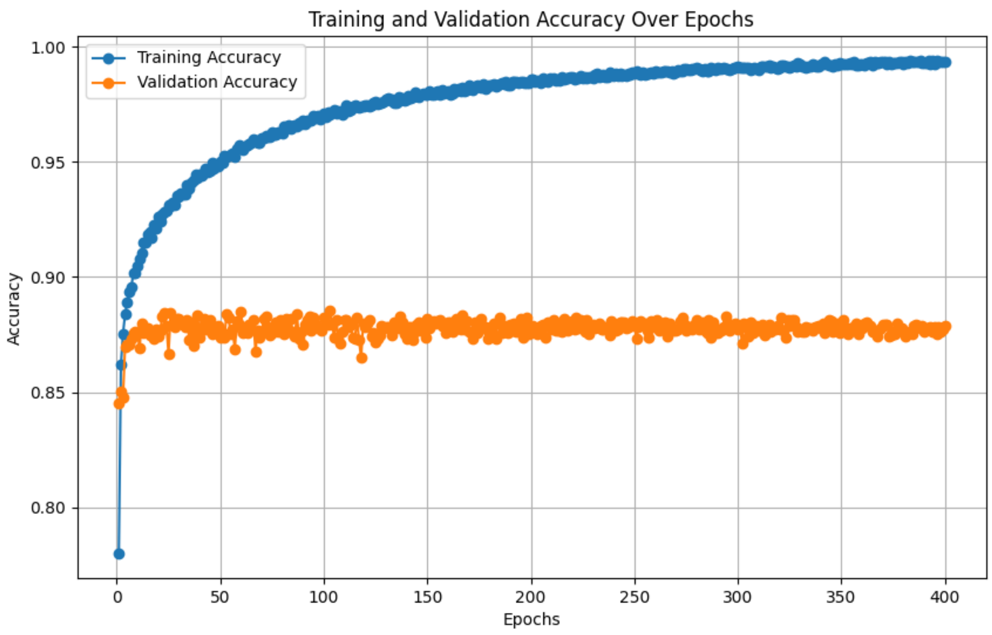


Figure 10: Comparison of accuracy values in training data with accuracy values in validation data in the MSRProp algorithm

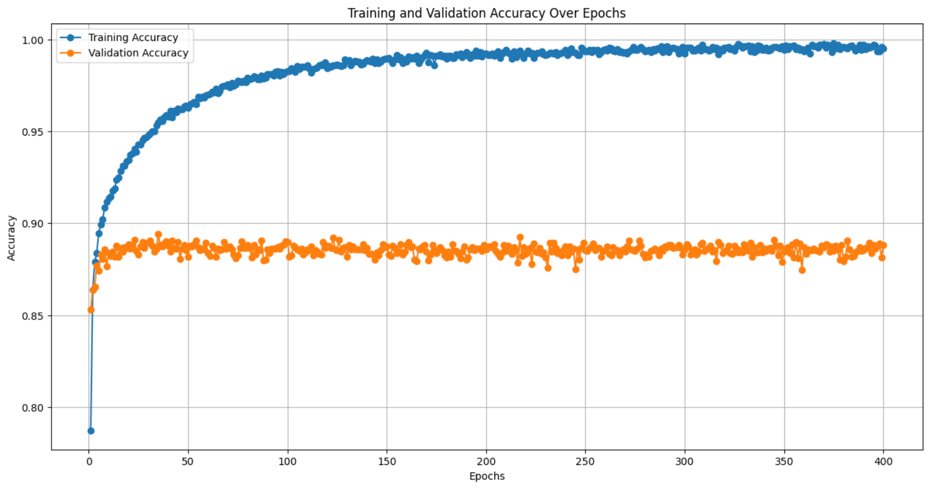


Figure 11: Comparison of accuracy values in training data with accuracy values in validation data in the NAdam algorithm

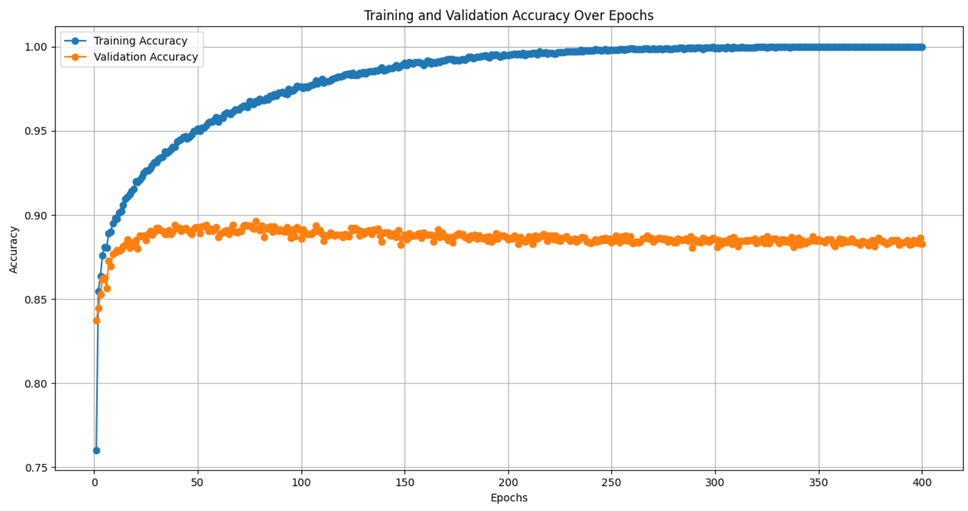


Figure 12: Comparison of accuracy values in training data with accuracy values in validation data in the AdamMax algorithm

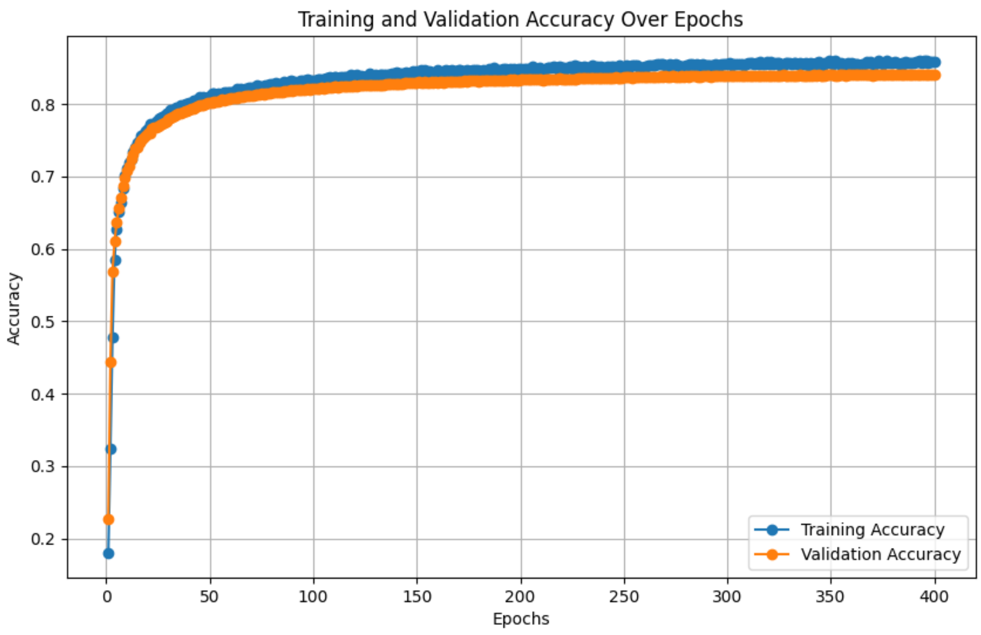


Figure 13: Comparison of accuracy values in training data with accuracy values in validation data in the Ftrl algorithm

At the same time, we collect data from the last epoch to compare the optimization algorithms.

Table 6: Comparison of algorithms

Algorithm	Epoch	Accuracy of Training	Accuracy of Validation
Adam	400	0.9953	0.8849
SGD	400	0.9934	0.8834
RMSProp	400	0.9936	0.8788
NAdam	400	0.9952	**0.8882**
AdamMax	400	**0.9999**	0.8826
Ftrl	400	0.8583	0.8408

Commentary on algorithm efficiency

From the analysis of the diagrams, various conclusions can be drawn regarding the effectiveness of the different algorithms applied to the neural network model. All algorithms achieve high training accuracy, with Adamax reaching almost 100%, indicating effective minimization of training losses. However, with the exception of the FTRL algorithm, most models exhibit overfitting, as evidenced by the notable discrepancy between training and validation accuracy. This suggests that while the models are highly accurate on the training data, they do not generalize well to new data, with RMSprop exhibiting the most severe overfitting. The FTRL algorithm exhibits the fastest convergence, reaching the peak of performance earlier than the others, although it also exhibits greater variance and a higher probability of overfitting. In terms of stability, NAdam, Adam, and Adamax provide the most stable performance, as indicated by smoother accuracy curves and lower variance.

To better understand these observations, it is useful to examine the mathematical formulations underlying each algorithm. The Stochastic Gradient Descent (SGD) algorithm updates weights iteratively based on the gradient approach, while RMSprop adjusts the learning rate using a moving average of the squared gradients to mitigate oscillations. Adam combines the benefits of SGD and RMSprop by using moving averages of both gradients and gradient squares. NAdam extends Adam by incorporating Nesterov momentum to accelerate convergence. Adamax, a variant of Adam, uses the infinite norm of previous gradients for stronger performance with sparse gradients. Finally, FTRL (Follow-the-Regularized-Leader) applies a combination of L1 and L2 regularization to the loss function, promoting sparsity and stability.

These mathematical clarifications explain the observed performance characteristics: Adam and its variants (NAdam and Adamax) offer a balanced approach with stable convergence, while RMSprop is more prone to overfitting due to its aggressive learning rate adjustment.

Convolutional Neural Network

In this exercise, we are asked to implement a more complex neural network, which is summarized in the following image:

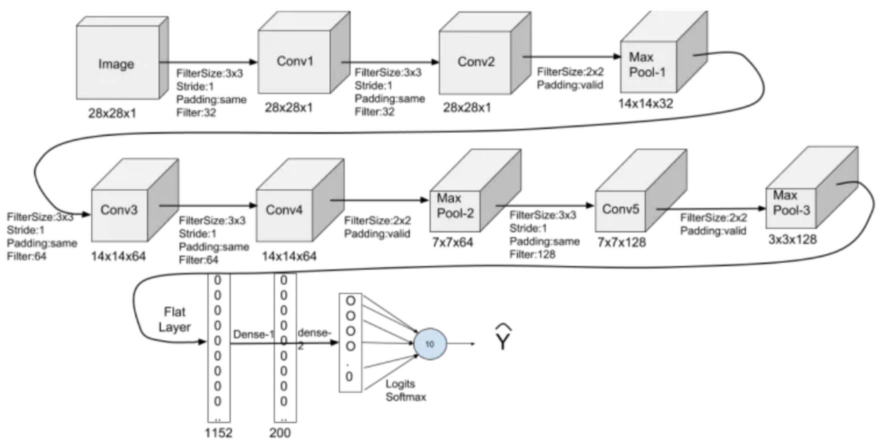


Figure 14: The neural network to be implemented

Implementing the artificial neural network in Python, we have:

Convolution Network

```
from __future__ import absolute_import, division, print_function,
 ↪  unicode_literals

# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
 ↪  BatchNormalization, Dropout

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

from plots import plot_some_data, plot_some_predictions

fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) =
 ↪  fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

```
# Scale these values to a range of 0 to 1 before feeding them to the
↪ neural network model.

train_images = train_images / 255.0
test_images = test_images / 255.0

num_train_images = train_images.shape[0]
num_test_images = test_images.shape[0]
height = 28
width = 28
channels = 1  # grayscale images have 1 channel

train_images_reshaped = train_images.reshape(num_train_images, height,
↪ width, channels)
test_images_reshaped = test_images.reshape(num_test_images, height, width,
↪ channels)

# Build the model
# Building the neural network requires configuring the layers of the model,
↪ then compiling the model.
model = Sequential([
    Conv2D(32, (3, 3), padding='same', input_shape=(28, 28, 1)),
    BatchNormalization(),
    keras.layers.ReLU(),

    Conv2D(32, (3, 3), padding='same'),
    BatchNormalization(),
    keras.layers.ReLU(),
    MaxPooling2D((2, 2)),
    Dropout(0.2),

    Conv2D(64, (3, 3), padding='same'),
    BatchNormalization(),
    keras.layers.ReLU(),

    Conv2D(64, (3, 3), padding='same'),
    BatchNormalization(),
    keras.layers.ReLU(),
    MaxPooling2D((2, 2)),
    Dropout(0.3),
    Conv2D(128, (3, 3), padding='same'),
    BatchNormalization(),
    keras.layers.ReLU(),
    MaxPooling2D((2, 2)),
    Dropout(0.4),

    Flatten(),
    Dense(200),
    BatchNormalization(),
    keras.layers.ReLU(),
    Dropout(0.5),

    Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])

# Train the model
model.fit(train_images_reshaped, train_labels, epochs=50,
 ↪  validation_data=(test_images_reshaped, test_labels))

# Evaluate accuracy
test_loss, test_acc = model.evaluate(test_images_reshaped, test_labels,
 ↪  verbose=2)

print('\nTest accuracy:', test_acc)

probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])

predictions = probability_model.predict(test_images_reshaped)

plot_some_predictions(test_images, test_labels, predictions, class_names)
```

We visualize our samples and we have:

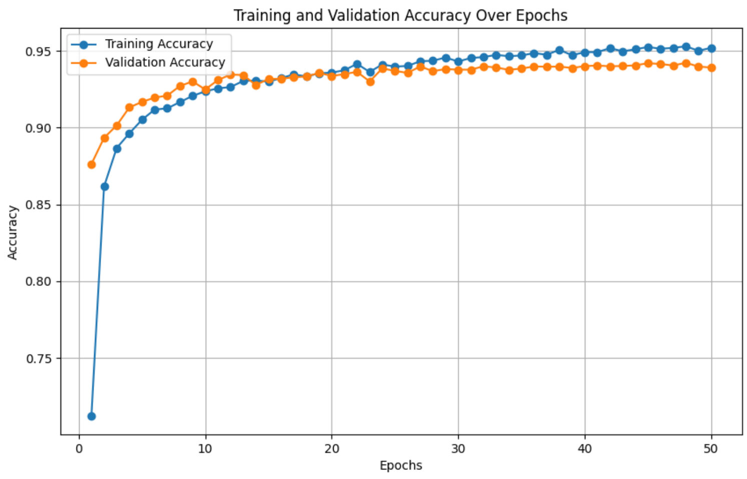


Figure 15: Neural network effectiveness

Commentary on a convolutional neural network

Our observations show that the neural network performs well on both training samples and validation data, successfully reducing the risk of overfitting. To fully understand the functionality of our algorithm, it is necessary to examine the technical details of the network architecture.

Conv2D layers form the basis of our neural network. These layers apply either 32 or 64 filters, depending on the specific configuration, and use ”same” padding to ensure that the output dimensions match the input dimensions. This design choice preserves the spatial resolution of the feature maps, which is critical for maintaining crucial edge and texture information throughout the network.

Batch normalization is used to normalize the output of each layer, thereby stabilizing and accelerating the training process. By correcting the internal shift of covariances, batch normalization allows for higher learning rates, ultimately improving the performance and convergence of the network.

MaxPooling layers are incorporated to downsample the spatial dimensions of feature maps by selecting the maximum value within a specified window, typically 2x2 in size. This function not only reduces the computational load, but also helps to highlight the most important features, allowing for faster convergence of the neural network.

Finally, the application of dropout regularization is vital to avoid overfitting of the model. Dropout works by randomly setting a fraction of the input units to zero at each update during training, forcing the network to learn more robust features that do not depend on specific neurons. This technique significantly improves the model’s generalization ability.

Appendix A

In this appendix, we present the architectures used to develop the first part of Exercise 5.

Neural Network with 5 layers

```
    network = [
    Dense(28 * 28, 256),
    Sigmoid(),
    Dense(256, 128),
    Sigmoid(),
    Dense(128, 64),
    Sigmoid(),
    Dense(64, 32),
    Sigmoid(),
    Dense(32, 10),
    Softmax()
]
```

Neural Network with 2 layers

```
    network = [
    Dense(28 * 28, 50),
    Sigmoid(),
    Dense(50, 10),
    Softmax()
]
```

Appendix B

The purpose of this appendix is to shed light on concepts that have already been mentioned, offering a deeper understanding of them.

Stochastic Gradient Descent (SGD)

- **Gradient Approximation**: SGD updates the model weights by calculating the gradient (partial derivatives) of the loss function with respect to the weights, using a small subset of the training data. This approach allows for faster updates compared to using the entire data set.
- **Weight update rule**:
$$w_{t+1} = w_t - \eta \nabla L(w_t)$$
  where
  - w_t: the weights during repetition t
  - η: learning rate
  - $\nabla L(w_t)$: slope of the loss function at w_t

RMSprop

- **Adapting to the rate of learning**: RMSprop adjusts the learning rate for each parameter using a moving average of the squared gradients to stabilize updates.
- **Update Rule**:
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$
where
  - $E[g^2]_t$: floating mean of square slopes
  - ϵ: small constant to avoid division by zero
  - g_t: slope in repetition t

Adam (Adaptive Moment Estimation)

- **Combines SGD and RMSprop**: Adam maintains the moving averages of both the slopes and their squares.
- **Update rule**:
$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \text{(floating mean of slopes)} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{(floating mean of the square gradients)} \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \quad \text{(bias-corrected first moment estimate)} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \quad \text{(bias-corrected second moment estimate)} \\ w_{t+1} &= w_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned}$$

NAdam (Nesterov-accelerated Adam)

- **Nesterov Momentum**: Επεκτείνει τον Adam ενσωματώνοντας την ορμή Nesterov, η οποία προσαρμόζει τις κλίσεις ώστε να βλέπει μπροστά στην κατεύθυνση του βήματος.
- **Κανόνας ενημέρωσης**: Παρόμοια με τον Adam, αλλά με έναν πρόσθετο όρο ορμής.

Adamax

- **Infinite Norm of Gradients**: A variant of Adam that uses the infinite norm (maximum absolute value) of gradients instead of the L2 norm (squared value) for stronger performance with sparse gradients.
- **Update rule**: Similar to Adam, but using the norm of infinity to estimate the second moment.

FTRL (Follow-the-Regularized-Leader)

- **L1 και L2 Regularization**: FTRL combines L1 and L2 regularization in the loss function, promoting both sparsity (L1) and stability (L2).
- **Update rule**: It includes normalization terms added to the standard descending gradient update to encourage rare solutions.

Infinite norm

We recall the family of p-norms for which the generalized form is as follows:

$$||x||_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}} \quad \text{με } p \geq 1 \tag{25}$$

So we have that the infinite norm or maximum norm $\mathbf{R}^n$ is defined as follows:

$$\|x\|_\infty = \max_{1 \leq k \leq n} |x_k| \tag{26}$$