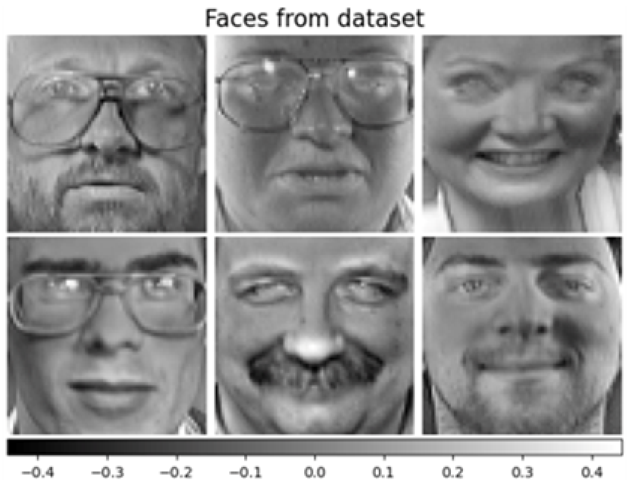
STAT 542 Midterm Project Report – Group 11

I. Problem Statement

As instructed, we will perform a supervised learning task focusing on image recognition. We select the Olivetti faces data set as the target of our analysis. Our objective is to accurately predict the identity of individuals featured in the Olivetti faces data set, which was developed by AT&T Laboratories Cambridge. This data set consists of 400 images, featuring 10 unique images of 40 different individuals. The images were captured at various times between April 1992 and April 1994, under different lighting conditions and facial expressions. However, all of the images were taken against a similar dark background, with subjects positioned front and center.



Each observation within the data set is labeled between 0 and 39, with 4,096 data columns representing every pixel in 64x64 images. Originally, the images were stored as unsigned 8-bit integers in a 256-gray-level format, but they were converted into floating values ranging from 0 to 1, to enhance compatibility with machine learning tasks. Our approach to training involves implementing three algorithms: Logistic Regression, Linear Discriminant Analysis (LDA), and Convolutional Neural Networks (CNN), in conjunction with Principal Component Analysis (PCA) to reduce dimensionality. A detailed explanation of each method can be found in the following section.

II. Methodology

1. Principal Component Analysis (PCA)

Principal component analysis (PCA) is a statistical method used to identify patterns in high-dimensional data by reducing the dimensionality of the data while retaining most of the variability in the data set. PCA works by transforming the original variables into a new set of variables, called principal components, which are linear combinations of the original variables. These principal components are chosen in such a way that they capture the maximum amount of variability in the data, while also being uncorrelated with each other. By reducing the dimensionality of the data, PCA can simplify complex data sets and make it easier to visualize and interpret the data. PCA is widely used in data analysis, machine learning, and pattern recognition applications, and is particularly useful when dealing with large data sets with many variables.

Each Principal Component z_i is a linear combination of the original variables $x_1, x_2, \dots, x_m$ with weights given by each column ui of matrix U:

$$z_i = u_{1i}x_1 + u_{2i}x_2 + \dots + u_{mi}x_m,$$

while U is the rotation matrix and Z is a version of the data rotated in such a way that the resulting principal components are orthogonal.

2. Logistic Regression Analysis

In logistic regression, the dependent variable (or response variable) is binary, usually represented by 0 and 1. The goal of logistic regression is to model the probability that the dependent variable equals 1 as a function of the independent variables (or predictors), which can be continuous, categorical or a mix of both. The logistic regression model uses a logistic function (also called a sigmoid function) to map the input variables to the output probabilities. The logistic regression model estimates the regression coefficients that best fit the training data by minimizing the log loss (also called cross-entropy loss) between the predicted probabilities and the true labels.

Once the model is trained, it can be used to predict the probability of the dependent variable being 1 for new input values, and the predicted probability can be converted into a binary decision using a threshold value (e.g., 0.5). If the predicted probability is above the threshold, the predicted label is 1, otherwise it is 0. In the case of multi-class classification problems, there are two common ways to extend logistic regression for multi-class classification: One-vs-all and Softmax regression approach.

3. Linear Discriminant Analysis (LDA)

LDA (Linear Discriminant Analysis) is a supervised machine learning algorithm that is commonly used for classification tasks. Like logistic regression, it is also used for binary classification problems, where the target variable takes on one of two possible values. However, LDA can also be extended to multi-class classification problems. The goal of LDA is to find a linear discriminant function that can distinguish between the different classes by projecting the data onto a lower-dimensional subspace.

The LDA algorithm works by first computing the mean and variance of each feature for each class in the training set. It then uses these statistics to estimate the between-class and within-class covariance matrices. The between-class covariance matrix measures the distance between the means of the different classes, while the within-class covariance matrix measures the spread of the data within each class. Next, LDA finds the eigenvectors and eigenvalues of the inverse of the within-class covariance matrix multiplied by the between-class covariance matrix. These eigenvectors are used to create the linear discriminant function that maximizes the separation between the classes. Finally, the LDA algorithm uses the linear discriminant function to classify new instances by projecting them onto the subspace defined by the eigenvectors and applying a decision rule to determine the predicted class label.

4. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized form of deep learning algorithm created explicitly for tasks in image processing and computer vision. They incorporate a blend of convolutional layers, which detect significant features within the input image, and pooling layers, which reduce the data's dimensionality. Typically, CNNs leverage activation functions and fully connected layers to carry out the final classification process. Within the convolutional layers, filters or kernels scan the image to pinpoint critical features such as edges, corners, and textures. Subsequently, these features are integrated and processed in successive layers to execute more intricate operations like object recognition and classification. An illustration of CNNs algorithm is included in **(Appendix 1)**. CNNs have demonstrated exceptional performance levels in a broad range of image processing applications, including image classification, object detection, and facial recognition.

III. Results and Comparisons

1. PCA

To perform dimensionality reduction while maintaining ideal accuracy, we included principal components up to a point where 90 percent of the variances could be explained, which is 60 components **(Appendix 2)**.

2. Logistics Regression and LDA

	Logistics Regression		**Linear Discriminant Analysis**	
	Accuracy	**Run Time**	**Accuracy**	**Run Time**
With PCA	**0.975**	**<u>0.093</u>**	**0.975**	**0.048**
Without PCA	**0.975**	**5.121**	**<u>0.986</u>**	**<u>0.491</u>**

In terms of accuracy, PCA does not improve the result of logistics regression. However, applying PCA sped up the process by a lot. For LDA, applying PCA barely affected the accuracy while reducing run time. Similarly, applying PCA reduced Logistics Regression running time, as well as slightly lowered its accuracy. Without PCA, LDA gave greater accuracy with a fair run time compared to logistics regression.

3. Convolutional Neural Network

	2 layers with 32, 64 nodes	**3 layers with 32, 64, 128 nodes**
Batch size = 32	**<u>0.9750</u>**	**0.9625**
Batch size = 64	**0.9375**	**0.9250**

The parameter tuning process would be specified in the next chapter. The optimal model has an accuracy of 0.975 (underlined). Other settings for the optimal model are as follows:

epoch = 25, a **3x3** kernel size of convolutional layer, **max-pooling** subsampling method (size 2x2), **relu** activation function and **adam** optimizer.

4. Comparison

Best Performance	Logistic Regression	LDA	CNN
Accuracy	**0.975**	**<u>0.986</u>**	**0.975**

While LDA had a slight edge, three models performed well on this dataset. The accuracy is comparable. However, we disregarded comparison of time consumption because CNNs, by its nature, usually take longer time than the others.

IV. Difficulties

1. Loss of Accuracy While Performing PCA

From the scree plot **(Appendix 3)**, we found that the ability to explain variances is relatively low starting from the eighth component. Therefore, originally we selected the first seven components to fit the model but the accuracy was bad. Finally, we ended up using 60 principal components as stated in the previous chapter.

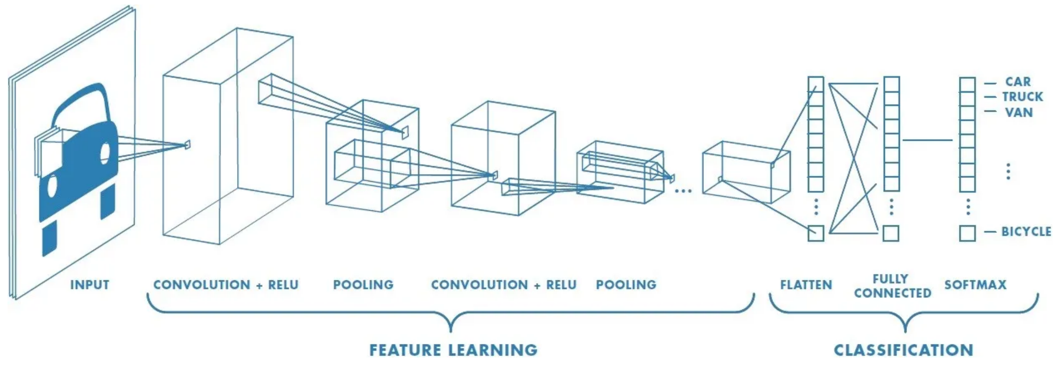
2. Parameter Tuning of CNN

<u>2 hidden layers with 32, 64 nodes</u>	**<u>Batch size of 32</u>**	**<u>25 epochs</u>**
3 layers with 32, 64 and 128 nodes	**Batch size of 64**	**60 epochs**

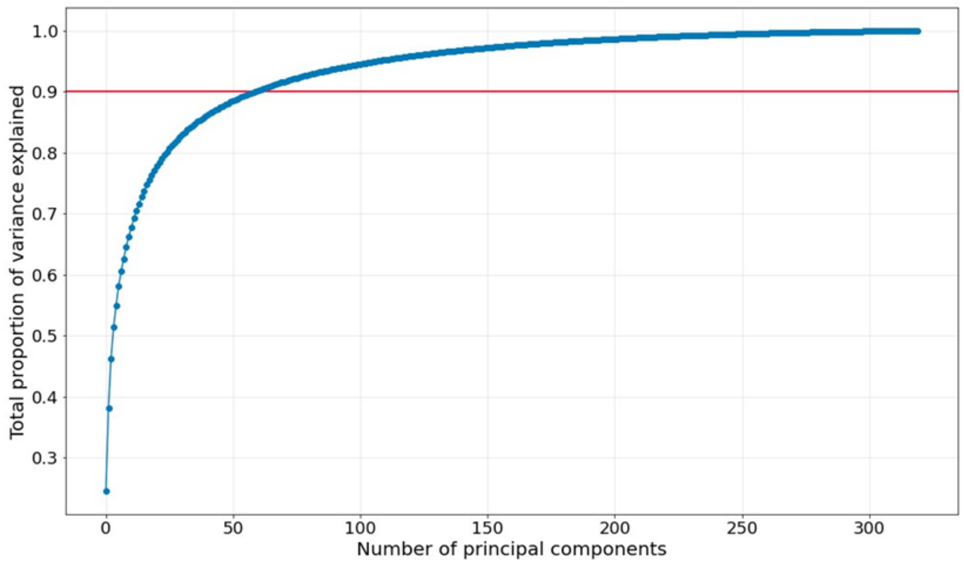
The table above consists of the different combinations of parameters that we have gone through. The underlined parameters would produce the optimal model. Some findings during the tuning process include:

1) Overfitting: We discovered that the third hidden layer would lead to overfitting (training accuracy = 1). Therefore, the third layer with 128 nodes was dropped.
2) The accuracy stopped improving long before epoch reached 60. Based on the graph **(Appendix 4)**, we let epoch = 25.

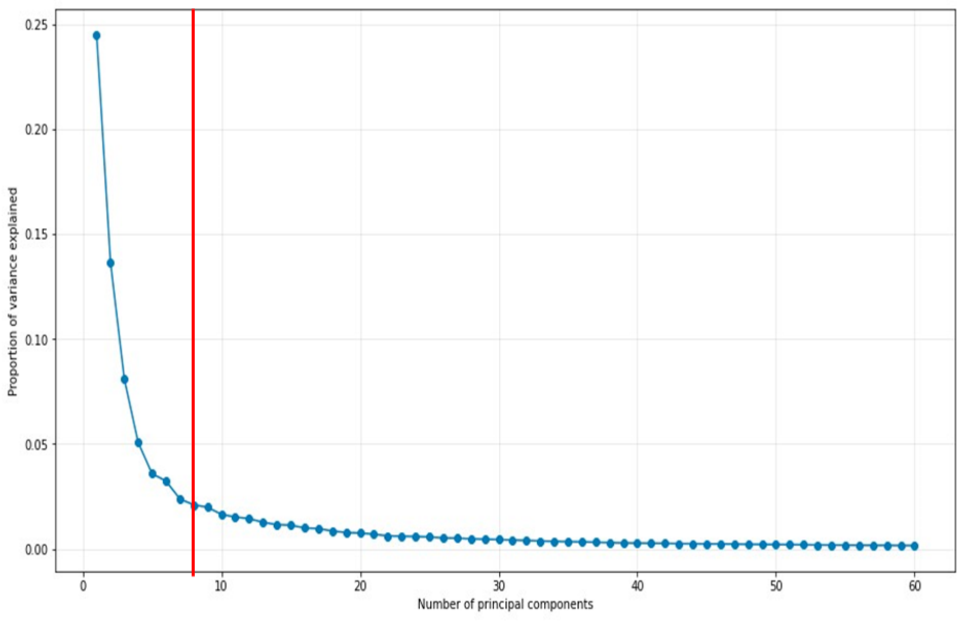
V. Appendices



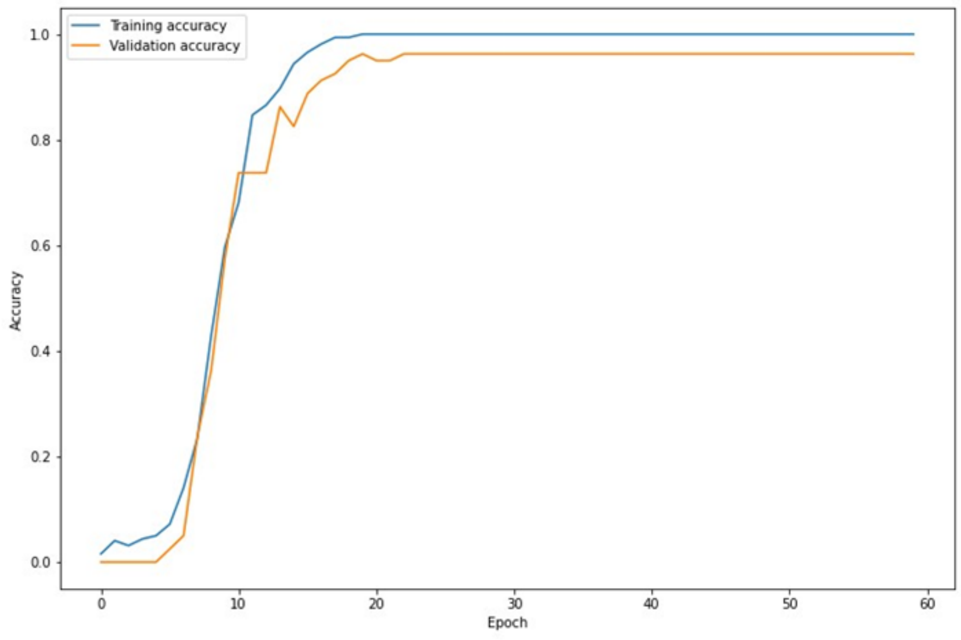
Appendix 1: Illustration of CNNs algorithm



Appendix 2: Principal Components and Cumulative Variance Explained



Appendix 3: Principal Components and Proportion of Variance Explained



Appendix 4: Number of Epoch and Accuracy

Appendix 5: Code and Result (attached below)

STAT 542: Midterm Project

Face Recognition

- Yu-Ching Liao ycliao3@illinois.edu

Basic Import

Package Import

In [1]:
```
# import necessary libraries and modules
from sklearn.datasets import fetch_olivetti_faces
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from keras.wrappers.scikit_learn import KerasClassifier
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
import time
```

Dataset Import

In [39]:
```
# load the Olivetti faces dataset
data = fetch_olivetti_faces()
X = data['data']
y = data['target']
```

Prepare train and test set

In [40]:
```
# split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
```

Statistical Learning

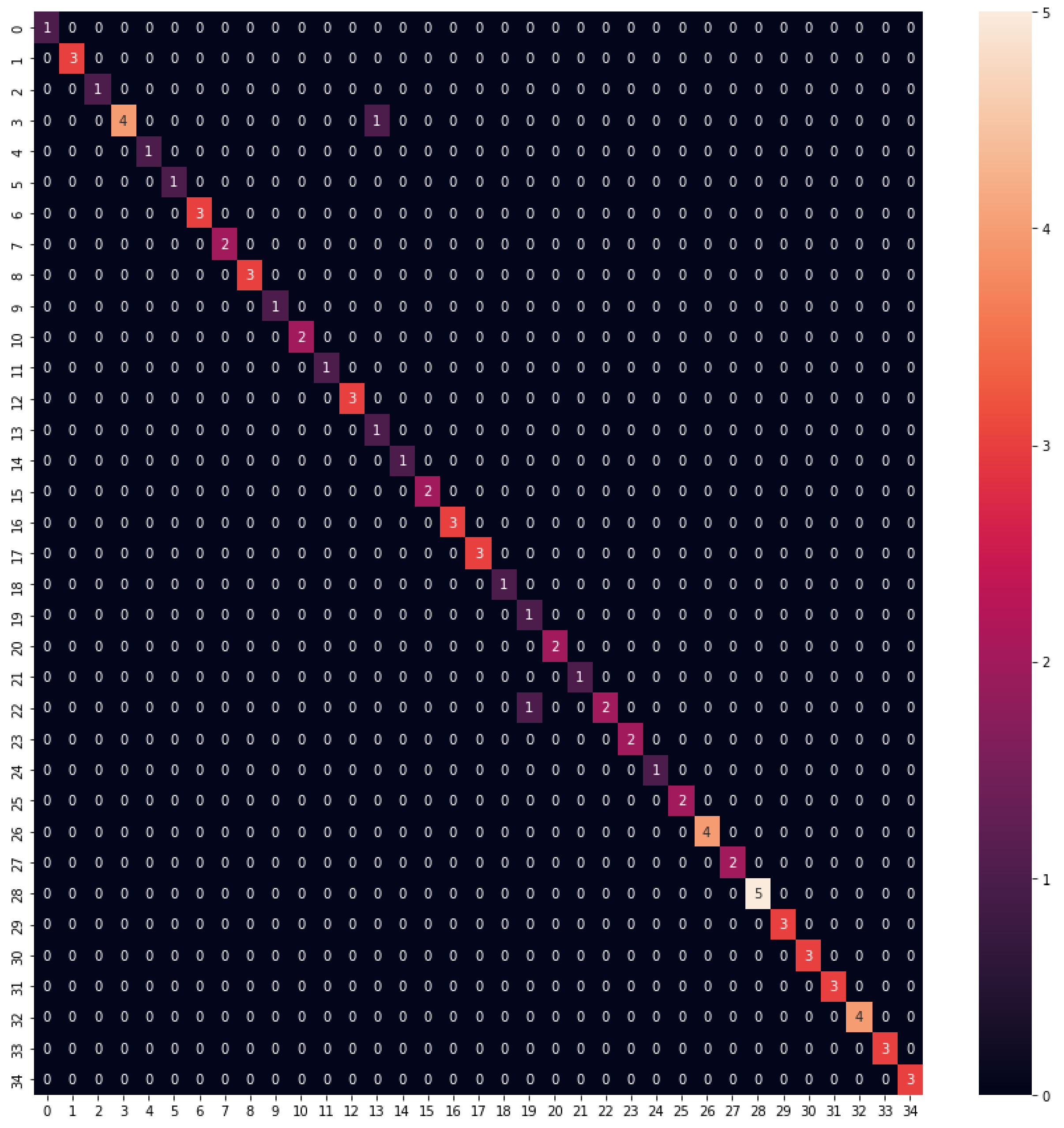
Running the Model without PCA

Logistic

In [6]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
import seaborn as sns

start = time.time()
# Create a logistic regression model
clf = LogisticRegression()

# Fit the model to the training data
clf.fit(X_train, y_train)

# Use the model to make predictions on the testing data
y_pred = clf.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
end = time.time()

print("Out-sample Accuracy for logistic regression:", accuracy)
print("Time Comsumption:", end - start, "sec.")

cm = confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots(figsize=(15,15))
sns.heatmap(cm, annot=True, fmt='d', ax=ax)
```

```
Accuracy for logistic regression: 0.975
Time Comsumption: 6.632861137390137 sec.
```

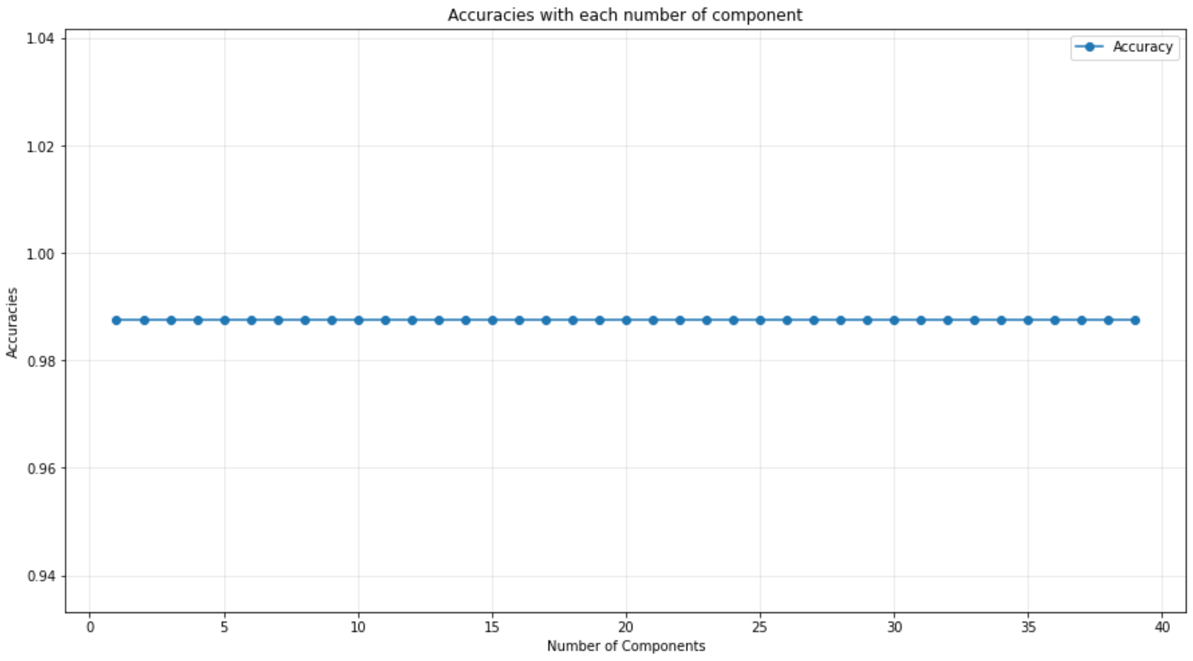
Out[6]:
```
<Axes: >
```



LDA

```
In [55]: #find the best parameters
         acc = []
         for n in range(1, 40):
           lda = LinearDiscriminantAnalysis(n_components=n)
           lda.fit(X_train, y_train)
           y_pred = lda.predict(X_test)
           accuracy = accuracy_score(y_test, y_pred)
           if n % 5 == 0:
             print("Out-sample Accuracy for LDA for", n, "components:", accuracy)

           acc.append(accuracy)
           accuracy = 0
         plt.figure(figsize=(15,8))
         plt.title("Accuracies with each number of component")
         plt.plot(range(1,40), acc, 'o-',  label = "Accuracy")
         plt.ylabel("Accuracies")
         plt.xlabel("Number of Components")
         plt.legend()
         plt.grid(alpha=0.3)
         plt.show()

         print("The best number of components:", max(acc))
```

```
Out-sample Accuracy for LDA for 5 components: 0.9875
Out-sample Accuracy for LDA for 10 components: 0.9875
Out-sample Accuracy for LDA for 15 components: 0.9875
Out-sample Accuracy for LDA for 20 components: 0.9875
Out-sample Accuracy for LDA for 25 components: 0.9875
Out-sample Accuracy for LDA for 30 components: 0.9875
Out-sample Accuracy for LDA for 35 components: 0.9875
```



```
The best number of components: 0.9875
```

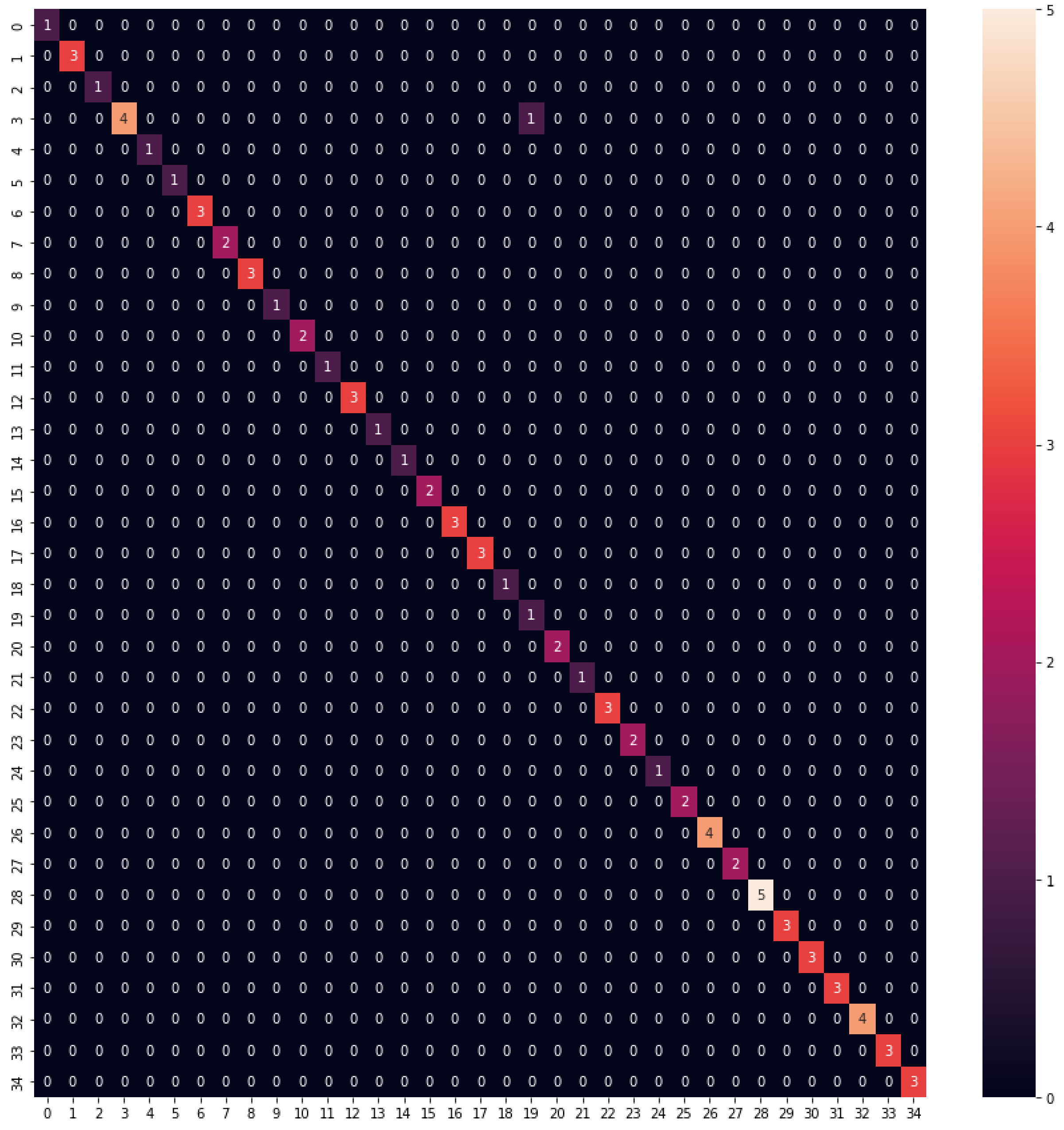
So basically all of those number of components have same performance so we are going to randomly pick a number ranged from 1 to 39 and see its performance.

In [8]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import accuracy_score
from mlxtend.plotting import plot_decision_regions

start = time.time()
# Create an instance of LDA
lda = LinearDiscriminantAnalysis(n_components=39)

# Train the LDA model
lda.fit(X_train, y_train)


# Make predictions on the test set
y_pred = lda.predict(X_test)
```

```
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

end = time.time()

# Print the accuracy
print("Out-sample Accuracy for LDA:", accuracy)
print("Time Comsumption:", end - start, "sec.")

cm = confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots(figsize=(15,15))
sns.heatmap(cm, annot=True, fmt='d', ax=ax)
```

```
Out-sample Accuracy for LDA: 0.9875
Time Comsumption: 0.794893741607666 sec.
```

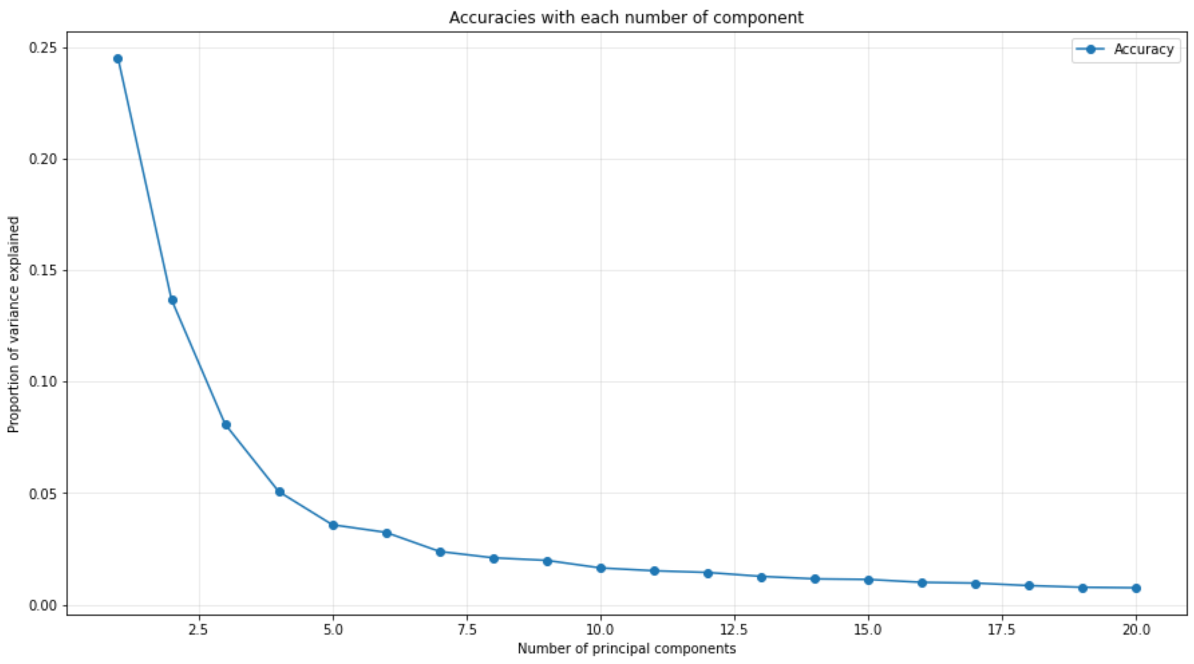
```
Out[8]: <Axes: >
```



Running the Model with PCA

PCA Visualization

In [56]:
```python
from sklearn.decomposition import PCA

# load the Olivetti faces dataset
data = fetch_olivetti_faces()
X = data['data']
y = data['target']

# split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran

pca = PCA(n_components=20)
X_train_pca = pca.fit_transform(X_train)

# Apply PCA to the testing set
X_test_pca = pca.transform(X_test)

# visualizw results
plt.figure(figsize=(15,8))
plt.plot(range(1, pca.n_components_+1), pca.explained_variance_ratio_, 'o-',
plt.title("Accuracies with each number of component")
plt.xlabel('Number of principal components')
plt.ylabel('Proportion of variance explained')
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

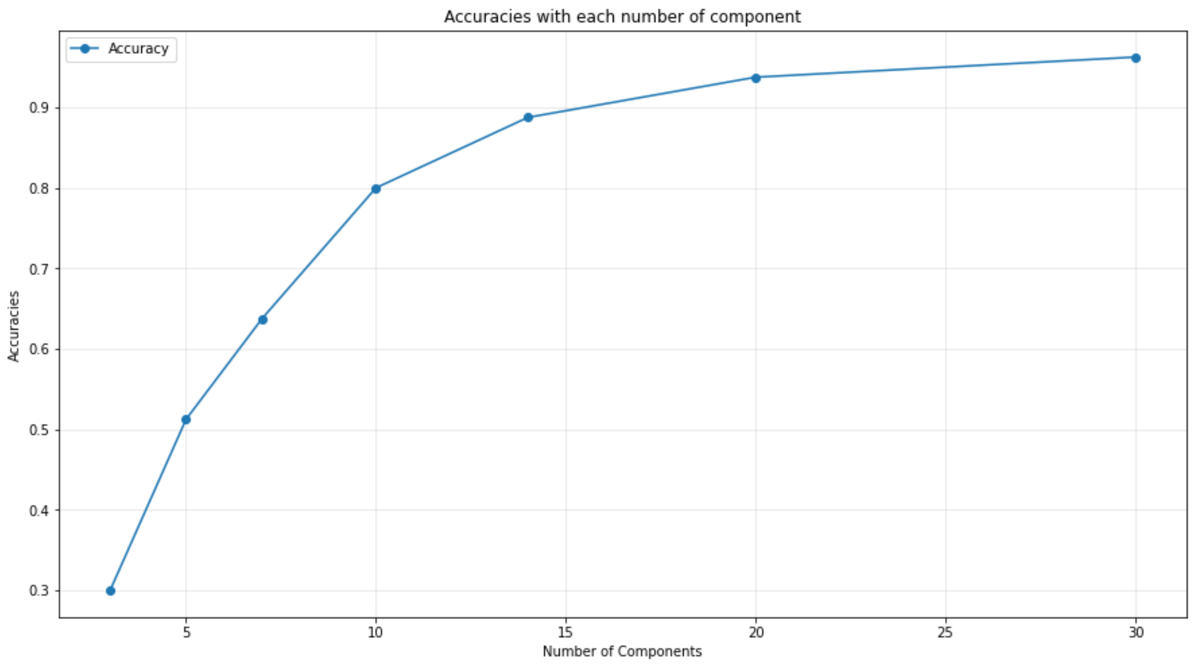


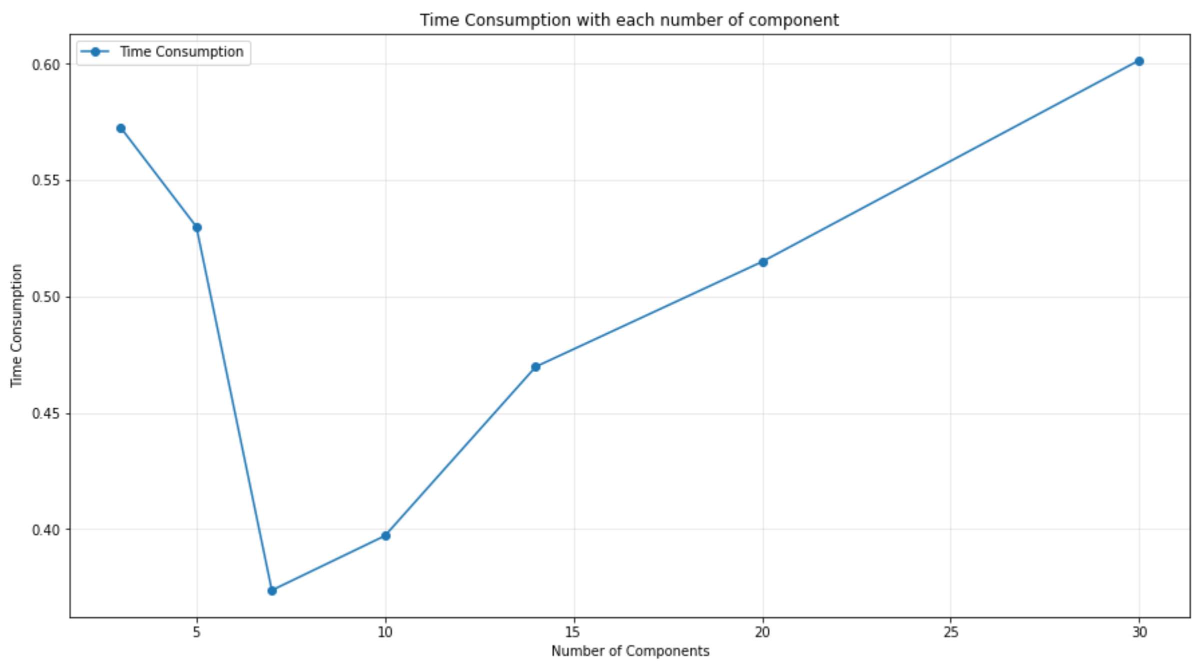
We are going to try the number of components with 3, 5, 7, 10, 14, 20, 30 to see the performance of PCA.

Logistic with PCA

In [62]:
```
acc = []
time_comp = []
for n in [3, 5, 7, 10, 14, 20, 30]:
  start = time.time()
  pca = PCA(n_components=n)
  X_train_pca = pca.fit_transform(X_train)
  X_test_pca = pca.transform(X_test)
  clf = LogisticRegression()
  clf.fit(X_train_pca, y_train)
  y_pred = clf.predict(X_test_pca)
  accuracy = accuracy_score(y_test, y_pred)
  end = time.time()
  print("Number of components:", n, ", Out-sample Accuracy for logistic regr
  print("Time Comsumption:", end - start, "sec.\n")
  acc.append(accuracy)
  time_comp.append(end - start)

plt.figure(figsize=(15,8))
plt.plot([3, 5, 7, 10, 14, 20, 30], acc, 'o-', label = 'Accuracy')
plt.ylabel("Accuracies")
plt.xlabel("Number of Components")
plt.title("Accuracies with each number of component")
plt.legend()
plt.grid(alpha=0.3)
plt.show()

plt.figure(figsize=(15,8))
plt.plot([3, 5, 7, 10, 14, 20, 30], time_comp, 'o-', label = 'Time Consumpti
plt.ylabel("Time Consumption")
plt.xlabel("Number of Components")
plt.title("Time Consumption with each number of component")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

```
Number of components: 3 , Out-sample Accuracy for logistic regression with
PCA: 0.3
Time Comsumption: 0.5724036693572998 sec.

Number of components: 5 , Out-sample Accuracy for logistic regression with
PCA: 0.5125
Time Comsumption: 0.5299580097198486 sec.

Number of components: 7 , Out-sample Accuracy for logistic regression with
PCA: 0.6375
Time Comsumption: 0.3737456798553467 sec.

Number of components: 10 , Out-sample Accuracy for logistic regression with
PCA: 0.8
Time Comsumption: 0.39716148376464844 sec.

Number of components: 14 , Out-sample Accuracy for logistic regression with
PCA: 0.8875
Time Comsumption: 0.46981143951416016 sec.

Number of components: 20 , Out-sample Accuracy for logistic regression with
PCA: 0.9375
Time Comsumption: 0.514904260635376 sec.

Number of components: 30 , Out-sample Accuracy for logistic regression with
PCA: 0.9625
Time Comsumption: 0.6014773845672607 sec.
```



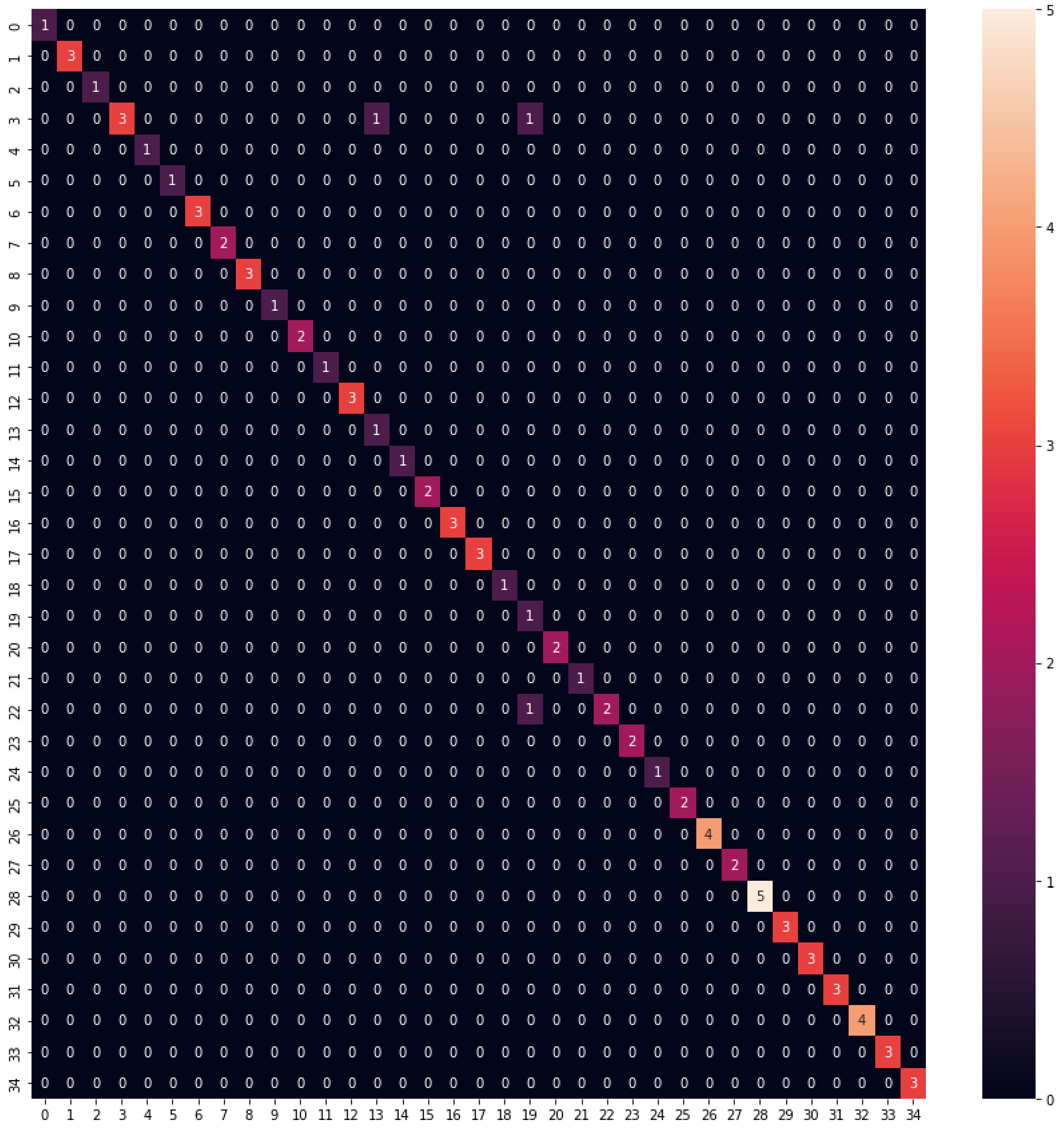


It seemed like the differences among time consumptions are not significant so we will try the number of conponents that have largest accuracy.

```
In [67]: start = time.time()

#Implementing PCA
pca = PCA(n_components=30)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create a logistic regression model
clf = LogisticRegression()

# Fit the model to the training data
clf.fit(X_train_pca, y_train)

# Use the model to make predictions on the testing data
y_pred = clf.predict(X_test_pca)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
end = time.time()

print("Number of components: 30")
print("Out-sample Accuracy for logistic regression with PCA:", accuracy)
print("Time Comsumption:", end - start, "sec.")

cm = confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots(figsize=(15,15))
sns.heatmap(cm, annot=True, fmt='d', ax=ax)
```

```
Number of components: 30
Out-sample Accuracy for logistic regression with PCA: 0.9625
Time Comsumption: 0.4608933925628662 sec.
```

Out[67]: <Axes: >

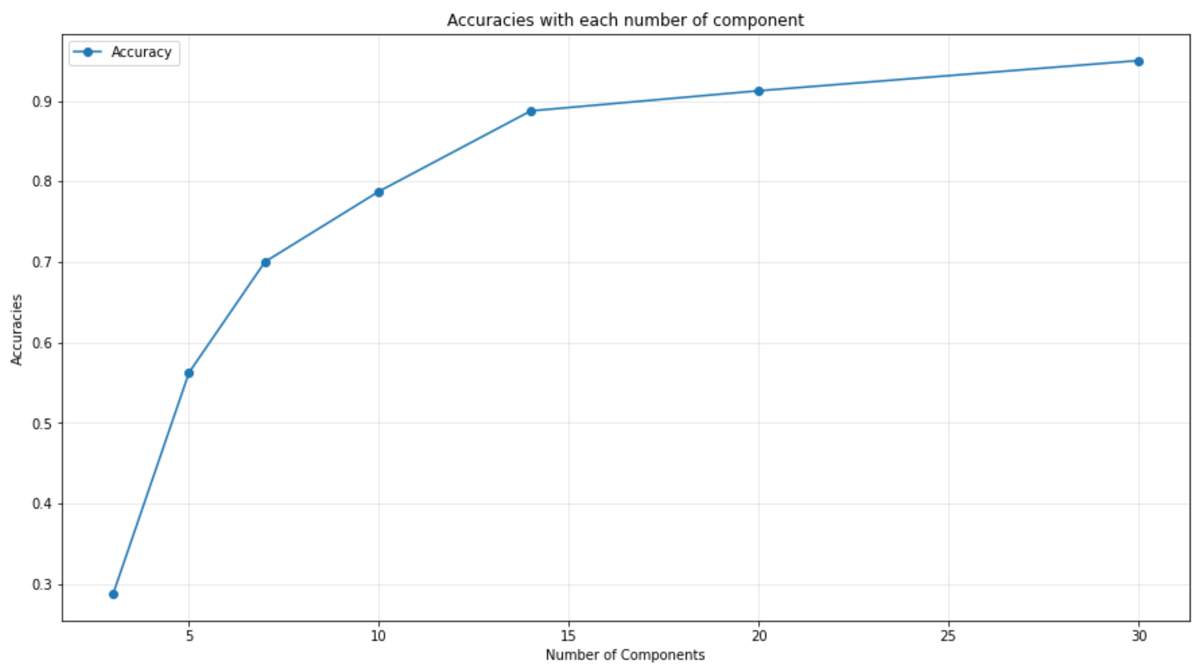


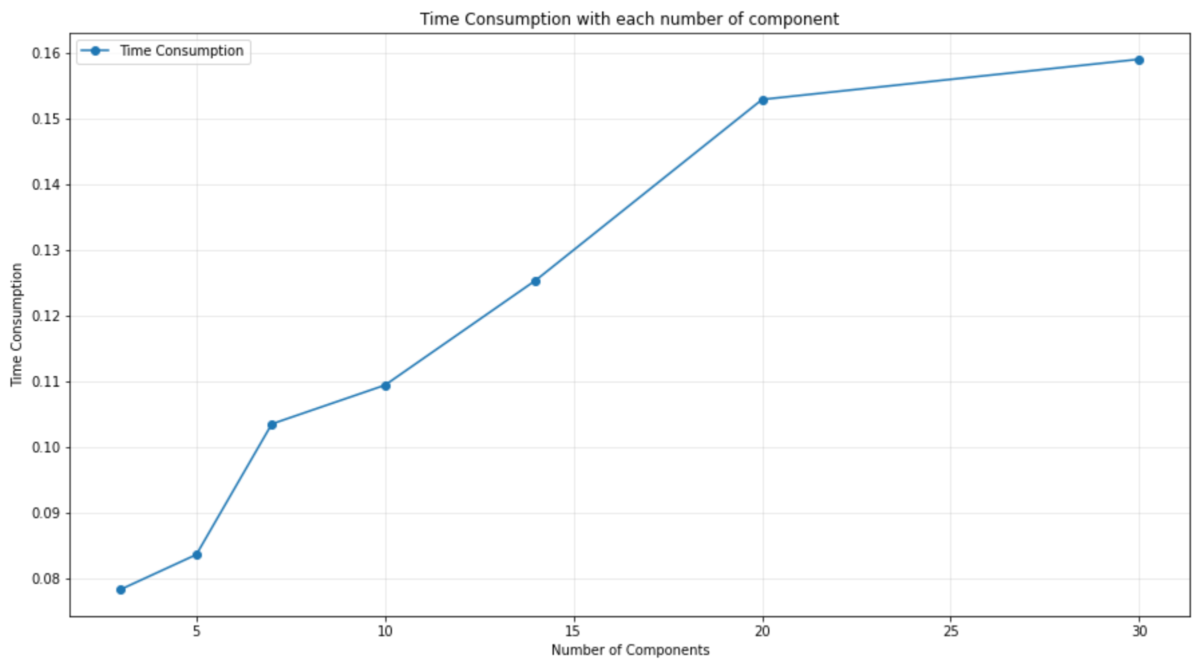
LDA with PCA

In [64]:
```
acc = []
time_comp = []
for n in [3, 5, 7, 10, 14, 20, 30]:
  start = time.time()
  pca = PCA(n_components=n)
  X_train_pca = pca.fit_transform(X_train)
  X_test_pca = pca.transform(X_test)
  lda = LinearDiscriminantAnalysis(n_components=n-2)
  lda.fit(X_train_pca, y_train)
  y_pred = lda.predict(X_test_pca)
  accuracy = accuracy_score(y_test, y_pred)
  end = time.time()
  print("Number of components:", n, ", Out-sample Accuracy for LDA with PCA:
```

```
  print("Time Comsumption:", end - start, "sec.\n")
  acc.append(accuracy)
  time_comp.append(end - start)

plt.figure(figsize=(15,8))
plt.plot([3, 5, 7, 10, 14, 20, 30], acc, 'o-', label = 'Accuracy')
plt.ylabel("Accuracies")
plt.xlabel("Number of Components")
plt.title("Accuracies with each number of component")
plt.legend()
plt.grid(alpha=0.3)
plt.show()

plt.figure(figsize=(15,8))
plt.plot([3, 5, 7, 10, 14, 20, 30], time_comp, 'o-', label = 'Time Consumpti
plt.ylabel("Time Consumption")
plt.xlabel("Number of Components")
plt.title("Time Consumption with each number of component")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

```
Number of components: 3 , Out-sample Accuracy for LDA with PCA: 0.2875
Time Comsumption: 0.07838582992553711 sec.

Number of components: 5 , Out-sample Accuracy for LDA with PCA: 0.5625
Time Comsumption: 0.08365988731384277 sec.

Number of components: 7 , Out-sample Accuracy for LDA with PCA: 0.7
Time Comsumption: 0.10351896286010742 sec.

Number of components: 10 , Out-sample Accuracy for LDA with PCA: 0.7875
Time Comsumption: 0.10941696166992188 sec.

Number of components: 14 , Out-sample Accuracy for LDA with PCA: 0.8875
Time Comsumption: 0.12537312507629395 sec.

Number of components: 20 , Out-sample Accuracy for LDA with PCA: 0.9125
Time Comsumption: 0.1528491973876953 sec.

Number of components: 30 , Out-sample Accuracy for LDA with PCA: 0.95
Time Comsumption: 0.1589670181274414 sec.
```





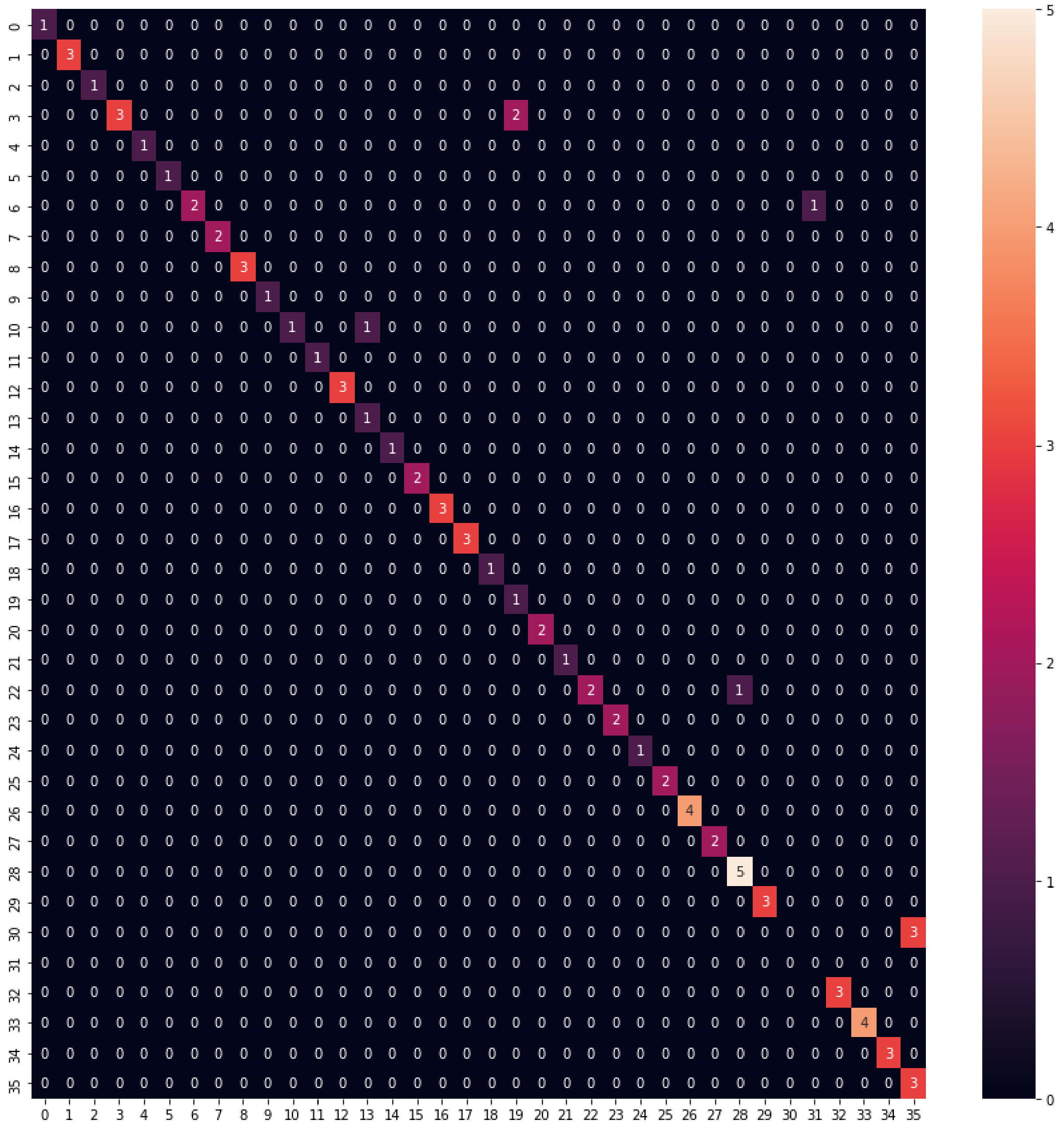
The time consumption will increase nearly twice as the number of components grown from 3 to 30. However, it seemed like the accuracies cease growing since the number of components reached 15. Thus we try number of components with 15 following.

In [66]:
```
start = time.time()

#PCA
pca = PCA(n_components=15)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create an instance of LDA
lda = LinearDiscriminantAnalysis(n_components=5)

# Train the LDA model
```

```
lda.fit(X_train_pca, y_train)


# Make predictions on the test set
y_pred = lda.predict(X_test_pca)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

end = time.time()

# Print the accuracy
print("Number of conponents: 15")
print("Out-sample Accuracy for LDA:", accuracy)
print("Time Comsumption:", end - start, "sec.")

cm = confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots(figsize=(15,15))
sns.heatmap(cm, annot=True, fmt='d', ax=ax)
```

```
Number of conponents: 15
Out-sample Accuracy for LDA: 0.9
Time Comsumption: 0.28126072883605957 sec.
```

```
Out[66]: <Axes: >
```



```
In [ ]:
```

Deep Learning: CNN

```
In [14]:
```

```python
# Import the necessary libraries
import numpy as np
from sklearn.datasets import fetch_olivetti_faces
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, models

# Load the Olivetti faces dataset
dataset = fetch_olivetti_faces()
X = dataset.data.reshape(-1, 64, 64, 1)  # Reshape the data into images
y = dataset.target
```

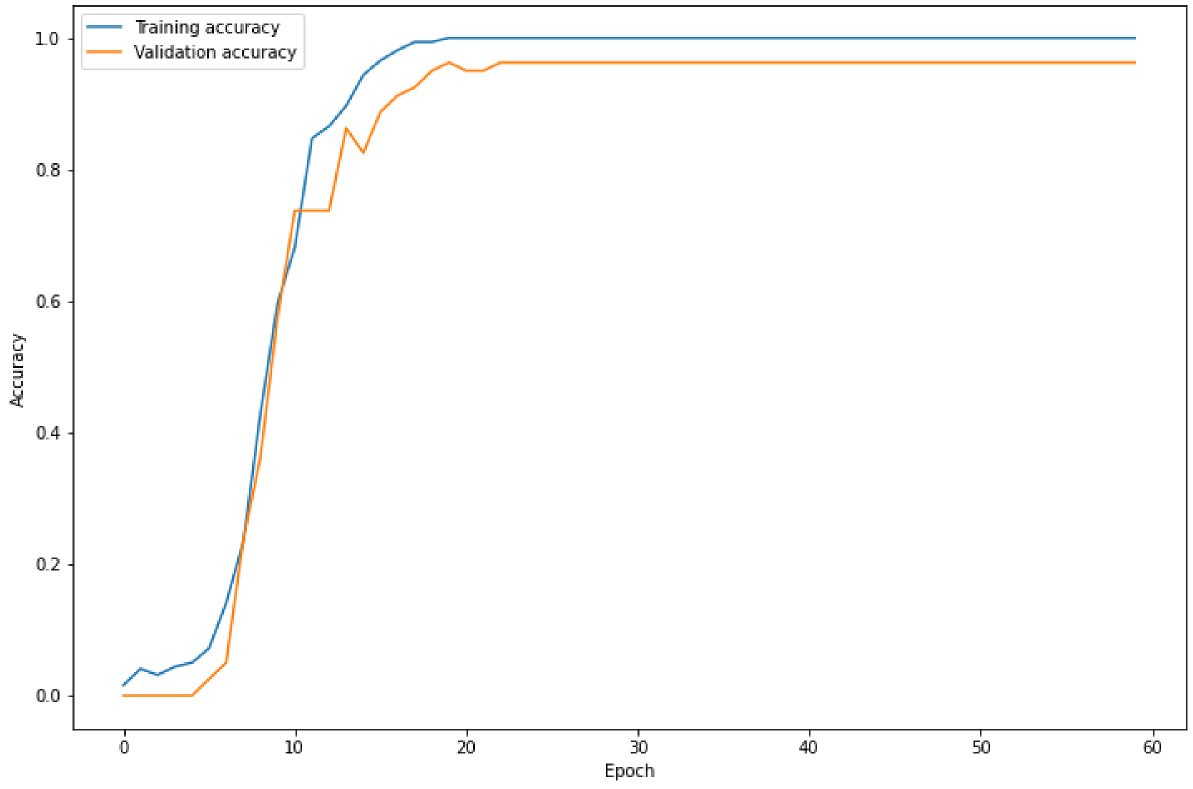
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

start = time.time()
# Define the CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(40, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=60, batch_size=32, validation_c


# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(X_test, y_test)

end = time.time()
print('Out-sample Test accuracy for CNN:', test_acc)
print("Time Comsumption:", end - start, "sec.")

# Plot the training and validation accuracy history
plt.figure(figsize=(12,8))
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
Epoch 1/60
10/10 [==============================] - 3s 228ms/step - loss: 3.7012 - acc
uracy: 0.0156 - val_loss: 3.6956 - val_accuracy: 0.0000e+00
Epoch 2/60
10/10 [==============================] - 2s 206ms/step - loss: 3.6867 - acc
uracy: 0.0406 - val_loss: 3.7047 - val_accuracy: 0.0000e+00
Epoch 3/60
10/10 [==============================] - 2s 204ms/step - loss: 3.6826 - acc
uracy: 0.0312 - val_loss: 3.7319 - val_accuracy: 0.0000e+00
Epoch 4/60
10/10 [==============================] - 3s 328ms/step - loss: 3.6654 - acc
uracy: 0.0437 - val_loss: 3.7314 - val_accuracy: 0.0000e+00
Epoch 5/60
10/10 [==============================] - 2s 212ms/step - loss: 3.6479 - acc
uracy: 0.0500 - val_loss: 3.7185 - val_accuracy: 0.0000e+00
Epoch 6/60
10/10 [==============================] - 2s 210ms/step - loss: 3.5900 - acc
uracy: 0.0719 - val_loss: 3.6808 - val_accuracy: 0.0250
Epoch 7/60
10/10 [==============================] - 2s 209ms/step - loss: 3.3892 - acc
uracy: 0.1406 - val_loss: 3.5236 - val_accuracy: 0.0500
Epoch 8/60
10/10 [==============================] - 2s 210ms/step - loss: 2.9935 - acc
uracy: 0.2344 - val_loss: 3.0909 - val_accuracy: 0.2375
Epoch 9/60
10/10 [==============================] - 2s 222ms/step - loss: 2.2842 - acc
uracy: 0.4281 - val_loss: 2.5594 - val_accuracy: 0.3625
Epoch 10/60
10/10 [==============================] - 4s 423ms/step - loss: 1.5825 - acc
uracy: 0.5969 - val_loss: 1.6420 - val_accuracy: 0.5750
Epoch 11/60
10/10 [==============================] - 4s 370ms/step - loss: 1.1012 - acc
uracy: 0.6812 - val_loss: 1.0202 - val_accuracy: 0.7375
Epoch 12/60
10/10 [==============================] - 4s 379ms/step - loss: 0.6678 - acc
uracy: 0.8469 - val_loss: 1.0121 - val_accuracy: 0.7375
Epoch 13/60
10/10 [==============================] - 4s 420ms/step - loss: 0.4729 - acc
uracy: 0.8656 - val_loss: 0.9264 - val_accuracy: 0.7375
Epoch 14/60
10/10 [==============================] - 4s 355ms/step - loss: 0.3435 - acc
uracy: 0.8969 - val_loss: 0.4453 - val_accuracy: 0.8625
Epoch 15/60
10/10 [==============================] - 2s 225ms/step - loss: 0.2453 - acc
uracy: 0.9438 - val_loss: 0.4538 - val_accuracy: 0.8250
Epoch 16/60
10/10 [==============================] - 2s 211ms/step - loss: 0.1426 - acc
uracy: 0.9656 - val_loss: 0.4761 - val_accuracy: 0.8875
Epoch 17/60
10/10 [==============================] - 2s 215ms/step - loss: 0.1031 - acc
uracy: 0.9812 - val_loss: 0.3277 - val_accuracy: 0.9125
Epoch 18/60
10/10 [==============================] - 3s 337ms/step - loss: 0.0666 - acc
uracy: 0.9937 - val_loss: 0.3167 - val_accuracy: 0.9250
Epoch 19/60
10/10 [==============================] - 2s 207ms/step - loss: 0.0471 - acc
```

```
uracy: 0.9937 - val_loss: 0.2262 - val_accuracy: 0.9500
Epoch 20/60
10/10 [==============================] - 2s 213ms/step - loss: 0.0283 - acc
uracy: 1.0000 - val_loss: 0.1460 - val_accuracy: 0.9625
Epoch 21/60
10/10 [==============================] - 2s 222ms/step - loss: 0.0189 - acc
uracy: 1.0000 - val_loss: 0.2014 - val_accuracy: 0.9500
Epoch 22/60
10/10 [==============================] - 2s 212ms/step - loss: 0.0112 - acc
uracy: 1.0000 - val_loss: 0.1818 - val_accuracy: 0.9500
Epoch 23/60
10/10 [==============================] - 3s 295ms/step - loss: 0.0091 - acc
uracy: 1.0000 - val_loss: 0.1586 - val_accuracy: 0.9625
Epoch 24/60
10/10 [==============================] - 3s 246ms/step - loss: 0.0072 - acc
uracy: 1.0000 - val_loss: 0.1492 - val_accuracy: 0.9625
Epoch 25/60
10/10 [==============================] - 2s 211ms/step - loss: 0.0068 - acc
uracy: 1.0000 - val_loss: 0.1343 - val_accuracy: 0.9625
Epoch 26/60
10/10 [==============================] - 2s 214ms/step - loss: 0.0051 - acc
uracy: 1.0000 - val_loss: 0.1554 - val_accuracy: 0.9625
Epoch 27/60
10/10 [==============================] - 2s 213ms/step - loss: 0.0046 - acc
uracy: 1.0000 - val_loss: 0.1454 - val_accuracy: 0.9625
Epoch 28/60
10/10 [==============================] - 2s 210ms/step - loss: 0.0041 - acc
uracy: 1.0000 - val_loss: 0.1333 - val_accuracy: 0.9625
Epoch 29/60
10/10 [==============================] - 3s 344ms/step - loss: 0.0036 - acc
uracy: 1.0000 - val_loss: 0.1405 - val_accuracy: 0.9625
Epoch 30/60
10/10 [==============================] - 2s 215ms/step - loss: 0.0033 - acc
uracy: 1.0000 - val_loss: 0.1398 - val_accuracy: 0.9625
Epoch 31/60
10/10 [==============================] - 2s 214ms/step - loss: 0.0030 - acc
uracy: 1.0000 - val_loss: 0.1377 - val_accuracy: 0.9625
Epoch 32/60
10/10 [==============================] - 2s 211ms/step - loss: 0.0028 - acc
uracy: 1.0000 - val_loss: 0.1403 - val_accuracy: 0.9625
Epoch 33/60
10/10 [==============================] - 2s 216ms/step - loss: 0.0027 - acc
uracy: 1.0000 - val_loss: 0.1415 - val_accuracy: 0.9625
Epoch 34/60
10/10 [==============================] - 3s 300ms/step - loss: 0.0025 - acc
uracy: 1.0000 - val_loss: 0.1385 - val_accuracy: 0.9625
Epoch 35/60
10/10 [==============================] - 3s 252ms/step - loss: 0.0023 - acc
uracy: 1.0000 - val_loss: 0.1375 - val_accuracy: 0.9625
Epoch 36/60
10/10 [==============================] - 3s 296ms/step - loss: 0.0022 - acc
uracy: 1.0000 - val_loss: 0.1405 - val_accuracy: 0.9625
Epoch 37/60
10/10 [==============================] - 4s 359ms/step - loss: 0.0021 - acc
uracy: 1.0000 - val_loss: 0.1403 - val_accuracy: 0.9625
Epoch 38/60
```

```
10/10 [==============================] - 2s 211ms/step - loss: 0.0020 - acc
uracy: 1.0000 - val_loss: 0.1419 - val_accuracy: 0.9625
Epoch 39/60
10/10 [==============================] - 3s 337ms/step - loss: 0.0019 - acc
uracy: 1.0000 - val_loss: 0.1406 - val_accuracy: 0.9625
Epoch 40/60
10/10 [==============================] - 2s 211ms/step - loss: 0.0018 - acc
uracy: 1.0000 - val_loss: 0.1391 - val_accuracy: 0.9625
Epoch 41/60
10/10 [==============================] - 2s 210ms/step - loss: 0.0017 - acc
uracy: 1.0000 - val_loss: 0.1428 - val_accuracy: 0.9625
Epoch 42/60
10/10 [==============================] - 2s 213ms/step - loss: 0.0016 - acc
uracy: 1.0000 - val_loss: 0.1415 - val_accuracy: 0.9625
Epoch 43/60
10/10 [==============================] - 2s 212ms/step - loss: 0.0015 - acc
uracy: 1.0000 - val_loss: 0.1431 - val_accuracy: 0.9625
Epoch 44/60
10/10 [==============================] - 3s 312ms/step - loss: 0.0014 - acc
uracy: 1.0000 - val_loss: 0.1428 - val_accuracy: 0.9625
Epoch 45/60
10/10 [==============================] - 2s 231ms/step - loss: 0.0014 - acc
uracy: 1.0000 - val_loss: 0.1404 - val_accuracy: 0.9625
Epoch 46/60
10/10 [==============================] - 2s 212ms/step - loss: 0.0013 - acc
uracy: 1.0000 - val_loss: 0.1435 - val_accuracy: 0.9625
Epoch 47/60
10/10 [==============================] - 2s 213ms/step - loss: 0.0013 - acc
uracy: 1.0000 - val_loss: 0.1444 - val_accuracy: 0.9625
Epoch 48/60
10/10 [==============================] - 2s 213ms/step - loss: 0.0012 - acc
uracy: 1.0000 - val_loss: 0.1439 - val_accuracy: 0.9625
Epoch 49/60
10/10 [==============================] - 2s 218ms/step - loss: 0.0011 - acc
uracy: 1.0000 - val_loss: 0.1442 - val_accuracy: 0.9625
Epoch 50/60
10/10 [==============================] - 3s 333ms/step - loss: 0.0011 - acc
uracy: 1.0000 - val_loss: 0.1469 - val_accuracy: 0.9625
Epoch 51/60
10/10 [==============================] - 2s 215ms/step - loss: 0.0011 - acc
uracy: 1.0000 - val_loss: 0.1420 - val_accuracy: 0.9625
Epoch 52/60
10/10 [==============================] - 2s 211ms/step - loss: 0.0010 - acc
uracy: 1.0000 - val_loss: 0.1464 - val_accuracy: 0.9625
Epoch 53/60
10/10 [==============================] - 2s 213ms/step - loss: 9.7355e-04 -
accuracy: 1.0000 - val_loss: 0.1446 - val_accuracy: 0.9625
Epoch 54/60
10/10 [==============================] - 2s 213ms/step - loss: 9.3644e-04 -
accuracy: 1.0000 - val_loss: 0.1475 - val_accuracy: 0.9625
Epoch 55/60
10/10 [==============================] - 3s 321ms/step - loss: 8.9502e-04 -
accuracy: 1.0000 - val_loss: 0.1475 - val_accuracy: 0.9625
Epoch 56/60
10/10 [==============================] - 2s 238ms/step - loss: 8.6761e-04 -
accuracy: 1.0000 - val_loss: 0.1458 - val_accuracy: 0.9625
```

```
Epoch 57/60
10/10 [==============================] - 2s 209ms/step - loss: 8.3691e-04 -
accuracy: 1.0000 - val_loss: 0.1482 - val_accuracy: 0.9625
Epoch 58/60
10/10 [==============================] - 2s 209ms/step - loss: 8.0988e-04 -
accuracy: 1.0000 - val_loss: 0.1491 - val_accuracy: 0.9625
Epoch 59/60
10/10 [==============================] - 2s 223ms/step - loss: 7.7744e-04 -
accuracy: 1.0000 - val_loss: 0.1458 - val_accuracy: 0.9625
Epoch 60/60
10/10 [==============================] - 2s 224ms/step - loss: 7.4348e-04 -
accuracy: 1.0000 - val_loss: 0.1464 - val_accuracy: 0.9625
3/3 [==============================] - 0s 42ms/step - loss: 0.1464 - accura
cy: 0.9625
Out-sample Test accuracy for CNN: 0.9624999761581421
Time Comsumption: 203.26114439964294 sec.
```



The accuracies reached its limit after 20 of epoches, which is 96.25% of accuracy.