1. Project Problem Statement

US Department of transportation is converting an existing state highway US 60/93 from Phoenix, AZ to Las Vegas, NV via Wickenburg, AZ and Kingsman, AZ into a new Interstate highway I-11. It is a 4 lane highway, which passes through green farms, cattle farms and ranches, near Wickenburg, AZ. Recently lots of traffic incidents have been reported where pet animals especially dogs and horses have been found wandering and crossing the highway. This has caused a sudden rise in safety related incidents, frequent collisions and accidents.

DOT wants to establish an image recognition solution which could do following:

1. Ability to capture images of pets and stray animals crossing the Interstate Highway.
2. Ability to correctly identify dogs and horses from the images being taken
3. Ability to identify dogs and horses through their ID tags. It may enable DoT to reach out to their owners to warn them, and in turn minimize the occurrence of such traffic incidents. (future project)

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier,
                            GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA, NMF
from sklearn.preprocessing import minmax_scale
from sklearn.metrics import accuracy_score


from tensorflow.keras.datasets import cifar10
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Conv2D,
                                    Flatten, MaxPooling2D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Dropout
```

2. Get the data

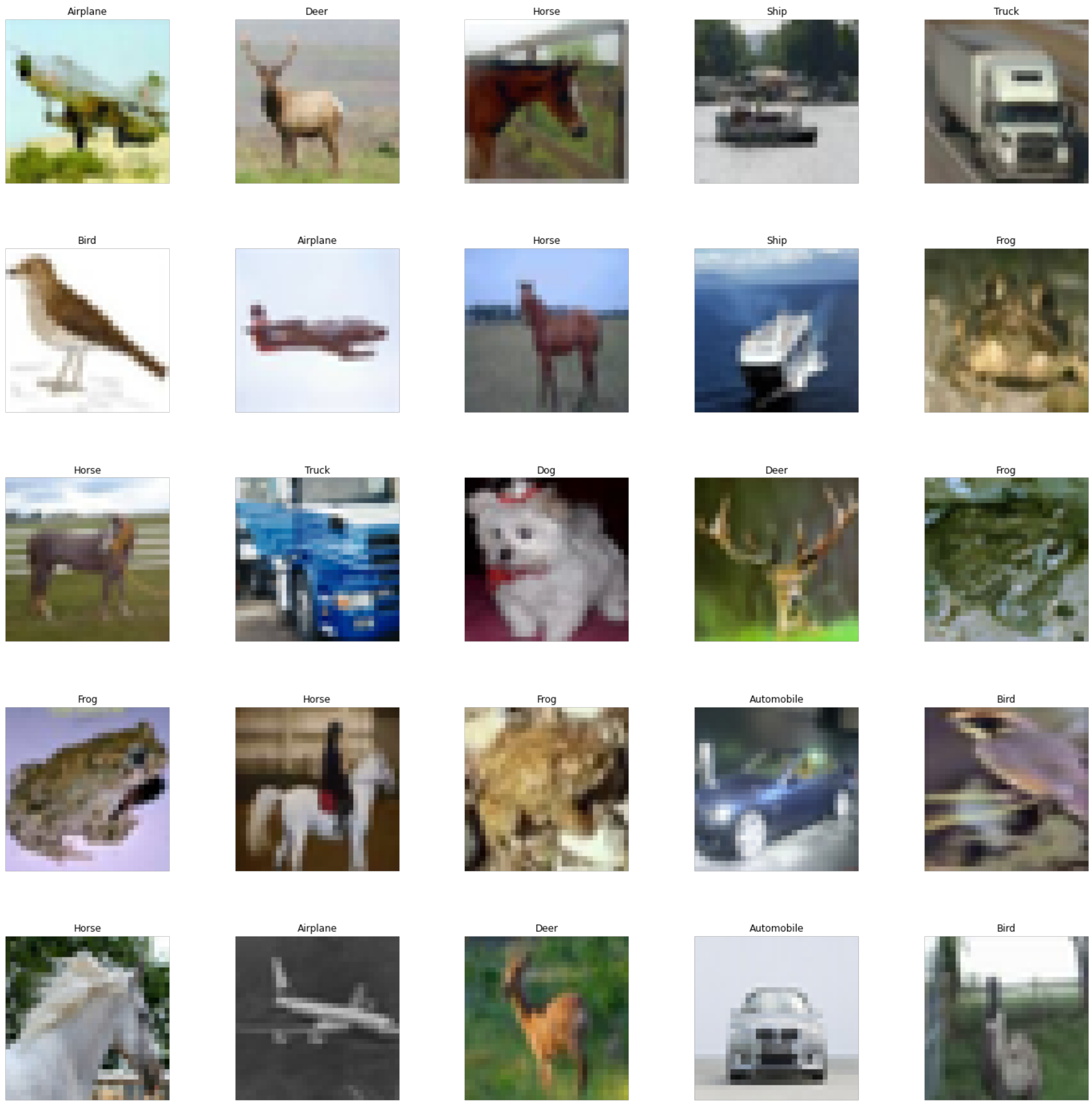
In [2]:
```
#loading the data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(50000, 32, 32, 3) (10000, 32, 32, 3) (50000, 1) (10000, 1)
```

In [3]:
```
##Count for each images
unique, counts = np.unique(y_train, return_counts=True)
dict(zip(unique, counts))
```

Out[3]:
```
{0: 5000,
 1: 5000,
 2: 5000,
 3: 5000,
 4: 5000,
 5: 5000,
 6: 5000,
 7: 5000,
 8: 5000,
 9: 5000}
```

In [4]:
```
#visualizing few samples of whole data set
labels = ['Airplane', 'Automobile', 'Bird',
          'Cat', 'Deer', 'Dog', 'Frog', 'Horse',
          'Ship', 'Truck']

fig, axes = plt.subplots(5, 5, figsize = (25, 25))
axes = axes.ravel()

n_training = len(X_train)

for i in range(0,5*5):
    index = np.random.randint(0,n_training) # pick a random number
    axes[i].imshow(X_train[index])
    index = y_train[index]
    axes[i].set_title(labels[int(index)])
    axes[i].axis('off')

plt.subplots_adjust(hspace = 0.4)
```



Only extract dog and horse images

In [5]:
```
## Filter training and test sets to dog and horse
dog = 5
horse = 7

train_ind = np.where((y_train==dog) | (y_train == horse))[0]
test_ind = np.where((y_test==dog) | (y_test == horse))[0]

X_train = X_train[train_ind]
y_train = y_train[train_ind]

X_test = X_test[test_ind]
y_test = y_test[test_ind]
```

In [6]:
```
#train and test shape
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

```
(10000, 32, 32, 3) (10000, 1) (2000, 32, 32, 3) (2000, 1)
```

Relabel the data

In [7]:
```
##Relabel dog as 0 and horse as 1 for binary classification
y_train[y_train == dog]=0
y_train[y_train == horse]=1
y_test[y_test == dog]=0
y_test[y_test== horse]=1
```

In [8]:
```
##Flatten
X_train = X_train.reshape(X_train.shape[0], -1)
X_test = X_test.reshape(X_test.shape[0], -1)
```

In [9]:
```
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

```
(10000, 3072) (10000, 1) (2000, 3072) (2000, 1)
```

Split the data

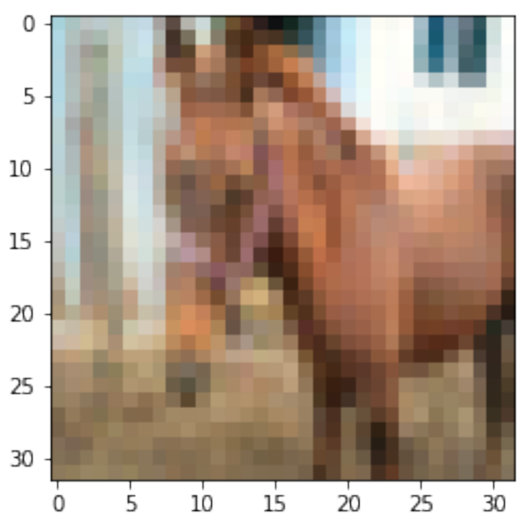
In [10]:
```
#Create validation set
X_train, X_valid, y_train, y_valid = train_test_split(X_train,
                                                      y_train,
                                                      test_size=0.2,
                                                      stratify = y_tra
in,
                                                      random_state=202
0)
print(X_train.shape, X_valid.shape,X_test.shape,
      y_train.shape,y_valid.shape, y_test.shape)
```

```
(8000, 3072) (2000, 3072) (2000, 3072) (8000, 1) (2000, 1) (2000, 1)
```

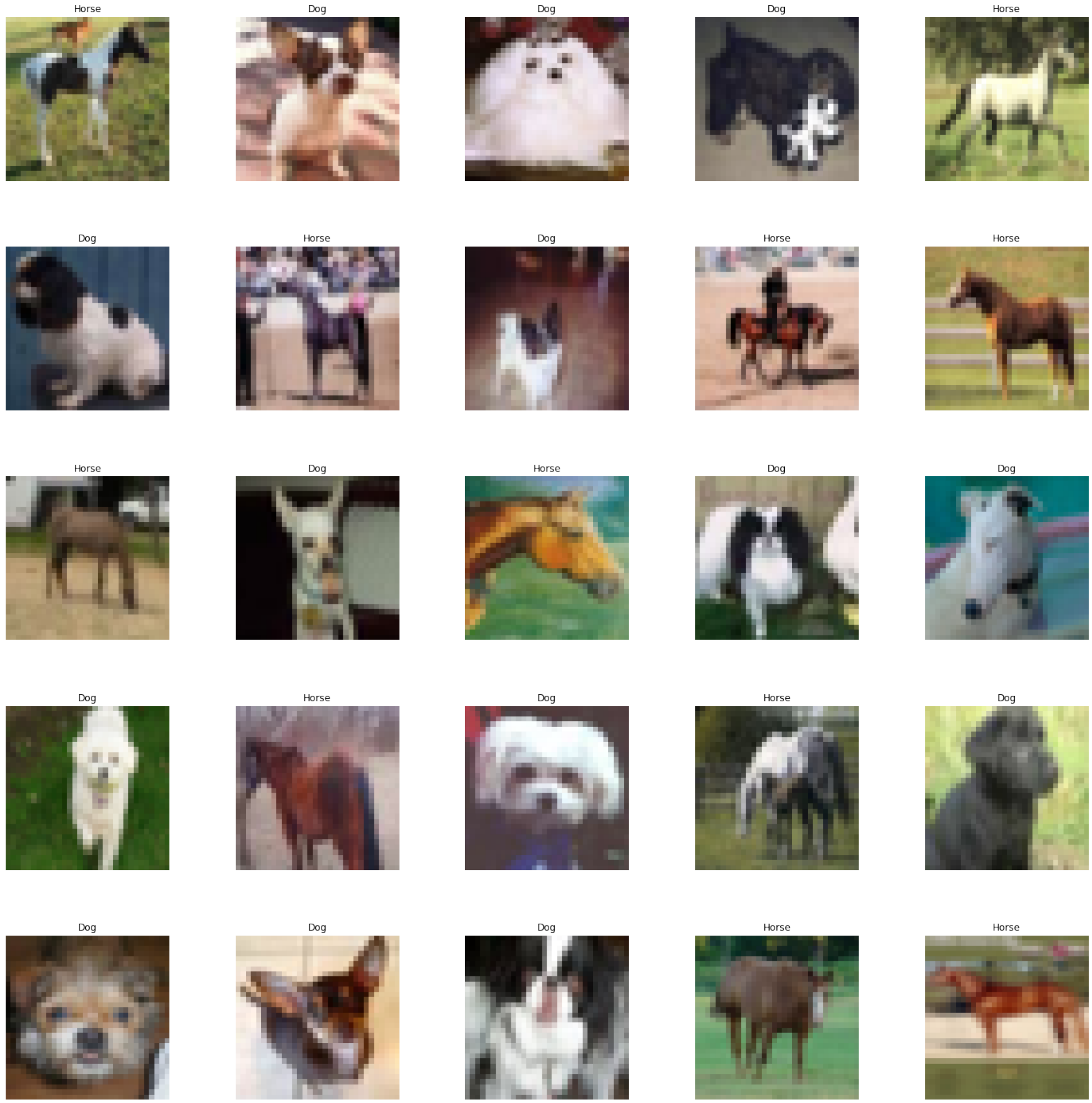
3. Exploratory Data Analysis (EDA)

In [11]:
```
#plot the first image of training data set.
plt.imshow(X_train[0].reshape(32,32,3))
print(y_train[0])
```

```
[1]
```



In [12]:
```
##Plot some images of dog and horse
labels2 = ['Dog', 'Horse']

fig, axes = plt.subplots(5, 5, figsize = (25, 25))
axes = axes.ravel()

n_training = len(X_train)

for i in range(0,5*5):
    index = np.random.randint(0,n_training) # pick a random number
    axes[i].imshow(X_train[index].reshape(32,32,3))
    index = y_train[index]
    axes[i].set_title(labels2[int(index)])
    axes[i].axis('off')

plt.subplots_adjust(hspace = 0.4)
```



4. Preprocessing

There are 3072 columns in the data set. In order to reduce computational cost, we are using Principal Component Analysis (PCA) to reduce the dimensions and make our machine learning algorithms run faster.

PCA

In [13]:
```
#Build a default PCA model
pca = PCA()
pca.fit_transform(X_train)
```

Out[13]:
```
array([[ 1.47846677e+03, -7.88784340e+02,  6.54600623e+02, ...,
        -6.36834475e-03, -5.92768861e-02, -1.79615376e-02],
       [ 1.19894500e+03, -2.86444942e+02, -9.15070578e+02, ...,
        -8.44842864e-03,  2.03756697e-01,  1.20015894e-01],
       [-1.34256119e+03,  2.12314158e+02, -1.26224793e+03, ...,
         6.76602584e-02,  5.84493544e-01, -4.66203285e-02],
       ...,
       [ 2.34404890e+03,  1.61195884e+03, -6.85811046e+02, ...,
        -3.00640930e-02,  4.94818862e-01,  1.57506047e-01],
       [ 1.58220735e+03, -1.08234880e+03,  2.59291539e+02, ...,
         1.96112036e-01,  3.98314375e-01,  1.49850794e-02],
       [-1.55954661e+03,  4.04595574e+02,  4.11727531e+02, ...,
        -2.25971549e-02,  6.14892029e-02,  3.27682535e-01]])
```

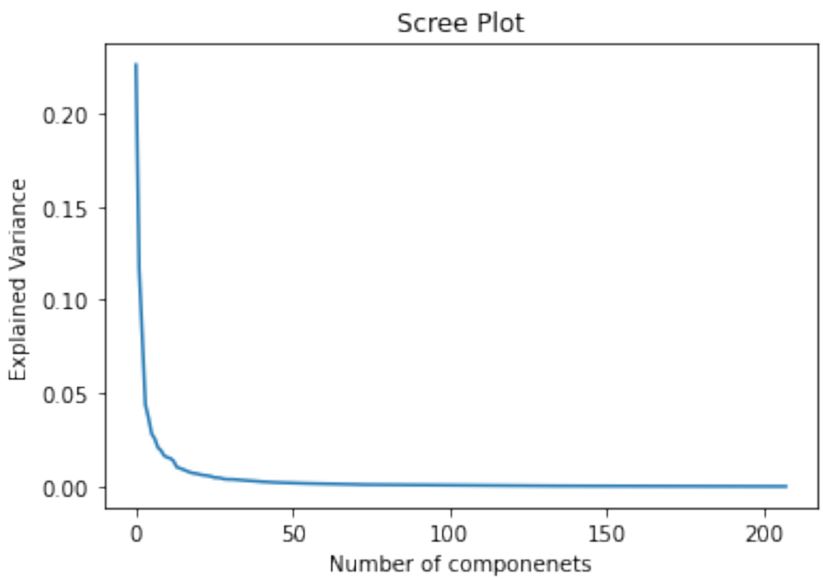
In [14]:
```
# Calculating optimal k to have 95% variance
k = 0
total = sum(pca.explained_variance_)
current_sum = 0

while(current_sum / total < 0.95):
    current_sum += pca.explained_variance_[k]
    k += 1
print(k)
```

```
208
```

In [15]:
```
## Applying PCA with k calcuated above
pca2 = PCA(n_components=k, whiten=True)

X_train_pca = pca2.fit_transform(X_train)
X_valid_pca = pca2.transform(X_valid)
X_test_pca = pca2.transform(X_test)
```

In [16]:
```
#Scree Plot
plt.plot(pca2.explained_variance_ratio_)
plt.title("Scree Plot")
plt.xlabel('Number of componenets')
plt.ylabel('Explained Variance')
```

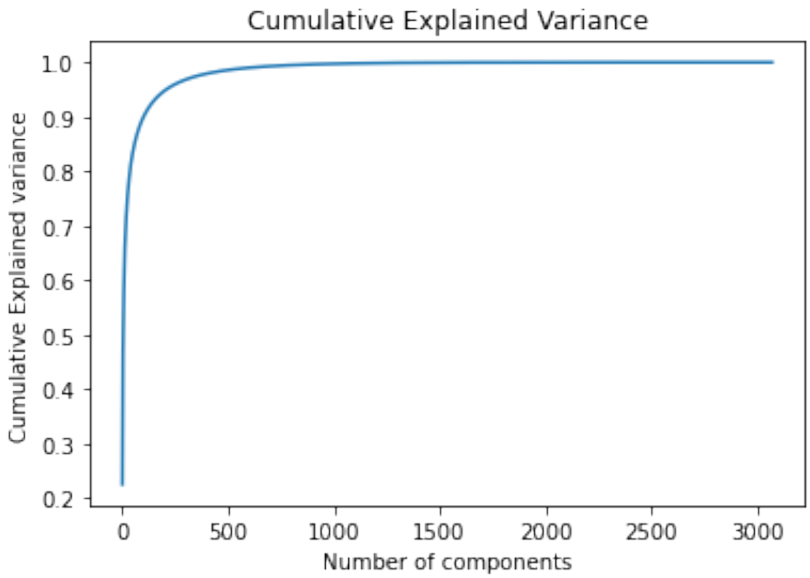
Out[16]:
```
Text(0, 0.5, 'Explained Variance')
```



In [17]:
```
#Cumulative Explained Variance
print('Top k components explain {:.0f}% of the variance.'
 .format(100*pca2.explained_variance_ratio_.sum()))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.title("Cumulative Explained Variance")
plt.xlabel('Number of components')
plt.ylabel('Cumulative Explained variance')
```

```
Top k components explain 95% of the variance.
```

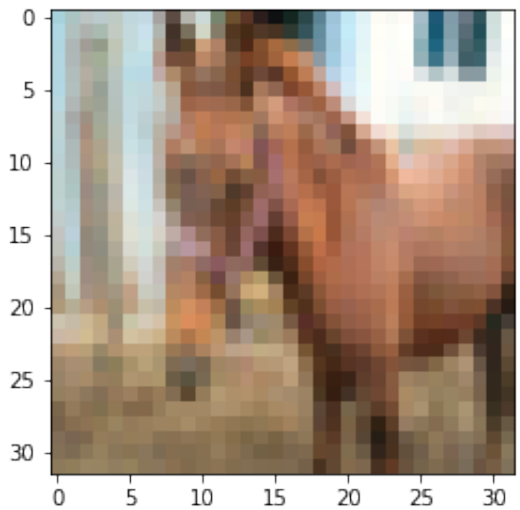
Out[17]:
```
Text(0, 0.5, 'Cumulative Explained variance')
```



In [18]:
```
loadings = minmax_scale(pca.components_, feature_range=(0,1), axis=1)
```

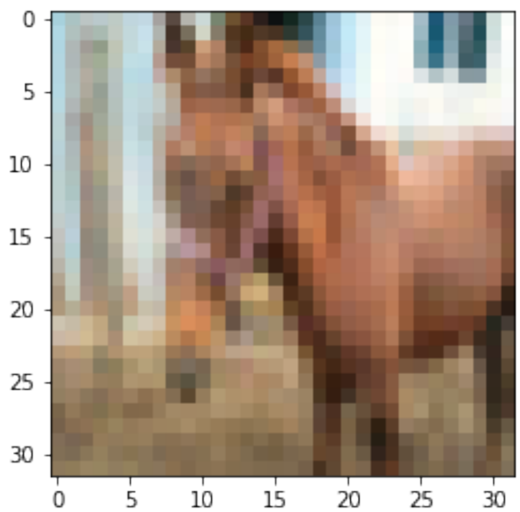
In [19]:
```
##image vs it's reconstruction using PCA
PCs = pca.transform(X_train)
X_recon = pca.inverse_transform(PCs).astype('int')
plt.imshow(X_train[0].reshape(32, 32, 3))
```

Out[19]: <matplotlib.image.AxesImage at 0x7fe665bf7b80>



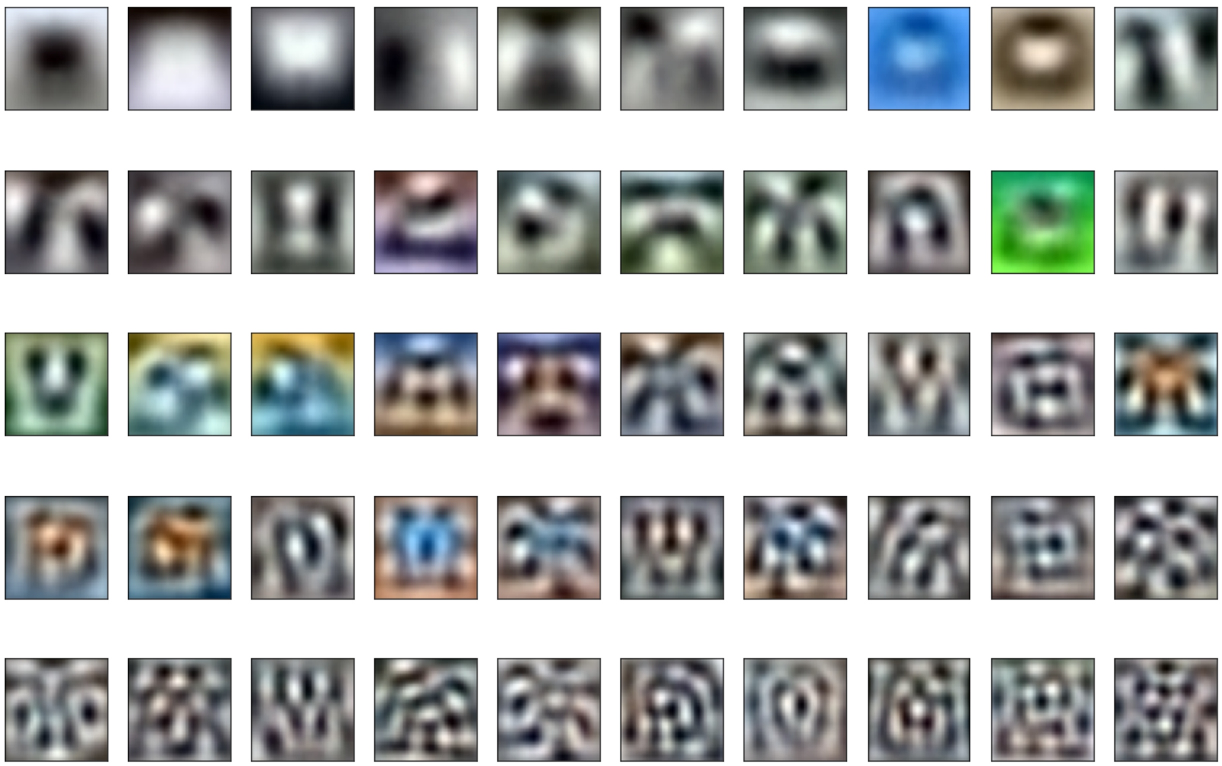
In [20]:
```
plt.imshow(X_recon[0].reshape(32, 32, 3))
```

Out[20]: <matplotlib.image.AxesImage at 0x7fe652c91a30>



In [21]:
```
# Plot the top 50 PCA components
plt.figure(figsize=(15, 10))
for j in range(50):
    plt.subplot(5, 10, j + 1)
    plt.imshow(loadings[j].reshape(32, 32, 3))
    plt.xticks(())
    plt.yticks(())
```

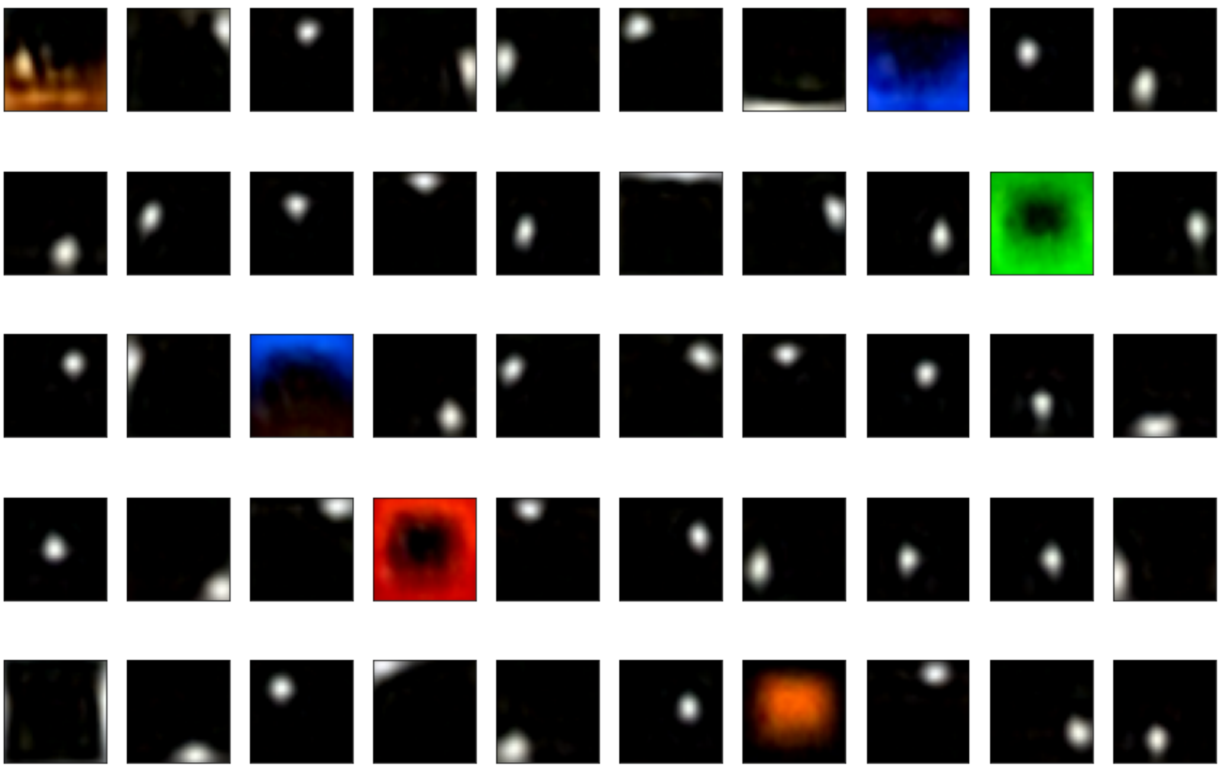
```
Clipping input data to the valid range for imshow with RGB data ([0.
.1] for floats or [0..255] for integers).
```



NMF

In [22]:
```
nmf = NMF(n_components=50).fit(X_train)
nmf_loadings = minmax_scale(nmf.components_,
                            feature_range=(0,1),
                            axis=1)

plt.figure(figsize=(15, 10))
for j in range(50):
    plt.subplot(5, 10, j + 1)
    plt.imshow(nmf_loadings[j].reshape(32, 32, 3))
    plt.xticks(())
    plt.yticks(())
```

```
/Users/tt/opt/anaconda3/lib/python3.8/site-packages/sklearn/decompos
ition/_nmf.py:1076: ConvergenceWarning: Maximum number of iterations
200 reached. Increase it to improve convergence.
  warnings.warn("Maximum number of iterations %d reached. Increase i
t to"
```



NMF loadings seem to highlight different parts of the image.

The ordering of the PC loadings are meaningful (they are ordered by variance explained). The ordering of the NMF loadings are irrelevant

5. Initial Models

1) Random Forest and Gradient Boosted Decision Tree

In [23]:
```
#Initial random forest and gradient boosted decision tree models
rf = RandomForestClassifier()
gbdt = GradientBoostingClassifier()

# Initialize lists to hold metrics:
models = [rf, gbdt]
model_names = ['Random Forest', 'Gradient Boosted Tree']
train_accuracy=[]
test_accuracy=[]
```

In [24]:
```
#calculate the accuracy of Random Forest and Gradient Bossted
#Decision Tree
for m in models:
    m.fit(X_train_pca, y_train.ravel())
    train_preds = m.predict(X_train_pca)
    test_preds = m.predict(X_test_pca)
    train_accuracy.append(accuracy_score(y_train, train_preds))
    test_accuracy.append(accuracy_score(y_test,test_preds))
```

In [25]:
```
#Compare the accuracy in train and test sets
accuracy = pd.DataFrame({'model':model_names,
             'Accuracy_train': train_accuracy,
             'Accuracy_test': test_accuracy})
accuracy
```

Out[25]:

	model	Accuracy_train	Accuracy_test
0	Random Forest	1.00000	0.7515
1	Gradient Boosted Tree	0.84625	0.7815

As we can tell the Gradient Boosted Decision Trees has better acuuracy score in test set.

2) Fully-Connected Neural Networks

In [26]:
```
##Reshape the X
X_train = X_train.reshape(8000,32,32,3)
X_valid = X_valid.reshape(2000,32,32,3)
X_test = X_test.reshape(2000,32,32,3)
```

In [27]:
```
##One hot encoder y to categorical data
y_train_cnn = to_categorical(y_train,2)
y_valid_cnn = to_categorical(y_valid,2)
y_test_cnn = to_categorical(y_test,2)
```

In [28]:
```
##Fully-connected neural networks by using PCA components
model_NN = Sequential()
model_NN.add(Dense(50, activation = 'relu',
                   input_shape=(X_train_pca.shape[1],)))
model_NN.add(Dense(20, activation = 'relu'))
model_NN.add(Dense(2, activation = 'softmax'))

model_NN.compile(optimizer='adam',
             loss='categorical_crossentropy',
             metrics=['accuracy'])
history=model_NN.fit(X_train_pca, y_train_cnn,
                     epochs=50,
                     batch_size=100,
                     validation_data=(X_valid_pca, y_valid_cnn))
```

```
Epoch 1/50
80/80 [==============================] - 0s 2ms/step - loss: 0.6906
- accuracy: 0.5965 - val_loss: 0.6016 - val_accuracy: 0.6820
Epoch 2/50
80/80 [==============================] - 0s 716us/step - loss: 0.520
6 - accuracy: 0.7494 - val_loss: 0.5370 - val_accuracy: 0.7475
Epoch 3/50
80/80 [==============================] - 0s 718us/step - loss: 0.446
2 - accuracy: 0.7951 - val_loss: 0.5219 - val_accuracy: 0.7520
Epoch 4/50
80/80 [==============================] - 0s 729us/step - loss: 0.395
9 - accuracy: 0.8234 - val_loss: 0.5137 - val_accuracy: 0.7590
Epoch 5/50
80/80 [==============================] - 0s 713us/step - loss: 0.354
5 - accuracy: 0.8486 - val_loss: 0.5142 - val_accuracy: 0.7630
Epoch 6/50
80/80 [==============================] - 0s 720us/step - loss: 0.315
9 - accuracy: 0.8717 - val_loss: 0.5190 - val_accuracy: 0.7625
Epoch 7/50
```

```
80/80 [==============================] - 0s 724us/step - loss: 0.277
6 - accuracy: 0.8923 - val_loss: 0.5246 - val_accuracy: 0.7710
Epoch 8/50
80/80 [==============================] - 0s 2ms/step - loss: 0.2420
- accuracy: 0.9101 - val_loss: 0.5338 - val_accuracy: 0.7795
Epoch 9/50
80/80 [==============================] - 0s 736us/step - loss: 0.207
3 - accuracy: 0.9300 - val_loss: 0.5490 - val_accuracy: 0.7795
Epoch 10/50
80/80 [==============================] - 0s 789us/step - loss: 0.174
2 - accuracy: 0.9410 - val_loss: 0.5768 - val_accuracy: 0.7815
Epoch 11/50
80/80 [==============================] - 0s 703us/step - loss: 0.145
7 - accuracy: 0.9565 - val_loss: 0.6068 - val_accuracy: 0.7800
Epoch 12/50
80/80 [==============================] - 0s 707us/step - loss: 0.120
5 - accuracy: 0.9660 - val_loss: 0.6401 - val_accuracy: 0.7775
Epoch 13/50
80/80 [==============================] - 0s 698us/step - loss: 0.095
5 - accuracy: 0.9771 - val_loss: 0.6851 - val_accuracy: 0.7790
Epoch 14/50
80/80 [==============================] - 0s 702us/step - loss: 0.075
6 - accuracy: 0.9862 - val_loss: 0.7243 - val_accuracy: 0.7875
Epoch 15/50
80/80 [==============================] - 0s 703us/step - loss: 0.058
2 - accuracy: 0.9914 - val_loss: 0.7687 - val_accuracy: 0.7865
Epoch 16/50
80/80 [==============================] - 0s 700us/step - loss: 0.046
1 - accuracy: 0.9954 - val_loss: 0.8307 - val_accuracy: 0.7810
Epoch 17/50
80/80 [==============================] - 0s 699us/step - loss: 0.034
1 - accuracy: 0.9974 - val_loss: 0.8711 - val_accuracy: 0.7860
Epoch 18/50
80/80 [==============================] - 0s 689us/step - loss: 0.025
5 - accuracy: 0.9985 - val_loss: 0.9284 - val_accuracy: 0.7800
Epoch 19/50
80/80 [==============================] - 0s 703us/step - loss: 0.019
8 - accuracy: 0.9990 - val_loss: 0.9755 - val_accuracy: 0.7840
Epoch 20/50
80/80 [==============================] - 0s 693us/step - loss: 0.015
0 - accuracy: 0.9995 - val_loss: 1.0139 - val_accuracy: 0.7820
Epoch 21/50
80/80 [==============================] - 0s 694us/step - loss: 0.011
7 - accuracy: 0.9998 - val_loss: 1.0559 - val_accuracy: 0.7825
Epoch 22/50
80/80 [==============================] - 0s 696us/step - loss: 0.009
2 - accuracy: 1.0000 - val_loss: 1.1029 - val_accuracy: 0.7825
Epoch 23/50
80/80 [==============================] - 0s 708us/step - loss: 0.007
6 - accuracy: 1.0000 - val_loss: 1.1434 - val_accuracy: 0.7860
```

```
Epoch 24/50
80/80 [==============================] - 0s 696us/step - loss: 0.006
2 - accuracy: 1.0000 - val_loss: 1.1749 - val_accuracy: 0.7805
Epoch 25/50
80/80 [==============================] - 0s 706us/step - loss: 0.005
1 - accuracy: 1.0000 - val_loss: 1.2092 - val_accuracy: 0.7835
Epoch 26/50
80/80 [==============================] - 0s 700us/step - loss: 0.004
4 - accuracy: 1.0000 - val_loss: 1.2384 - val_accuracy: 0.7815
Epoch 27/50
80/80 [==============================] - 0s 725us/step - loss: 0.003
7 - accuracy: 1.0000 - val_loss: 1.2675 - val_accuracy: 0.7805
Epoch 28/50
80/80 [==============================] - 0s 711us/step - loss: 0.003
2 - accuracy: 1.0000 - val_loss: 1.2968 - val_accuracy: 0.7840
Epoch 29/50
80/80 [==============================] - 0s 691us/step - loss: 0.002
8 - accuracy: 1.0000 - val_loss: 1.3262 - val_accuracy: 0.7845
Epoch 30/50
80/80 [==============================] - 0s 701us/step - loss: 0.002
4 - accuracy: 1.0000 - val_loss: 1.3453 - val_accuracy: 0.7830
Epoch 31/50
80/80 [==============================] - 0s 702us/step - loss: 0.002
1 - accuracy: 1.0000 - val_loss: 1.3752 - val_accuracy: 0.7820
Epoch 32/50
80/80 [==============================] - 0s 695us/step - loss: 0.001
9 - accuracy: 1.0000 - val_loss: 1.3972 - val_accuracy: 0.7830
Epoch 33/50
80/80 [==============================] - 0s 710us/step - loss: 0.001
7 - accuracy: 1.0000 - val_loss: 1.4202 - val_accuracy: 0.7800
Epoch 34/50
80/80 [==============================] - 0s 694us/step - loss: 0.001
5 - accuracy: 1.0000 - val_loss: 1.4429 - val_accuracy: 0.7815
Epoch 35/50
80/80 [==============================] - 0s 711us/step - loss: 0.001
4 - accuracy: 1.0000 - val_loss: 1.4653 - val_accuracy: 0.7805
Epoch 36/50
80/80 [==============================] - 0s 691us/step - loss: 0.001
2 - accuracy: 1.0000 - val_loss: 1.4859 - val_accuracy: 0.7825
Epoch 37/50
80/80 [==============================] - 0s 702us/step - loss: 0.001
1 - accuracy: 1.0000 - val_loss: 1.5057 - val_accuracy: 0.7790
Epoch 38/50
80/80 [==============================] - 0s 712us/step - loss: 9.884
1e-04 - accuracy: 1.0000 - val_loss: 1.5258 - val_accuracy: 0.7795
Epoch 39/50
80/80 [==============================] - 0s 740us/step - loss: 9.002
3e-04 - accuracy: 1.0000 - val_loss: 1.5439 - val_accuracy: 0.7795
Epoch 40/50
80/80 [==============================] - 0s 764us/step - loss: 8.216
```

```
8e-04 - accuracy: 1.0000 - val_loss: 1.5598 - val_accuracy: 0.7790
Epoch 41/50
80/80 [==============================] - 0s 738us/step - loss: 7.485
8e-04 - accuracy: 1.0000 - val_loss: 1.5784 - val_accuracy: 0.7790
Epoch 42/50
80/80 [==============================] - 0s 711us/step - loss: 6.878
5e-04 - accuracy: 1.0000 - val_loss: 1.5970 - val_accuracy: 0.7780
Epoch 43/50
80/80 [==============================] - 0s 721us/step - loss: 6.300
9e-04 - accuracy: 1.0000 - val_loss: 1.6124 - val_accuracy: 0.7785
Epoch 44/50
80/80 [==============================] - 0s 753us/step - loss: 5.807
2e-04 - accuracy: 1.0000 - val_loss: 1.6297 - val_accuracy: 0.7780
Epoch 45/50
80/80 [==============================] - 0s 712us/step - loss: 5.332
9e-04 - accuracy: 1.0000 - val_loss: 1.6450 - val_accuracy: 0.7785
Epoch 46/50
80/80 [==============================] - 0s 712us/step - loss: 4.927
3e-04 - accuracy: 1.0000 - val_loss: 1.6613 - val_accuracy: 0.7800
Epoch 47/50
80/80 [==============================] - 0s 707us/step - loss: 4.561
4e-04 - accuracy: 1.0000 - val_loss: 1.6778 - val_accuracy: 0.7790
Epoch 48/50
80/80 [==============================] - 0s 756us/step - loss: 4.207
6e-04 - accuracy: 1.0000 - val_loss: 1.6925 - val_accuracy: 0.7785
Epoch 49/50
80/80 [==============================] - 0s 716us/step - loss: 3.911
3e-04 - accuracy: 1.0000 - val_loss: 1.7082 - val_accuracy: 0.7775
Epoch 50/50
80/80 [==============================] - 0s 759us/step - loss: 3.639
4e-04 - accuracy: 1.0000 - val_loss: 1.7229 - val_accuracy: 0.7785
```

In [29]:

```
NN_accuracy = model_NN.evaluate(X_test_pca, y_test_cnn)[1]
print(NN_accuracy)
```

```
63/63 [==============================] - 0s 648us/step - loss: 1.674
1 - accuracy: 0.7760
0.7760000228881836
```

In [30]:
```
##Compare accuracy with the three models
accuracy.append({'model':'Neural Network','Accuracy_test':NN_accuracy}
, ignore_index=True)
```

Out[30]:

	model	Accuracy_train	Accuracy_test
0	Random Forest	1.00000	0.7515
1	Gradient Boosted Tree	0.84625	0.7815
2	Neural Network	NaN	0.7760

In [31]:
```
##Fully-connected neural networks without using PCA
model_NN2 = Sequential()
model_NN2.add(Flatten(input_shape=(32,32,3)))
model_NN2.add(Dense(50, activation = 'relu'))
model_NN2.add(Dense(20, activation='relu'))
model_NN2.add(Dense(2, activation='softmax'))

model_NN2.compile(optimizer='adam',
             loss='categorical_crossentropy',
             metrics=['accuracy'])
history=model_NN2.fit(X_train, y_train_cnn,
                      epochs=50, batch_size=100,
                      validation_data=(X_valid, y_valid_cnn))
```

```
Epoch 1/50
80/80 [==============================] - 0s 2ms/step - loss: 2.2755
- accuracy: 0.4996 - val_loss: 0.6963 - val_accuracy: 0.5035
Epoch 2/50
80/80 [==============================] - 0s 2ms/step - loss: 0.6874
- accuracy: 0.5483 - val_loss: 0.6811 - val_accuracy: 0.5815
Epoch 3/50
80/80 [==============================] - 0s 1ms/step - loss: 0.6674
- accuracy: 0.6037 - val_loss: 0.6653 - val_accuracy: 0.5935
Epoch 4/50
80/80 [==============================] - 0s 1ms/step - loss: 0.6429
- accuracy: 0.6451 - val_loss: 0.6360 - val_accuracy: 0.6525
Epoch 5/50
80/80 [==============================] - 0s 1ms/step - loss: 0.6180
- accuracy: 0.6752 - val_loss: 0.6252 - val_accuracy: 0.6540
Epoch 6/50
80/80 [==============================] - 0s 1ms/step - loss: 0.6069
- accuracy: 0.6758 - val_loss: 0.6092 - val_accuracy: 0.6795
Epoch 7/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5888
- accuracy: 0.7015 - val_loss: 0.5965 - val_accuracy: 0.6775
Epoch 8/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5762
```

```
- accuracy: 0.7086 - val_loss: 0.5925 - val_accuracy: 0.6910
Epoch 9/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5705
- accuracy: 0.7101 - val_loss: 0.5859 - val_accuracy: 0.6960
Epoch 10/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5645
- accuracy: 0.7178 - val_loss: 0.6018 - val_accuracy: 0.6835
Epoch 11/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5559
- accuracy: 0.7260 - val_loss: 0.5688 - val_accuracy: 0.7115
Epoch 12/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5506
- accuracy: 0.7295 - val_loss: 0.5630 - val_accuracy: 0.7170
Epoch 13/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5501
- accuracy: 0.7289 - val_loss: 0.5609 - val_accuracy: 0.7180
Epoch 14/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5358
- accuracy: 0.7419 - val_loss: 0.5554 - val_accuracy: 0.7215
Epoch 15/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5462
- accuracy: 0.7336 - val_loss: 0.5835 - val_accuracy: 0.7070
Epoch 16/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5299
- accuracy: 0.7489 - val_loss: 0.5553 - val_accuracy: 0.7255
Epoch 17/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5271
- accuracy: 0.7487 - val_loss: 0.5474 - val_accuracy: 0.7295
Epoch 18/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5377
- accuracy: 0.7383 - val_loss: 0.5441 - val_accuracy: 0.7315
Epoch 19/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5171
- accuracy: 0.7555 - val_loss: 0.5456 - val_accuracy: 0.7365
Epoch 20/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5142
- accuracy: 0.7575 - val_loss: 0.5380 - val_accuracy: 0.7335
Epoch 21/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5244
- accuracy: 0.7461 - val_loss: 0.5361 - val_accuracy: 0.7385
Epoch 22/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5078
- accuracy: 0.7592 - val_loss: 0.5412 - val_accuracy: 0.7295
Epoch 23/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5331
- accuracy: 0.7440 - val_loss: 0.5802 - val_accuracy: 0.7255
Epoch 24/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5147
- accuracy: 0.7564 - val_loss: 0.5353 - val_accuracy: 0.7360
Epoch 25/50
```

```
80/80 [==============================] - 0s 2ms/step - loss: 0.5084
- accuracy: 0.7613 - val_loss: 0.5343 - val_accuracy: 0.7355
Epoch 26/50
80/80 [==============================] - 0s 2ms/step - loss: 0.5015
- accuracy: 0.7663 - val_loss: 0.5315 - val_accuracy: 0.7430
Epoch 27/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5051
- accuracy: 0.7681 - val_loss: 0.6078 - val_accuracy: 0.6770
Epoch 28/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5162
- accuracy: 0.7548 - val_loss: 0.5215 - val_accuracy: 0.7505
Epoch 29/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4963
- accuracy: 0.7716 - val_loss: 0.5199 - val_accuracy: 0.7545
Epoch 30/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5046
- accuracy: 0.7630 - val_loss: 0.5175 - val_accuracy: 0.7545
Epoch 31/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5054
- accuracy: 0.7654 - val_loss: 0.6098 - val_accuracy: 0.7035
Epoch 32/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5084
- accuracy: 0.7614 - val_loss: 0.5398 - val_accuracy: 0.7435
Epoch 33/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4873
- accuracy: 0.7765 - val_loss: 0.5515 - val_accuracy: 0.7210
Epoch 34/50
80/80 [==============================] - 0s 2ms/step - loss: 0.4754
- accuracy: 0.7845 - val_loss: 0.5347 - val_accuracy: 0.7550
Epoch 35/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4955
- accuracy: 0.7706 - val_loss: 0.5110 - val_accuracy: 0.7595
Epoch 36/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4874
- accuracy: 0.7738 - val_loss: 0.5582 - val_accuracy: 0.7165
Epoch 37/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4900
- accuracy: 0.7764 - val_loss: 0.5401 - val_accuracy: 0.7590
Epoch 38/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4745
- accuracy: 0.7853 - val_loss: 0.5135 - val_accuracy: 0.7530
Epoch 39/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4736
- accuracy: 0.7846 - val_loss: 0.5108 - val_accuracy: 0.7590
Epoch 40/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4834
- accuracy: 0.7784 - val_loss: 0.5276 - val_accuracy: 0.7425
Epoch 41/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4850
- accuracy: 0.7790 - val_loss: 0.5376 - val_accuracy: 0.7305
```

```
Epoch 42/50
80/80 [==============================] - 0s 1ms/step - loss: 0.5003
- accuracy: 0.7676 - val_loss: 0.5256 - val_accuracy: 0.7425
Epoch 43/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4770
- accuracy: 0.7828 - val_loss: 0.5596 - val_accuracy: 0.7205
Epoch 44/50
80/80 [==============================] - 0s 2ms/step - loss: 0.4697
- accuracy: 0.7880 - val_loss: 0.5183 - val_accuracy: 0.7620
Epoch 45/50
80/80 [==============================] - 0s 2ms/step - loss: 0.4687
- accuracy: 0.7856 - val_loss: 0.5254 - val_accuracy: 0.7560
Epoch 46/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4805
- accuracy: 0.7794 - val_loss: 0.5549 - val_accuracy: 0.7335
Epoch 47/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4711
- accuracy: 0.7840 - val_loss: 0.5152 - val_accuracy: 0.7700
Epoch 48/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4639
- accuracy: 0.7905 - val_loss: 0.5116 - val_accuracy: 0.7560
Epoch 49/50
80/80 [==============================] - 0s 1ms/step - loss: 0.4822
- accuracy: 0.7795 - val_loss: 0.5096 - val_accuracy: 0.7670
Epoch 50/50
80/80 [==============================] - 0s 2ms/step - loss: 0.4578
- accuracy: 0.7974 - val_loss: 0.5404 - val_accuracy: 0.7310
```

In [32]:

```
model_NN2.evaluate(X_test, y_test_cnn)
```

```
63/63 [==============================] - 0s 708us/step - loss: 0.523
6 - accuracy: 0.7475
```

Out[32]:

```
[0.5235821604728699, 0.7475000023841858]
```

3) Convolutional Neural Networks (CNN)

In [52]:
```
model_CNN=Sequential()
model_CNN.add(Conv2D(32,
                    kernel_size=(3,3),
                    strides=1,
                    activation ='relu',
                    padding='same', input_shape=(32,32,3)))
model_CNN.add(MaxPooling2D(pool_size=(2,2),strides=2))
model_CNN.add(Conv2D(64,
                    kernel_size=(3,3),
                    strides=1,
                    activation ='relu',
                    padding='same'))
model_CNN.add(Flatten())
model_CNN.add(Dense(50, activation = 'relu'))
model_CNN.add(Dense(20, activation='relu'))
model_CNN.add(Dense(2, activation='softmax'))
```

In [53]:
```
model_CNN.compile(optimizer='adam',
              loss ='categorical_crossentropy',
              metrics=['accuracy'])
history_CNN = model_CNN.fit(X_train, y_train_cnn,
                            epochs=50, batch_size=100,
                            validation_data =(X_valid,y_valid_cnn))
```

```
Epoch 1/50
80/80 [==============================] - 4s 45ms/step - loss: 3.7811
- accuracy: 0.6819 - val_loss: 0.4947 - val_accuracy: 0.7660
Epoch 2/50
80/80 [==============================] - 3s 42ms/step - loss: 0.4472
- accuracy: 0.7918 - val_loss: 0.5077 - val_accuracy: 0.7590
Epoch 3/50
80/80 [==============================] - 3s 43ms/step - loss: 0.3803
- accuracy: 0.8314 - val_loss: 0.5098 - val_accuracy: 0.7780
Epoch 4/50
80/80 [==============================] - 3s 43ms/step - loss: 0.3293
- accuracy: 0.8589 - val_loss: 0.4722 - val_accuracy: 0.8115
Epoch 5/50
80/80 [==============================] - 3s 43ms/step - loss: 0.2806
- accuracy: 0.8764 - val_loss: 0.5084 - val_accuracy: 0.8065
Epoch 6/50
80/80 [==============================] - 3s 43ms/step - loss: 0.2609
- accuracy: 0.8924 - val_loss: 0.5321 - val_accuracy: 0.8140
Epoch 7/50
80/80 [==============================] - 3s 43ms/step - loss: 0.1953
- accuracy: 0.9221 - val_loss: 0.5801 - val_accuracy: 0.8065
Epoch 8/50
80/80 [==============================] - 3s 43ms/step - loss: 0.1551
- accuracy: 0.9391 - val_loss: 0.5412 - val_accuracy: 0.8210
Epoch 9/50
```

```
80/80 [==============================] - 4s 44ms/step - loss: 0.1683
- accuracy: 0.9320 - val_loss: 0.6559 - val_accuracy: 0.7930
Epoch 10/50
80/80 [==============================] - 3s 43ms/step - loss: 0.1300
- accuracy: 0.9510 - val_loss: 0.6884 - val_accuracy: 0.7990
Epoch 11/50
80/80 [==============================] - 3s 44ms/step - loss: 0.0913
- accuracy: 0.9641 - val_loss: 0.6723 - val_accuracy: 0.8240
Epoch 12/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0556
- accuracy: 0.9810 - val_loss: 0.8278 - val_accuracy: 0.8105
Epoch 13/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0723
- accuracy: 0.9789 - val_loss: 0.8408 - val_accuracy: 0.8105
Epoch 14/50
80/80 [==============================] - 3s 44ms/step - loss: 0.0668
- accuracy: 0.9768 - val_loss: 0.7921 - val_accuracy: 0.8110
Epoch 15/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0522
- accuracy: 0.9829 - val_loss: 0.8367 - val_accuracy: 0.8145
Epoch 16/50
80/80 [==============================] - 3s 42ms/step - loss: 0.0606
- accuracy: 0.9824 - val_loss: 0.8580 - val_accuracy: 0.8135
Epoch 17/50
80/80 [==============================] - 3s 42ms/step - loss: 0.0796
- accuracy: 0.9736 - val_loss: 1.0619 - val_accuracy: 0.8140
Epoch 18/50
80/80 [==============================] - 3s 42ms/step - loss: 0.0479
- accuracy: 0.9839 - val_loss: 0.8822 - val_accuracy: 0.8140
Epoch 19/50
80/80 [==============================] - 3s 44ms/step - loss: 0.0500
- accuracy: 0.9821 - val_loss: 0.8605 - val_accuracy: 0.8200
Epoch 20/50
80/80 [==============================] - 3s 44ms/step - loss: 0.0419
- accuracy: 0.9868 - val_loss: 1.0284 - val_accuracy: 0.8250
Epoch 21/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0372
- accuracy: 0.9879 - val_loss: 1.1394 - val_accuracy: 0.7845
Epoch 22/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0488
- accuracy: 0.9834 - val_loss: 1.0857 - val_accuracy: 0.8125
Epoch 23/50
80/80 [==============================] - 3s 42ms/step - loss: 0.0213
- accuracy: 0.9941 - val_loss: 1.0201 - val_accuracy: 0.8195
Epoch 24/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0222
- accuracy: 0.9941 - val_loss: 1.0093 - val_accuracy: 0.8060
Epoch 25/50
80/80 [==============================] - 4s 44ms/step - loss: 0.0189
- accuracy: 0.9940 - val_loss: 1.1535 - val_accuracy: 0.8145
```

```
Epoch 26/50
80/80 [==============================] - 4s 45ms/step - loss: 0.0090
- accuracy: 0.9976 - val_loss: 1.3746 - val_accuracy: 0.8260
Epoch 27/50
80/80 [==============================] - 4s 44ms/step - loss: 0.0166
- accuracy: 0.9950 - val_loss: 1.2878 - val_accuracy: 0.8175
Epoch 28/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0316
- accuracy: 0.9891 - val_loss: 1.3309 - val_accuracy: 0.7945
Epoch 29/50
80/80 [==============================] - 3s 44ms/step - loss: 0.0348
- accuracy: 0.9896 - val_loss: 1.3477 - val_accuracy: 0.8145
Epoch 30/50
80/80 [==============================] - 4s 44ms/step - loss: 0.0585
- accuracy: 0.9803 - val_loss: 1.2394 - val_accuracy: 0.8200
Epoch 31/50
80/80 [==============================] - 4s 44ms/step - loss: 0.0688
- accuracy: 0.9812 - val_loss: 1.1356 - val_accuracy: 0.8200
Epoch 32/50
80/80 [==============================] - 3s 43ms/step - loss: 0.0296
- accuracy: 0.9905 - val_loss: 1.3074 - val_accuracy: 0.8130
Epoch 33/50
80/80 [==============================] - 3s 44ms/step - loss: 0.0086
- accuracy: 0.9980 - val_loss: 1.4255 - val_accuracy: 0.8185
Epoch 34/50
80/80 [==============================] - 4s 45ms/step - loss: 0.0047
- accuracy: 0.9991 - val_loss: 1.3831 - val_accuracy: 0.8200
Epoch 35/50
80/80 [==============================] - 4s 45ms/step - loss: 9.7085
e-04 - accuracy: 1.0000 - val_loss: 1.5201 - val_accuracy: 0.8220
Epoch 36/50
80/80 [==============================] - 4s 46ms/step - loss: 4.7500
e-04 - accuracy: 1.0000 - val_loss: 1.5635 - val_accuracy: 0.8275
Epoch 37/50
80/80 [==============================] - 4s 46ms/step - loss: 3.3742
e-04 - accuracy: 1.0000 - val_loss: 1.6060 - val_accuracy: 0.8255
Epoch 38/50
80/80 [==============================] - 4s 44ms/step - loss: 2.6764
e-04 - accuracy: 1.0000 - val_loss: 1.6266 - val_accuracy: 0.8270
Epoch 39/50
80/80 [==============================] - 3s 43ms/step - loss: 2.2777
e-04 - accuracy: 1.0000 - val_loss: 1.6527 - val_accuracy: 0.8250
Epoch 40/50
80/80 [==============================] - 4s 46ms/step - loss: 1.9558
e-04 - accuracy: 1.0000 - val_loss: 1.6720 - val_accuracy: 0.8250
Epoch 41/50
80/80 [==============================] - 4s 46ms/step - loss: 1.7125
e-04 - accuracy: 1.0000 - val_loss: 1.6924 - val_accuracy: 0.8250
Epoch 42/50
80/80 [==============================] - 4s 44ms/step - loss: 1.5437
```

```
e-04 - accuracy: 1.0000 - val_loss: 1.7079 - val_accuracy: 0.8260
Epoch 43/50
80/80 [==============================] - 4s 44ms/step - loss: 1.4013
e-04 - accuracy: 1.0000 - val_loss: 1.7210 - val_accuracy: 0.8260
Epoch 44/50
80/80 [==============================] - 4s 44ms/step - loss: 1.2603
e-04 - accuracy: 1.0000 - val_loss: 1.7370 - val_accuracy: 0.8255
Epoch 45/50
80/80 [==============================] - 3s 44ms/step - loss: 1.1452
e-04 - accuracy: 1.0000 - val_loss: 1.7434 - val_accuracy: 0.8270
Epoch 46/50
80/80 [==============================] - 3s 43ms/step - loss: 1.0695
e-04 - accuracy: 1.0000 - val_loss: 1.7572 - val_accuracy: 0.8270
Epoch 47/50
80/80 [==============================] - 4s 44ms/step - loss: 9.8312
e-05 - accuracy: 1.0000 - val_loss: 1.7686 - val_accuracy: 0.8280
Epoch 48/50
80/80 [==============================] - 4s 44ms/step - loss: 9.0891
e-05 - accuracy: 1.0000 - val_loss: 1.7740 - val_accuracy: 0.8280
Epoch 49/50
80/80 [==============================] - 4s 45ms/step - loss: 8.3985
e-05 - accuracy: 1.0000 - val_loss: 1.7895 - val_accuracy: 0.8275
Epoch 50/50
80/80 [==============================] - 4s 46ms/step - loss: 7.8878
e-05 - accuracy: 1.0000 - val_loss: 1.8002 - val_accuracy: 0.8280
```

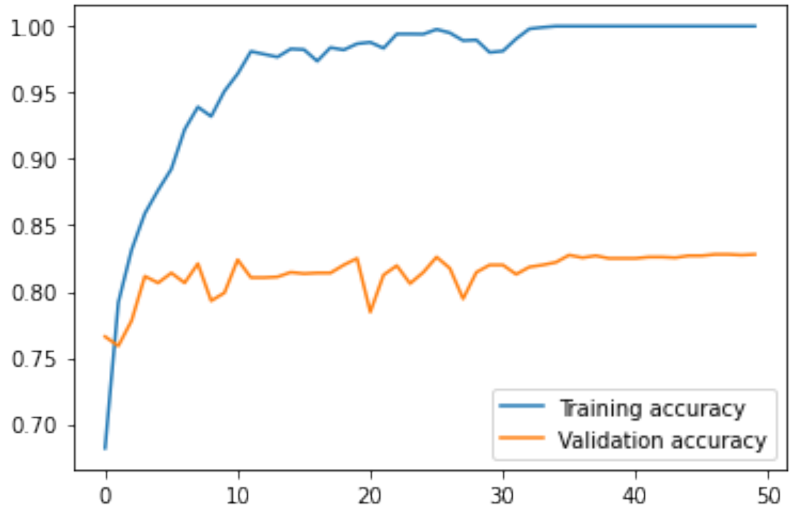
In [54]:
```
model_CNN.evaluate(X_test, y_test_cnn)
```

```
63/63 [==============================] - 0s 4ms/step - loss: 1.6902
- accuracy: 0.8435
```

Out[54]:
```
[1.6902180910110474, 0.843500018119812]
```

In [55]:
```
#Accuracy plot
plt.plot(history_CNN.history['accuracy'],
         label='Training accuracy')
plt.plot(history_CNN.history['val_accuracy'],
         label='Validation accuracy')
plt.legend()
```

Out[55]: <matplotlib.legend.Legend at 0x7fe63b6e1760>



In [56]:
```
## Compare NN without using PCA
pd.DataFrame({'model':['NN_PCA','NN_without/PCA','CNN'],
             'Accuracy_test':[NN_accuracy,
                              model_NN2.evaluate(X_test,
                                                 y_test_cnn,
                                                 verbose=0)[1],
                              model_CNN.evaluate(X_test,
                                                 y_test_cnn,
                                                 verbose=0)[1]]})
```

Out[56]:

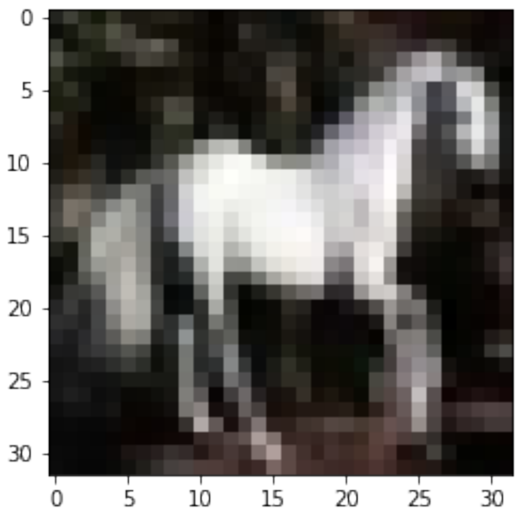
	model	Accuracy_test
0	NN_PCA	0.7760
1	NN_without/PCA	0.7475
2	CNN	0.8435

Comparing with the accuracy of all the models, we go with convolutional neural networks.

Plot an example image and its feature maps

In [37]:
```
## CNN Example
img_index=1
example = X_test[img_index].reshape(32,32,3)
plt.imshow(example)
```

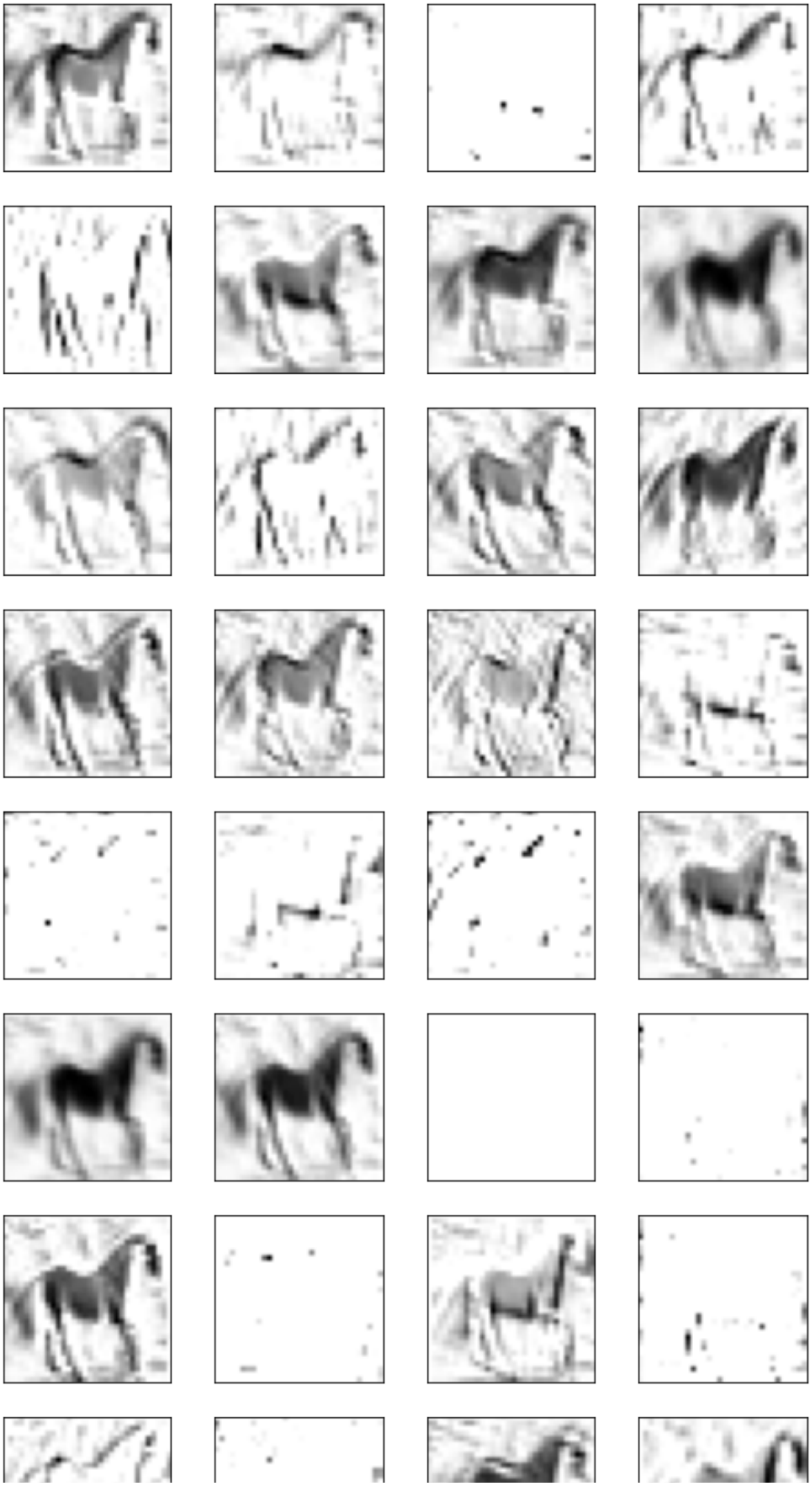
Out[37]: <matplotlib.image.AxesImage at 0x7fe6bfe6d8b0>

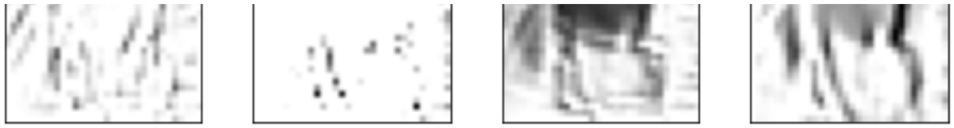


In [38]:
```
# Create list of layer outputs
layer_outputs = [layer.output for layer in model_CNN.layers]

# Create a model that will return the outputs at each layer
layers_model = keras.Model(inputs=model_CNN.input,
                           outputs=layer_outputs)

# Get predictions for each layer of the network
outputs = layers_model.predict(example.reshape(1,32, 32,3))
```

In [39]:
```
#Plot the first convolutional layer and its feature
layer = 0
n_col = 4
n_row = 8
plt.figure(figsize=(2*n_col, 2*n_row))
for j in range(n_row * n_col):
    plt.subplot(n_row, n_col, j + 1)
    plt.imshow(outputs[layer][0, :, :, j], plt.cm.binary)
    plt.xticks(())
    plt.yticks(())
plt.show
```

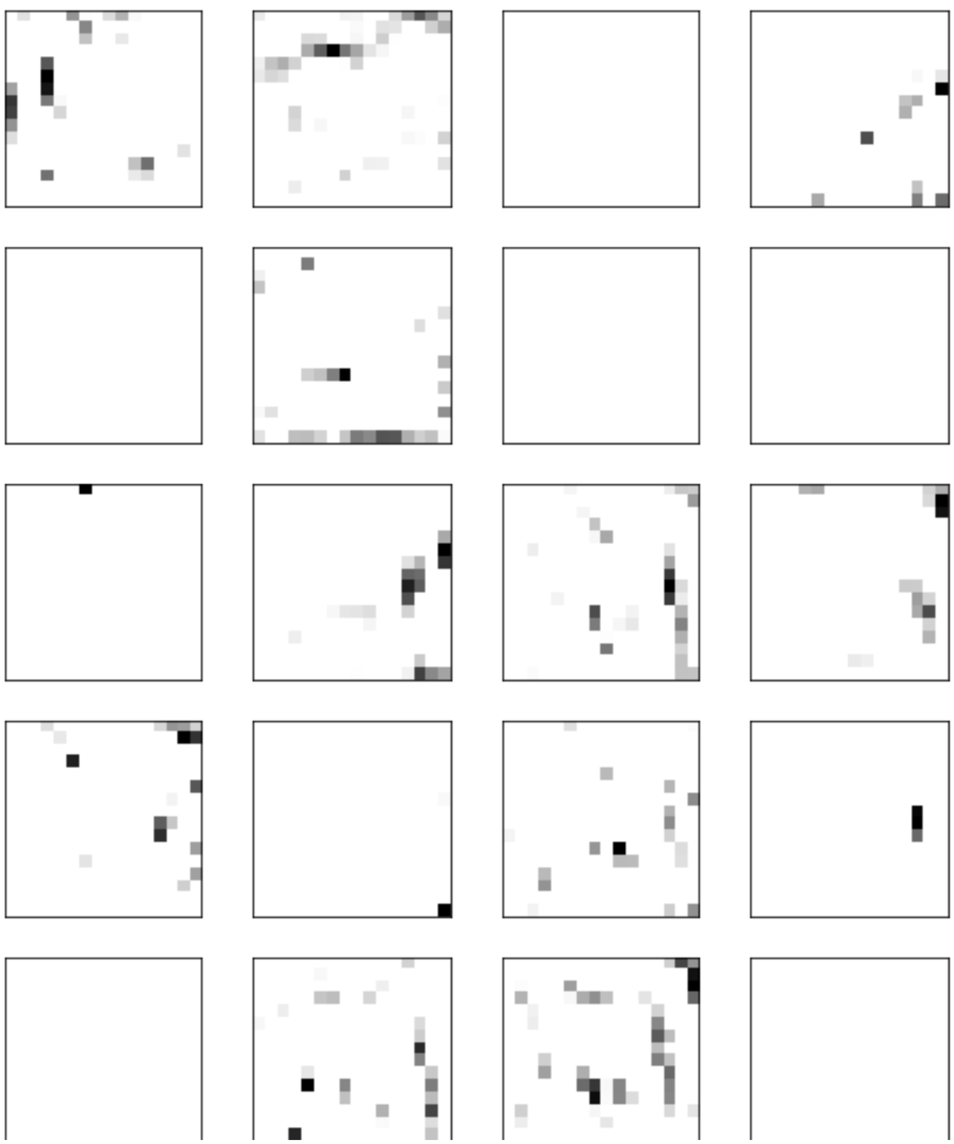
Out[39]: <function matplotlib.pyplot.show(*args, **kw)>

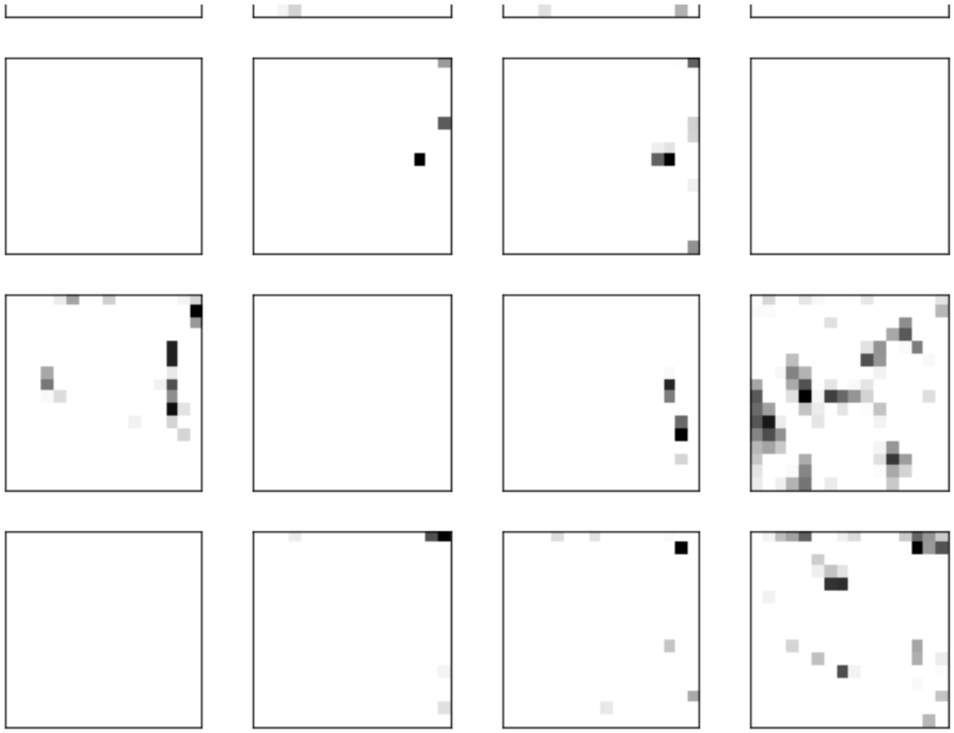




In [40]:
```
#Plot the second convolutional layer and its feature
layer = 2
n_col = 4
n_row = 8
plt.figure(figsize=(2*n_col, 2*n_row))
for j in range(n_row * n_col):
    plt.subplot(n_row, n_col, j + 1)
    plt.imshow(outputs[layer][0, :, :, j], plt.cm.binary)
    plt.xticks(())
    plt.yticks(())
plt.show
```

Out[40]: <function matplotlib.pyplot.show(*args, **kw)>





6. Model Optimization (CNN)

We are stacking convolutional layers with small 3×3 filters followed by a max pooling layer. Together, these layers form a block, and these blocks can be repeated where the number of filters in each block is increased with the depth of the network such as 32, 64, 128 for the first three blocks of the model. In addition, adding Dropout layers after each max pooling layer and after the fully connected layer, and using a fixed dropout rate of 20% (retain 80% of the nodes).

Then we increasing to 3 fully connected layers, and using L2 weight regularization 0.001 on dense layer 3.

Last, we are increasing batch size to 128.

In [41]:
```
model_opt =Sequential()
model_opt.add(Conv2D(32,
                     kernel_size=(3,3),
                     strides=1,
                     activation ='relu',
                     padding='same', input_shape=(32,32,3)))
model_opt.add(Conv2D(32,
                     kernel_size=(3,3),
                     strides=1,
                     activation ='relu',
                     padding='same'))
model_opt.add(MaxPooling2D(pool_size=(2,2),strides=2))
model_opt.add(Dropout(0.2))
model_opt.add(Conv2D(64,
                     kernel_size=(3,3),
                     strides=1,
                     activation ='relu',
                     padding='same'))
model_opt.add(Conv2D(64,
                     kernel_size=(3,3),
                     strides=1,
                     activation ='relu',
                     padding='same'))
model_opt.add(MaxPooling2D(pool_size=(2,2),strides=2))
model_opt.add(Dropout(0.2))
model_opt.add(Conv2D(128,
                     kernel_size=(3,3),
                     strides=1,
                     activation ='relu',
                     padding='same'))
model_opt.add(Conv2D(128,
                     kernel_size=(3,3),
                     strides=1,
                     activation ='relu',
                     padding='same'))
model_opt.add(MaxPooling2D(pool_size=(2,2),strides=2))
model_opt.add(Dropout(0.2))
model_opt.add(Flatten())
model_opt.add(Dense(128, activation = 'relu'))
model_opt.add(Dropout(0.2))
model_opt.add(Dense(50, activation = 'relu'))
model_opt.add(Dropout(0.2))
model_opt.add(Dense(20, activation = 'relu',
                    kernel_regularizer=keras.regularizers.l2(.001)))
model_opt.add(Dropout(0.2))
model_opt.add(Dense(2, activation='softmax'))
```

In [42]:
```
model_opt.compile(optimizer='adam',
                  loss ='categorical_crossentropy',
                  metrics=['accuracy'])
history_opt = model_opt.fit(X_train, y_train_cnn,
                            epochs=50, batch_size=128,
                            validation_data =(X_valid,y_valid_cnn))
```

```
Epoch 1/50
63/63 [==============================] - 15s 242ms/step - loss: 1.31
79 - accuracy: 0.5167 - val_loss: 0.7214 - val_accuracy: 0.4570
Epoch 2/50
63/63 [==============================] - 15s 241ms/step - loss: 0.71
99 - accuracy: 0.5159 - val_loss: 0.7104 - val_accuracy: 0.6495
Epoch 3/50
63/63 [==============================] - 15s 232ms/step - loss: 0.69
39 - accuracy: 0.5863 - val_loss: 0.6788 - val_accuracy: 0.6440
Epoch 4/50
63/63 [==============================] - 15s 235ms/step - loss: 0.62
58 - accuracy: 0.6800 - val_loss: 0.5268 - val_accuracy: 0.7645
Epoch 5/50
63/63 [==============================] - 15s 242ms/step - loss: 0.55
02 - accuracy: 0.7464 - val_loss: 0.4981 - val_accuracy: 0.8030
Epoch 6/50
63/63 [==============================] - 15s 241ms/step - loss: 0.49
45 - accuracy: 0.7865 - val_loss: 0.4095 - val_accuracy: 0.8370
Epoch 7/50
63/63 [==============================] - 15s 231ms/step - loss: 0.45
82 - accuracy: 0.8126 - val_loss: 0.3882 - val_accuracy: 0.8495
Epoch 8/50
63/63 [==============================] - 15s 230ms/step - loss: 0.42
39 - accuracy: 0.8257 - val_loss: 0.3959 - val_accuracy: 0.8380
Epoch 9/50
63/63 [==============================] - 15s 231ms/step - loss: 0.41
41 - accuracy: 0.8346 - val_loss: 0.4155 - val_accuracy: 0.8300
Epoch 10/50
63/63 [==============================] - 14s 229ms/step - loss: 0.36
86 - accuracy: 0.8565 - val_loss: 0.3396 - val_accuracy: 0.8710
Epoch 11/50
63/63 [==============================] - 15s 232ms/step - loss: 0.35
49 - accuracy: 0.8651 - val_loss: 0.3392 - val_accuracy: 0.8685
Epoch 12/50
63/63 [==============================] - 15s 232ms/step - loss: 0.34
04 - accuracy: 0.8687 - val_loss: 0.3028 - val_accuracy: 0.8780
Epoch 13/50
63/63 [==============================] - 15s 233ms/step - loss: 0.33
55 - accuracy: 0.8648 - val_loss: 0.3059 - val_accuracy: 0.8855
Epoch 14/50
63/63 [==============================] - 15s 234ms/step - loss: 0.30
13 - accuracy: 0.8857 - val_loss: 0.2920 - val_accuracy: 0.8820
```

```
Epoch 15/50
63/63 [==============================] - 14s 229ms/step - loss: 0.29
57 - accuracy: 0.8880 - val_loss: 0.2966 - val_accuracy: 0.8880
Epoch 16/50
63/63 [==============================] - 15s 234ms/step - loss: 0.28
16 - accuracy: 0.8913 - val_loss: 0.2803 - val_accuracy: 0.8915
Epoch 17/50
63/63 [==============================] - 15s 231ms/step - loss: 0.26
84 - accuracy: 0.9029 - val_loss: 0.2724 - val_accuracy: 0.8975
Epoch 18/50
63/63 [==============================] - 15s 233ms/step - loss: 0.25
29 - accuracy: 0.9055 - val_loss: 0.2703 - val_accuracy: 0.9035
Epoch 19/50
63/63 [==============================] - 15s 231ms/step - loss: 0.24
12 - accuracy: 0.9118 - val_loss: 0.2573 - val_accuracy: 0.8985
Epoch 20/50
63/63 [==============================] - 15s 237ms/step - loss: 0.23
25 - accuracy: 0.9166 - val_loss: 0.2878 - val_accuracy: 0.8865
Epoch 21/50
63/63 [==============================] - 16s 256ms/step - loss: 0.22
63 - accuracy: 0.9191 - val_loss: 0.2575 - val_accuracy: 0.9010
Epoch 22/50
63/63 [==============================] - 15s 236ms/step - loss: 0.23
10 - accuracy: 0.9159 - val_loss: 0.2793 - val_accuracy: 0.8975
Epoch 23/50
63/63 [==============================] - 15s 240ms/step - loss: 0.21
05 - accuracy: 0.9251 - val_loss: 0.2651 - val_accuracy: 0.9020
Epoch 24/50
63/63 [==============================] - 18s 279ms/step - loss: 0.20
05 - accuracy: 0.9277 - val_loss: 0.2634 - val_accuracy: 0.9000
Epoch 25/50
63/63 [==============================] - 17s 276ms/step - loss: 0.18
96 - accuracy: 0.9339 - val_loss: 0.2619 - val_accuracy: 0.9085
Epoch 26/50
63/63 [==============================] - 19s 294ms/step - loss: 0.17
69 - accuracy: 0.9367 - val_loss: 0.2689 - val_accuracy: 0.9030
Epoch 27/50
63/63 [==============================] - 18s 279ms/step - loss: 0.16
93 - accuracy: 0.9406 - val_loss: 0.2588 - val_accuracy: 0.9060
Epoch 28/50
63/63 [==============================] - 19s 295ms/step - loss: 0.18
15 - accuracy: 0.9354 - val_loss: 0.2691 - val_accuracy: 0.8985
Epoch 29/50
63/63 [==============================] - 19s 294ms/step - loss: 0.15
64 - accuracy: 0.9477 - val_loss: 0.2903 - val_accuracy: 0.9010
Epoch 30/50
63/63 [==============================] - 19s 305ms/step - loss: 0.15
45 - accuracy: 0.9463 - val_loss: 0.2629 - val_accuracy: 0.9075
Epoch 31/50
63/63 [==============================] - 18s 290ms/step - loss: 0.15
```

```
37 - accuracy: 0.9484 - val_loss: 0.2637 - val_accuracy: 0.9105
Epoch 32/50
63/63 [==============================] - 19s 309ms/step - loss: 0.13
06 - accuracy: 0.9535 - val_loss: 0.2740 - val_accuracy: 0.9160
Epoch 33/50
63/63 [==============================] - 17s 271ms/step - loss: 0.12
48 - accuracy: 0.9582 - val_loss: 0.2621 - val_accuracy: 0.9125
Epoch 34/50
63/63 [==============================] - 15s 241ms/step - loss: 0.13
81 - accuracy: 0.9516 - val_loss: 0.2823 - val_accuracy: 0.9065
Epoch 35/50
63/63 [==============================] - 16s 249ms/step - loss: 0.13
92 - accuracy: 0.9529 - val_loss: 0.2512 - val_accuracy: 0.9155
Epoch 36/50
63/63 [==============================] - 15s 243ms/step - loss: 0.12
83 - accuracy: 0.9553 - val_loss: 0.2385 - val_accuracy: 0.9200
Epoch 37/50
63/63 [==============================] - 15s 232ms/step - loss: 0.11
81 - accuracy: 0.9601 - val_loss: 0.2523 - val_accuracy: 0.9185
Epoch 38/50
63/63 [==============================] - 15s 245ms/step - loss: 0.11
21 - accuracy: 0.9616 - val_loss: 0.2483 - val_accuracy: 0.9140
Epoch 39/50
63/63 [==============================] - 15s 237ms/step - loss: 0.11
31 - accuracy: 0.9631 - val_loss: 0.2954 - val_accuracy: 0.9180
Epoch 40/50
63/63 [==============================] - 16s 246ms/step - loss: 0.10
19 - accuracy: 0.9647 - val_loss: 0.2672 - val_accuracy: 0.9210
Epoch 41/50
63/63 [==============================] - 15s 232ms/step - loss: 0.12
02 - accuracy: 0.9601 - val_loss: 0.2747 - val_accuracy: 0.9090
Epoch 42/50
63/63 [==============================] - 15s 238ms/step - loss: 0.10
34 - accuracy: 0.9660 - val_loss: 0.2797 - val_accuracy: 0.9090
Epoch 43/50
63/63 [==============================] - 15s 238ms/step - loss: 0.09
35 - accuracy: 0.9689 - val_loss: 0.2972 - val_accuracy: 0.9130
Epoch 44/50
63/63 [==============================] - 15s 234ms/step - loss: 0.10
92 - accuracy: 0.9649 - val_loss: 0.3716 - val_accuracy: 0.8760
Epoch 45/50
63/63 [==============================] - 15s 231ms/step - loss: 0.10
22 - accuracy: 0.9656 - val_loss: 0.2718 - val_accuracy: 0.9185
Epoch 46/50
63/63 [==============================] - 15s 233ms/step - loss: 0.08
45 - accuracy: 0.9734 - val_loss: 0.3172 - val_accuracy: 0.9150
Epoch 47/50
63/63 [==============================] - 15s 236ms/step - loss: 0.08
81 - accuracy: 0.9725 - val_loss: 0.3206 - val_accuracy: 0.9090
Epoch 48/50
```

```
63/63 [==============================] - 15s 236ms/step - loss: 0.09
61 - accuracy: 0.9689 - val_loss: 0.2702 - val_accuracy: 0.9085
Epoch 49/50
63/63 [==============================] - 15s 237ms/step - loss: 0.09
77 - accuracy: 0.9672 - val_loss: 0.3151 - val_accuracy: 0.9075
Epoch 50/50
63/63 [==============================] - 15s 243ms/step - loss: 0.08
16 - accuracy: 0.9735 - val_loss: 0.2766 - val_accuracy: 0.9105
```

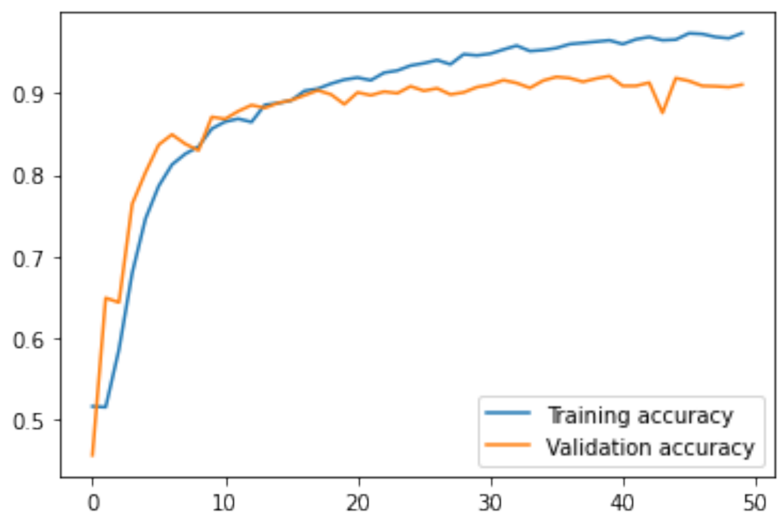
In [43]:
```
#Accuracy on test set
model_opt.evaluate(X_test, y_test_cnn)
```

```
63/63 [==============================] - 1s 9ms/step - loss: 0.2768
- accuracy: 0.9090
```

Out[43]:
```
[0.27681854367256165, 0.9089999794960022]
```

In [44]:
```
#Accuracy plot
plt.plot(history_opt.history['accuracy'],
         label='Training accuracy')
plt.plot(history_opt.history['val_accuracy'],
         label='Validation accuracy')
plt.legend()
```

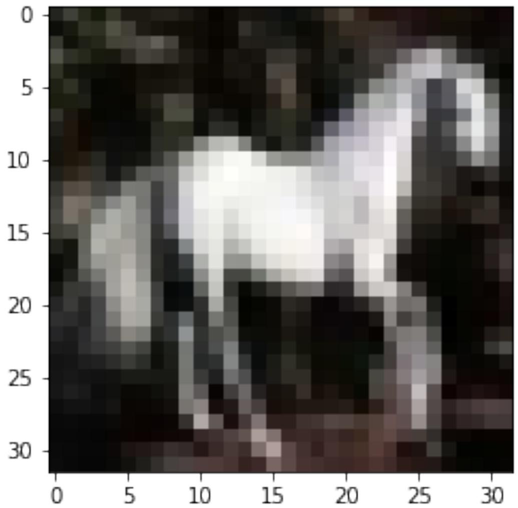
Out[44]:
```
<matplotlib.legend.Legend at 0x7fe634b13790>
```



Plot an example image and its feature maps

In [45]:
```
img_index=1
example = X_test[img_index].reshape(32,32,3)
plt.imshow(example)
```

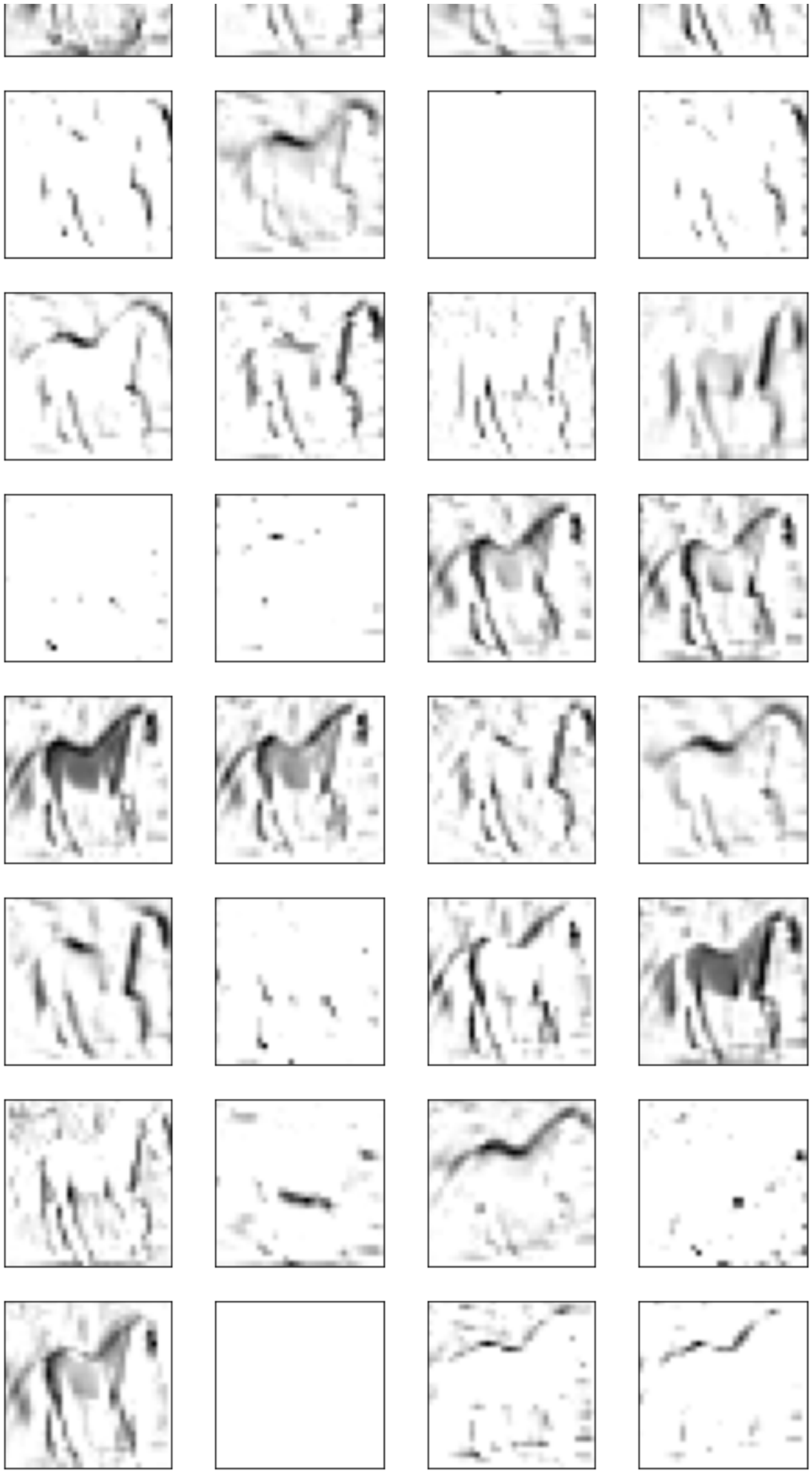
Out[45]: <matplotlib.image.AxesImage at 0x7fe636224190>



In [46]:
```
# Create list of layer outputs
layer_outputs2 = [layer.output for layer in model_opt.layers]

# Create a model that will return the outputs at each layer
layers_model2 = keras.Model(inputs=model_opt.input,
                            outputs=layer_outputs2)

# Get predictions for each layer of the network
outputs2 = layers_model2.predict(example.reshape(1,32, 32, 3))
```

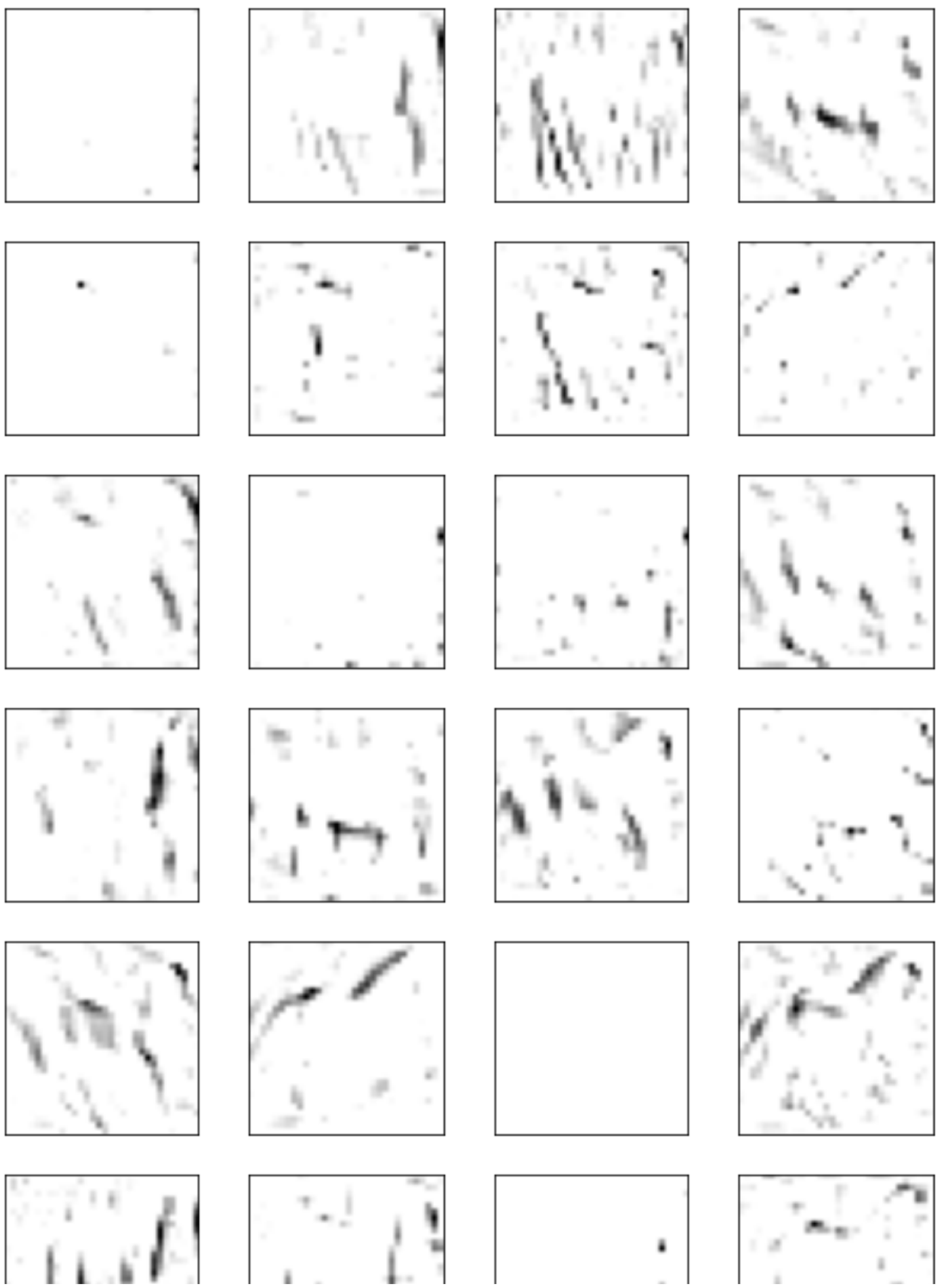
In [47]:
```
#Plot the first convolutional layer and its feature
layer = 0
n_col = 4
n_row = 8
plt.figure(figsize=(2*n_col, 2*n_row))
for j in range(n_row * n_col):
    plt.subplot(n_row, n_col, j + 1)
    plt.imshow(outputs2[layer][0, :, :, j], plt.cm.binary)
    plt.xticks(())
    plt.yticks(())
plt.show
```

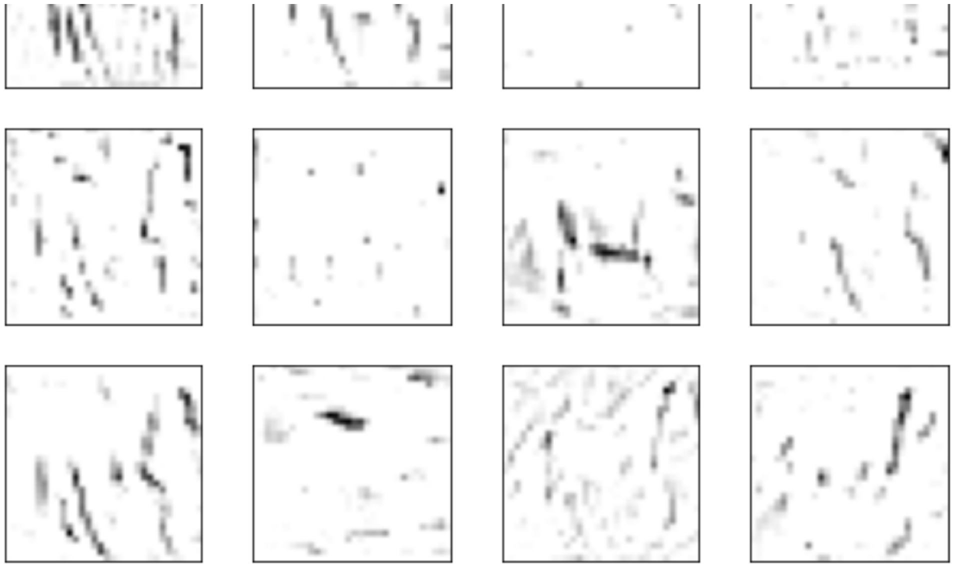
Out[47]: <function matplotlib.pyplot.show(*args, **kw)>



In [48]:
```
#Plot the second convolutional layer and its feature
layer = 1
n_col = 4
n_row = 8
plt.figure(figsize=(2*n_col, 2*n_row))
for j in range(n_row * n_col):
    plt.subplot(n_row, n_col, j + 1)
    plt.imshow(outputs2[layer][0, :, :, j], plt.cm.binary)
    plt.xticks(())
    plt.yticks(())
plt.show
```

Out[48]: <function matplotlib.pyplot.show(*args, **kw)>



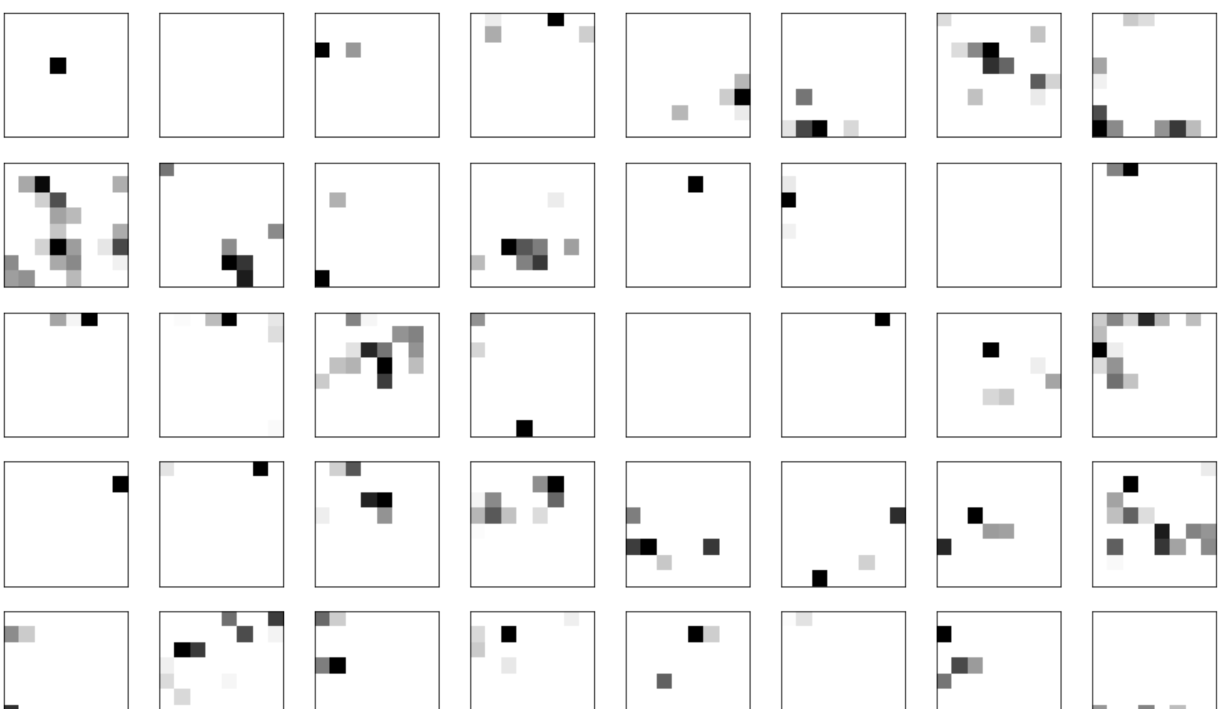


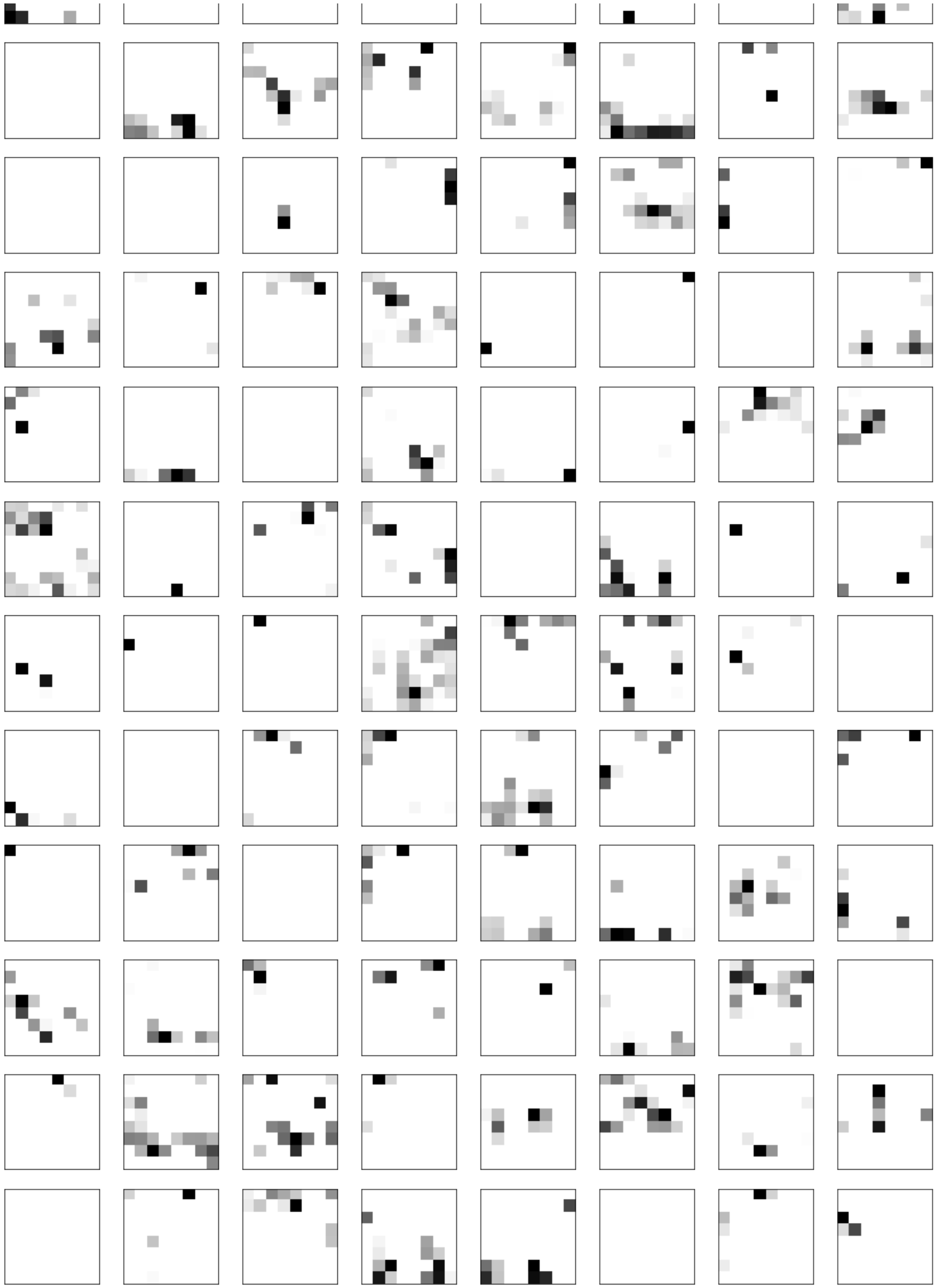
In [49]:

```
#Plot the last convolutional layer and its feature
layer = 9
n_col = 8
n_row = 16
plt.figure(figsize=(2*n_col, 2*n_row))
for j in range(n_row * n_col):
    plt.subplot(n_row, n_col, j + 1)
    plt.imshow(outputs2[layer][0, :, :, j], plt.cm.binary)
    plt.xticks(())
    plt.yticks(())
plt.show
```

Out[49]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```





7. Additional Code

Function for CNN Image Classification

In [50]:
```
# # ------------------------------------------------------------------------
--
# # Creating a function for building a CNN based on arguments supplied
# # ------------------------------------------------------------------------
--
# def f_build_CNN(arg_X_train, arg_y_train, arg_X_test, arg_y_test,
#                 arg_conv_layers,
#                 arg_pool_layers,
#                 arg_out_layers,
#                 arg_compile_parms,
#                 arg_fit_parms):
#     # ----------------------------------------------------------------------
------
#     # Initializing the model
#     # ----------------------------------------------------------------------
------
#     model = Sequential()

#     # ----------------------------------------------------------------------
------
#     # Adding Convolution and Pool Layers alternatively
#     # ----------------------------------------------------------------------
------
#     conv_layers = len(arg_conv_layers)
#     pool_layers = len(arg_pool_layers)
#     all_layers = conv_layers + pool_layers

#     conv_used = 0
#     pool_used = 0
#     prev_layer = None
#     curr_layer = None

#     for i in range(all_layers):
#         if i == 0 and prev_layer is None:
#             if conv_layers > 0:
#                 curr_layer = 'CONV'
#                 conv_used = conv_used + 1
#             else:
#                 curr_layer = 'POOL'
#                 pool_used = pool_used + 1
#         elif prev_layer == 'POOL':
```

```
#             if conv_used < conv_layers:
#                 curr_layer = 'CONV'
#                 conv_used = conv_used + 1
#             else:
#                 curr_layer = 'POOL'
#                 pool_used = pool_used + 1
#         elif prev_layer == 'CONV':
#             if pool_used < pool_layers:
#                 curr_layer = 'POOL'
#                 pool_used = pool_used + 1
#             else:
#                 curr_layer = 'CONV'
#                 conv_used = conv_used + 1
#         else:
#             None

#         if curr_layer == 'CONV':
#             conv_layer = arg_conv_layers[conv_used - 1]
#             model.add(Conv2D(conv_layer[0],
#                              kernel_size = conv_layer[1],
#                              activation  = conv_layer[2],
#                              padding     = conv_layer[3],
#                              input_shape = conv_layer[4]))

#             #print('Current CONV Item : ',
#                     arg_conv_layers[conv_used - 1])
#         else:
#             pool_layer = arg_pool_layers[pool_used - 1]
#             model.add(MaxPooling2D(pool_size = pool_layer[0],
#                                    strides   = pool_layer[1]))
#             #print('Current POOL Item : ',
#                     arg_pool_layers[pool_used - 1])

#         prev_layer = curr_layer
#         curr_layer = None

#     # -------------------------------------------------------------
------
#     # Flattening the image data
#     # -------------------------------------------------------------
------
#     model.add(Flatten())

#     # -------------------------------------------------------------
------
#     # Adding the output layer
#     # -------------------------------------------------------------
------
#     for out_layer in arg_out_layers:
#         model.add(Dense(out_layer[0], activation = out_layer[1]))
```

```
#     # ------------------------------------------------------------------
------
#     # Printing Model Structure
#     # ------------------------------------------------------------------
------
#     print(model.summary())

#     # ------------------------------------------------------------------
------
#     # Compiling the Model
#     # ------------------------------------------------------------------
------
#     model.compile(optimizer = arg_compile_parms[0][0],
#                   loss      = arg_compile_parms[0][1],
#                   metrics   = arg_compile_parms[0][2])

#     # ------------------------------------------------------------------
------
#     # Fitting the Model
#     # ------------------------------------------------------------------
------
#     model_history = model.fit(arg_X_train,
#                               arg_y_train,
#                               epochs   = arg_fit_parms[0][0],
#                               batch_size = arg_fit_parms[0][1],
#                               validation_data= (arg_X_test, arg_y_test))

#     # ------------------------------------------------------------------
------
#     # Printing the model metrics
#     # ------------------------------------------------------------------
------
#     model_accuracy=model.evaluate(arg_X_test, arg_y_test, verbose =
0)[1]
#     print('Model Accuracy is ', model_accuracy)

#     return(model)
```

Function for Fully Connected NN Image Classification

In [51]:

```
# # ----------------------------------------------------------------------
--
# # Creating a function for building a CNN based on arguments supplied
# # ----------------------------------------------------------------------
--
# def f_build_ANN(arg_X_train, arg_y_train, arg_X_test, arg_y_test,
```

```
#                  arg_in_layers,
#                  arg_out_layers,
#                  arg_compile_parms,
#                  arg_fit_parms):
#     # ---------------------------------------------------------------
------
#     # Initializing the model
#     # ---------------------------------------------------------------
------
#     model = Sequential()
#     # ---------------------------------------------------------------
---------------------------------------------
#     # Neural Netwoek Architecture
#     # ---------------------------------------------------------------
---------------------------------------------

#     for in_layer in arg_in_layers:
#         model.add(Dense(units = in_layer[0],
#                        activation = in_layer[1],
#                        input_shape = in_layer[2]))

#     model.add(Flatten())
#     # ---------------------------------------------------------------
------
#     # Adding the output layer
#     # ---------------------------------------------------------------
------
#     for out_layer in arg_out_layers:
#         model.add(Dense(out_layer[0], activation = out_layer[1]))

#     # ---------------------------------------------------------------
---------------------------------------------
#     # Generating Model Summary
#     # ---------------------------------------------------------------
---------------------------------------------
#     model.summary()

#     # ---------------------------------------------------------------
---------------------------------------------
#     # Neural Netwoek Model Compilation
#     # ---------------------------------------------------------------
---------------------------------------------
#     #model.compile(optimizer = 'adam',
#.                   loss = 'mean_squared_error',
#                    metrics = ['mse'])
#     model.compile(optimizer = arg_compile_parms[0][0],
#                   loss      = arg_compile_parms[0][1],
#                   metrics   = arg_compile_parms[0][2])
#     # ---------------------------------------------------------------
------
```

```
#     # Fitting the Model
#     # --------------------------------------------------------------
------
#     model_history = model.fit(arg_X_train,
#                               arg_y_train,
#                               epochs  = arg_fit_parms[0][0],
#                               batch_size = arg_fit_parms[0][1],
#                               validation_data=(arg_X_test, arg_y_test))

#     # --------------------------------------------------------------
------
#     # Printing the model Accuracy
#     # --------------------------------------------------------------
------
#     model_accuracy = model.evaluate(X_test, y_test, verbose = 0)[1]
#     print('Model Accuracy is ', model_accuracy)

#     return model
```