9 X_Xception_centr

April 3, 2023

1 Date: 9 2022

2 Method: Cross_Inception

3 Data: Pavia

4 Results v.05

[ ]:
```
# Libraries
import pandas as pd
import numpy as np
import seaborn as sn
from sklearn.decomposition import PCA
```

[ ]:
```
# Read dataset Pavia
from scipy.io import loadmat

def read_HSI():
  X = loadmat('Pavia.mat')['pavia']
  y = loadmat('Pavia_gt.mat')['pavia_gt']
  print(f"X shape: {X.shape}\ny shape: {y.shape}")
  return X, y

X, y = read_HSI()
```

```
X shape: (1096, 715, 102)
y shape: (1096, 715)
```

[ ]:
```
# PCA
def applyPCA(X, numComponents): # numComponents=64
    newX = np.reshape(X, (-1, X.shape[2]))
    print(newX.shape)
    pca = PCA(n_components=numComponents, whiten=True)
    newX = pca.fit_transform(newX)
    newX = np.reshape(newX, (X.shape[0],X.shape[1], numComponents))
    return newX, pca, pca.explained_variance_ratio_
```

[ ]:
```python
# channel_wise_shift
def channel_wise_shift(X,numComponents):
    X_copy = np.zeros((X.shape[0] , X.shape[1], X.shape[2]))
    half = int(numComponents/2)
    for i in range(0,half-1):
        X_copy[:,:,i] = X[:,:,(half-i)*2-1]
    for i in range(half,numComponents):
        X_copy[:,:,i] = X[:,:,(i-half)*2]
    X = X_copy
    return X
```

[ ]:
```python
# Split the hyperspectral image into patches of size windowSize-by-windowSize␣
↪pixels
def Patches_Creating(X, y, windowSize, removeZeroLabels = True):  #␣
↪windowSize=15, 25
    margin = int((windowSize - 1) / 2)
    zeroPaddedX = padWithZeros(X, margin=margin)
    # split patches
    patchesData = np.zeros((X.shape[0] * X.shape[1], windowSize, windowSize, X.
↪shape[2]),dtype="float16")
    patchesLabels = np.zeros((X.shape[0] * X.shape[1]),dtype="float16")
    patchIndex = 0
    for r in range(margin, zeroPaddedX.shape[0] - margin):
        for c in range(margin, zeroPaddedX.shape[1] - margin):
            patch = zeroPaddedX[r - margin:r + margin + 1, c - margin:c +␣
↪margin + 1]
            patchesData[patchIndex, :, :, :] = patch
            patchesLabels[patchIndex] = y[r-margin, c-margin]
            patchIndex = patchIndex + 1
    if removeZeroLabels:
        patchesData = patchesData[patchesLabels>0,:,:,:]
        patchesLabels = patchesLabels[patchesLabels>0]
        patchesLabels -= 1
    return patchesData, patchesLabels
# pading With Zeros
def padWithZeros(X, margin=2):
    newX = np.zeros((X.shape[0] + 2 * margin, X.shape[1] + 2* margin, X.
↪shape[2]),dtype="float16")
    x_offset = margin
    y_offset = margin
    newX[x_offset:X.shape[0] + x_offset, y_offset:X.shape[1] + y_offset, :] = X
    return newX
```

[ ]:
```python
# Split Data
from sklearn.model_selection import train_test_split

def splitTrainTestSet(X, y, testRatio, randomState=345):
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=testRatio, random_state=randomState,stratify=y)
    return X_train, X_test, y_train, y_test
```

[ ]:
```
test_ratio = 0.5

# Load and reshape data for training
X0, y0 = read_HSI()
#X=X0
#y=y0

windowSize=9  # accuracy of
# Score for fold 1: loss of 0.34631192684173584; accuracy of 89.49999809265137%

# to test: 7, 9, 13, 15,

width = windowSize
height = windowSize
img_width, img_height, img_num_channels = windowSize, windowSize, 3

input_image_size=windowSize
INPUT_IMG_SIZE=windowSize

dimReduction=3

InputShape=(windowSize, windowSize, dimReduction)

#X, y = loadData(dataset) channel_wise_shift
X1,pca,ratio = applyPCA(X0,numComponents=dimReduction)
X2_shifted = channel_wise_shift(X1,dimReduction) # channel-wise shift
#X2=X1

#print(f"X0 shape: {X0.shape}\ny0 shape: {y0.shape}")
#print(f"X1 shape: {X1.shape}\nX2 shape: {X2.shape}")

X3, y3 = Patches_Creating(X2_shifted, y0, windowSize=windowSize)
Xtrain, Xtest, ytrain, ytest = splitTrainTestSet(X3, y3, test_ratio)
```

```
X shape: (1096, 715, 102)
y shape: (1096, 715)
(783640, 102)
```

[ ]:
```
# Compile the model
#incept_model.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
↪metrics=['accuracy'])
```

[ ]:
```python
print()

import warnings
warnings.filterwarnings("ignore")

# load libraries
from keras.initializers import VarianceScaling
from keras.regularizers import l2
from keras.models import Sequential
from keras.layers import Dense
from sklearn import datasets
from sklearn.model_selection import StratifiedKFold
import numpy as np
```

[ ]:
```python
# 9 classes names

names = ['1. Water', '2. Trees', '3. Asphalt', '4. Self-Blocking Bricks',
                 '5. Bitumen','6. Tiles', '7. Shadows',
                 '8. Meadows', '9. Bare Soil']
```

[ ]:
```python
from tensorflow.keras.applications import EfficientNetB0
from keras.applications import densenet, inception_v3, mobilenet, resnet,␣
↪vgg16, vgg19, xception
from tensorflow.keras import layers
from keras.layers import Dense, GlobalAveragePooling2D, Dropout, Flatten
import tensorflow as tf

'''
#model = EfficientNetB0(weights='imagenet')


def build_model(num_classes):
    inputs = layers.Input(shape=(windowSize, windowSize, 3))
    #x = img_augmentation(inputs)
    model = xception.Xception(weights='imagenet', include_top=False,␣
↪input_tensor=inputs)

    #model1 = resnet.ResNet50(weights='imagenet')


    # Freeze the pretrained weights
    model.trainable = False

    # Rebuild top
    x = layers.GlobalAveragePooling2D(name="avg_pool")(model.output)
```

```
    x = layers.BatchNormalization()(x)

    x = model.output


    x = GlobalAveragePooling2D()(x)
    # let's add a fully-connected layer
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.25)(x)


    top_dropout_rate = 0.2
    #x = layers.Dropout(top_dropout_rate, name="top_dropout")(x)
    outputs = layers.Dense(9, activation="softmax", name="pred")(x)

    # Compile
    model = tf.keras.Model(inputs, outputs, name="EfficientNet")
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
    model.compile(
        optimizer=optimizer, loss="categorical_crossentropy",␣
 ↪metrics=["accuracy"]
    )
    return model
'''
```

[ ]:
```
'\'\'\n#model = EfficientNetB0(weights=\'imagenet\')\n\n\ndef
build_model(num_classes):\n    inputs = layers.Input(shape=(windowSize,
windowSize, 3))\n    #x = img_augmentation(inputs)\n    model =
xception.Xception(weights=\'imagenet\', include_top=False,
input_tensor=inputs)\n\n    #model1 =
resnet.ResNet50(weights=\'imagenet\')\n\n\n    # Freeze the pretrained weights\n
model.trainable = False\n\n    # Rebuild top\n    x =
layers.GlobalAveragePooling2D(name="avg_pool")(model.output)\n    x =
layers.BatchNormalization()(x)\n\n    x = model.output\n\n\n    x =
GlobalAveragePooling2D()(x)\n    # let\'s add a fully-connected layer\n    x =
Dense(256, activation=\'relu\')(x)\n    x = Dropout(0.25)(x)\n    \n\n
top_dropout_rate = 0.2\n    #x = layers.Dropout(top_dropout_rate,
name="top_dropout")(x)\n    outputs = layers.Dense(9, activation="softmax",
name="pred")(x)\n\n    # Compile\n    model = tf.keras.Model(inputs, outputs,
name="EfficientNet")\n    optimizer =
tf.keras.optimizers.Adam(learning_rate=1e-3)\n    model.compile(\n
optimizer=optimizer, loss="categorical_crossentropy", metrics=["accuracy"]\n
)\n    return model\n'
```

[ ]:
```
from tensorflow.keras.applications import EfficientNetB0
```

```
def build_model(num_classes):
    inputs = layers.Input(shape=(windowSize, windowSize, 3))
    #x = img_augmentation(inputs)
    #model = EfficientNetB0(include_top=False,  input_tensor=inputs,␣
↪weights="imagenet")
    model = xception.Xception(weights='imagenet', include_top=False,␣
↪input_tensor=inputs)


    # Freeze the pretrained weights
   #model.trainable = False

    # Rebuild top
    x = layers.GlobalAveragePooling2D(name="avg_pool")(model.output)
    x = layers.BatchNormalization()(x)

    top_dropout_rate = 0.2
    x = layers.Dropout(top_dropout_rate, name="top_dropout")(x)
    outputs = layers.Dense(9, activation="softmax", name="pred")(x)

    # Compile
    model = tf.keras.Model(inputs, outputs, name="EfficientNet")
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
    model.compile(
        optimizer=optimizer, loss="categorical_crossentropy",␣
↪metrics=["accuracy"]
    )
    return model
```

[ ]:
```
model = build_model(num_classes=9)
```

[ ]:
```
def unfreeze_model(model):
    # We unfreeze the top 20 layers while leaving BatchNorm layers frozen
    for layer in model.layers[-20:]:
        if not isinstance(layer, layers.BatchNormalization):
            layer.trainable = True

    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
    model.compile(
        optimizer=optimizer, loss="categorical_crossentropy",␣
↪metrics=["accuracy"]
    )
```

[ ]:
```
import matplotlib.pyplot as plt


def plot_hist(hist):
```

```
    plt.plot(hist.history["accuracy"])
    plt.plot(hist.history["val_accuracy"])
    plt.title("model accuracy")
    plt.ylabel("accuracy")
    plt.xlabel("epoch")
    plt.legend(["train", "validation"], loc="upper left")
    plt.show()
```

[ ]:
```
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
from tensorflow.keras import layers

import numpy as np
from sklearn.metrics import confusion_matrix, accuracy_score,␣
↪classification_report, cohen_kappa_score
import matplotlib.pyplot as plt
from keras.applications.inception_resnet_v2 import InceptionResNetV2,␣
↪preprocess_input
from keras.layers import Dense, GlobalAveragePooling2D, Dropout, Flatten
from keras.models import Model

import tensorflow as tf

# configuration
confmat = 0
batch_size = 50
loss_function = sparse_categorical_crossentropy
no_classes = 9
no_epochs = 10
optimizer = Adam()
verbosity = 1
num_folds = 5

NN=len(Xtrain)
NN=500
#NN=5000


input_train=Xtrain[0:NN]
target_train=ytrain[0:NN]

input_test=Xtest[0:NN]
target_test=ytest[0:NN]

# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)
```

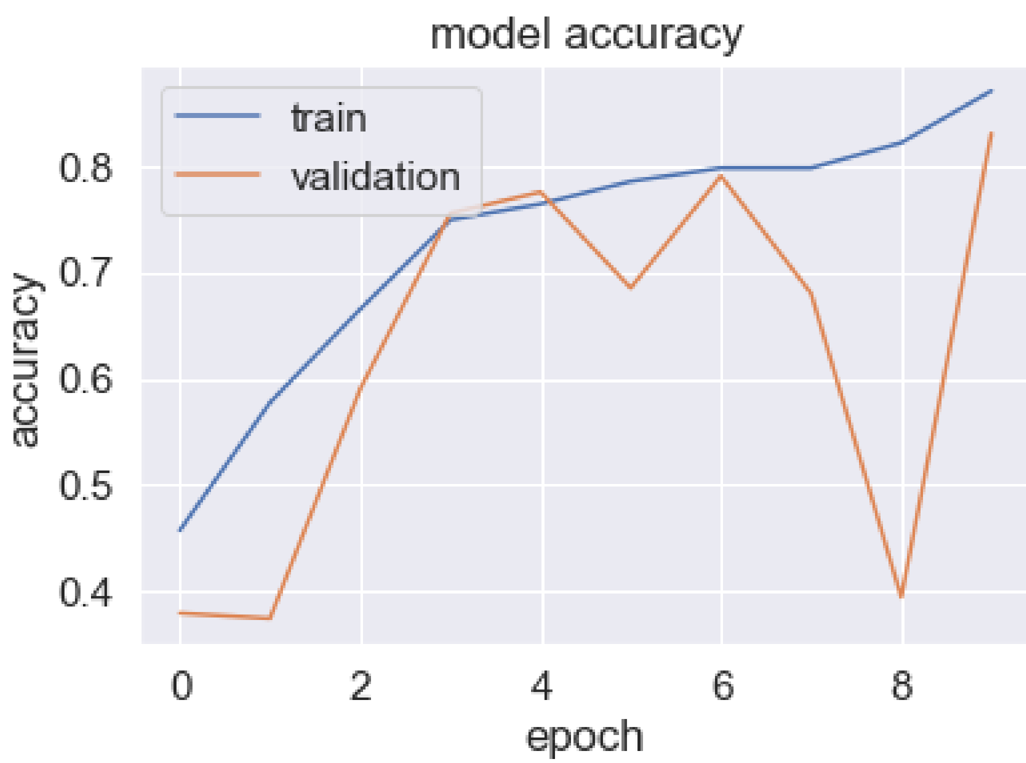
```
# Parse numbers as floats
#input_train = input_train.astype('float32')
#input_test = input_test.astype('float32')

# Normalize data
#input_train = input_train / 255
#input_test = input_test / 255

# Define per-fold score containers
acc_per_fold = []
loss_per_fold = []

Y_pred=[]
y_pred=[]
# Merge inputs and targets
inputs = np.concatenate((input_train, input_test), axis=0)
targets = np.concatenate((target_train, target_test), axis=0)

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(inputs, targets):

  #  model architecture

  # Compile the model
  #model.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])

   # Compile the model
 # model.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])

  model = build_model(num_classes=9)
  #model.compile(loss=loss_function, optimizer='rmsprop',metrics=['accuracy'])

  #model.summary()

  #unfreeze_model(model)
  model.compile(loss=loss_function, optimizer='rmsprop',metrics=['accuracy'])
```

```
# Generate a print
␣
↪print('------------------------------------------------------------------------')
print(f'Training for fold {fold_no} ...')

# Fit data to model
#model.summary()

history = model.fit(inputs[train], targets[train],
            validation_data = (inputs[test],targets[test]),
            epochs=no_epochs,verbose=2 )
plt.figure()
plot_hist(history)
# hist = model.fit(inputs[train], targets[train],
 #                  steps_per_epoch=(29943/batch_size),
 #                  epochs=5,
 #                  validation_data=(inputs[test],targets[test]),
 #                  validation_steps=(8000/batch_size),
 #                  initial_epoch=20,
 #                  verbose=1 )
plt.figure()



# Generate generalization metrics
scores = model.evaluate(inputs[test], targets[test],verbose=0)
print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]};␣
↪{model.metrics_names[1]} of {scores[1]*100}%')
acc_per_fold.append(scores[1] * 100)
loss_per_fold.append(scores[0])

# confusion_matrix
Y_pred = model.predict(inputs[test])
y_pred = np.argmax(Y_pred, axis=1)
#target_test=targets[test]

confusion = confusion_matrix(targets[test], y_pred)
df_cm = pd.DataFrame(confusion, columns=np.unique(names), index = np.
↪unique(names))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (9,9))
sn.set(font_scale=1.4)#for label size
sn.heatmap(df_cm, cmap="Reds", annot=True,annot_kws={"size": 16}, fmt='d')
plt.savefig('cmap.png', dpi=300)
print(confusion_matrix(targets[test], y_pred))
```

```
  confmat    = confmat + confusion;


  # Increase fold number
  fold_no = fold_no + 1

# == average scores ==
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
 ␣
 ↪print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy:␣
 ↪{acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'> Loss: {np.mean(loss_per_fold)}')
print('------------------------------------------------------------------------')

Overall_Conf = pd.DataFrame(confmat, columns=np.unique(names), index = np.
 ↪unique(names))
Overall_Conf.index.name = 'Actual Overall'
Overall_Conf.columns.name = 'Predicted Overall'
plt.figure(figsize = (10,8))
sn.set(font_scale=1.4)#for label size
sn.heatmap(Overall_Conf, cmap="Reds", annot=True,annot_kws={"size": 16},␣
 ↪fmt='d')
plt.savefig('cmap.png', dpi=300)
print(Overall_Conf)


# Notes for next trial

# windowsize=25 __> will work
# windowsize=5  --> Only Basyesian will work
# Need to test (7, 9, 11, 13, 15) window sizes
# When the accuracy is decreasing, it's not right.
# When need to get acc over 0.7
```

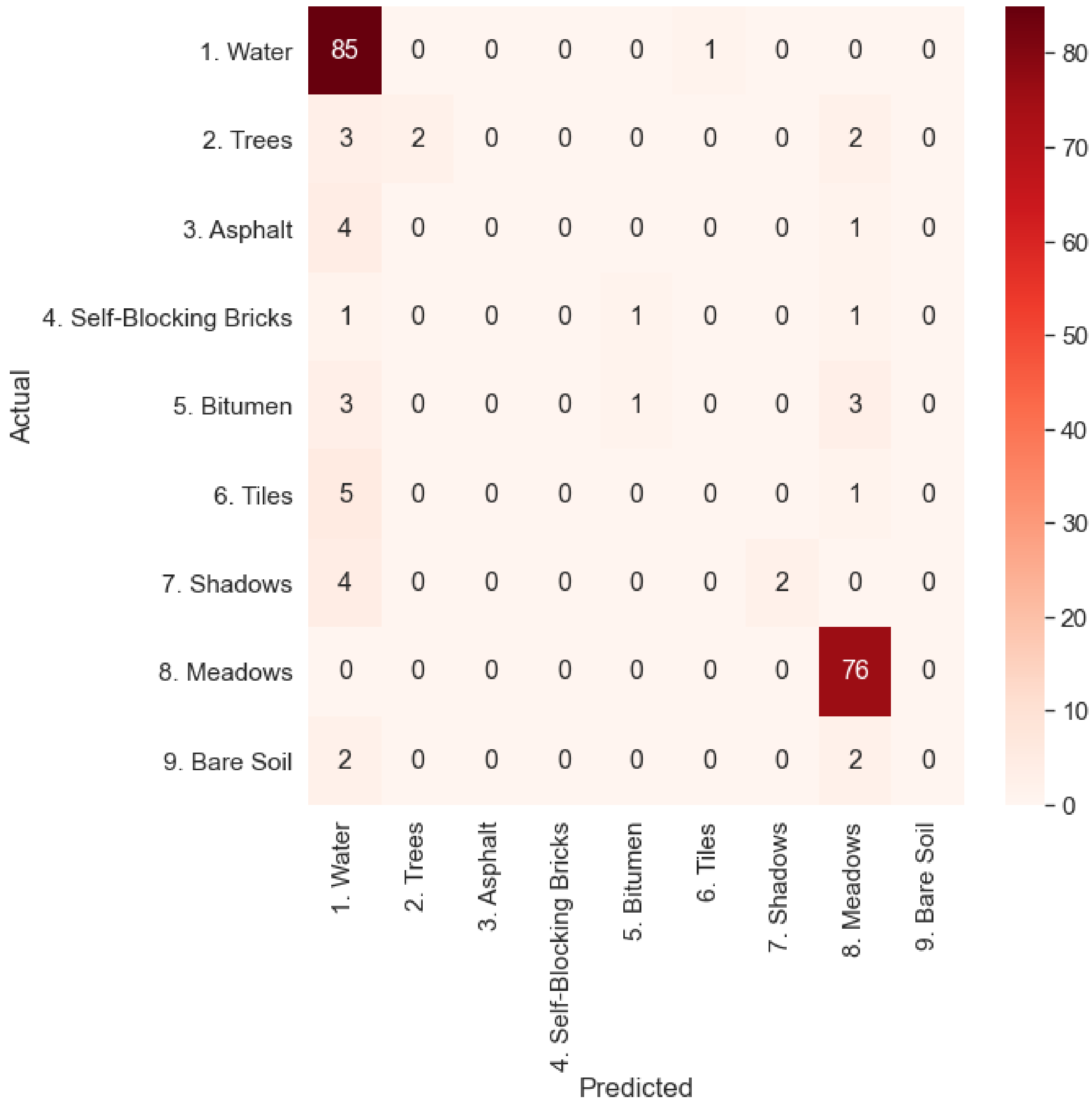
```
------------------------------------------------------------------------
Training for fold 1 …
Epoch 1/10
25/25 - 29s - loss: 1.7123 - accuracy: 0.4575 - val_loss: 1.4662 - val_accuracy:
0.3800 - 29s/epoch - 1s/step
Epoch 2/10
25/25 - 21s - loss: 1.4465 - accuracy: 0.5775 - val_loss: 1.5099 - val_accuracy:
```

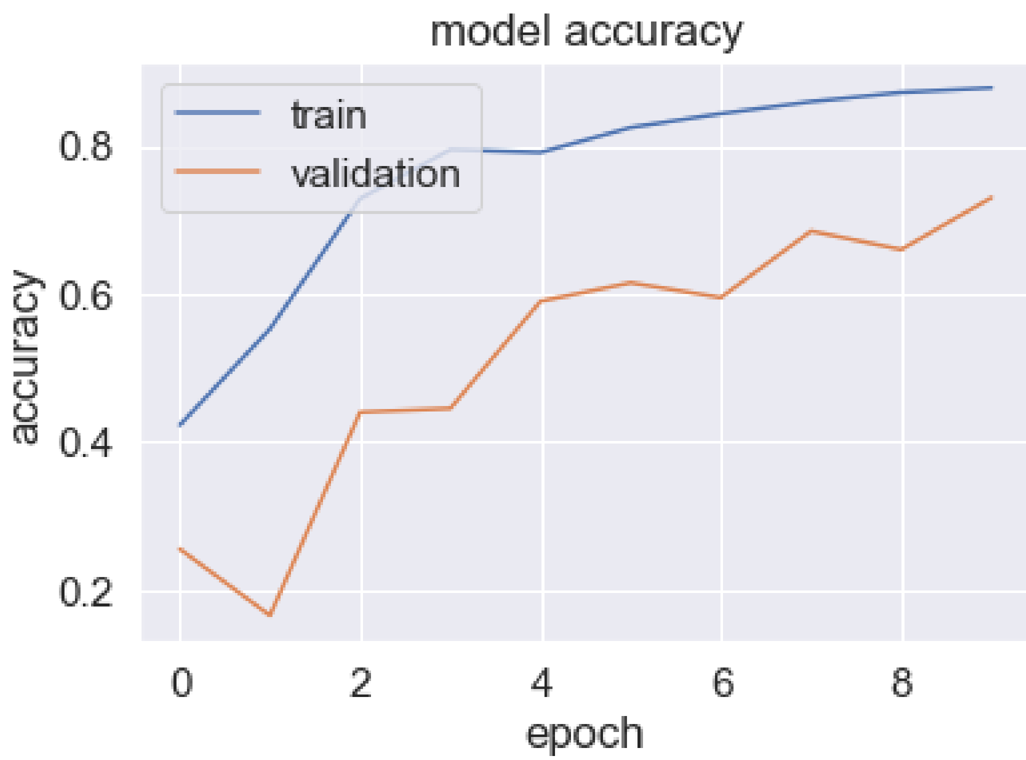
```
0.3750 - 21s/epoch - 856ms/step
Epoch 3/10
25/25 - 20s - loss: 1.2599 - accuracy: 0.6650 - val_loss: 1.3686 - val_accuracy:
0.5900 - 20s/epoch - 784ms/step
Epoch 4/10
25/25 - 21s - loss: 0.8477 - accuracy: 0.7487 - val_loss: 1.0948 - val_accuracy:
0.7550 - 21s/epoch - 826ms/step
Epoch 5/10
25/25 - 20s - loss: 0.7333 - accuracy: 0.7638 - val_loss: 1.0152 - val_accuracy:
0.7750 - 20s/epoch - 799ms/step
Epoch 6/10
25/25 - 21s - loss: 0.6453 - accuracy: 0.7850 - val_loss: 1.1753 - val_accuracy:
0.6850 - 21s/epoch - 853ms/step
Epoch 7/10
25/25 - 23s - loss: 0.6242 - accuracy: 0.7975 - val_loss: 1.1643 - val_accuracy:
0.7900 - 23s/epoch - 928ms/step
Epoch 8/10
25/25 - 18s - loss: 0.7174 - accuracy: 0.7975 - val_loss: 1.9180 - val_accuracy:
0.6800 - 18s/epoch - 735ms/step
Epoch 9/10
25/25 - 18s - loss: 0.5705 - accuracy: 0.8213 - val_loss: 46.4534 -
val_accuracy: 0.3950 - 18s/epoch - 709ms/step
Epoch 10/10
25/25 - 18s - loss: 0.4174 - accuracy: 0.8700 - val_loss: 0.5095 - val_accuracy:
0.8300 - 18s/epoch - 732ms/step
```



```
Score for fold 1: loss of 0.5095426440238953; accuracy of 82.99999833106995%
7/7 [==============================] - 2s 29ms/step
[[85  0  0  0  0  1  0  0  0]
 [ 3  2  0  0  0  0  0  2  0]
 [ 4  0  0  0  0  0  0  1  0]
 [ 1  0  0  0  1  0  0  1  0]
 [ 3  0  0  0  1  0  0  3  0]
 [ 5  0  0  0  0  0  0  1  0]
 [ 4  0  0  0  0  0  2  0  0]
 [ 0  0  0  0  0  0  0 76  0]
 [ 2  0  0  0  0  0  0  2  0]]
------------------------------------------------------------------------
Training for fold 2 …
Epoch 1/10
25/25 - 24s - loss: 1.6941 - accuracy: 0.4225 - val_loss: 1.6912 - val_accuracy:
0.2550 - 24s/epoch - 945ms/step
Epoch 2/10
25/25 - 18s - loss: 1.4432 - accuracy: 0.5525 - val_loss: 1.7553 - val_accuracy:
0.1650 - 18s/epoch - 724ms/step
Epoch 3/10
25/25 - 18s - loss: 0.9514 - accuracy: 0.7287 - val_loss: 1.6424 - val_accuracy:
0.4400 - 18s/epoch - 731ms/step
Epoch 4/10
25/25 - 18s - loss: 0.6897 - accuracy: 0.7950 - val_loss: 1.6467 - val_accuracy:
0.4450 - 18s/epoch - 725ms/step
Epoch 5/10
25/25 - 19s - loss: 0.6490 - accuracy: 0.7912 - val_loss: 1.4570 - val_accuracy:
0.5900 - 19s/epoch - 748ms/step
Epoch 6/10
25/25 - 18s - loss: 0.5817 - accuracy: 0.8250 - val_loss: 1.3576 - val_accuracy:
0.6150 - 18s/epoch - 729ms/step
Epoch 7/10
25/25 - 18s - loss: 0.5123 - accuracy: 0.8438 - val_loss: 1.7497 - val_accuracy:
0.5950 - 18s/epoch - 739ms/step
Epoch 8/10
25/25 - 20s - loss: 0.4430 - accuracy: 0.8600 - val_loss: 1.1898 - val_accuracy:
0.6850 - 20s/epoch - 802ms/step
Epoch 9/10
25/25 - 19s - loss: 0.4242 - accuracy: 0.8725 - val_loss: 1.0147 - val_accuracy:
0.6600 - 19s/epoch - 768ms/step
Epoch 10/10
25/25 - 19s - loss: 0.4063 - accuracy: 0.8788 - val_loss: 1.1509 - val_accuracy:
0.7300 - 19s/epoch - 744ms/step
```

```
<Figure size 432x288 with 0 Axes>
```

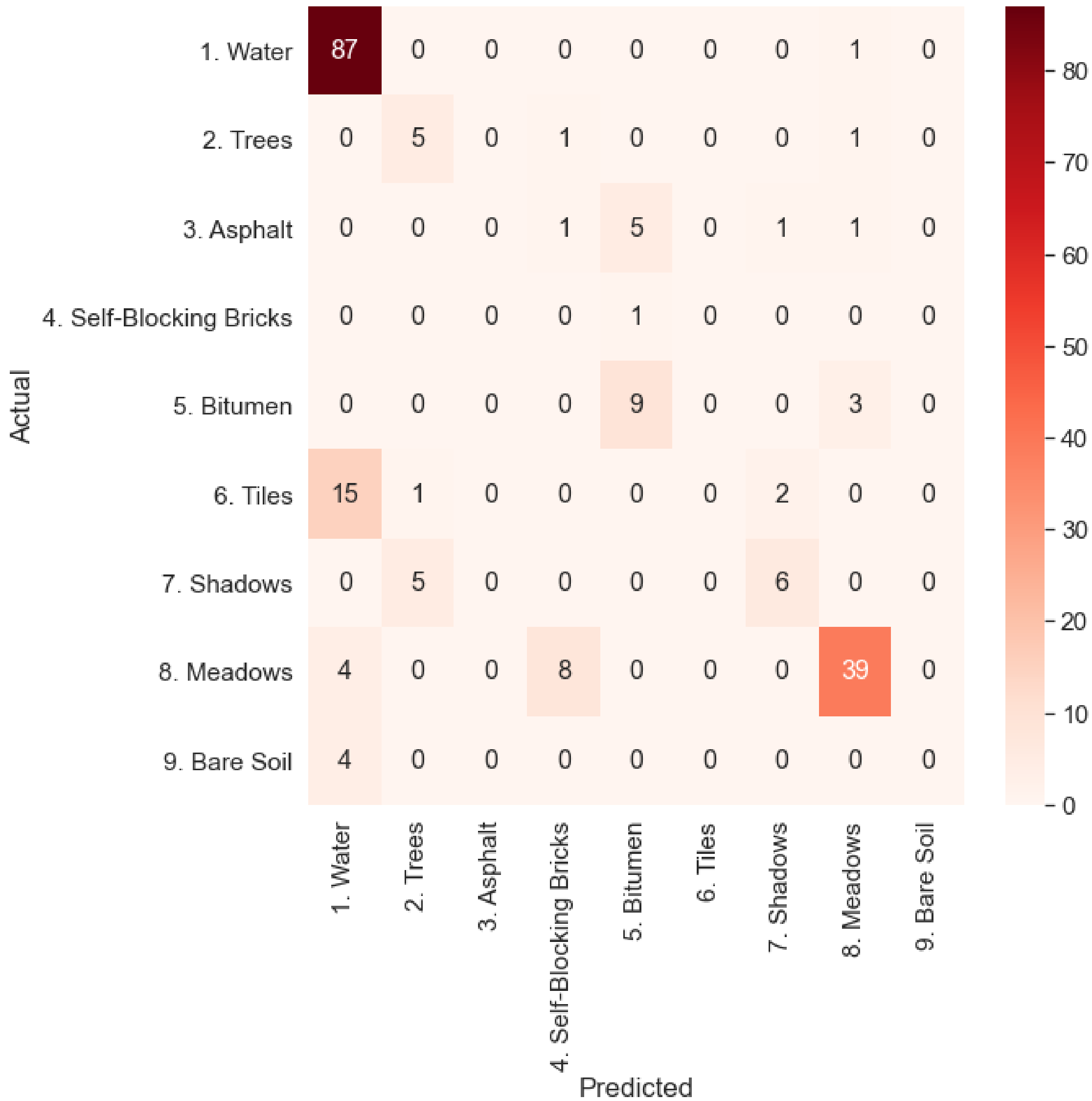

Actual
Predicted
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
85 0 0 0 0 1 0 0 0
3 2 0 0 0 0 0 2 0
4 0 0 0 0 0 0 1 0
1 0 0 0 1 0 0 1 0
3 0 0 0 1 0 0 3 0
5 0 0 0 0 0 0 1 0
4 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 76 0
2 0 0 0 0 0 0 2 0
80
70
60
50
40
30
20
10
0

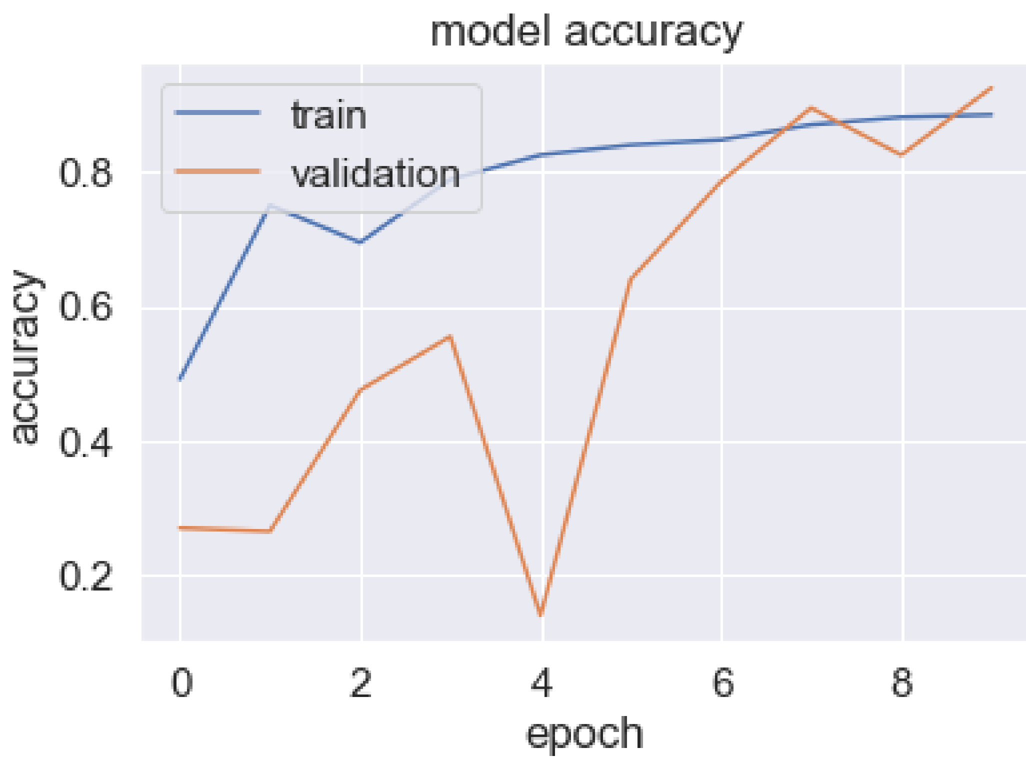



```
Score for fold 2: loss of 1.1509346961975098; accuracy of 73.00000190734863%
7/7 [==============================] - 1s 24ms/step
[[87  0  0  0  0  0  0  1  0]
 [ 0  5  0  1  0  0  0  1  0]
 [ 0  0  0  1  5  0  1  1  0]
 [ 0  0  0  0  1  0  0  0  0]
 [ 0  0  0  0  9  0  0  3  0]
 [15  1  0  0  0  0  2  0  0]
 [ 0  5  0  0  0  0  6  0  0]
 [ 4  0  0  8  0  0  0 39  0]
 [ 4  0  0  0  0  0  0  0  0]]
------------------------------------------------------------------------
Training for fold 3 …
Epoch 1/10
25/25 - 24s - loss: 1.6827 - accuracy: 0.4913 - val_loss: 1.7753 - val_accuracy:
0.2700 - 24s/epoch - 971ms/step
Epoch 2/10
25/25 - 18s - loss: 0.9101 - accuracy: 0.7500 - val_loss: 1.5698 - val_accuracy:
0.2650 - 18s/epoch - 740ms/step
Epoch 3/10
25/25 - 18s - loss: 0.9171 - accuracy: 0.6950 - val_loss: 1.5099 - val_accuracy:
0.4750 - 18s/epoch - 722ms/step
Epoch 4/10
```

```
25/25 - 19s - loss: 0.6719 - accuracy: 0.7887 - val_loss: 1.4117 - val_accuracy:
0.5550 - 19s/epoch - 745ms/step
Epoch 5/10
25/25 - 19s - loss: 0.6556 - accuracy: 0.8250 - val_loss: 14.1231 -
val_accuracy: 0.1400 - 19s/epoch - 775ms/step
Epoch 6/10
25/25 - 19s - loss: 0.5747 - accuracy: 0.8400 - val_loss: 1.2833 - val_accuracy:
0.6400 - 19s/epoch - 754ms/step
Epoch 7/10
25/25 - 20s - loss: 0.4872 - accuracy: 0.8475 - val_loss: 0.7609 - val_accuracy:
0.7850 - 20s/epoch - 814ms/step
Epoch 8/10
25/25 - 18s - loss: 0.4045 - accuracy: 0.8700 - val_loss: 0.4981 - val_accuracy:
0.8950 - 18s/epoch - 717ms/step
Epoch 9/10
25/25 - 18s - loss: 0.3560 - accuracy: 0.8813 - val_loss: 0.7362 - val_accuracy:
0.8250 - 18s/epoch - 719ms/step
Epoch 10/10
25/25 - 18s - loss: 0.3612 - accuracy: 0.8850 - val_loss: 0.3687 - val_accuracy:
0.9250 - 18s/epoch - 721ms/step
```

```
<Figure size 432x288 with 0 Axes>
```

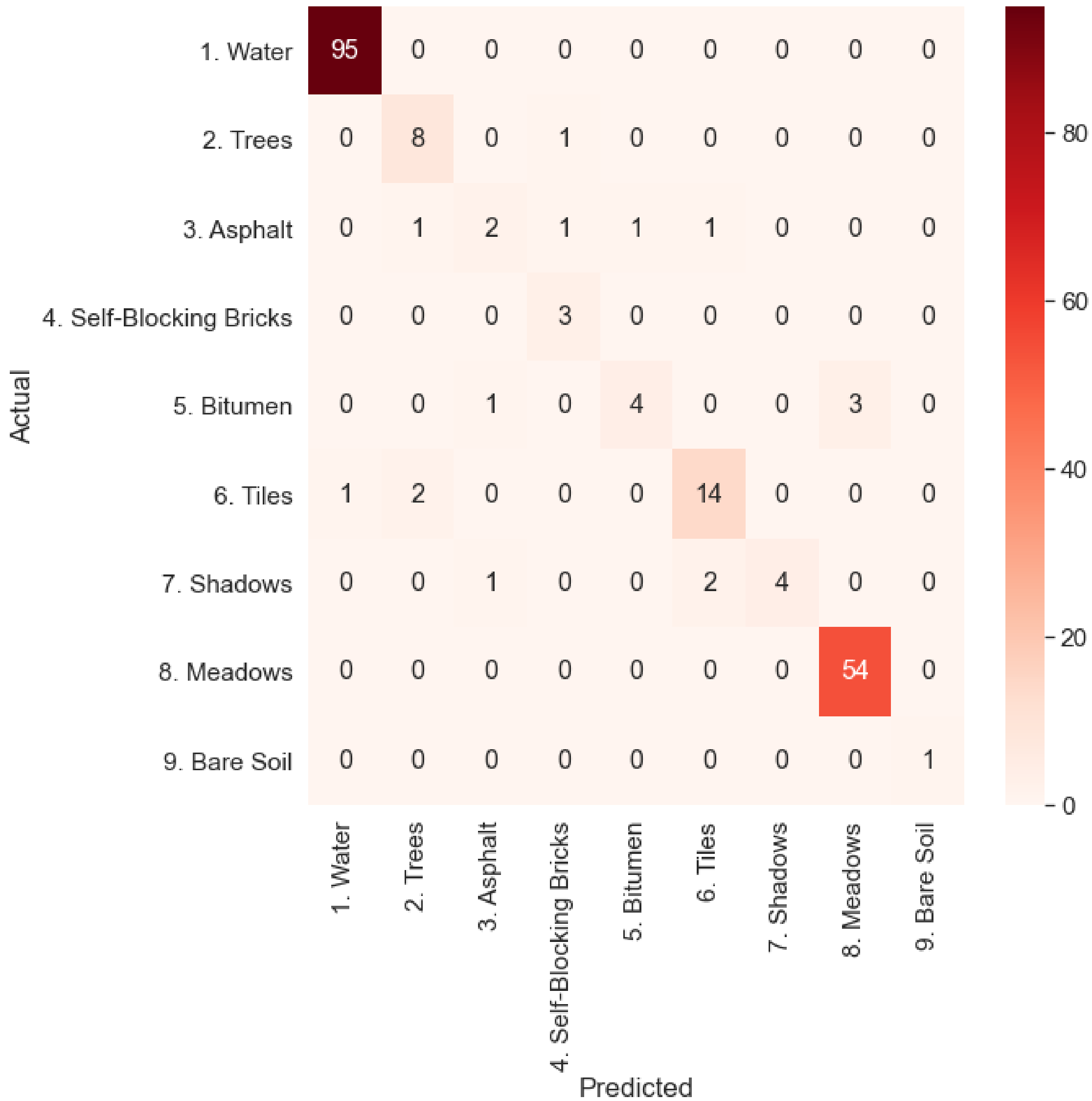

1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
87 0 0 0 0 0 0 1 0
0 5 0 1 0 0 0 1 0
0 0 0 1 5 0 1 1 0
0 0 0 0 1 0 0 0 0
0 0 0 0 9 0 0 3 0
15 1 0 0 0 0 2 0 0
0 5 0 0 0 0 6 0 0
4 0 0 8 0 0 0 39 0
4 0 0 0 0 0 0 0 0
80
70
60
50
40
30
20
10
0

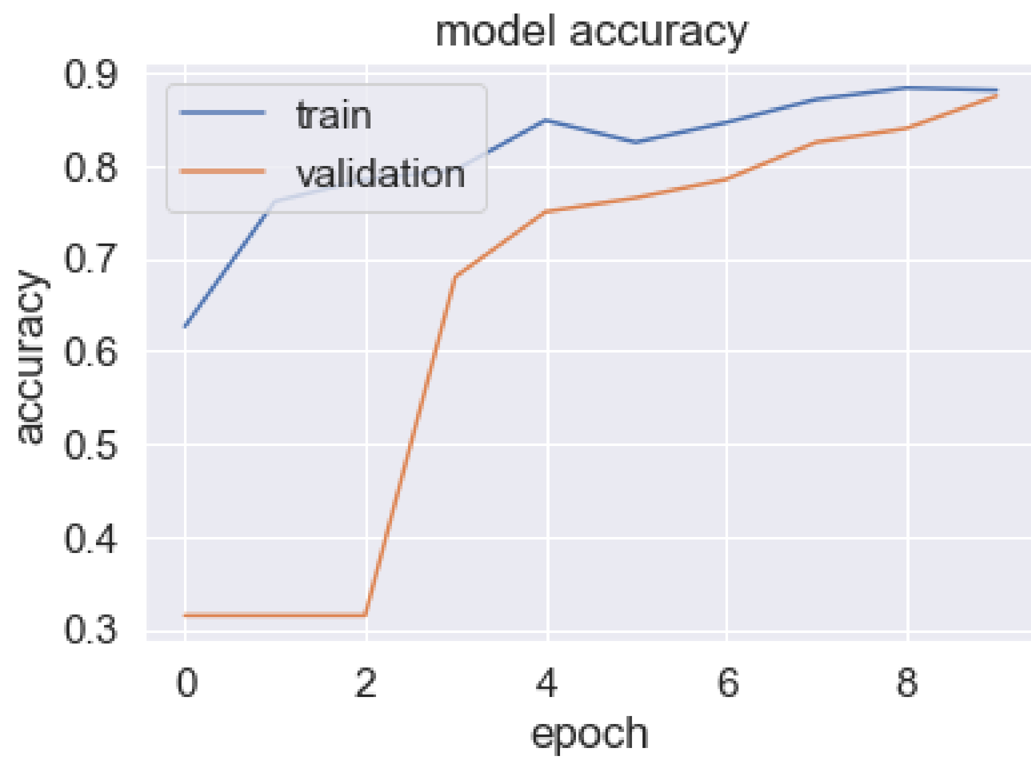



```
Score for fold 3: loss of 0.3687174320220947; accuracy of 92.5000011920929%
7/7 [==============================] - 1s 26ms/step
[[95  0  0  0  0  0  0  0  0]
 [ 0  8  0  1  0  0  0  0  0]
 [ 0  1  2  1  1  1  0  0  0]
 [ 0  0  0  3  0  0  0  0  0]
 [ 0  0  1  0  4  0  0  3  0]
 [ 1  2  0  0  0 14  0  0  0]
 [ 0  0  1  0  0  2  4  0  0]
 [ 0  0  0  0  0  0  0 54  0]
 [ 0  0  0  0  0  0  0  0  1]]
------------------------------------------------------------------------
Training for fold 4 …
Epoch 1/10
25/25 - 24s - loss: 1.3866 - accuracy: 0.6263 - val_loss: 3.0797 - val_accuracy:
0.3150 - 24s/epoch - 966ms/step
Epoch 2/10
25/25 - 18s - loss: 0.8369 - accuracy: 0.7613 - val_loss: 14.4341 -
val_accuracy: 0.3150 - 18s/epoch - 715ms/step
Epoch 3/10
25/25 - 18s - loss: 0.6867 - accuracy: 0.7850 - val_loss: 28.7018 -
val_accuracy: 0.3150 - 18s/epoch - 724ms/step
Epoch 4/10
```

```
25/25 - 18s - loss: 0.7099 - accuracy: 0.7950 - val_loss: 2.6674 - val_accuracy:
0.6800 - 18s/epoch - 726ms/step
Epoch 5/10
25/25 - 18s - loss: 0.5495 - accuracy: 0.8487 - val_loss: 0.9990 - val_accuracy:
0.7500 - 18s/epoch - 723ms/step
Epoch 6/10
25/25 - 18s - loss: 0.6161 - accuracy: 0.8250 - val_loss: 0.8497 - val_accuracy:
0.7650 - 18s/epoch - 712ms/step
Epoch 7/10
25/25 - 20s - loss: 0.4480 - accuracy: 0.8462 - val_loss: 0.7280 - val_accuracy:
0.7850 - 20s/epoch - 781ms/step
Epoch 8/10
25/25 - 20s - loss: 0.4856 - accuracy: 0.8712 - val_loss: 0.5983 - val_accuracy:
0.8250 - 20s/epoch - 791ms/step
Epoch 9/10
25/25 - 20s - loss: 0.3723 - accuracy: 0.8838 - val_loss: 0.5022 - val_accuracy:
0.8400 - 20s/epoch - 817ms/step
Epoch 10/10
25/25 - 21s - loss: 0.4019 - accuracy: 0.8813 - val_loss: 3.4994 - val_accuracy:
0.8750 - 21s/epoch - 829ms/step
```

```
<Figure size 432x288 with 0 Axes>
```

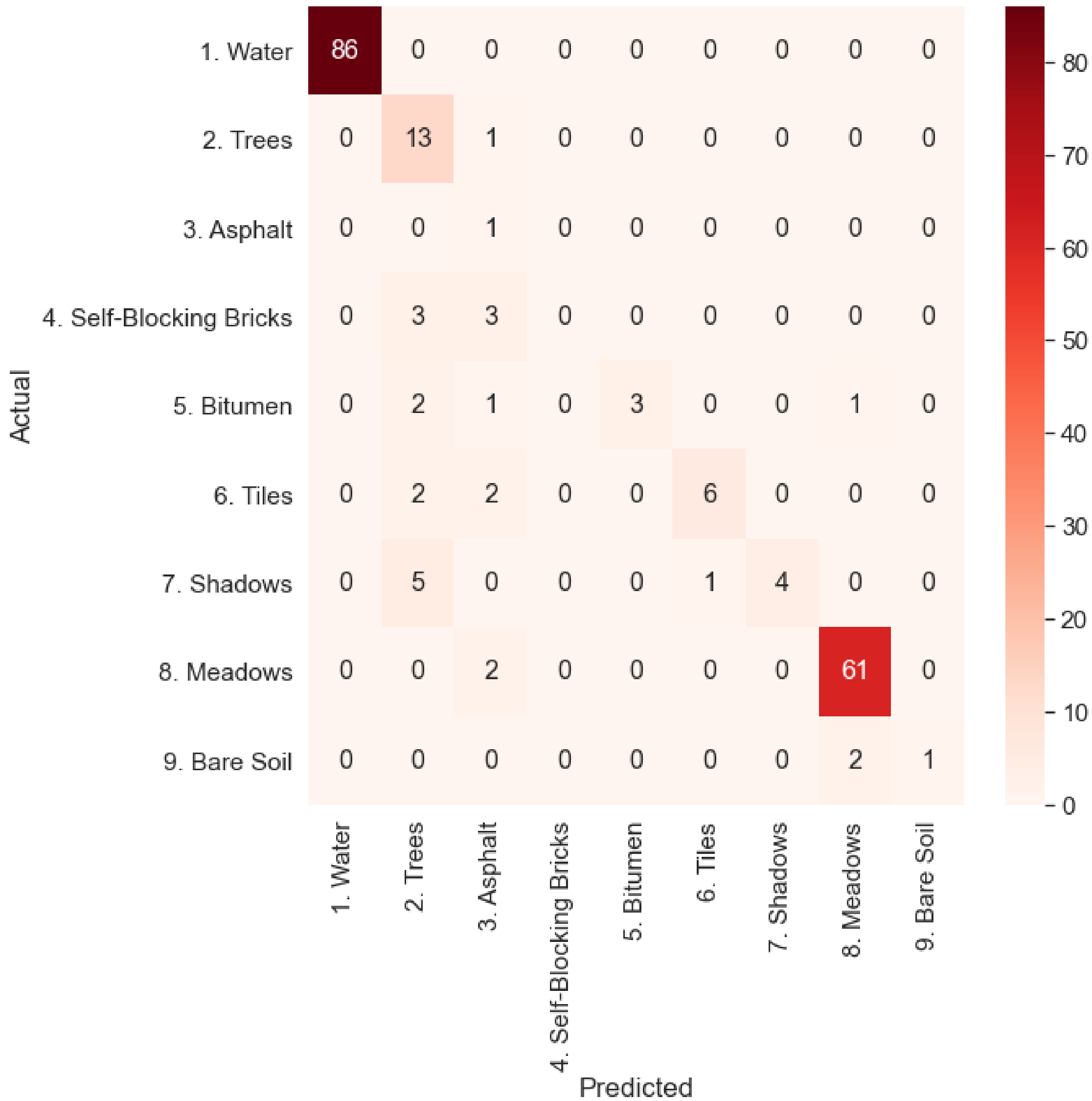

Actual
Predicted
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
95 0 0 0 0 0 0 0 0
0 8 0 1 0 0 0 0 0
0 1 2 1 1 1 0 0 0
0 0 0 3 0 0 0 0 0
0 0 1 0 4 0 0 3 0
1 2 0 0 0 14 0 0 0
0 0 1 0 0 2 4 0 0
0 0 0 0 0 0 0 54 0
0 0 0 0 0 0 0 0 1
0
20
40
60
80

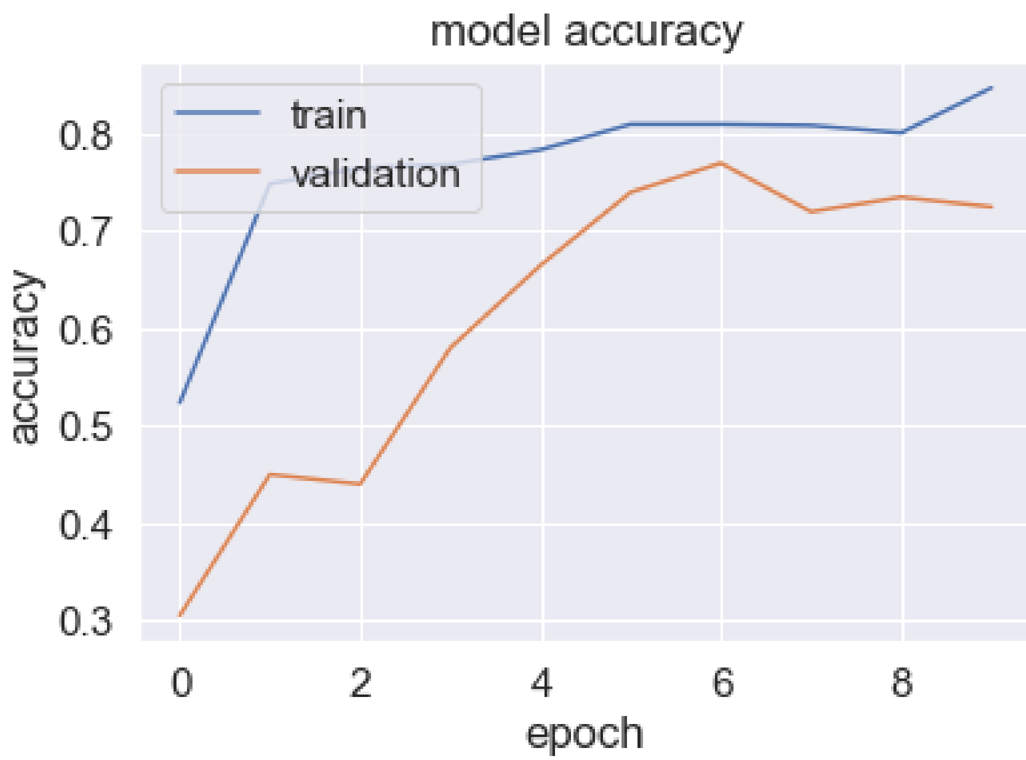



```
Score for fold 4: loss of 3.4993598461151123; accuracy of 87.5%
7/7 [==============================] - 1s 27ms/step
[[86  0  0  0  0  0  0  0  0]
 [ 0 13  1  0  0  0  0  0  0]
 [ 0  0  1  0  0  0  0  0  0]
 [ 0  3  3  0  0  0  0  0  0]
 [ 0  2  1  0  3  0  0  1  0]
 [ 0  2  2  0  0  6  0  0  0]
 [ 0  5  0  0  0  1  4  0  0]
 [ 0  0  2  0  0  0  0 61  0]
 [ 0  0  0  0  0  0  0  2  1]]
------------------------------------------------------------------------
Training for fold 5 …
Epoch 1/10
25/25 - 24s - loss: 1.6170 - accuracy: 0.5238 - val_loss: 1.9182 - val_accuracy:
0.3050 - 24s/epoch - 948ms/step
Epoch 2/10
25/25 - 17s - loss: 0.8814 - accuracy: 0.7487 - val_loss: 1.5470 - val_accuracy:
0.4500 - 17s/epoch - 697ms/step
Epoch 3/10
25/25 - 18s - loss: 0.8046 - accuracy: 0.7650 - val_loss: 1.5658 - val_accuracy:
0.4400 - 18s/epoch - 718ms/step
Epoch 4/10
```

```
25/25 - 18s - loss: 0.7514 - accuracy: 0.7688 - val_loss: 1.3317 - val_accuracy:
0.5800 - 18s/epoch - 707ms/step
Epoch 5/10
25/25 - 18s - loss: 0.7173 - accuracy: 0.7837 - val_loss: 1.1516 - val_accuracy:
0.6650 - 18s/epoch - 706ms/step
Epoch 6/10
25/25 - 18s - loss: 0.6109 - accuracy: 0.8100 - val_loss: 0.9179 - val_accuracy:
0.7400 - 18s/epoch - 722ms/step
Epoch 7/10
25/25 - 19s - loss: 0.6155 - accuracy: 0.8100 - val_loss: 0.9234 - val_accuracy:
0.7700 - 19s/epoch - 743ms/step
Epoch 8/10
25/25 - 18s - loss: 0.6522 - accuracy: 0.8087 - val_loss: 1.7151 - val_accuracy:
0.7200 - 18s/epoch - 711ms/step
Epoch 9/10
25/25 - 18s - loss: 0.5887 - accuracy: 0.8012 - val_loss: 1.8540 - val_accuracy:
0.7350 - 18s/epoch - 715ms/step
Epoch 10/10
25/25 - 19s - loss: 0.4861 - accuracy: 0.8475 - val_loss: 1.9776 - val_accuracy:
0.7250 - 19s/epoch - 762ms/step
```

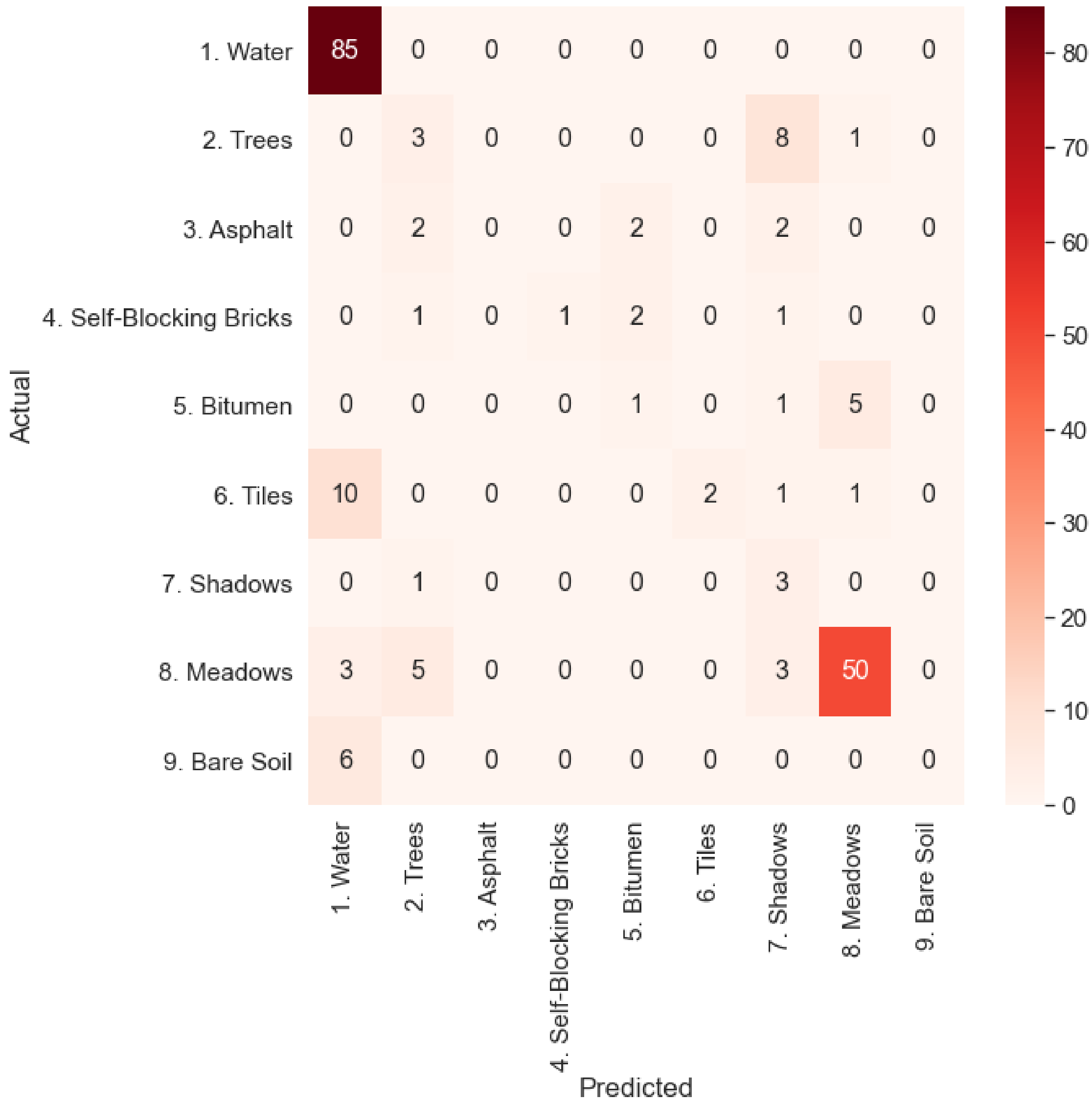
```
<Figure size 432x288 with 0 Axes>
```


1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
86 0 0 0 0 0 0 0 0
0 13 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 3 3 0 0 0 0 0 0
0 2 1 0 3 0 0 1 0
0 2 2 0 0 6 0 0 0
0 5 0 0 0 1 4 0 0
0 0 2 0 0 0 0 61 0
0 0 0 0 0 0 0 2 1
80
70
60
50
40
30
20
10
0




```
Score for fold 5: loss of 1.9775854349136353; accuracy of 72.50000238418579%
7/7 [==============================] - 1s 38ms/step
[[85  0  0  0  0  0  0  0  0]
 [ 0  3  0  0  0  0  8  1  0]
 [ 0  2  0  0  2  0  2  0  0]
 [ 0  1  0  1  2  0  1  0  0]
 [ 0  0  0  0  1  0  1  5  0]
 [10  0  0  0  0  2  1  1  0]
 [ 0  1  0  0  0  0  3  0  0]
 [ 3  5  0  0  0  0  3 50  0]
 [ 6  0  0  0  0  0  0  0  0]]
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5095426440238953 - Accuracy: 82.99999833106995%
------------------------------------------------------------------------
> Fold 2 - Loss: 1.1509346961975098 - Accuracy: 73.00000190734863%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.3687174320220947 - Accuracy: 92.5000011920929%
------------------------------------------------------------------------
> Fold 4 - Loss: 3.4993598461151123 - Accuracy: 87.5%
------------------------------------------------------------------------
> Fold 5 - Loss: 1.9775854349136353 - Accuracy: 72.50000238418579%
```

```
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 81.70000076293945 (+- 7.903163219282698)
> Loss: 1.5012280106544496
------------------------------------------------------------------------
Predicted Overall        1. Water  2. Trees  3. Asphalt  \
Actual Overall
1. Water                      438         0           0
2. Trees                        3        31           1
3. Asphalt                      4         3           3
4. Self-Blocking Bricks         1         4           3
5. Bitumen                      3         2           2
6. Tiles                       31         5           2
7. Shadows                      4        11           1
8. Meadows                      7         5           2
9. Bare Soil                   12         0           0

Predicted Overall        4. Self-Blocking Bricks  5. Bitumen  6. Tiles  \
Actual Overall
1. Water                                       0           0         1
2. Trees                                       2           0         0
3. Asphalt                                     2           8         1
4. Self-Blocking Bricks                        4           4         0
5. Bitumen                                     0          18         0
6. Tiles                                       0           0        22
7. Shadows                                     0           0         3
8. Meadows                                     8           0         0
9. Bare Soil                                   0           0         0

Predicted Overall        7. Shadows  8. Meadows  9. Bare Soil
Actual Overall
1. Water                          0           1             0
2. Trees                          8           4             0
3. Asphalt                        3           2             0
4. Self-Blocking Bricks           1           1             0
5. Bitumen                        1          15             0
6. Tiles                          3           2             0
7. Shadows                       19           0             0
8. Meadows                        3         280             0
9. Bare Soil                      0           4             2

<Figure size 432x288 with 0 Axes>
```


Actual
Predicted
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
85 0 0 0 0 0 0 0 0
0 3 0 0 0 0 8 1 0
0 2 0 0 2 0 2 0 0
0 1 0 1 2 0 1 0 0
0 0 0 0 1 0 1 5 0
10 0 0 0 0 2 1 1 0
0 1 0 0 0 0 3 0 0
3 5 0 0 0 0 3 50 0
6 0 0 0 0 0 0 0 0
0
10
20
30
40
50
60
70
80


1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual Overall
Predicted Overall
438 0 0 0 0 1 0 1 0
3 31 1 2 0 0 8 4 0
4 3 3 2 8 1 3 2 0
1 4 3 4 4 0 1 1 0
3 2 2 0 18 0 1 15 0
31 5 2 0 0 22 3 2 0
4 11 1 0 0 3 19 0 0
7 5 2 8 0 0 3 280 0
12 0 0 0 0 0 0 4 2
400
350
300
250
200
150
100
50
0