# 15 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=15  # 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
```

```python
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
```

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

[ ]:
```python
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"]
    )
```

[ ]:
```python
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)
```

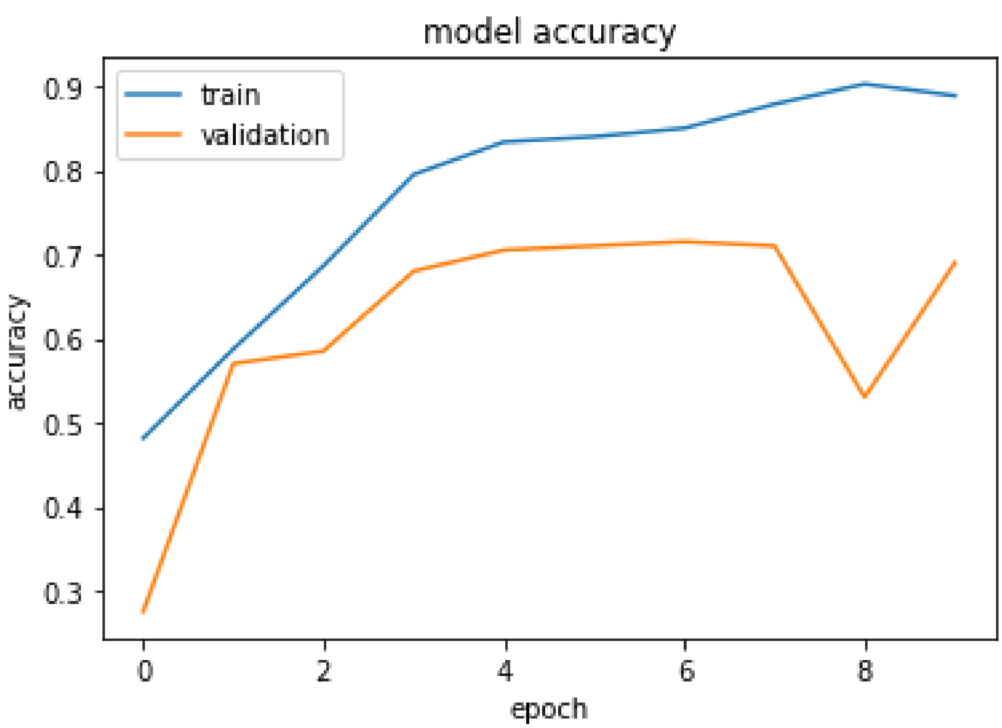
```
# Parse numbers as floats
#input_train = input_train.astype('float32')
#input_test = input_test.astype('float32')

# 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
```

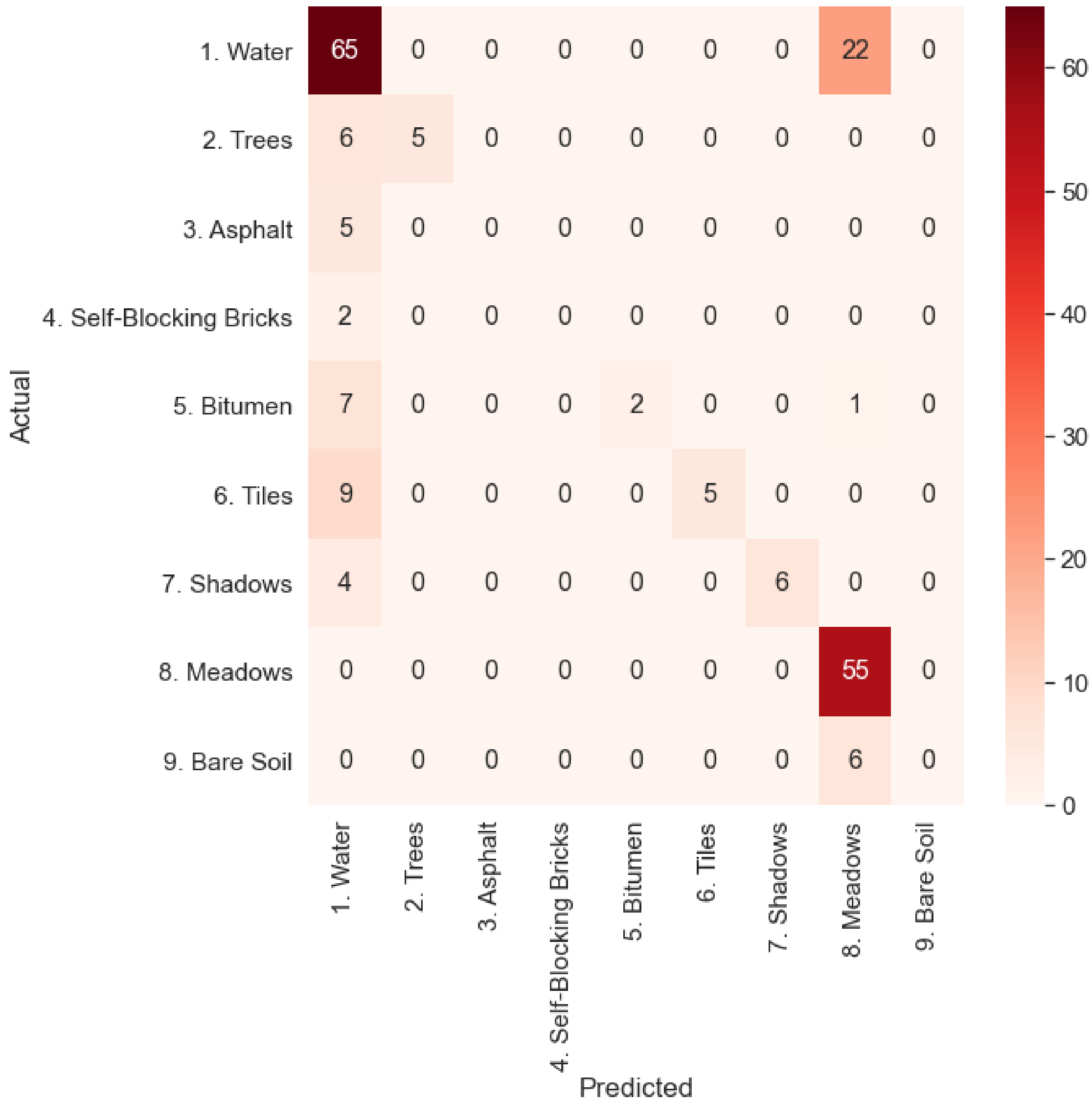
```
------------------------------------------------------------------------
Training for fold 1 …
Epoch 1/10
25/25 - 31s - loss: 1.9761 - accuracy: 0.4812 - val_loss: 1.6973 - val_accuracy:
0.2750 - 31s/epoch - 1s/step
Epoch 2/10
25/25 - 22s - loss: 1.3998 - accuracy: 0.5875 - val_loss: 1.5661 - val_accuracy:
```

```
0.5700 - 22s/epoch - 882ms/step
Epoch 3/10
25/25 - 23s - loss: 1.1627 - accuracy: 0.6862 - val_loss: 1.4803 - val_accuracy:
0.5850 - 23s/epoch - 924ms/step
Epoch 4/10
25/25 - 23s - loss: 0.6702 - accuracy: 0.7950 - val_loss: 1.2447 - val_accuracy:
0.6800 - 23s/epoch - 930ms/step
Epoch 5/10
25/25 - 22s - loss: 0.5200 - accuracy: 0.8338 - val_loss: 1.0614 - val_accuracy:
0.7050 - 22s/epoch - 893ms/step
Epoch 6/10
25/25 - 22s - loss: 0.5597 - accuracy: 0.8400 - val_loss: 1.0162 - val_accuracy:
0.7100 - 22s/epoch - 882ms/step
Epoch 7/10
25/25 - 23s - loss: 0.5119 - accuracy: 0.8500 - val_loss: 0.9808 - val_accuracy:
0.7150 - 23s/epoch - 909ms/step
Epoch 8/10
25/25 - 23s - loss: 0.4279 - accuracy: 0.8788 - val_loss: 0.8744 - val_accuracy:
0.7100 - 23s/epoch - 901ms/step
Epoch 9/10
25/25 - 22s - loss: 0.3261 - accuracy: 0.9025 - val_loss: 28.5196 -
val_accuracy: 0.5300 - 22s/epoch - 869ms/step
Epoch 10/10
25/25 - 22s - loss: 0.3451 - accuracy: 0.8888 - val_loss: 2.5156 - val_accuracy:
0.6900 - 22s/epoch - 897ms/step
```

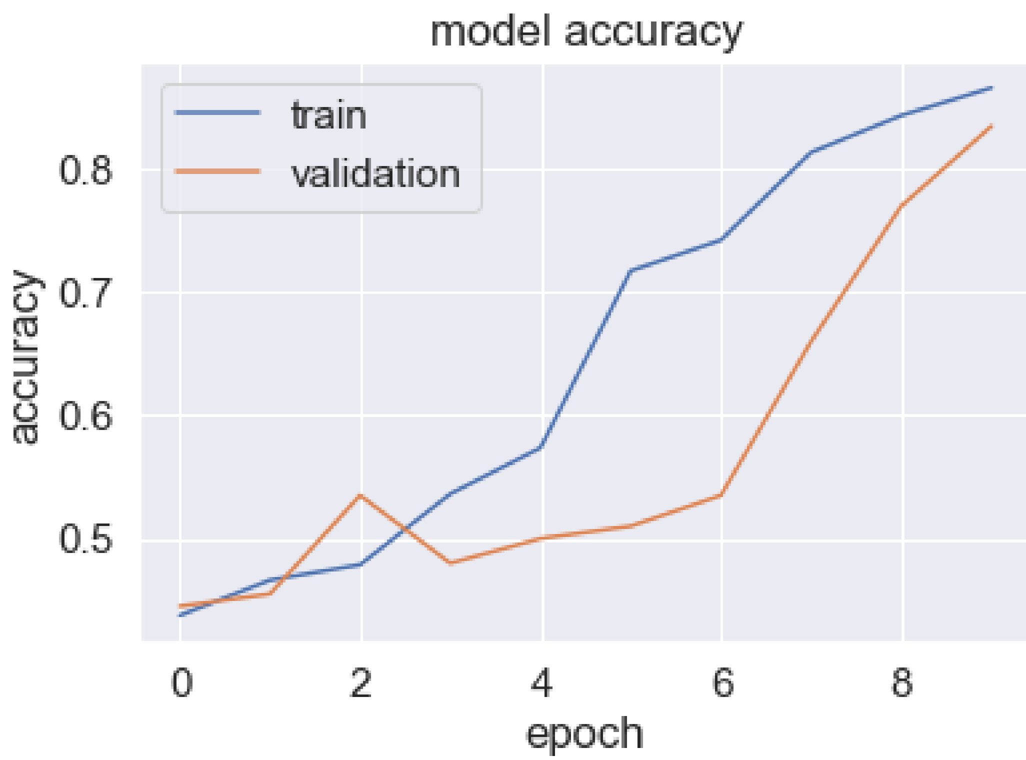


```
Score for fold 1: loss of 2.5155649185180664; accuracy of 68.99999976158142%
7/7 [==============================] - 1s 35ms/step
[[65  0  0  0  0  0  0 22  0]
 [ 6  5  0  0  0  0  0  0  0]
 [ 5  0  0  0  0  0  0  0  0]
 [ 2  0  0  0  0  0  0  0  0]
 [ 7  0  0  0  2  0  0  1  0]
 [ 9  0  0  0  0  5  0  0  0]
 [ 4  0  0  0  0  0  6  0  0]
 [ 0  0  0  0  0  0  0 55  0]
 [ 0  0  0  0  0  0  0  6  0]]
------------------------------------------------------------------------
Training for fold 2 …
Epoch 1/10
25/25 - 28s - loss: 1.6926 - accuracy: 0.4375 - val_loss: 1.5356 - val_accuracy:
0.4450 - 28s/epoch - 1s/step
Epoch 2/10
25/25 - 21s - loss: 1.5549 - accuracy: 0.4663 - val_loss: 1.4700 - val_accuracy:
0.4550 - 21s/epoch - 826ms/step
Epoch 3/10
25/25 - 21s - loss: 1.4440 - accuracy: 0.4787 - val_loss: 1.4366 - val_accuracy:
0.5350 - 21s/epoch - 831ms/step
Epoch 4/10
25/25 - 21s - loss: 1.5554 - accuracy: 0.5362 - val_loss: 1.6234 - val_accuracy:
0.4800 - 21s/epoch - 826ms/step
Epoch 5/10
25/25 - 21s - loss: 1.2882 - accuracy: 0.5738 - val_loss: 1.3957 - val_accuracy:
0.5000 - 21s/epoch - 825ms/step
Epoch 6/10
25/25 - 22s - loss: 0.9186 - accuracy: 0.7175 - val_loss: 1.3526 - val_accuracy:
0.5100 - 22s/epoch - 874ms/step
Epoch 7/10
25/25 - 24s - loss: 0.8097 - accuracy: 0.7425 - val_loss: 4.4956 - val_accuracy:
0.5350 - 24s/epoch - 960ms/step
Epoch 8/10
25/25 - 23s - loss: 0.5370 - accuracy: 0.8138 - val_loss: 1.6603 - val_accuracy:
0.6600 - 23s/epoch - 938ms/step
Epoch 9/10
25/25 - 22s - loss: 0.5152 - accuracy: 0.8438 - val_loss: 1.3340 - val_accuracy:
0.7700 - 22s/epoch - 884ms/step
Epoch 10/10
25/25 - 22s - loss: 0.4691 - accuracy: 0.8662 - val_loss: 1.1336 - val_accuracy:
0.8350 - 22s/epoch - 881ms/step
```

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



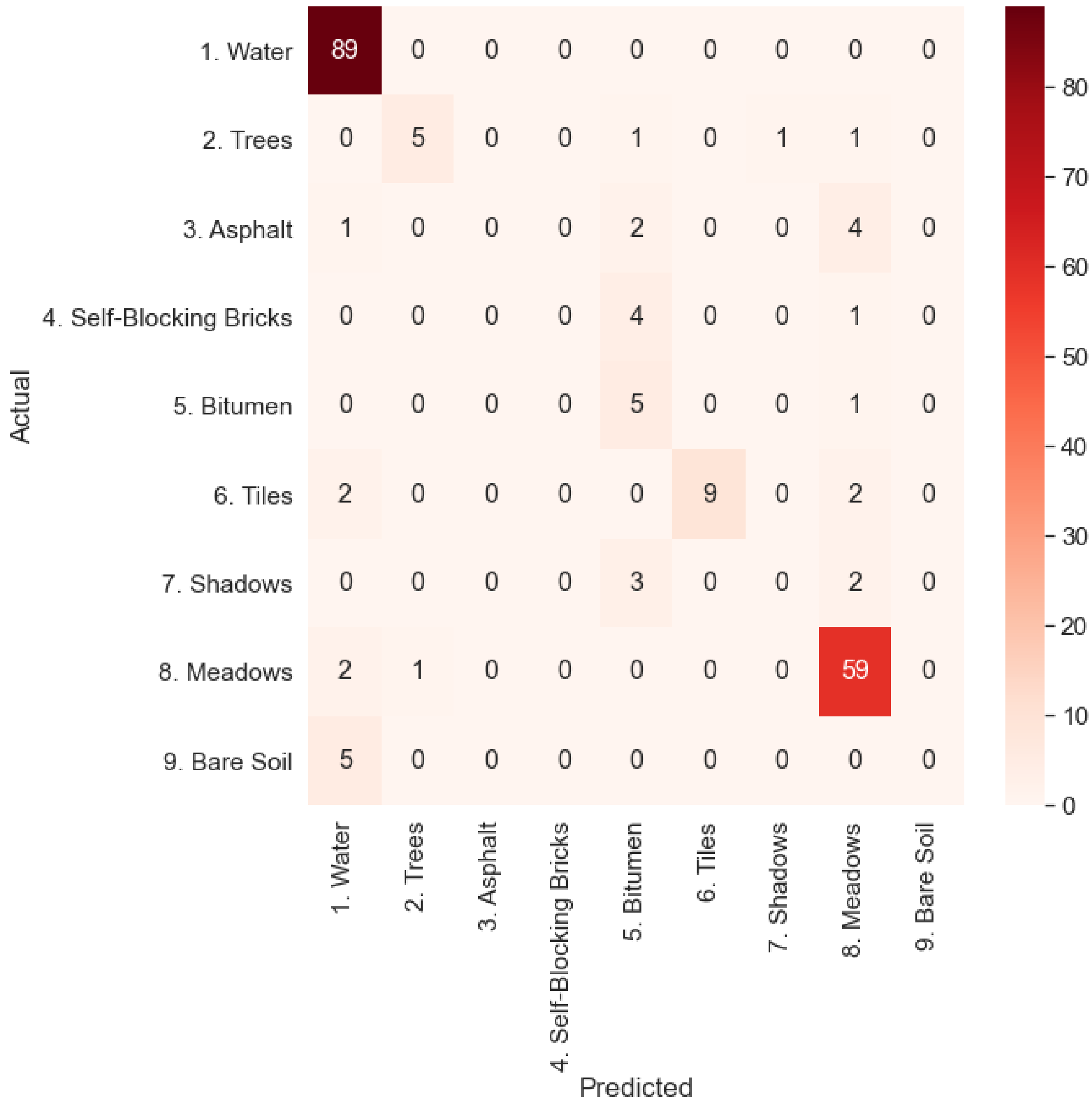
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
65 0 0 0 0 0 0 22 0
6 5 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0
7 0 0 0 2 0 0 1 0
9 0 0 0 0 5 0 0 0
4 0 0 0 0 0 6 0 0
0 0 0 0 0 0 0 55 0
0 0 0 0 0 0 0 6 0
60
50
40
30
20
10
0

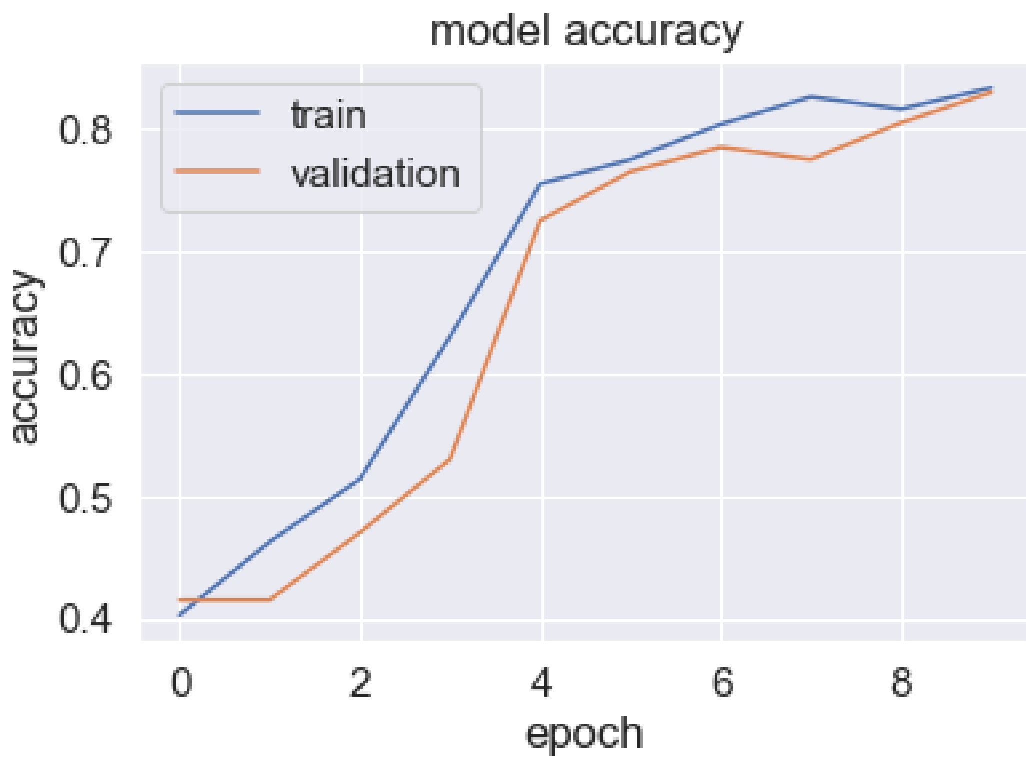



```
Score for fold 2: loss of 1.133560299873352; accuracy of 83.49999785423279%
7/7 [==============================] - 1s 39ms/step
[[89  0  0  0  0  0  0  0  0]
 [ 0  5  0  0  1  0  1  1  0]
 [ 1  0  0  0  2  0  0  4  0]
 [ 0  0  0  0  4  0  0  1  0]
 [ 0  0  0  0  5  0  0  1  0]
 [ 2  0  0  0  0  9  0  2  0]
 [ 0  0  0  0  3  0  0  2  0]
 [ 2  1  0  0  0  0  0 59  0]
 [ 5  0  0  0  0  0  0  0  0]]
------------------------------------------------------------------------
Training for fold 3 …
Epoch 1/10
25/25 - 29s - loss: 1.6885 - accuracy: 0.4025 - val_loss: 1.5774 - val_accuracy:
0.4150 - 29s/epoch - 1s/step
Epoch 2/10
25/25 - 22s - loss: 1.4947 - accuracy: 0.4625 - val_loss: 1.5829 - val_accuracy:
0.4150 - 22s/epoch - 889ms/step
Epoch 3/10
25/25 - 22s - loss: 1.4474 - accuracy: 0.5138 - val_loss: 1.4748 - val_accuracy:
0.4700 - 22s/epoch - 896ms/step
Epoch 4/10
```

```
25/25 - 22s - loss: 1.2244 - accuracy: 0.6300 - val_loss: 1.4403 - val_accuracy:
0.5300 - 22s/epoch - 893ms/step
Epoch 5/10
25/25 - 22s - loss: 0.7864 - accuracy: 0.7550 - val_loss: 1.2246 - val_accuracy:
0.7250 - 22s/epoch - 893ms/step
Epoch 6/10
25/25 - 22s - loss: 0.7326 - accuracy: 0.7750 - val_loss: 1.0265 - val_accuracy:
0.7650 - 22s/epoch - 888ms/step
Epoch 7/10
25/25 - 23s - loss: 0.5833 - accuracy: 0.8037 - val_loss: 0.7828 - val_accuracy:
0.7850 - 23s/epoch - 919ms/step
Epoch 8/10
25/25 - 23s - loss: 0.5387 - accuracy: 0.8263 - val_loss: 0.6490 - val_accuracy:
0.7750 - 23s/epoch - 903ms/step
Epoch 9/10
25/25 - 22s - loss: 0.5836 - accuracy: 0.8163 - val_loss: 0.4879 - val_accuracy:
0.8050 - 22s/epoch - 893ms/step
Epoch 10/10
25/25 - 22s - loss: 0.5363 - accuracy: 0.8338 - val_loss: 0.4259 - val_accuracy:
0.8300 - 22s/epoch - 877ms/step
```

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

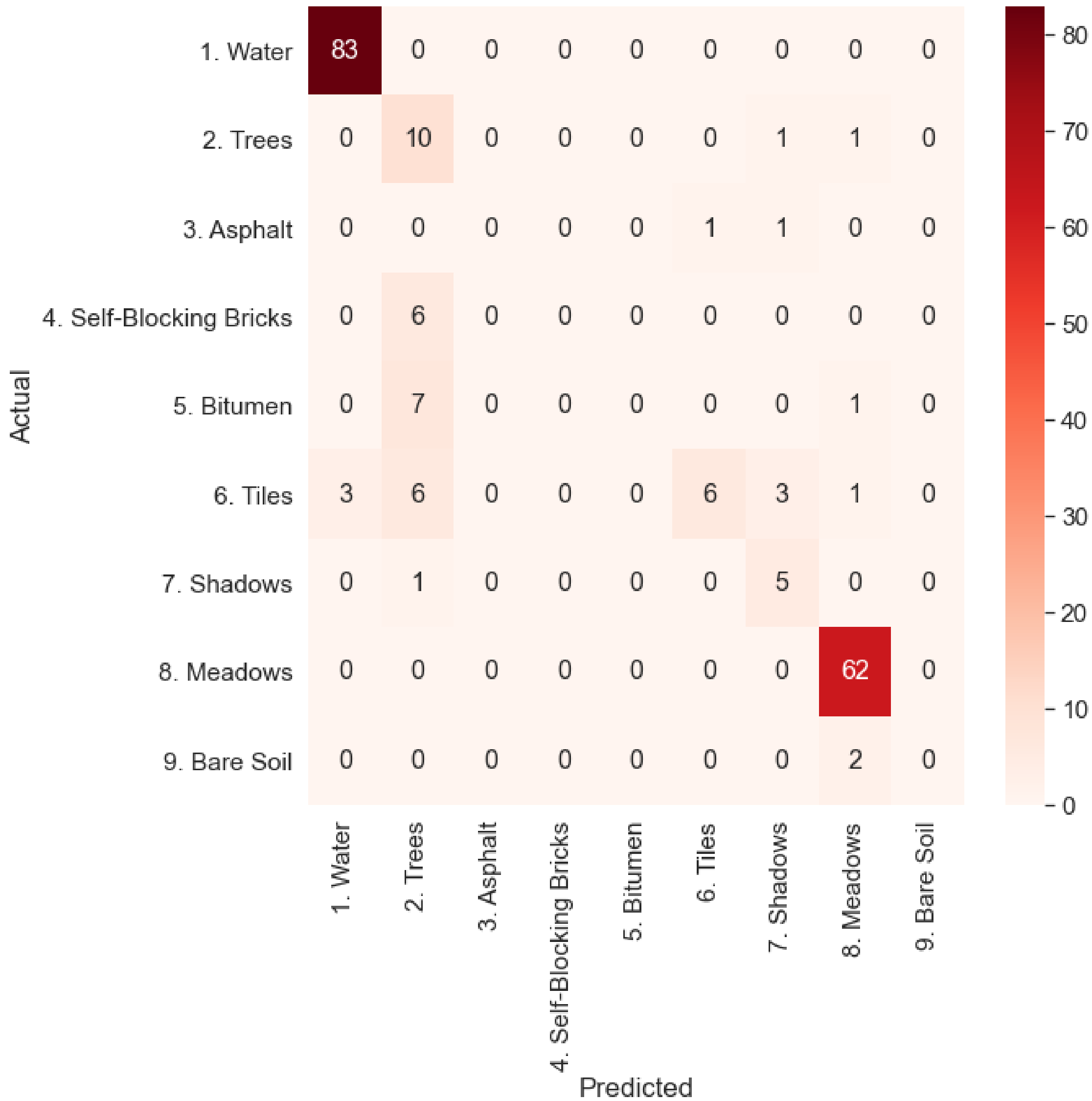

1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
89 0 0 0 0 0 0 0 0
0 5 0 0 1 0 1 1 0
1 0 0 0 2 0 0 4 0
0 0 0 0 4 0 0 1 0
0 0 0 0 5 0 0 1 0
2 0 0 0 0 9 0 2 0
0 0 0 0 3 0 0 2 0
2 1 0 0 0 0 0 59 0
5 0 0 0 0 0 0 0 0
80
70
60
50
40
30
20
10
0



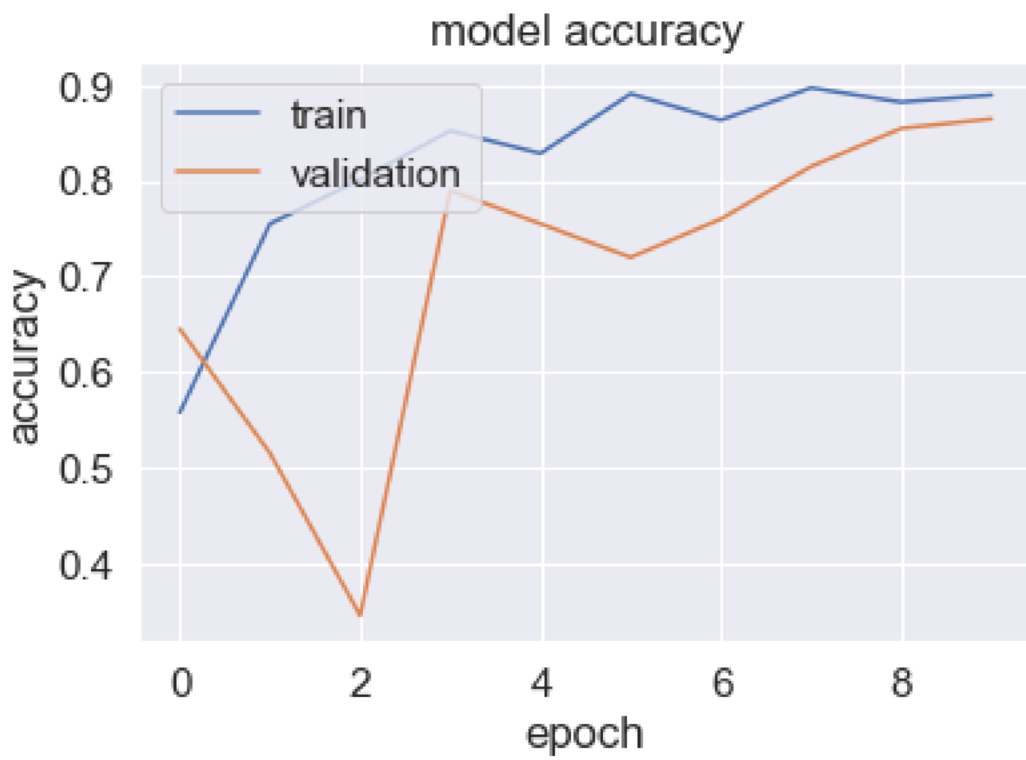

```
Score for fold 3: loss of 0.42586690187454224; accuracy of 82.99999833106995%
7/7 [==============================] - 1s 28ms/step
[[83  0  0  0  0  0  0  0  0]
 [ 0 10  0  0  0  0  1  1  0]
 [ 0  0  0  0  0  1  1  0  0]
 [ 0  6  0  0  0  0  0  0  0]
 [ 0  7  0  0  0  0  0  1  0]
 [ 3  6  0  0  0  6  3  1  0]
 [ 0  1  0  0  0  0  5  0  0]
 [ 0  0  0  0  0  0  0 62  0]
 [ 0  0  0  0  0  0  0  2  0]]
------------------------------------------------------------------------
Training for fold 4 …
Epoch 1/10
25/25 - 25s - loss: 1.5079 - accuracy: 0.5575 - val_loss: 1.5057 - val_accuracy:
0.6450 - 25s/epoch - 997ms/step
Epoch 2/10
25/25 - 20s - loss: 0.7890 - accuracy: 0.7550 - val_loss: 1.3790 - val_accuracy:
0.5150 - 20s/epoch - 792ms/step
Epoch 3/10
25/25 - 19s - loss: 0.6278 - accuracy: 0.8012 - val_loss: 4.8824 - val_accuracy:
0.3450 - 19s/epoch - 749ms/step
Epoch 4/10
```

```
25/25 - 19s - loss: 0.4900 - accuracy: 0.8525 - val_loss: 0.8901 - val_accuracy:
0.7900 - 19s/epoch - 772ms/step
Epoch 5/10
25/25 - 19s - loss: 0.7452 - accuracy: 0.8288 - val_loss: 0.8425 - val_accuracy:
0.7550 - 19s/epoch - 762ms/step
Epoch 6/10
25/25 - 20s - loss: 0.4130 - accuracy: 0.8913 - val_loss: 1.1858 - val_accuracy:
0.7200 - 20s/epoch - 793ms/step
Epoch 7/10
25/25 - 20s - loss: 0.4266 - accuracy: 0.8637 - val_loss: 0.7714 - val_accuracy:
0.7600 - 20s/epoch - 781ms/step
Epoch 8/10
25/25 - 19s - loss: 0.3403 - accuracy: 0.8975 - val_loss: 0.7194 - val_accuracy:
0.8150 - 19s/epoch - 764ms/step
Epoch 9/10
25/25 - 19s - loss: 0.4731 - accuracy: 0.8825 - val_loss: 0.5783 - val_accuracy:
0.8550 - 19s/epoch - 772ms/step
Epoch 10/10
25/25 - 20s - loss: 0.3778 - accuracy: 0.8900 - val_loss: 0.5993 - val_accuracy:
0.8650 - 20s/epoch - 781ms/step
```

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



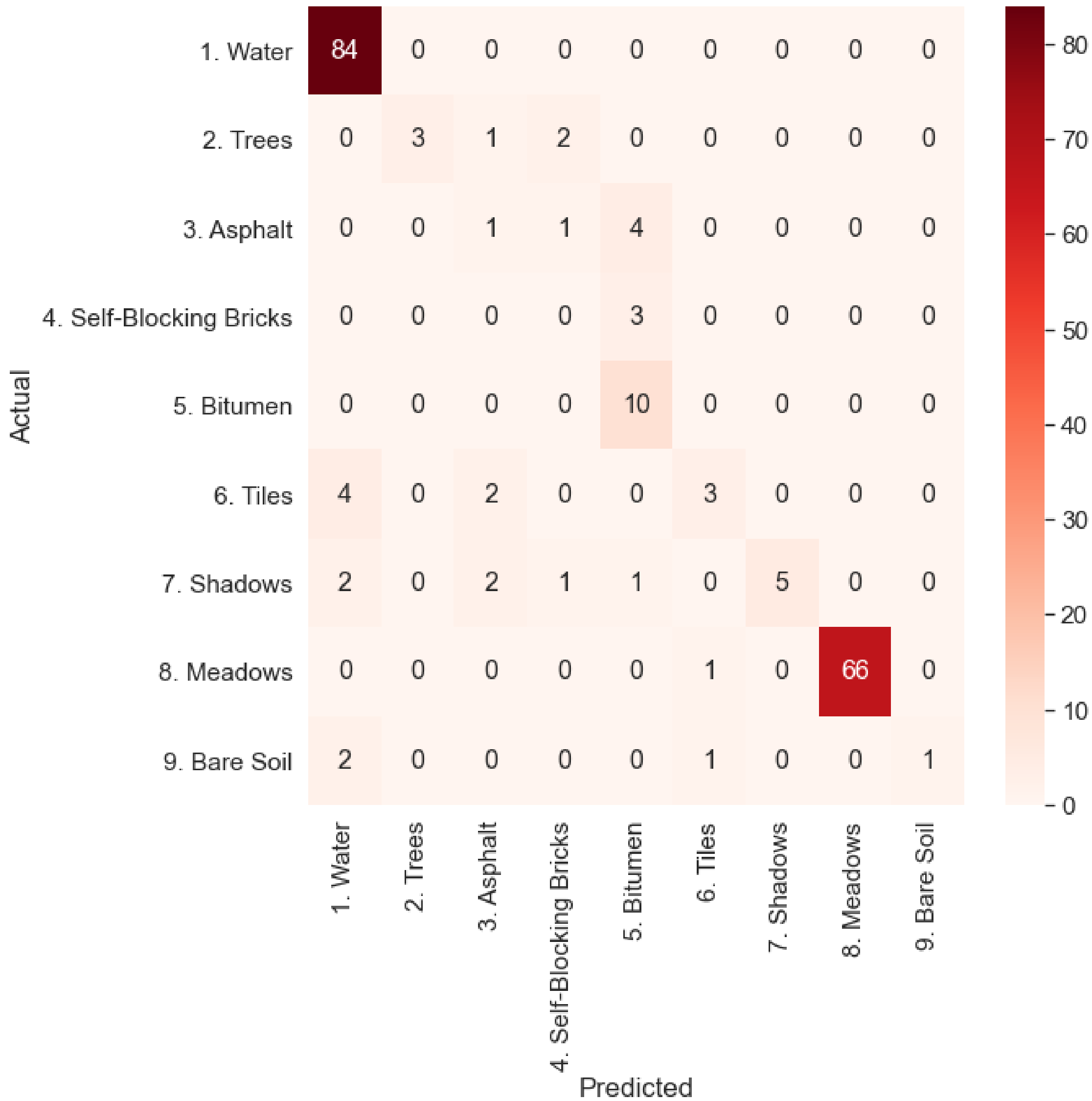
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
83 0 0 0 0 0 0 0 0
0 10 0 0 0 0 1 1 0
0 0 0 0 0 1 1 0 0
0 6 0 0 0 0 0 0 0
0 7 0 0 0 0 0 1 0
3 6 0 0 0 6 3 1 0
0 1 0 0 0 0 5 0 0
0 0 0 0 0 0 0 62 0
0 0 0 0 0 0 0 2 0
0
10
20
30
40
50
60
70
80

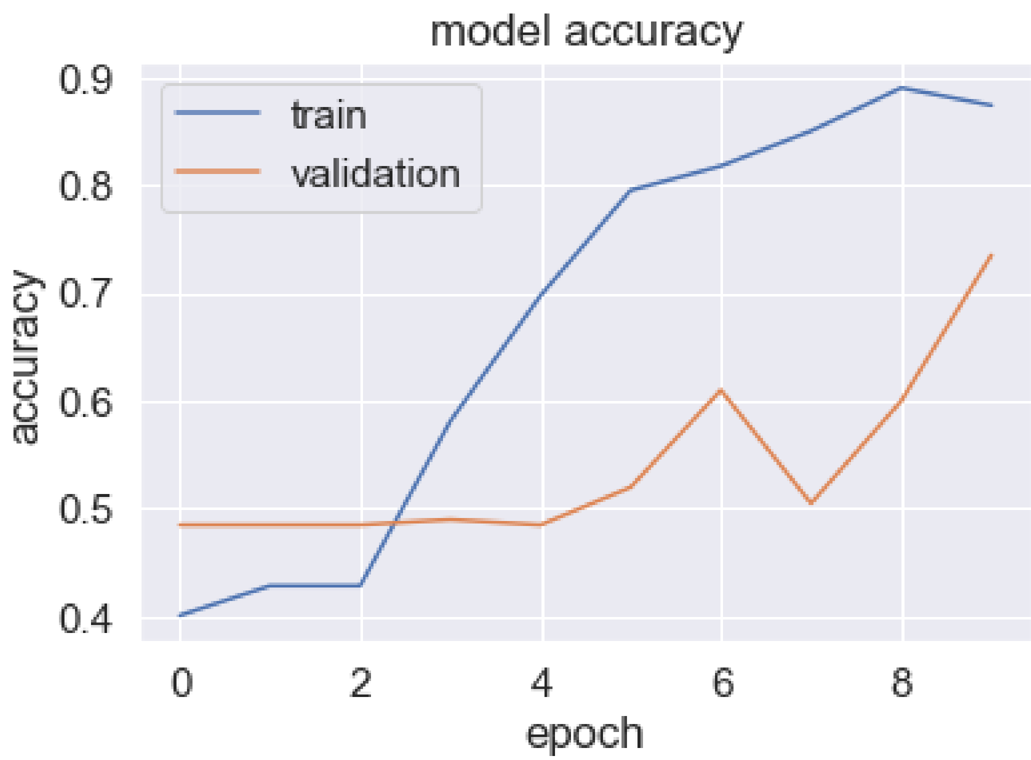



```
Score for fold 4: loss of 0.5992723703384399; accuracy of 86.50000095367432%
7/7 [==============================] - 1s 26ms/step
[[84  0  0  0  0  0  0  0  0]
 [ 0  3  1  2  0  0  0  0  0]
 [ 0  0  1  1  4  0  0  0  0]
 [ 0  0  0  0  3  0  0  0  0]
 [ 0  0  0  0 10  0  0  0  0]
 [ 4  0  2  0  0  3  0  0  0]
 [ 2  0  2  1  1  0  5  0  0]
 [ 0  0  0  0  0  1  0 66  0]
 [ 2  0  0  0  0  1  0  0  1]]
------------------------------------------------------------------------
Training for fold 5 …
Epoch 1/10
25/25 - 27s - loss: 1.7063 - accuracy: 0.4013 - val_loss: 1.4647 - val_accuracy:
0.4850 - 27s/epoch - 1s/step
Epoch 2/10
25/25 - 20s - loss: 1.5760 - accuracy: 0.4288 - val_loss: 1.4806 - val_accuracy:
0.4850 - 20s/epoch - 780ms/step
Epoch 3/10
25/25 - 19s - loss: 1.5772 - accuracy: 0.4288 - val_loss: 1.4592 - val_accuracy:
0.4850 - 19s/epoch - 763ms/step
Epoch 4/10
```

```
25/25 - 19s - loss: 1.3029 - accuracy: 0.5813 - val_loss: 1.5699 - val_accuracy:
0.4900 - 19s/epoch - 752ms/step
Epoch 5/10
25/25 - 19s - loss: 1.0493 - accuracy: 0.6975 - val_loss: 1.4742 - val_accuracy:
0.4850 - 19s/epoch - 749ms/step
Epoch 6/10
25/25 - 19s - loss: 0.6845 - accuracy: 0.7950 - val_loss: 1.6503 - val_accuracy:
0.5200 - 19s/epoch - 751ms/step
Epoch 7/10
25/25 - 19s - loss: 0.5666 - accuracy: 0.8175 - val_loss: 1.2286 - val_accuracy:
0.6100 - 19s/epoch - 754ms/step
Epoch 8/10
25/25 - 19s - loss: 0.5084 - accuracy: 0.8500 - val_loss: 1.2463 - val_accuracy:
0.5050 - 19s/epoch - 754ms/step
Epoch 9/10
25/25 - 19s - loss: 0.3833 - accuracy: 0.8900 - val_loss: 0.9953 - val_accuracy:
0.6000 - 19s/epoch - 750ms/step
Epoch 10/10
25/25 - 19s - loss: 0.4476 - accuracy: 0.8737 - val_loss: 3.2806 - val_accuracy:
0.7350 - 19s/epoch - 749ms/step
```

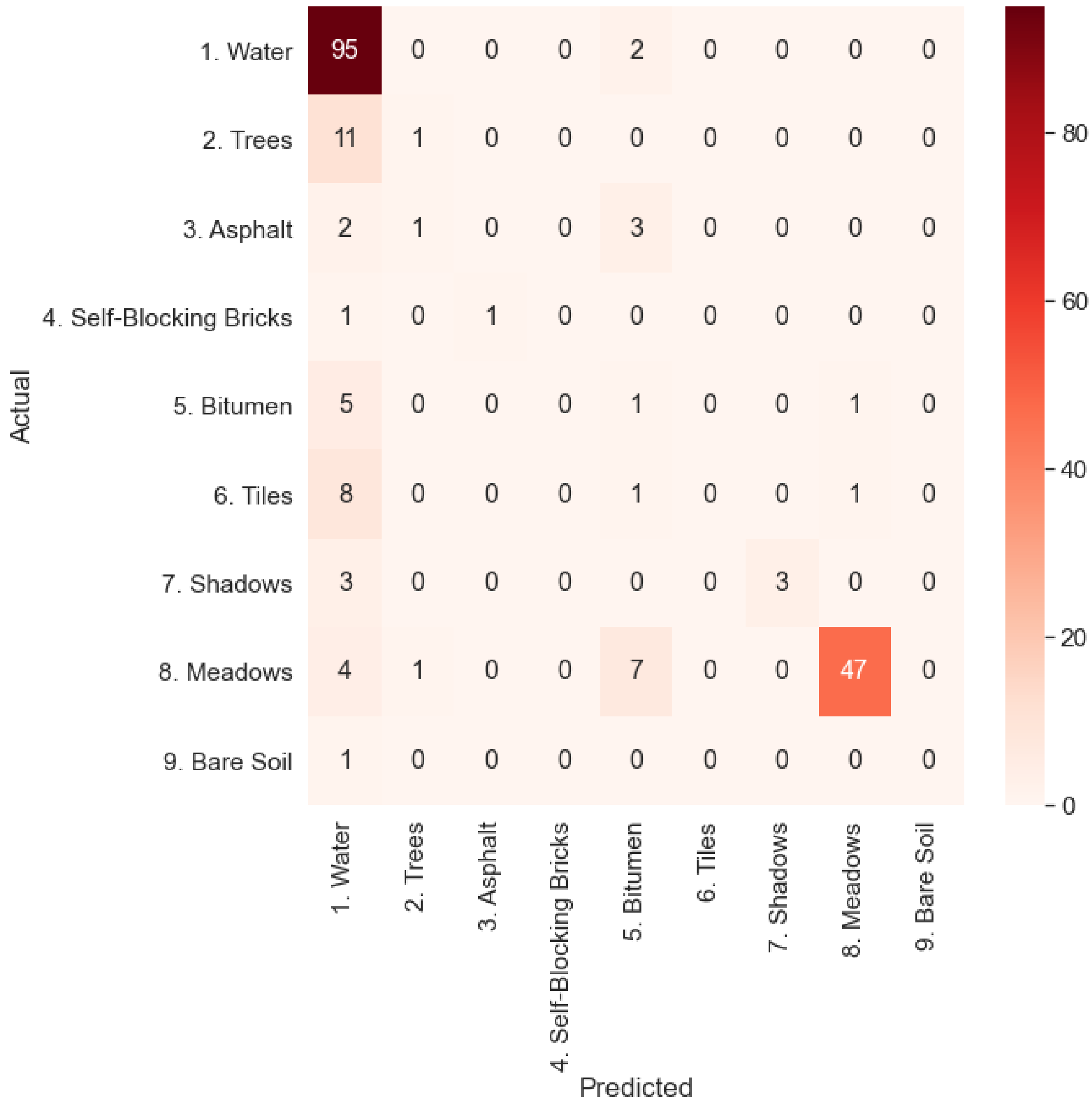
```
<Figure size 432x288 with 0 Axes>
```


1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
84 0 0 0 0 0 0 0 0
0 3 1 2 0 0 0 0 0
0 0 1 1 4 0 0 0 0
0 0 0 0 3 0 0 0 0
0 0 0 0 10 0 0 0 0
4 0 2 0 0 3 0 0 0
2 0 2 1 1 0 5 0 0
0 0 0 0 0 1 0 66 0
2 0 0 0 0 1 0 0 1
80
70
60
50
40
30
20
10
0




```
Score for fold 5: loss of 3.2805755138397217; accuracy of 73.50000143051147%
7/7 [==============================] - 1s 28ms/step
[[95  0  0  0  2  0  0  0  0]
 [11  1  0  0  0  0  0  0  0]
 [ 2  1  0  0  3  0  0  0  0]
 [ 1  0  1  0  0  0  0  0  0]
 [ 5  0  0  0  1  0  0  1  0]
 [ 8  0  0  0  1  0  0  1  0]
 [ 3  0  0  0  0  0  3  0  0]
 [ 4  1  0  0  7  0  0 47  0]
 [ 1  0  0  0  0  0  0  0  0]]
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 2.5155649185180664 - Accuracy: 68.99999976158142%
------------------------------------------------------------------------
> Fold 2 - Loss: 1.133560299873352 - Accuracy: 83.49999785423279%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.42586690187454224 - Accuracy: 82.99999833106995%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5992723703384399 - Accuracy: 86.50000095367432%
------------------------------------------------------------------------
> Fold 5 - Loss: 3.2805755138397217 - Accuracy: 73.50000143051147%
```

```
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 79.09999966621399 (+- 6.673829050976817)
> Loss: 1.5909680008888245
------------------------------------------------------------------------
Predicted Overall        1. Water  2. Trees  3. Asphalt  \
Actual Overall
1. Water                      416         0           0
2. Trees                       17        24           1
3. Asphalt                      8         1           1
4. Self-Blocking Bricks         3         6           1
5. Bitumen                     12         7           0
6. Tiles                       26         6           2
7. Shadows                      9         1           2
8. Meadows                      6         2           0
9. Bare Soil                    8         0           0

Predicted Overall        4. Self-Blocking Bricks  5. Bitumen  6. Tiles  \
Actual Overall
1. Water                                       0           2         0
2. Trees                                       2           1         0
3. Asphalt                                     1           9         1
4. Self-Blocking Bricks                        0           7         0
5. Bitumen                                     0          18         0
6. Tiles                                       0           1        23
7. Shadows                                     1           4         0
8. Meadows                                     0           7         1
9. Bare Soil                                   0           0         1

Predicted Overall        7. Shadows  8. Meadows  9. Bare Soil
Actual Overall
1. Water                          0          22             0
2. Trees                          2           2             0
3. Asphalt                        1           4             0
4. Self-Blocking Bricks           0           1             0
5. Bitumen                        0           4             0
6. Tiles                          3           4             0
7. Shadows                       19           2             0
8. Meadows                        0         289             0
9. Bare Soil                      0           8             1

<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
95 0 0 0 2 0 0 0 0
11 1 0 0 0 0 0 0 0
2 1 0 0 3 0 0 0 0
1 0 1 0 0 0 0 0 0
5 0 0 0 1 0 0 1 0
8 0 0 0 1 0 0 1 0
3 0 0 0 0 0 3 0 0
4 1 0 0 7 0 0 47 0
1 0 0 0 0 0 0 0 0
80
60
40
20
0


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
416 0 0 0 2 0 0 22 0
17 24 1 2 1 0 2 2 0
8 1 1 1 9 1 1 4 0
3 6 1 0 7 0 0 1 0
12 7 0 0 18 0 0 4 0
26 6 2 0 1 23 3 4 0
9 1 2 1 4 0 19 2 0
6 2 0 0 7 1 0 289 0
8 0 0 0 0 1 0 8 1
400
350
300
250
200
150
100
50
0