9 X_ResNet_Center

April 3, 2023

1 Date: 9 2022

2 Method: Cross_Inception

3 Data: Pavia

4 Results v.05

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

[ ]:
```python
# 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)
```

[ ]:
```python
# 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
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'])
```

[ ]:
```
print()

import warnings
warnings.filterwarnings("ignore")
```

```python
# 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

model = EfficientNetB0(weights='imagenet')


def build_model(num_classes):
    inputs = layers.Input(shape=(windowSize, windowSize, 3))
    #x = img_augmentation(inputs)
    model =  resnet.ResNet50(include_top=False,  input_tensor=inputs,␣
↪weights="imagenet")
    #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)

    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
```

[ ]:
```
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
```

```python
batch_size = 50
loss_function = sparse_categorical_crossentropy
no_classes = 9
no_epochs = 20
optimizer = Adam()
verbosity = 1
num_folds = 5

NN=len(Xtrain)
NN=1000

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
```

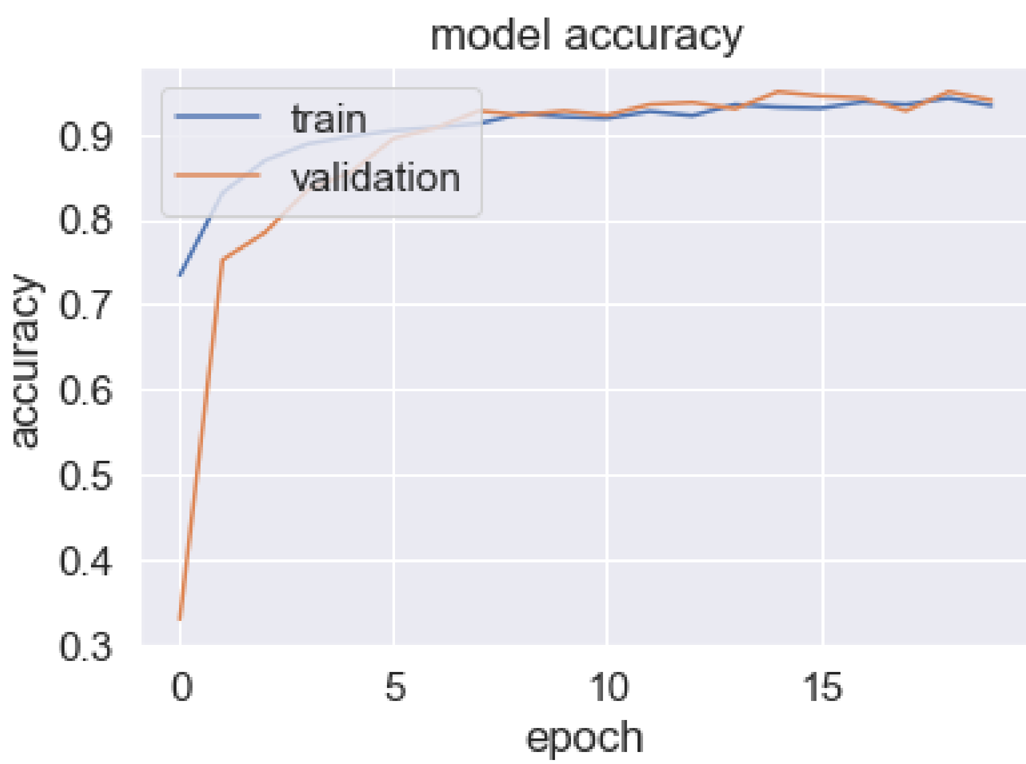
```
 # Compile the model
 #model.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
↪metrics=['accuracy'])

  # 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)
```

```python
    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)
```

```python
print(Overall_Conf)
```

```
------------------------------------------------------------------------
Training for fold 1 …
Epoch 1/20
50/50 - 5s - loss: 0.9955 - accuracy: 0.7344 - val_loss: 1.2988 - val_accuracy:
0.3300 - 5s/epoch - 101ms/step
Epoch 2/20
50/50 - 2s - loss: 0.5629 - accuracy: 0.8313 - val_loss: 1.0561 - val_accuracy:
0.7525 - 2s/epoch - 49ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4356 - accuracy: 0.8694 - val_loss: 0.7973 - val_accuracy:
0.7850 - 2s/epoch - 46ms/step
Epoch 4/20
50/50 - 2s - loss: 0.3798 - accuracy: 0.8888 - val_loss: 0.6189 - val_accuracy:
0.8350 - 2s/epoch - 43ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3352 - accuracy: 0.8975 - val_loss: 0.4650 - val_accuracy:
0.8550 - 2s/epoch - 47ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3166 - accuracy: 0.9044 - val_loss: 0.4091 - val_accuracy:
0.8950 - 2s/epoch - 46ms/step
Epoch 7/20
50/50 - 2s - loss: 0.2812 - accuracy: 0.9081 - val_loss: 0.3343 - val_accuracy:
0.9075 - 2s/epoch - 44ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2815 - accuracy: 0.9125 - val_loss: 0.2796 - val_accuracy:
0.9275 - 2s/epoch - 44ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2440 - accuracy: 0.9244 - val_loss: 0.2522 - val_accuracy:
0.9225 - 2s/epoch - 48ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2504 - accuracy: 0.9206 - val_loss: 0.2368 - val_accuracy:
0.9275 - 2s/epoch - 45ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2517 - accuracy: 0.9187 - val_loss: 0.2140 - val_accuracy:
0.9225 - 2s/epoch - 47ms/step
Epoch 12/20
50/50 - 3s - loss: 0.2270 - accuracy: 0.9269 - val_loss: 0.2010 - val_accuracy:
0.9350 - 3s/epoch - 52ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2349 - accuracy: 0.9219 - val_loss: 0.2022 - val_accuracy:
0.9375 - 2s/epoch - 43ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2117 - accuracy: 0.9350 - val_loss: 0.1897 - val_accuracy:
0.9300 - 2s/epoch - 42ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2100 - accuracy: 0.9319 - val_loss: 0.1762 - val_accuracy:
```

```
0.9500 - 2s/epoch - 45ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2053 - accuracy: 0.9306 - val_loss: 0.1759 - val_accuracy:
0.9450 - 2s/epoch - 49ms/step
Epoch 17/20
50/50 - 3s - loss: 0.1939 - accuracy: 0.9381 - val_loss: 0.1711 - val_accuracy:
0.9425 - 3s/epoch - 51ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2058 - accuracy: 0.9350 - val_loss: 0.1637 - val_accuracy:
0.9275 - 2s/epoch - 44ms/step
Epoch 19/20
50/50 - 3s - loss: 0.1905 - accuracy: 0.9425 - val_loss: 0.1569 - val_accuracy:
0.9500 - 3s/epoch - 51ms/step
Epoch 20/20
50/50 - 3s - loss: 0.2008 - accuracy: 0.9337 - val_loss: 0.1570 - val_accuracy:
0.9400 - 3s/epoch - 51ms/step
```

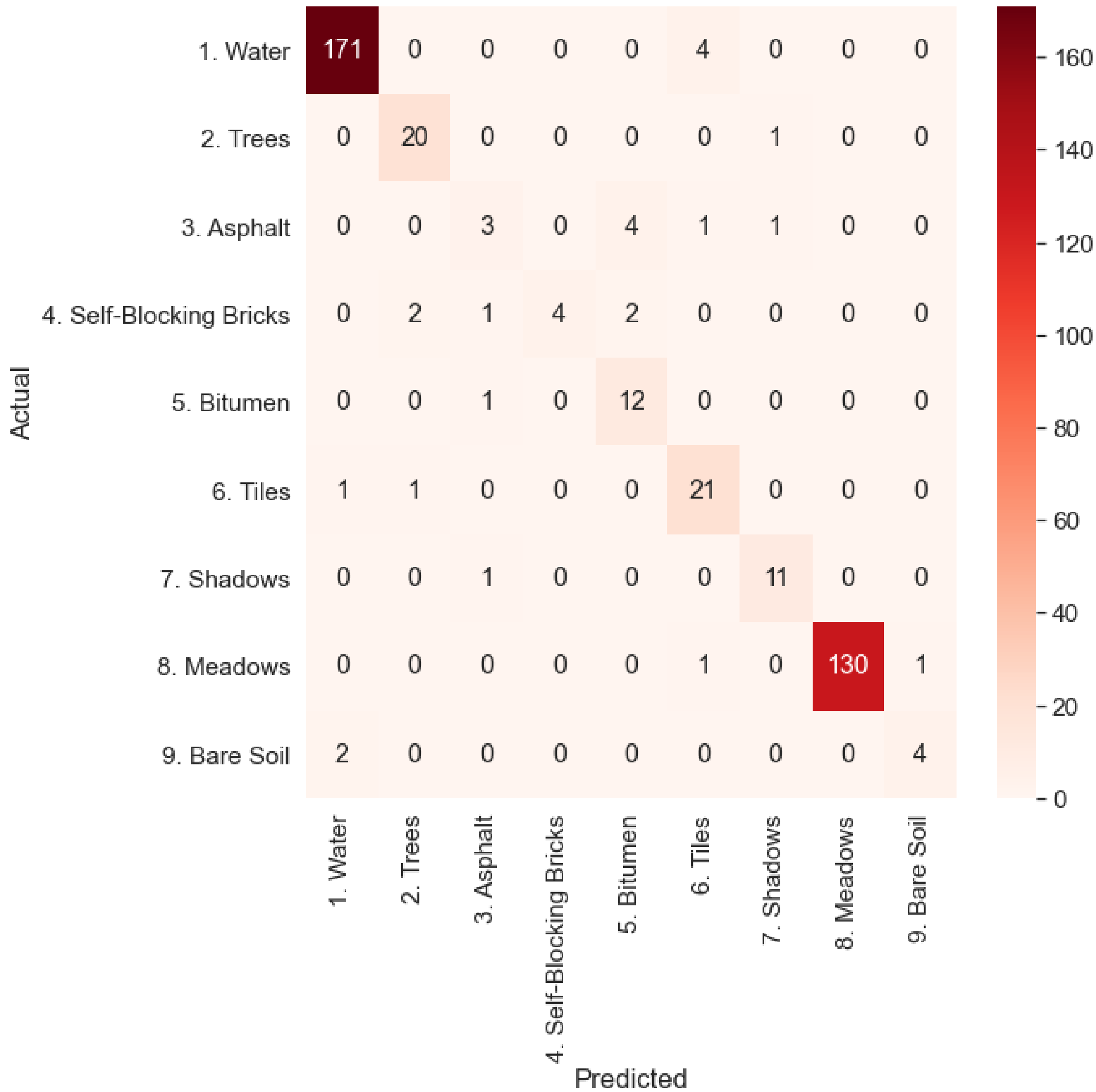


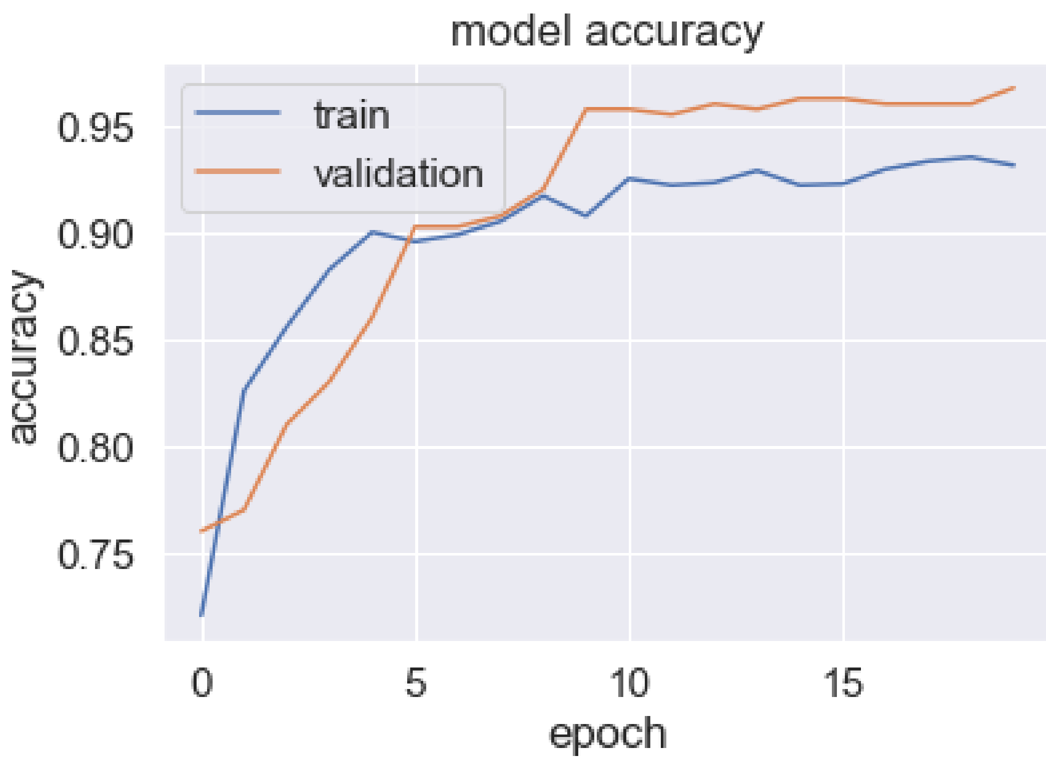

```
Score for fold 1: loss of 0.15701694786548615; accuracy of 93.99999976158142%
13/13 [==============================] - 1s 33ms/step
[[171   0   0   0   0   4   0   0   0]
 [  0  20   0   0   0   0   1   0   0]
 [  0   0   3   0   4   1   1   0   0]
 [  0   2   1   4   2   0   0   0   0]
```

```
 [  0   0   1   0  12   0   0   0   0]
 [  1   1   0   0   0  21   0   0   0]
 [  0   0   1   0   0   0  11   0   0]
 [  0   0   0   0   0   1   0 130   1]
 [  2   0   0   0   0   0   0   0   4]]
------------------------------------------------------------------------
Training for fold 2 …
Epoch 1/20
50/50 - 5s - loss: 1.0113 - accuracy: 0.7206 - val_loss: 1.3136 - val_accuracy:
0.7600 - 5s/epoch - 100ms/step
Epoch 2/20
50/50 - 2s - loss: 0.5744 - accuracy: 0.8256 - val_loss: 1.0159 - val_accuracy:
0.7700 - 2s/epoch - 49ms/step
Epoch 3/20
50/50 - 3s - loss: 0.4525 - accuracy: 0.8556 - val_loss: 0.8067 - val_accuracy:
0.8100 - 3s/epoch - 50ms/step
Epoch 4/20
50/50 - 2s - loss: 0.3875 - accuracy: 0.8825 - val_loss: 0.6081 - val_accuracy:
0.8300 - 2s/epoch - 50ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3412 - accuracy: 0.9000 - val_loss: 0.4593 - val_accuracy:
0.8600 - 2s/epoch - 48ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3164 - accuracy: 0.8956 - val_loss: 0.3525 - val_accuracy:
0.9025 - 2s/epoch - 49ms/step
Epoch 7/20
50/50 - 3s - loss: 0.3033 - accuracy: 0.8988 - val_loss: 0.2996 - val_accuracy:
0.9025 - 3s/epoch - 50ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2866 - accuracy: 0.9050 - val_loss: 0.2602 - val_accuracy:
0.9075 - 2s/epoch - 49ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2692 - accuracy: 0.9169 - val_loss: 0.2206 - val_accuracy:
0.9200 - 2s/epoch - 47ms/step
Epoch 10/20
50/50 - 3s - loss: 0.2677 - accuracy: 0.9075 - val_loss: 0.1998 - val_accuracy:
0.9575 - 3s/epoch - 51ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2519 - accuracy: 0.9250 - val_loss: 0.1747 - val_accuracy:
0.9575 - 2s/epoch - 46ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2372 - accuracy: 0.9219 - val_loss: 0.1603 - val_accuracy:
0.9550 - 2s/epoch - 48ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2309 - accuracy: 0.9231 - val_loss: 0.1526 - val_accuracy:
0.9600 - 2s/epoch - 50ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2165 - accuracy: 0.9287 - val_loss: 0.1468 - val_accuracy:
```

```
0.9575 - 2s/epoch - 49ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2250 - accuracy: 0.9219 - val_loss: 0.1433 - val_accuracy:
0.9625 - 2s/epoch - 48ms/step
Epoch 16/20
50/50 - 3s - loss: 0.2461 - accuracy: 0.9225 - val_loss: 0.1359 - val_accuracy:
0.9625 - 3s/epoch - 51ms/step
Epoch 17/20
50/50 - 2s - loss: 0.2152 - accuracy: 0.9294 - val_loss: 0.1377 - val_accuracy:
0.9600 - 2s/epoch - 45ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2141 - accuracy: 0.9331 - val_loss: 0.1346 - val_accuracy:
0.9600 - 2s/epoch - 46ms/step
Epoch 19/20
50/50 - 2s - loss: 0.1990 - accuracy: 0.9350 - val_loss: 0.1433 - val_accuracy:
0.9600 - 2s/epoch - 48ms/step
Epoch 20/20
50/50 - 2s - loss: 0.2008 - accuracy: 0.9312 - val_loss: 0.1270 - val_accuracy:
0.9675 - 2s/epoch - 48ms/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
171 0 0 0 0 4 0 0 0
0 20 0 0 0 0 1 0 0
0 0 3 0 4 1 1 0 0
0 2 1 4 2 0 0 0 0
0 0 1 0 12 0 0 0 0
1 1 0 0 0 21 0 0 0
0 0 1 0 0 0 11 0 0
0 0 0 0 0 1 0 130 1
2 0 0 0 0 0 0 0 4
160
140
120
100
80
60
40
20
0

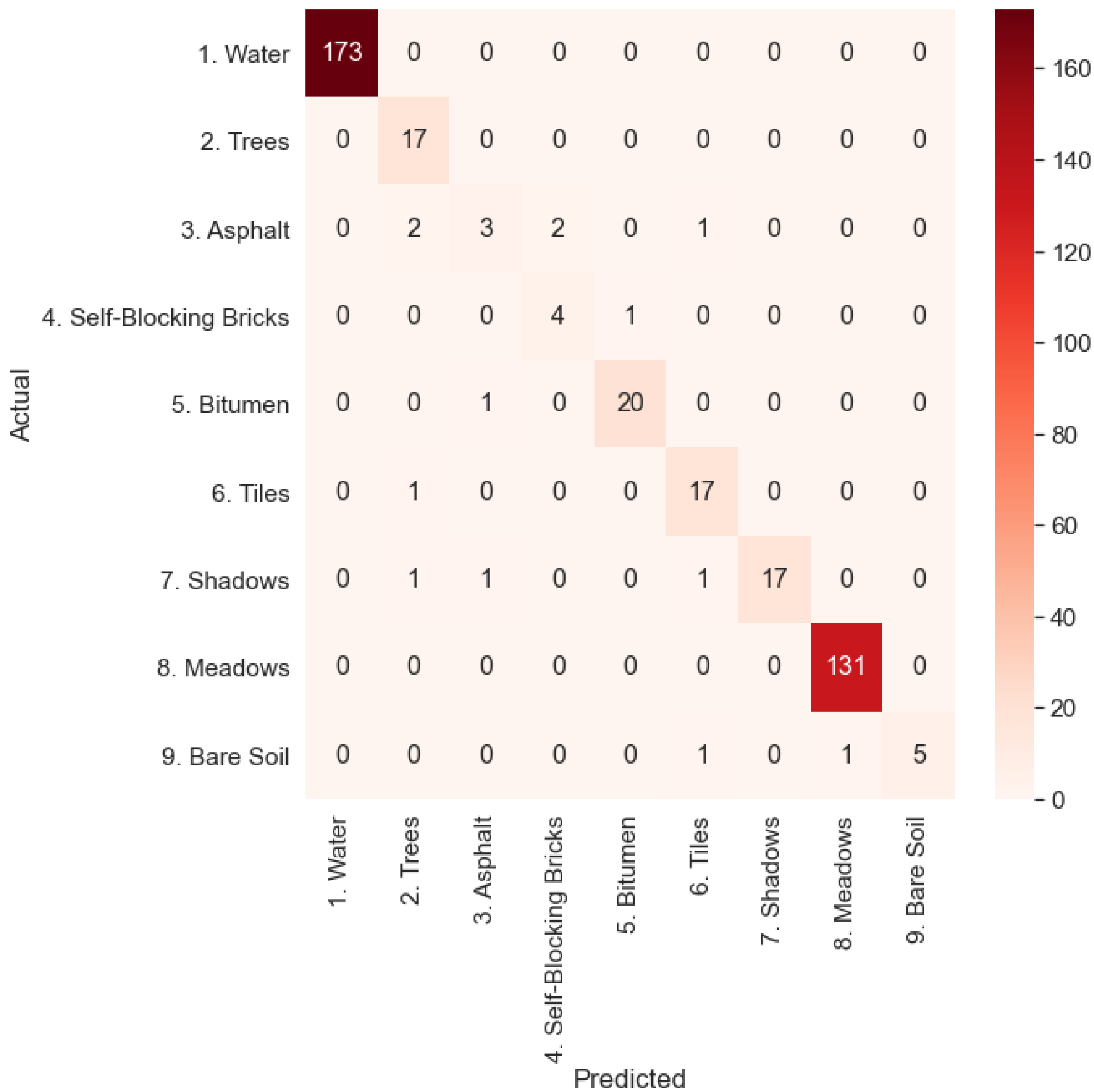


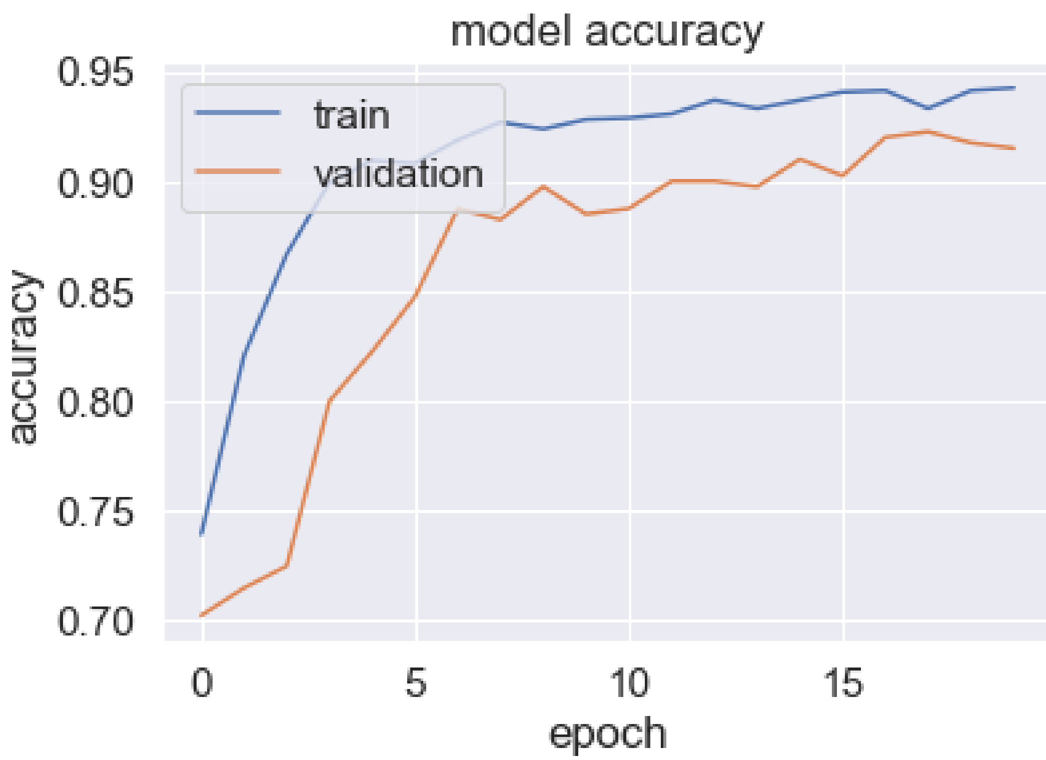

```
Score for fold 2: loss of 0.12703128159046173; accuracy of 96.74999713897705%
13/13 [==============================] - 1s 37ms/step
[[173   0   0   0   0   0   0   0   0]
 [  0  17   0   0   0   0   0   0   0]
 [  0   2   3   2   0   1   0   0   0]
 [  0   0   0   4   1   0   0   0   0]
 [  0   0   1   0  20   0   0   0   0]
 [  0   1   0   0   0  17   0   0   0]
 [  0   1   1   0   0   1  17   0   0]
 [  0   0   0   0   0   0   0 131   0]
 [  0   0   0   0   0   1   0   1   5]]
------------------------------------------------------------------------
Training for fold 3 …
Epoch 1/20
50/50 - 5s - loss: 0.9890 - accuracy: 0.7394 - val_loss: 1.4107 - val_accuracy:
0.7025 - 5s/epoch - 101ms/step
Epoch 2/20
50/50 - 3s - loss: 0.5586 - accuracy: 0.8206 - val_loss: 1.0355 - val_accuracy:
0.7150 - 3s/epoch - 50ms/step
Epoch 3/20
50/50 - 3s - loss: 0.4301 - accuracy: 0.8669 - val_loss: 0.8088 - val_accuracy:
0.7250 - 3s/epoch - 51ms/step
Epoch 4/20
```

```
50/50 - 2s - loss: 0.3555 - accuracy: 0.8994 - val_loss: 0.6545 - val_accuracy:
0.8000 - 2s/epoch - 48ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3140 - accuracy: 0.9094 - val_loss: 0.5335 - val_accuracy:
0.8225 - 2s/epoch - 47ms/step
Epoch 6/20
50/50 - 2s - loss: 0.2932 - accuracy: 0.9081 - val_loss: 0.4600 - val_accuracy:
0.8475 - 2s/epoch - 47ms/step
Epoch 7/20
50/50 - 2s - loss: 0.2657 - accuracy: 0.9187 - val_loss: 0.3891 - val_accuracy:
0.8875 - 2s/epoch - 49ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2501 - accuracy: 0.9269 - val_loss: 0.3453 - val_accuracy:
0.8825 - 2s/epoch - 48ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2456 - accuracy: 0.9237 - val_loss: 0.3168 - val_accuracy:
0.8975 - 2s/epoch - 49ms/step
Epoch 10/20
50/50 - 3s - loss: 0.2294 - accuracy: 0.9281 - val_loss: 0.3057 - val_accuracy:
0.8850 - 3s/epoch - 52ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2209 - accuracy: 0.9287 - val_loss: 0.2847 - val_accuracy:
0.8875 - 2s/epoch - 47ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2264 - accuracy: 0.9306 - val_loss: 0.2721 - val_accuracy:
0.9000 - 2s/epoch - 50ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2037 - accuracy: 0.9369 - val_loss: 0.2635 - val_accuracy:
0.9000 - 2s/epoch - 48ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2114 - accuracy: 0.9331 - val_loss: 0.2534 - val_accuracy:
0.8975 - 2s/epoch - 47ms/step
Epoch 15/20
50/50 - 2s - loss: 0.1993 - accuracy: 0.9369 - val_loss: 0.2442 - val_accuracy:
0.9100 - 2s/epoch - 47ms/step
Epoch 16/20
50/50 - 3s - loss: 0.1893 - accuracy: 0.9406 - val_loss: 0.2439 - val_accuracy:
0.9025 - 3s/epoch - 54ms/step
Epoch 17/20
50/50 - 2s - loss: 0.1881 - accuracy: 0.9413 - val_loss: 0.2358 - val_accuracy:
0.9200 - 2s/epoch - 45ms/step
Epoch 18/20
50/50 - 2s - loss: 0.1915 - accuracy: 0.9331 - val_loss: 0.2371 - val_accuracy:
0.9225 - 2s/epoch - 48ms/step
Epoch 19/20
50/50 - 2s - loss: 0.1801 - accuracy: 0.9413 - val_loss: 0.2303 - val_accuracy:
0.9175 - 2s/epoch - 45ms/step
Epoch 20/20
```

```
50/50 - 2s - loss: 0.1762 - accuracy: 0.9425 - val_loss: 0.2355 - val_accuracy:
0.9150 - 2s/epoch - 45ms/step
```

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



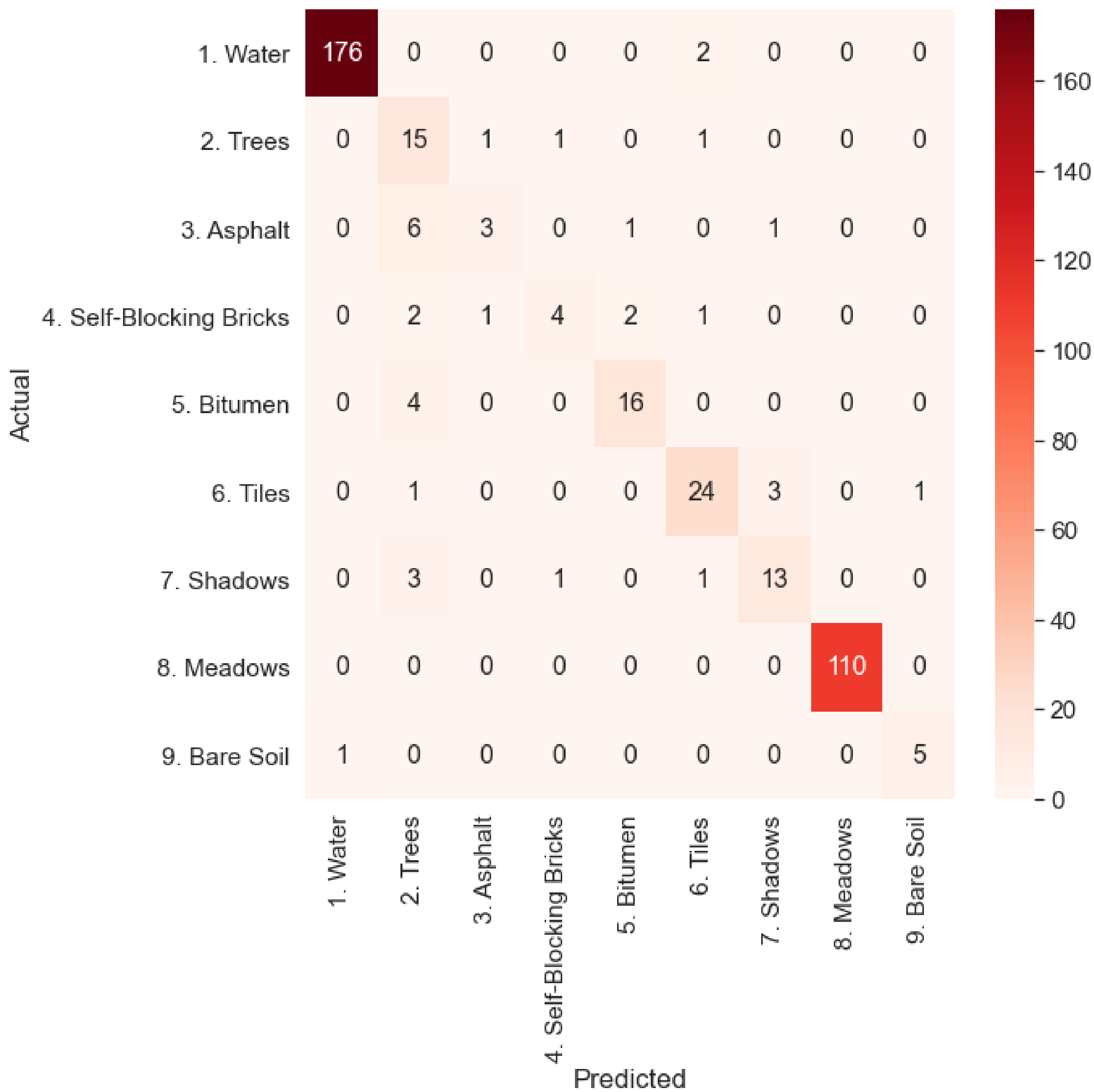


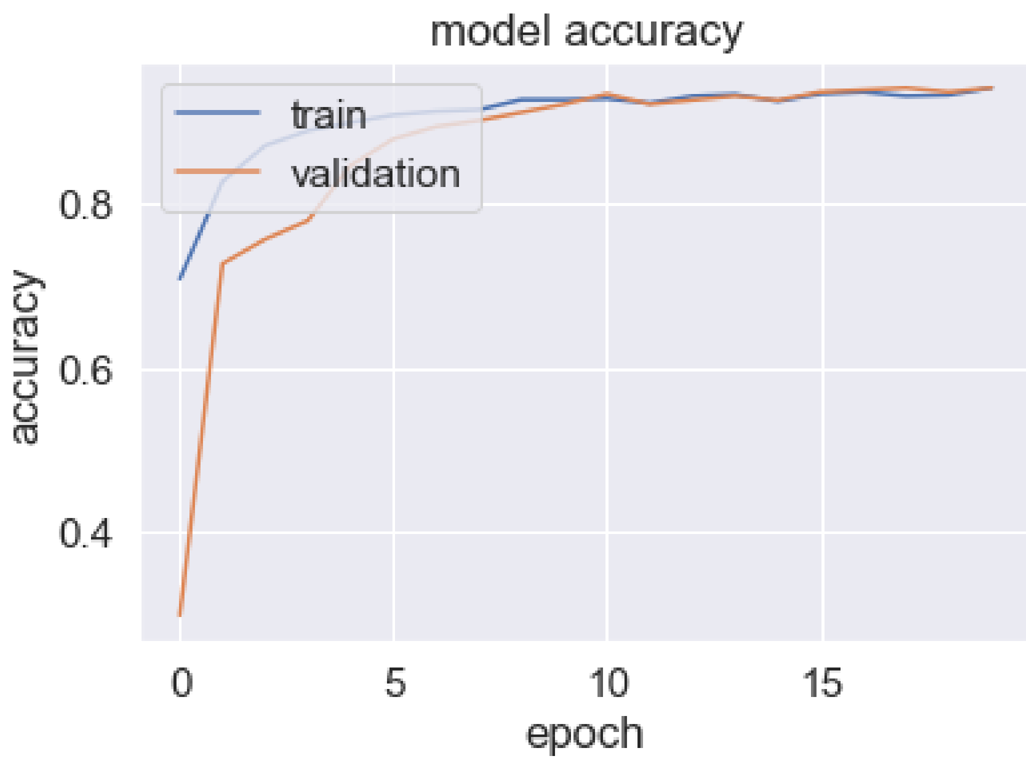

```
Score for fold 3: loss of 0.23551227152347565; accuracy of 91.50000214576721%
13/13 [==============================] - 1s 34ms/step
[[176   0   0   0   0   2   0   0   0]
 [  0  15   1   1   0   1   0   0   0]
 [  0   6   3   0   1   0   1   0   0]
 [  0   2   1   4   2   1   0   0   0]
 [  0   4   0   0  16   0   0   0   0]
 [  0   1   0   0   0  24   3   0   1]
 [  0   3   0   1   0   1  13   0   0]
 [  0   0   0   0   0   0   0 110   0]
 [  1   0   0   0   0   0   0   0   5]]
------------------------------------------------------------------------
Training for fold 4 …
Epoch 1/20
50/50 - 5s - loss: 1.0290 - accuracy: 0.7088 - val_loss: 1.6688 - val_accuracy:
0.2975 - 5s/epoch - 101ms/step
Epoch 2/20
50/50 - 2s - loss: 0.5529 - accuracy: 0.8281 - val_loss: 1.1567 - val_accuracy:
0.7275 - 2s/epoch - 46ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4295 - accuracy: 0.8719 - val_loss: 0.8895 - val_accuracy:
0.7575 - 2s/epoch - 46ms/step
Epoch 4/20
```

```
50/50 - 2s - loss: 0.3695 - accuracy: 0.8900 - val_loss: 0.6275 - val_accuracy:
0.7800 - 2s/epoch - 46ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3201 - accuracy: 0.9000 - val_loss: 0.5113 - val_accuracy:
0.8475 - 2s/epoch - 47ms/step
Epoch 6/20
50/50 - 2s - loss: 0.2919 - accuracy: 0.9094 - val_loss: 0.4328 - val_accuracy:
0.8800 - 2s/epoch - 47ms/step
Epoch 7/20
50/50 - 2s - loss: 0.2836 - accuracy: 0.9137 - val_loss: 0.3941 - val_accuracy:
0.8950 - 2s/epoch - 46ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2541 - accuracy: 0.9156 - val_loss: 0.3530 - val_accuracy:
0.9025 - 2s/epoch - 48ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2346 - accuracy: 0.9281 - val_loss: 0.3199 - val_accuracy:
0.9125 - 2s/epoch - 46ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2308 - accuracy: 0.9281 - val_loss: 0.3038 - val_accuracy:
0.9225 - 2s/epoch - 46ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2232 - accuracy: 0.9294 - val_loss: 0.2939 - val_accuracy:
0.9350 - 2s/epoch - 47ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2227 - accuracy: 0.9237 - val_loss: 0.2750 - val_accuracy:
0.9225 - 2s/epoch - 49ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2007 - accuracy: 0.9325 - val_loss: 0.2663 - val_accuracy:
0.9275 - 2s/epoch - 47ms/step
Epoch 14/20
50/50 - 3s - loss: 0.1998 - accuracy: 0.9344 - val_loss: 0.2639 - val_accuracy:
0.9325 - 3s/epoch - 52ms/step
Epoch 15/20
50/50 - 3s - loss: 0.2015 - accuracy: 0.9262 - val_loss: 0.2514 - val_accuracy:
0.9275 - 3s/epoch - 51ms/step
Epoch 16/20
50/50 - 2s - loss: 0.1916 - accuracy: 0.9350 - val_loss: 0.2476 - val_accuracy:
0.9375 - 2s/epoch - 46ms/step
Epoch 17/20
50/50 - 2s - loss: 0.1901 - accuracy: 0.9375 - val_loss: 0.2440 - val_accuracy:
0.9400 - 2s/epoch - 49ms/step
Epoch 18/20
50/50 - 2s - loss: 0.1869 - accuracy: 0.9325 - val_loss: 0.2383 - val_accuracy:
0.9425 - 2s/epoch - 48ms/step
Epoch 19/20
50/50 - 3s - loss: 0.1841 - accuracy: 0.9337 - val_loss: 0.2289 - val_accuracy:
0.9375 - 3s/epoch - 52ms/step
Epoch 20/20
```

```
50/50 - 3s - loss: 0.1751 - accuracy: 0.9425 - val_loss: 0.2312 - val_accuracy:
0.9425 - 3s/epoch - 52ms/step
```

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



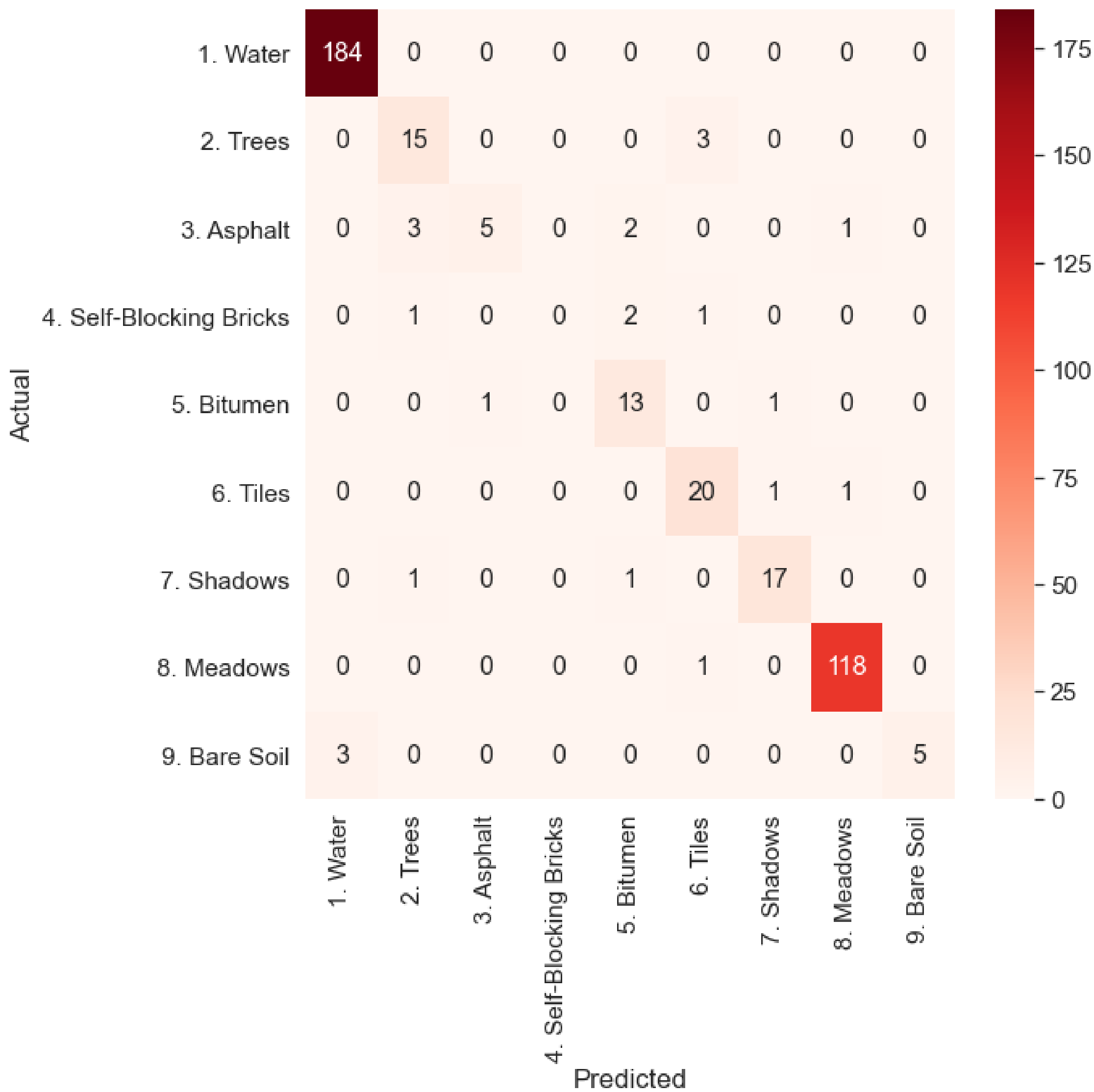


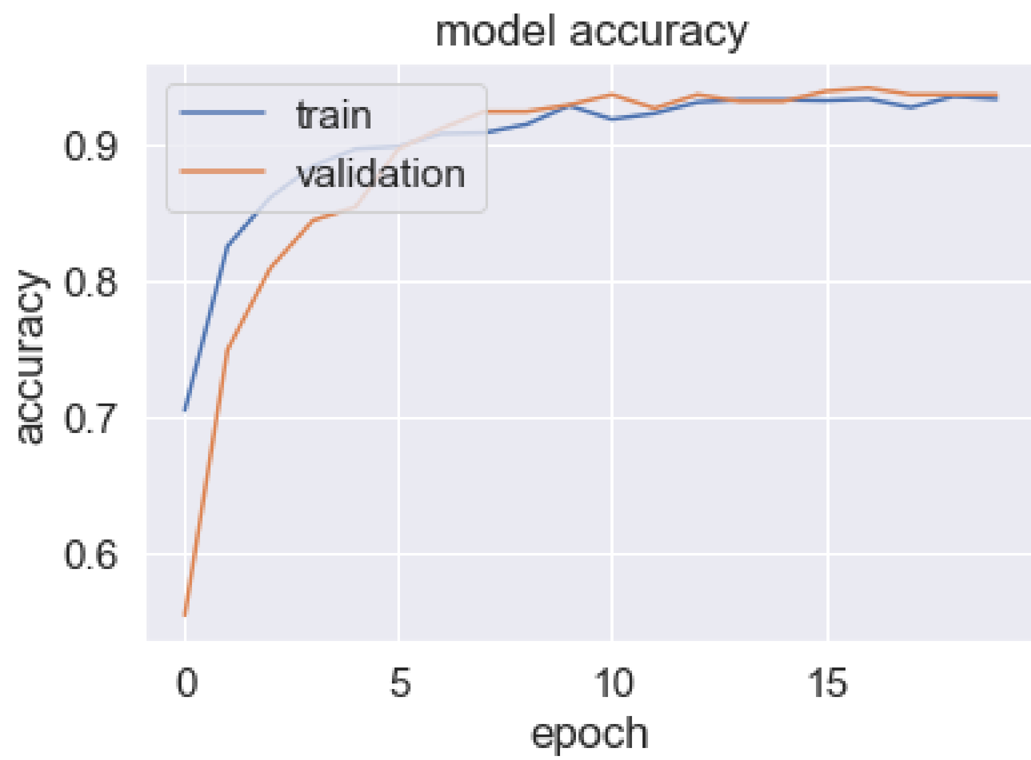

```
Score for fold 4: loss of 0.2312161773443222; accuracy of 94.24999952316284%
13/13 [==============================] - 1s 37ms/step
[[184   0   0   0   0   0   0   0   0]
 [  0  15   0   0   0   3   0   0   0]
 [  0   3   5   0   2   0   0   1   0]
 [  0   1   0   0   2   1   0   0   0]
 [  0   0   1   0  13   0   1   0   0]
 [  0   0   0   0   0  20   1   1   0]
 [  0   1   0   0   1   0  17   0   0]
 [  0   0   0   0   0   1   0 118   0]
 [  3   0   0   0   0   0   0   0   5]]
------------------------------------------------------------------------
Training for fold 5 …
Epoch 1/20
50/50 - 6s - loss: 1.0211 - accuracy: 0.7056 - val_loss: 1.1931 - val_accuracy:
0.5550 - 6s/epoch - 111ms/step
Epoch 2/20
50/50 - 3s - loss: 0.5491 - accuracy: 0.8263 - val_loss: 0.9119 - val_accuracy:
0.7500 - 3s/epoch - 50ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4389 - accuracy: 0.8619 - val_loss: 0.7499 - val_accuracy:
0.8100 - 2s/epoch - 47ms/step
Epoch 4/20
```

```
50/50 - 2s - loss: 0.3768 - accuracy: 0.8850 - val_loss: 0.5863 - val_accuracy:
0.8450 - 2s/epoch - 49ms/step
Epoch 5/20
50/50 - 3s - loss: 0.3324 - accuracy: 0.8975 - val_loss: 0.4646 - val_accuracy:
0.8550 - 3s/epoch - 52ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3100 - accuracy: 0.8994 - val_loss: 0.3914 - val_accuracy:
0.8975 - 2s/epoch - 49ms/step
Epoch 7/20
50/50 - 2s - loss: 0.2913 - accuracy: 0.9087 - val_loss: 0.3308 - val_accuracy:
0.9125 - 2s/epoch - 49ms/step
Epoch 8/20
50/50 - 3s - loss: 0.2771 - accuracy: 0.9094 - val_loss: 0.2848 - val_accuracy:
0.9250 - 3s/epoch - 55ms/step
Epoch 9/20
50/50 - 3s - loss: 0.2583 - accuracy: 0.9156 - val_loss: 0.2542 - val_accuracy:
0.9250 - 3s/epoch - 56ms/step
Epoch 10/20
50/50 - 3s - loss: 0.2375 - accuracy: 0.9294 - val_loss: 0.2331 - val_accuracy:
0.9300 - 3s/epoch - 55ms/step
Epoch 11/20
50/50 - 3s - loss: 0.2467 - accuracy: 0.9194 - val_loss: 0.2126 - val_accuracy:
0.9375 - 3s/epoch - 54ms/step
Epoch 12/20
50/50 - 3s - loss: 0.2279 - accuracy: 0.9237 - val_loss: 0.1992 - val_accuracy:
0.9275 - 3s/epoch - 52ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2159 - accuracy: 0.9319 - val_loss: 0.1909 - val_accuracy:
0.9375 - 2s/epoch - 48ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2146 - accuracy: 0.9337 - val_loss: 0.1877 - val_accuracy:
0.9325 - 2s/epoch - 49ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2068 - accuracy: 0.9337 - val_loss: 0.1895 - val_accuracy:
0.9325 - 2s/epoch - 47ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2185 - accuracy: 0.9331 - val_loss: 0.1843 - val_accuracy:
0.9400 - 2s/epoch - 47ms/step
Epoch 17/20
50/50 - 2s - loss: 0.2049 - accuracy: 0.9344 - val_loss: 0.1763 - val_accuracy:
0.9425 - 2s/epoch - 49ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2135 - accuracy: 0.9281 - val_loss: 0.1745 - val_accuracy:
0.9375 - 2s/epoch - 49ms/step
Epoch 19/20
50/50 - 3s - loss: 0.1929 - accuracy: 0.9362 - val_loss: 0.1680 - val_accuracy:
0.9375 - 3s/epoch - 51ms/step
Epoch 20/20
```

```
50/50 - 2s - loss: 0.1868 - accuracy: 0.9344 - val_loss: 0.1686 - val_accuracy:
0.9375 - 2s/epoch - 49ms/step
```

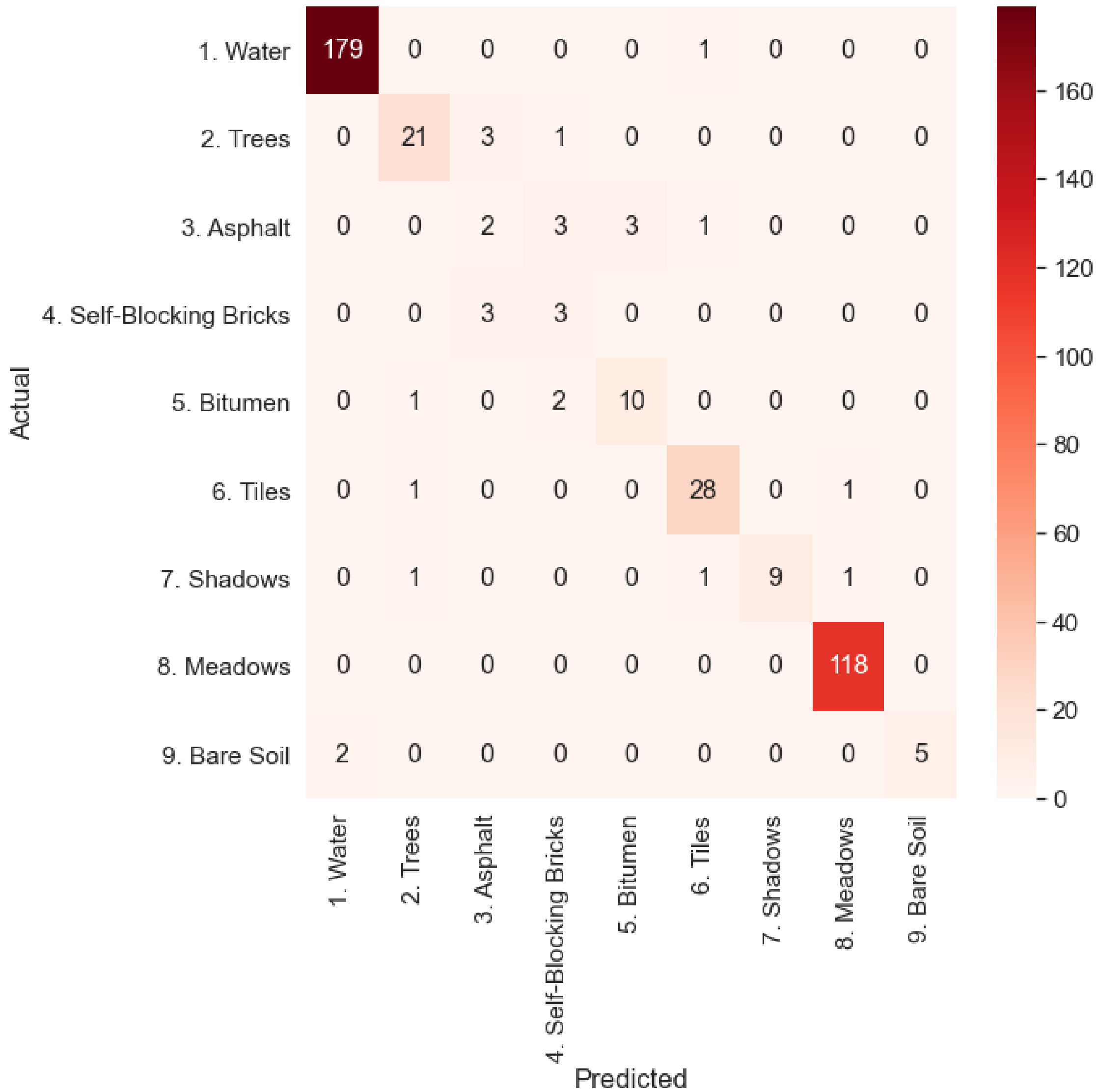
```
<Figure size 432x288 with 0 Axes>
```



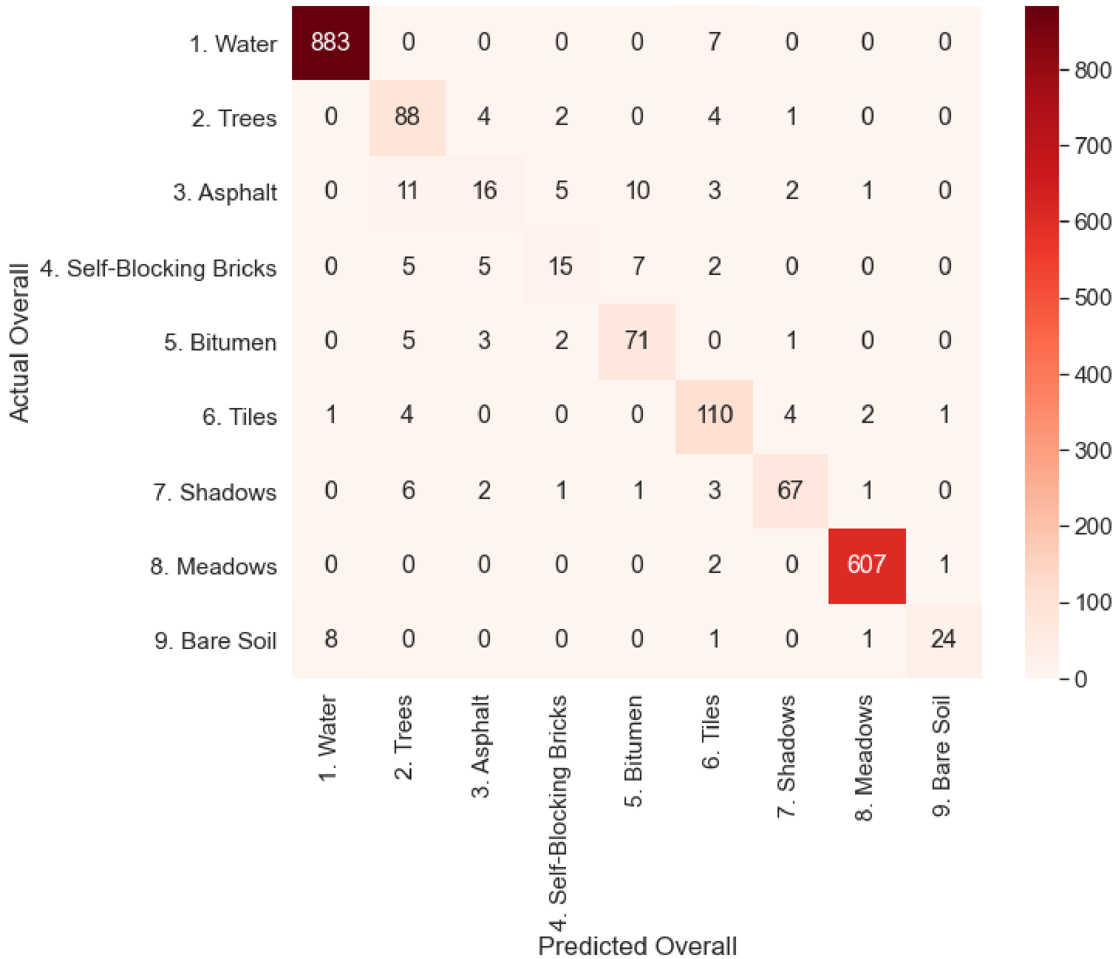



```
Score for fold 5: loss of 0.16860586404800415; accuracy of 93.75%
13/13 [==============================] - 1s 42ms/step
[[179   0   0   0   0   1   0   0   0]
 [  0  21   3   1   0   0   0   0   0]
 [  0   0   2   3   3   1   0   0   0]
 [  0   0   3   3   0   0   0   0   0]
 [  0   1   0   2  10   0   0   0   0]
 [  0   1   0   0   0  28   0   1   0]
 [  0   1   0   0   0   1   9   1   0]
 [  0   0   0   0   0   0   0 118   0]
 [  2   0   0   0   0   0   0   0   5]]
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.15701694786548615 - Accuracy: 93.99999976158142%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.12703128159046173 - Accuracy: 96.74999713897705%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.23551227152347565 - Accuracy: 91.50000214576721%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.2312161773443222 - Accuracy: 94.24999952316284%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.16860586404800415 - Accuracy: 93.75%
```

```
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 94.0499997138977 (+- 1.6688303353057166)
> Loss: 0.18387650847434997
------------------------------------------------------------------------
Predicted Overall        1. Water  2. Trees  3. Asphalt  \
Actual Overall
1. Water                      883         0           0
2. Trees                        0        88           4
3. Asphalt                      0        11          16
4. Self-Blocking Bricks         0         5           5
5. Bitumen                      0         5           3
6. Tiles                        1         4           0
7. Shadows                      0         6           2
8. Meadows                      0         0           0
9. Bare Soil                    8         0           0

Predicted Overall        4. Self-Blocking Bricks  5. Bitumen  6. Tiles  \
Actual Overall
1. Water                                       0           0         7
2. Trees                                       2           0         4
3. Asphalt                                     5          10         3
4. Self-Blocking Bricks                       15           7         2
5. Bitumen                                     2          71         0
6. Tiles                                       0           0       110
7. Shadows                                     1           1         3
8. Meadows                                     0           0         2
9. Bare Soil                                   0           0         1

Predicted Overall        7. Shadows  8. Meadows  9. Bare Soil
Actual Overall
1. Water                          0           0             0
2. Trees                          1           0             0
3. Asphalt                        2           1             0
4. Self-Blocking Bricks           0           0             0
5. Bitumen                        1           0             0
6. Tiles                          4           2             1
7. Shadows                       67           1             0
8. Meadows                        0         607             1
9. Bare Soil                      0           1            24

<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
179 0 0 0 0 1 0 0 0
0 21 3 1 0 0 0 0 0
0 0 2 3 3 1 0 0 0
0 0 3 3 0 0 0 0 0
0 1 0 2 10 0 0 0 0
0 1 0 0 0 28 0 1 0
0 1 0 0 0 1 9 1 0
0 0 0 0 0 0 0 118 0
2 0 0 0 0 0 0 0 5
160
140
120
100
80
60
40
20
0



Actual Overall
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
883 0 0 0 0 7 0 0 0
0 88 4 2 0 4 1 0 0
0 11 16 5 10 3 2 1 0
0 5 5 15 7 2 0 0 0
0 5 3 2 71 0 1 0 0
1 4 0 0 0 110 4 2 1
0 6 2 1 1 3 67 1 0
0 0 0 0 0 2 0 607 1
8 0 0 0 0 1 0 1 24
Predicted Overall
800
700
600
500
400
300
200
100
0