5 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_
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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=5
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")
```

```
# 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
```

[ ]:
```
# 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']
```

[ ]:
```
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
```

```
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
```

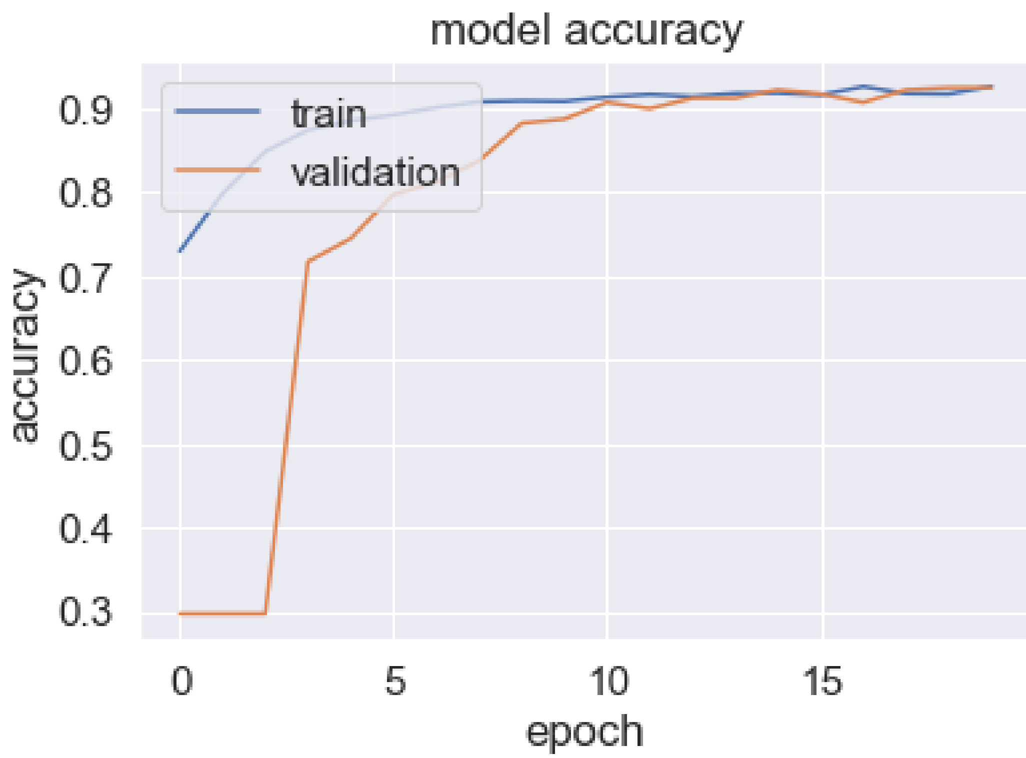
```
 # Compile the model
 #model.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
↪metrics=['accuracy'])

  # 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)
```

```
    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 - 6s - loss: 1.0164 - accuracy: 0.7300 - val_loss: 1.8869 - val_accuracy:
0.2975 - 6s/epoch - 118ms/step
Epoch 2/20
50/50 - 2s - loss: 0.6023 - accuracy: 0.7987 - val_loss: 1.8307 - val_accuracy:
0.2975 - 2s/epoch - 48ms/step
Epoch 3/20
50/50 - 3s - loss: 0.4664 - accuracy: 0.8487 - val_loss: 1.5462 - val_accuracy:
0.2975 - 3s/epoch - 51ms/step
Epoch 4/20
50/50 - 2s - loss: 0.3986 - accuracy: 0.8744 - val_loss: 1.2071 - val_accuracy:
0.7175 - 2s/epoch - 49ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3521 - accuracy: 0.8856 - val_loss: 0.9352 - val_accuracy:
0.7450 - 2s/epoch - 41ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3190 - accuracy: 0.8925 - val_loss: 0.6929 - val_accuracy:
0.7975 - 2s/epoch - 42ms/step
Epoch 7/20
50/50 - 2s - loss: 0.3025 - accuracy: 0.9013 - val_loss: 0.5778 - val_accuracy:
0.8125 - 2s/epoch - 42ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2736 - accuracy: 0.9081 - val_loss: 0.4628 - val_accuracy:
0.8375 - 2s/epoch - 40ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2723 - accuracy: 0.9094 - val_loss: 0.3827 - val_accuracy:
0.8825 - 2s/epoch - 42ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2678 - accuracy: 0.9087 - val_loss: 0.3301 - val_accuracy:
0.8875 - 2s/epoch - 38ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2494 - accuracy: 0.9137 - val_loss: 0.3018 - val_accuracy:
0.9075 - 2s/epoch - 39ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2427 - accuracy: 0.9169 - val_loss: 0.2898 - val_accuracy:
0.9000 - 2s/epoch - 44ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2470 - accuracy: 0.9144 - val_loss: 0.2741 - val_accuracy:
0.9125 - 2s/epoch - 43ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2377 - accuracy: 0.9181 - val_loss: 0.2647 - val_accuracy:
0.9125 - 2s/epoch - 46ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2352 - accuracy: 0.9187 - val_loss: 0.2517 - val_accuracy:
```

```
0.9225 - 2s/epoch - 45ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2362 - accuracy: 0.9162 - val_loss: 0.2538 - val_accuracy:
0.9175 - 2s/epoch - 47ms/step
Epoch 17/20
50/50 - 2s - loss: 0.2164 - accuracy: 0.9262 - val_loss: 0.2609 - val_accuracy:
0.9075 - 2s/epoch - 46ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2187 - accuracy: 0.9181 - val_loss: 0.2493 - val_accuracy:
0.9225 - 2s/epoch - 46ms/step
Epoch 19/20
50/50 - 2s - loss: 0.2249 - accuracy: 0.9175 - val_loss: 0.2444 - val_accuracy:
0.9250 - 2s/epoch - 50ms/step
Epoch 20/20
50/50 - 2s - loss: 0.2149 - accuracy: 0.9262 - val_loss: 0.2395 - val_accuracy:
0.9250 - 2s/epoch - 49ms/step
```

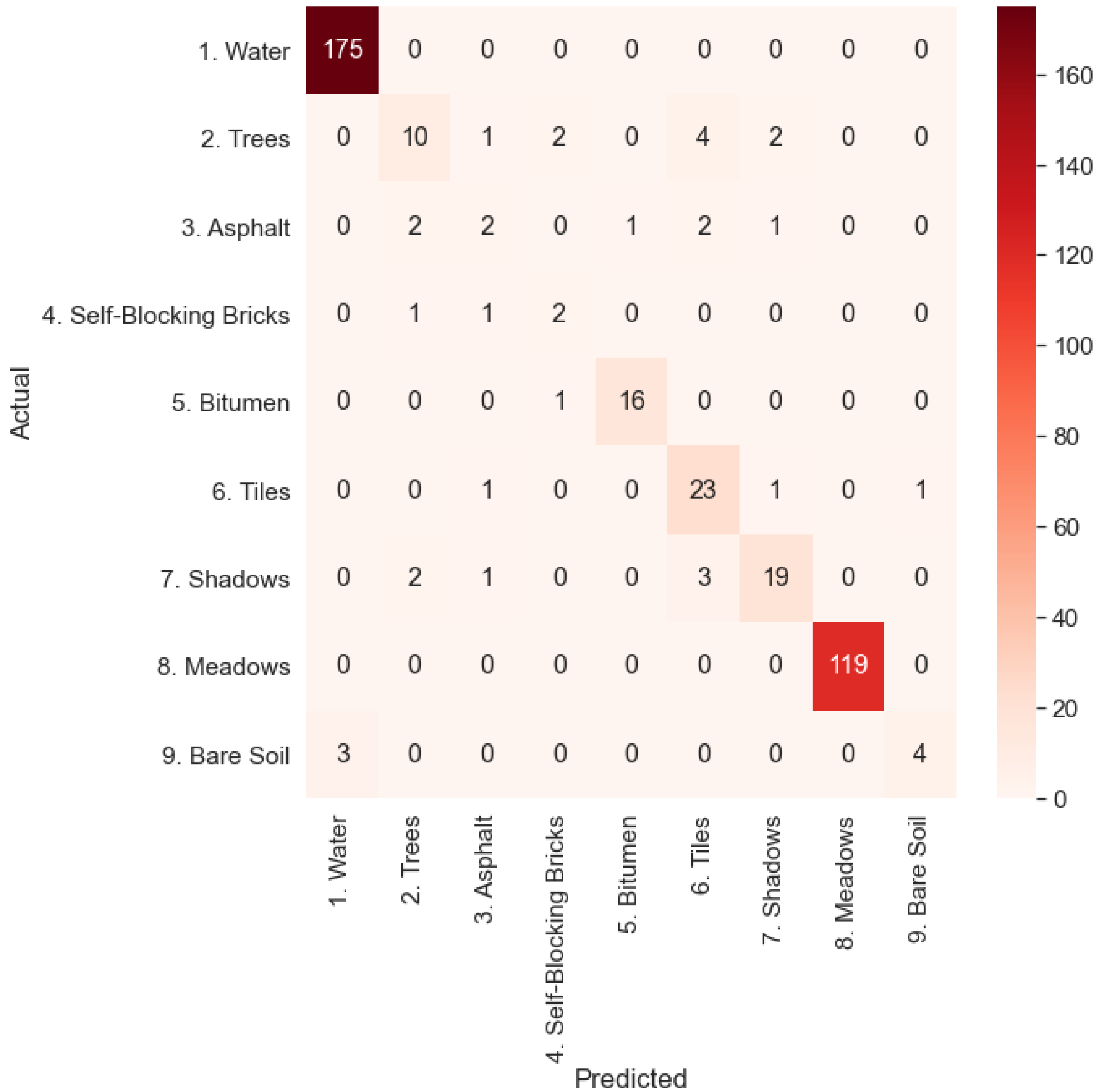


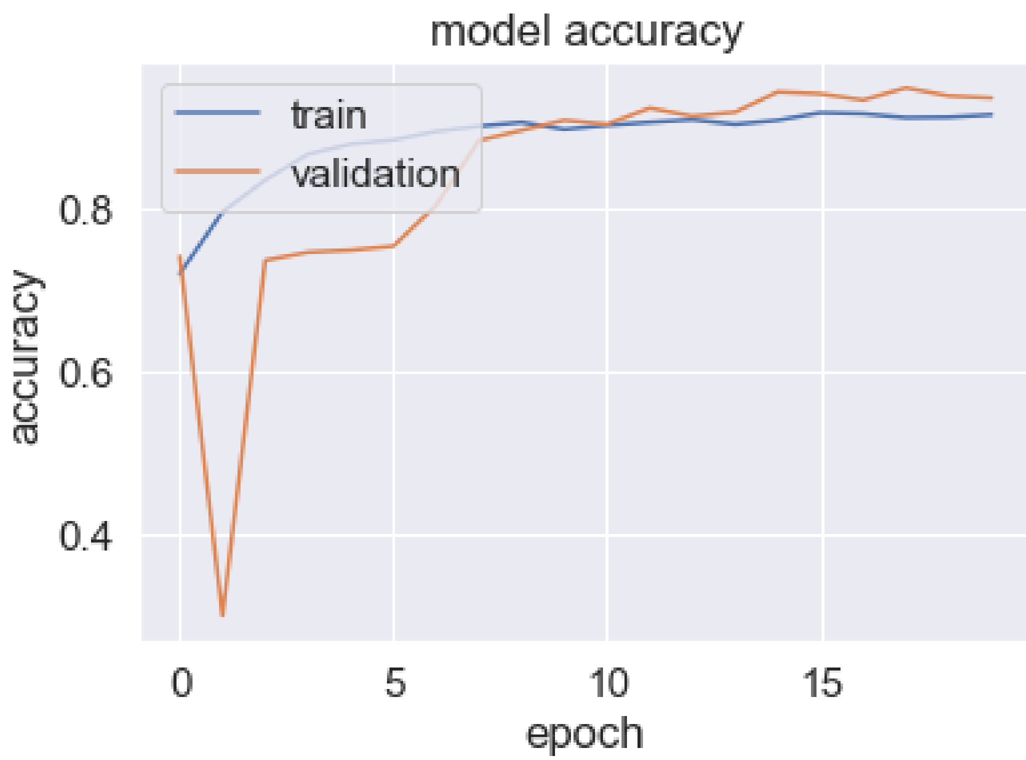

```
Score for fold 1: loss of 0.23945873975753784; accuracy of 92.5000011920929%
13/13 [==============================] - 1s 32ms/step
[[175   0   0   0   0   0   0   0   0]
 [  0  10   1   2   0   4   2   0   0]
 [  0   2   2   0   1   2   1   0   0]
 [  0   1   1   2   0   0   0   0   0]
```

```
 [  0   0   0   1  16   0   0   0   0]
 [  0   0   1   0   0  23   1   0   1]
 [  0   2   1   0   0   3  19   0   0]
 [  0   0   0   0   0   0   0 119   0]
 [  3   0   0   0   0   0   0   0   4]]
------------------------------------------------------------------------
Training for fold 2 …
Epoch 1/20
50/50 - 5s - loss: 1.0625 - accuracy: 0.7212 - val_loss: 1.4126 - val_accuracy:
0.7425 - 5s/epoch - 98ms/step
Epoch 2/20
50/50 - 2s - loss: 0.6231 - accuracy: 0.7969 - val_loss: 1.4739 - val_accuracy:
0.3000 - 2s/epoch - 44ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4954 - accuracy: 0.8363 - val_loss: 1.2333 - val_accuracy:
0.7375 - 2s/epoch - 44ms/step
Epoch 4/20
50/50 - 2s - loss: 0.4155 - accuracy: 0.8681 - val_loss: 1.0447 - val_accuracy:
0.7475 - 2s/epoch - 43ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3592 - accuracy: 0.8806 - val_loss: 0.8304 - val_accuracy:
0.7500 - 2s/epoch - 42ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3412 - accuracy: 0.8856 - val_loss: 0.6548 - val_accuracy:
0.7550 - 2s/epoch - 44ms/step
Epoch 7/20
50/50 - 2s - loss: 0.3125 - accuracy: 0.8963 - val_loss: 0.5245 - val_accuracy:
0.8050 - 2s/epoch - 43ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2928 - accuracy: 0.9025 - val_loss: 0.4081 - val_accuracy:
0.8850 - 2s/epoch - 44ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2815 - accuracy: 0.9075 - val_loss: 0.3397 - val_accuracy:
0.8975 - 2s/epoch - 50ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2814 - accuracy: 0.8988 - val_loss: 0.2898 - val_accuracy:
0.9100 - 2s/epoch - 44ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2648 - accuracy: 0.9038 - val_loss: 0.2538 - val_accuracy:
0.9050 - 2s/epoch - 44ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2586 - accuracy: 0.9075 - val_loss: 0.2353 - val_accuracy:
0.9250 - 2s/epoch - 41ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2524 - accuracy: 0.9106 - val_loss: 0.2200 - val_accuracy:
0.9150 - 2s/epoch - 41ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2496 - accuracy: 0.9050 - val_loss: 0.2119 - val_accuracy:
```

```
0.9200 - 2s/epoch - 41ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2439 - accuracy: 0.9100 - val_loss: 0.2005 - val_accuracy:
0.9450 - 2s/epoch - 41ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2385 - accuracy: 0.9194 - val_loss: 0.1974 - val_accuracy:
0.9425 - 2s/epoch - 41ms/step
Epoch 17/20
50/50 - 2s - loss: 0.2345 - accuracy: 0.9181 - val_loss: 0.1928 - val_accuracy:
0.9350 - 2s/epoch - 42ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2434 - accuracy: 0.9131 - val_loss: 0.1868 - val_accuracy:
0.9500 - 2s/epoch - 41ms/step
Epoch 19/20
50/50 - 2s - loss: 0.2338 - accuracy: 0.9137 - val_loss: 0.1841 - val_accuracy:
0.9400 - 2s/epoch - 41ms/step
Epoch 20/20
50/50 - 2s - loss: 0.2279 - accuracy: 0.9169 - val_loss: 0.1900 - val_accuracy:
0.9375 - 2s/epoch - 41ms/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
175 0 0 0 0 0 0 0 0
0 10 1 2 0 4 2 0 0
0 2 2 0 1 2 1 0 0
0 1 1 2 0 0 0 0 0
0 0 0 1 16 0 0 0 0
0 0 1 0 0 23 1 0 1
0 2 1 0 0 3 19 0 0
0 0 0 0 0 0 0 119 0
3 0 0 0 0 0 0 0 4
160
140
120
100
80
60
40
20
0

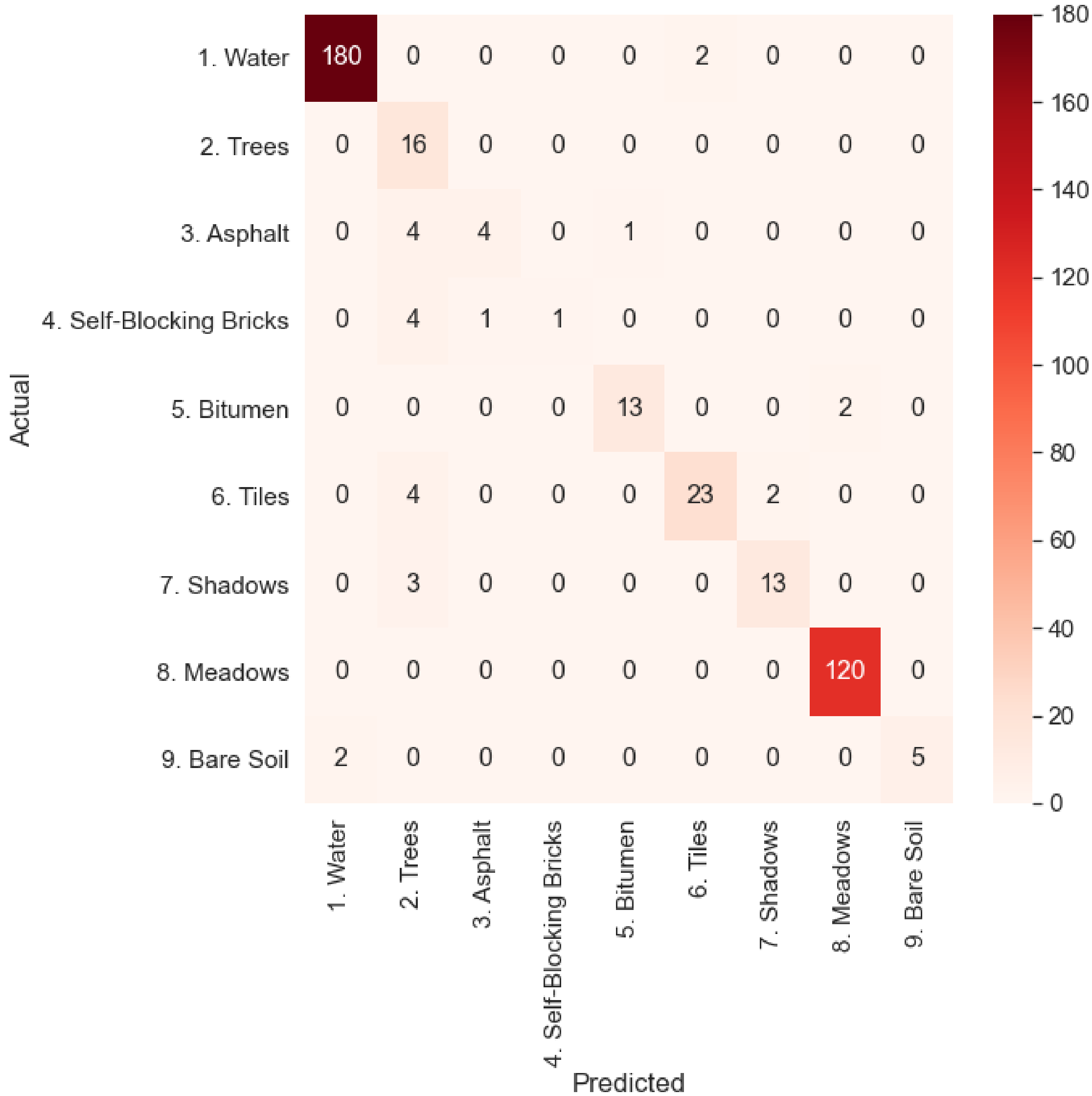


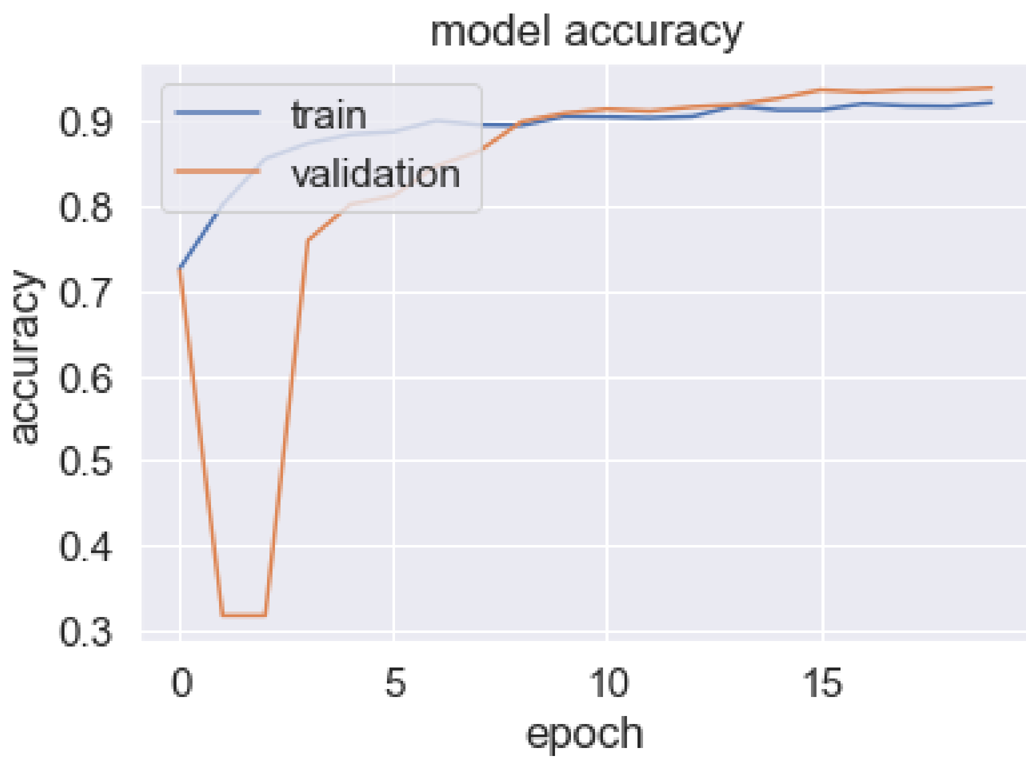

```
Score for fold 2: loss of 0.19002245366573334; accuracy of 93.75%
13/13 [==============================] - 1s 32ms/step
[[180   0   0   0   0   2   0   0   0]
 [  0  16   0   0   0   0   0   0   0]
 [  0   4   4   0   1   0   0   0   0]
 [  0   4   1   1   0   0   0   0   0]
 [  0   0   0   0  13   0   0   2   0]
 [  0   4   0   0   0  23   2   0   0]
 [  0   3   0   0   0   0  13   0   0]
 [  0   0   0   0   0   0   0 120   0]
 [  2   0   0   0   0   0   0   0   5]]
------------------------------------------------------------------------
Training for fold 3 …
Epoch 1/20
50/50 - 5s - loss: 1.0549 - accuracy: 0.7269 - val_loss: 1.4536 - val_accuracy:
0.7250 - 5s/epoch - 103ms/step
Epoch 2/20
50/50 - 2s - loss: 0.6122 - accuracy: 0.8019 - val_loss: 1.5666 - val_accuracy:
0.3175 - 2s/epoch - 47ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4702 - accuracy: 0.8562 - val_loss: 1.4650 - val_accuracy:
0.3175 - 2s/epoch - 47ms/step
Epoch 4/20
```

```
50/50 - 2s - loss: 0.4098 - accuracy: 0.8744 - val_loss: 1.1166 - val_accuracy:
0.7600 - 2s/epoch - 46ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3621 - accuracy: 0.8850 - val_loss: 0.9006 - val_accuracy:
0.8025 - 2s/epoch - 47ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3322 - accuracy: 0.8881 - val_loss: 0.6935 - val_accuracy:
0.8125 - 2s/epoch - 47ms/step
Epoch 7/20
50/50 - 2s - loss: 0.3018 - accuracy: 0.9013 - val_loss: 0.5284 - val_accuracy:
0.8475 - 2s/epoch - 46ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2957 - accuracy: 0.8963 - val_loss: 0.4153 - val_accuracy:
0.8650 - 2s/epoch - 46ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2814 - accuracy: 0.8956 - val_loss: 0.3369 - val_accuracy:
0.9000 - 2s/epoch - 47ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2707 - accuracy: 0.9062 - val_loss: 0.2903 - val_accuracy:
0.9100 - 2s/epoch - 47ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2605 - accuracy: 0.9056 - val_loss: 0.2583 - val_accuracy:
0.9150 - 2s/epoch - 46ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2616 - accuracy: 0.9044 - val_loss: 0.2372 - val_accuracy:
0.9125 - 2s/epoch - 46ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2660 - accuracy: 0.9062 - val_loss: 0.2240 - val_accuracy:
0.9175 - 2s/epoch - 46ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2416 - accuracy: 0.9187 - val_loss: 0.2139 - val_accuracy:
0.9200 - 2s/epoch - 47ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2441 - accuracy: 0.9137 - val_loss: 0.2065 - val_accuracy:
0.9275 - 2s/epoch - 47ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2343 - accuracy: 0.9137 - val_loss: 0.1970 - val_accuracy:
0.9375 - 2s/epoch - 47ms/step
Epoch 17/20
50/50 - 2s - loss: 0.2398 - accuracy: 0.9212 - val_loss: 0.1923 - val_accuracy:
0.9350 - 2s/epoch - 46ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2354 - accuracy: 0.9187 - val_loss: 0.1947 - val_accuracy:
0.9375 - 2s/epoch - 46ms/step
Epoch 19/20
50/50 - 2s - loss: 0.2237 - accuracy: 0.9181 - val_loss: 0.1909 - val_accuracy:
0.9375 - 2s/epoch - 46ms/step
Epoch 20/20
```

```
50/50 - 2s - loss: 0.2303 - accuracy: 0.9225 - val_loss: 0.1833 - val_accuracy:
0.9400 - 2s/epoch - 46ms/step
```

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



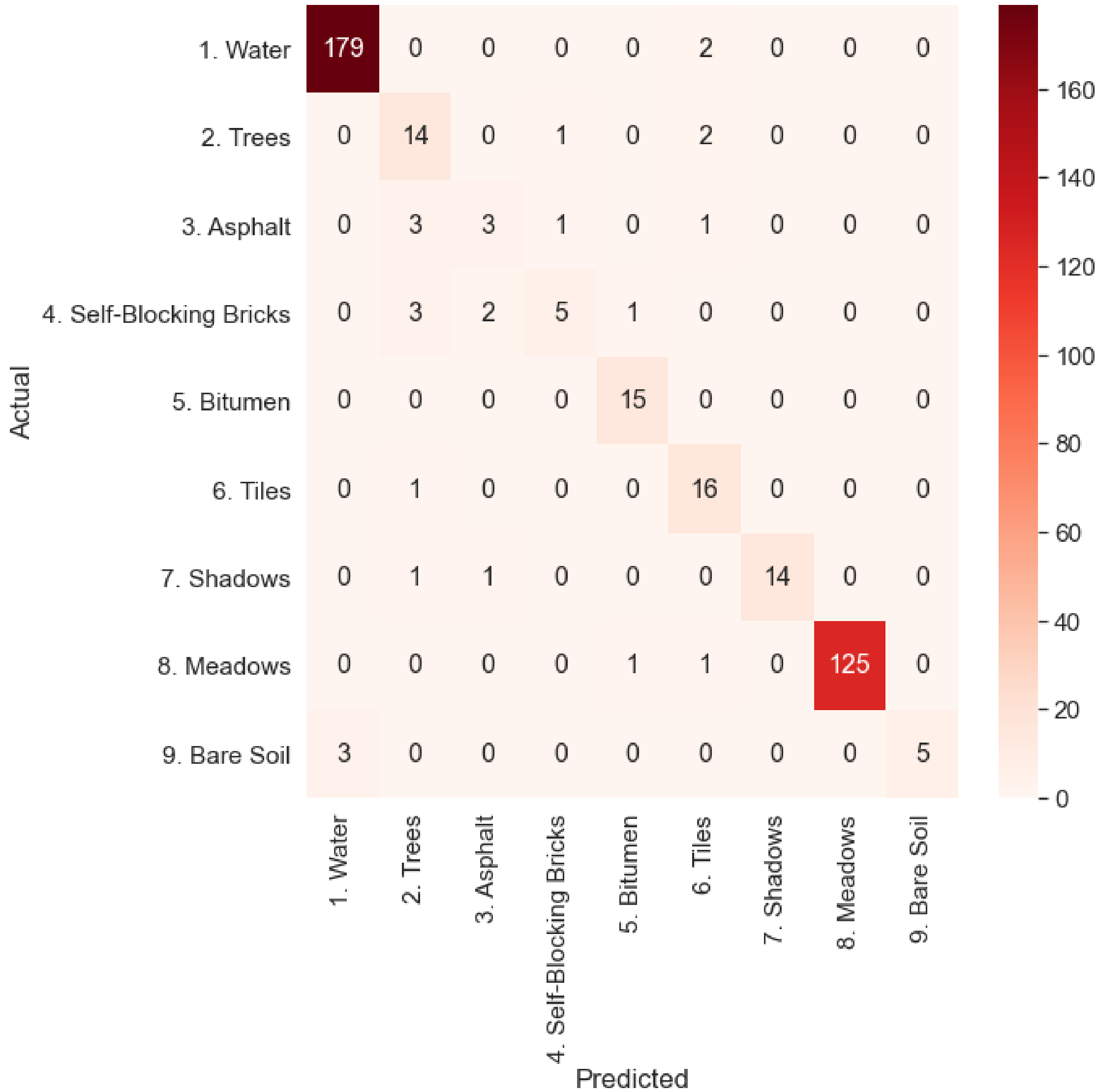


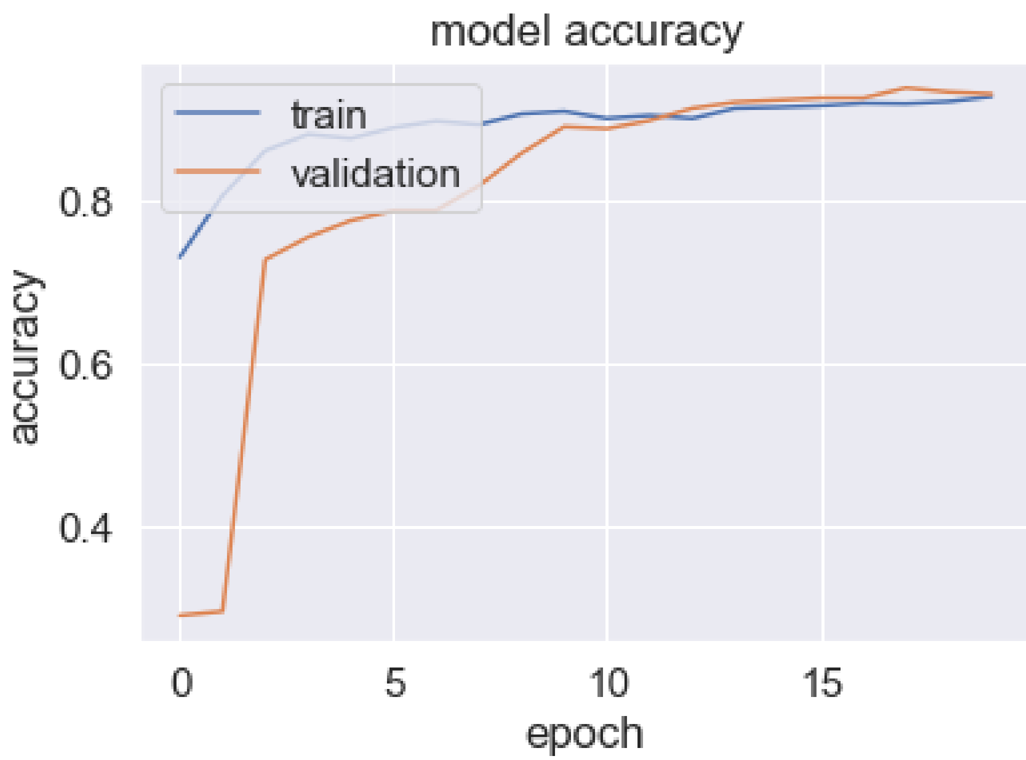

```
Score for fold 3: loss of 0.18330445885658264; accuracy of 93.99999976158142%
13/13 [==============================] - 1s 36ms/step
[[179   0   0   0   0   2   0   0   0]
 [  0  14   0   1   0   2   0   0   0]
 [  0   3   3   1   0   1   0   0   0]
 [  0   3   2   5   1   0   0   0   0]
 [  0   0   0   0  15   0   0   0   0]
 [  0   1   0   0   0  16   0   0   0]
 [  0   1   1   0   0   0  14   0   0]
 [  0   0   0   0   1   1   0 125   0]
 [  3   0   0   0   0   0   0   0   5]]
------------------------------------------------------------------------
Training for fold 4 …
Epoch 1/20
50/50 - 4s - loss: 1.0347 - accuracy: 0.7306 - val_loss: 1.7849 - val_accuracy:
0.2925 - 4s/epoch - 90ms/step
Epoch 2/20
50/50 - 2s - loss: 0.6130 - accuracy: 0.8056 - val_loss: 1.7970 - val_accuracy:
0.2975 - 2s/epoch - 45ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4680 - accuracy: 0.8606 - val_loss: 1.6113 - val_accuracy:
0.7275 - 2s/epoch - 45ms/step
Epoch 4/20
```

```
50/50 - 2s - loss: 0.3945 - accuracy: 0.8806 - val_loss: 1.2656 - val_accuracy:
0.7550 - 2s/epoch - 45ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3739 - accuracy: 0.8756 - val_loss: 1.0165 - val_accuracy:
0.7750 - 2s/epoch - 45ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3334 - accuracy: 0.8888 - val_loss: 0.7590 - val_accuracy:
0.7875 - 2s/epoch - 45ms/step
Epoch 7/20
50/50 - 2s - loss: 0.3080 - accuracy: 0.8969 - val_loss: 0.6516 - val_accuracy:
0.7875 - 2s/epoch - 45ms/step
Epoch 8/20
50/50 - 2s - loss: 0.3038 - accuracy: 0.8925 - val_loss: 0.4955 - val_accuracy:
0.8175 - 2s/epoch - 45ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2741 - accuracy: 0.9056 - val_loss: 0.3769 - val_accuracy:
0.8575 - 2s/epoch - 45ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2610 - accuracy: 0.9087 - val_loss: 0.3184 - val_accuracy:
0.8900 - 2s/epoch - 47ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2729 - accuracy: 0.9000 - val_loss: 0.2946 - val_accuracy:
0.8875 - 2s/epoch - 46ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2653 - accuracy: 0.9038 - val_loss: 0.2697 - val_accuracy:
0.8975 - 2s/epoch - 45ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2532 - accuracy: 0.9000 - val_loss: 0.2440 - val_accuracy:
0.9125 - 2s/epoch - 45ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2391 - accuracy: 0.9125 - val_loss: 0.2290 - val_accuracy:
0.9200 - 2s/epoch - 46ms/step
Epoch 15/20
50/50 - 3s - loss: 0.2462 - accuracy: 0.9137 - val_loss: 0.2237 - val_accuracy:
0.9225 - 3s/epoch - 58ms/step
Epoch 16/20
50/50 - 3s - loss: 0.2425 - accuracy: 0.9156 - val_loss: 0.2120 - val_accuracy:
0.9250 - 3s/epoch - 57ms/step
Epoch 17/20
50/50 - 3s - loss: 0.2355 - accuracy: 0.9187 - val_loss: 0.2113 - val_accuracy:
0.9250 - 3s/epoch - 53ms/step
Epoch 18/20
50/50 - 3s - loss: 0.2408 - accuracy: 0.9181 - val_loss: 0.2024 - val_accuracy:
0.9375 - 3s/epoch - 51ms/step
Epoch 19/20
50/50 - 3s - loss: 0.2185 - accuracy: 0.9206 - val_loss: 0.1992 - val_accuracy:
0.9325 - 3s/epoch - 52ms/step
Epoch 20/20
```

```
50/50 - 2s - loss: 0.2214 - accuracy: 0.9269 - val_loss: 0.1989 - val_accuracy:
0.9300 - 2s/epoch - 50ms/step
```

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



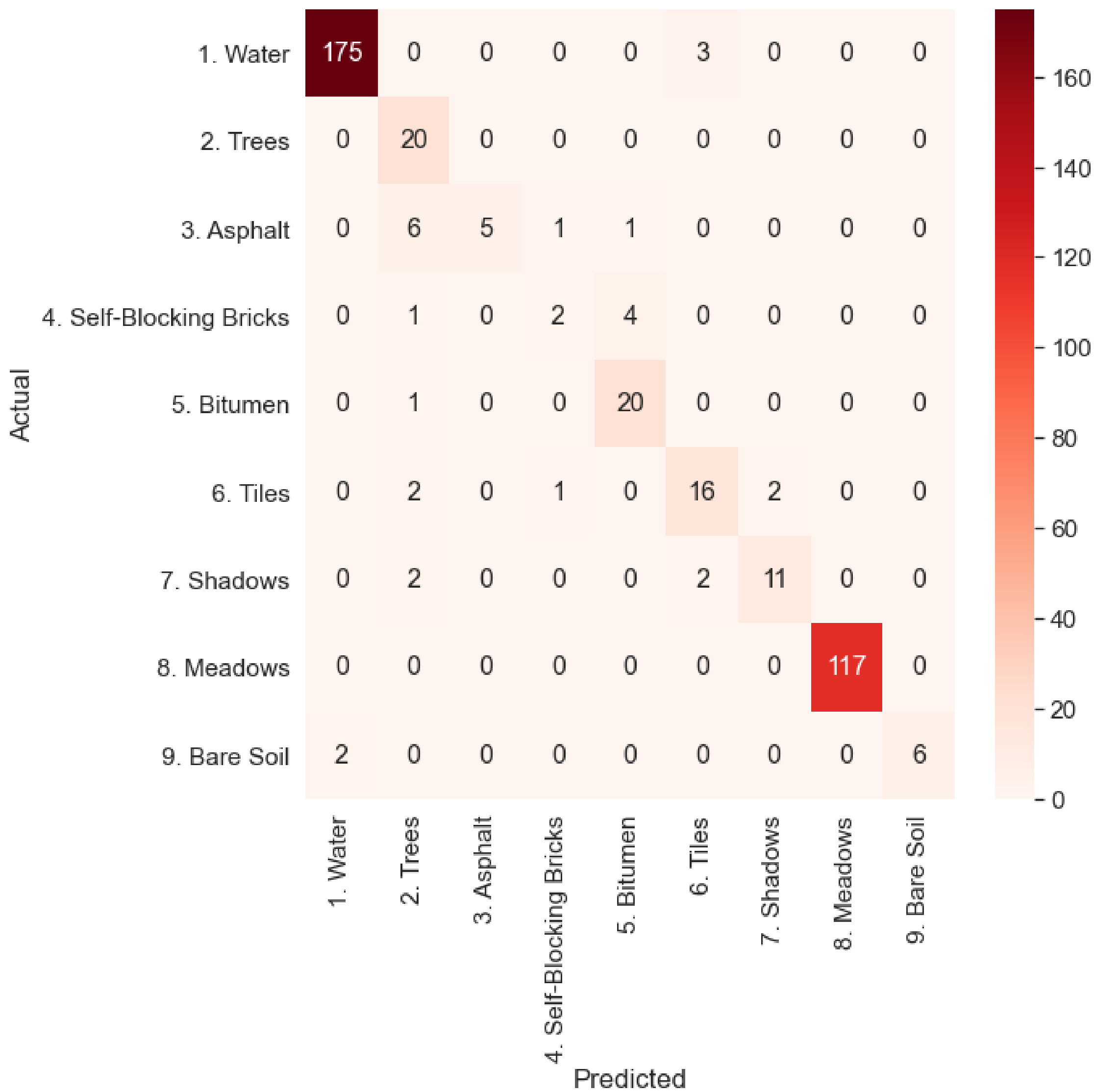


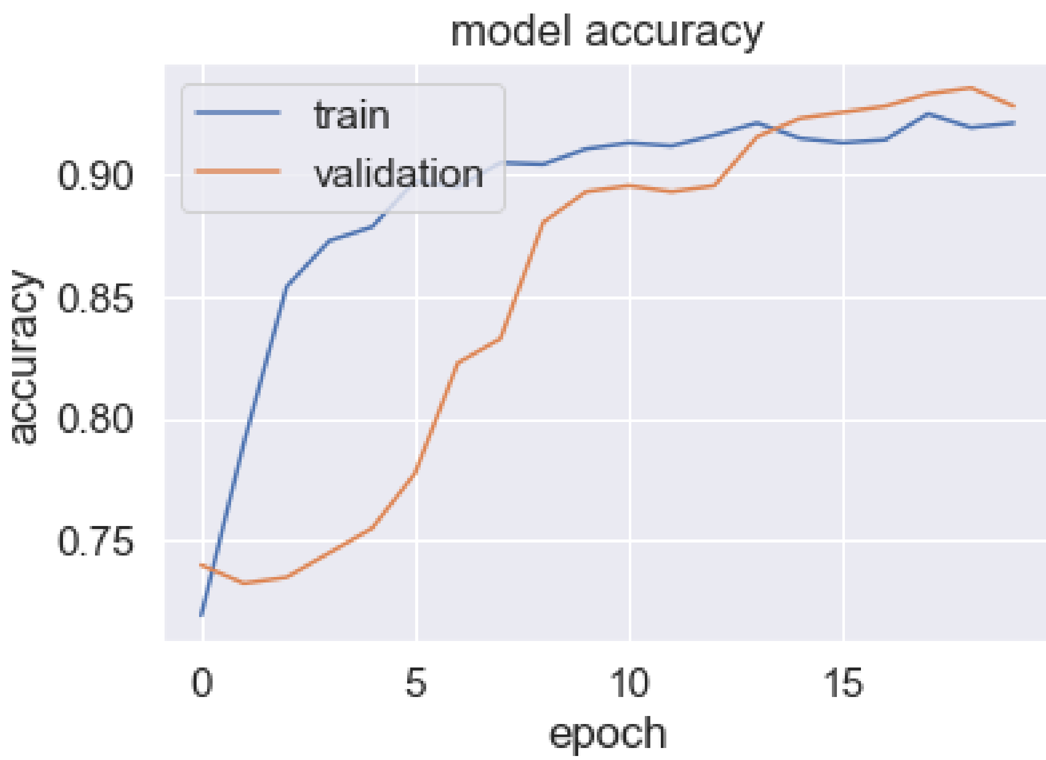

```
Score for fold 4: loss of 0.19887861609458923; accuracy of 93.00000071525574%
13/13 [==============================] - 1s 45ms/step
[[175   0   0   0   0   3   0   0   0]
 [  0  20   0   0   0   0   0   0   0]
 [  0   6   5   1   1   0   0   0   0]
 [  0   1   0   2   4   0   0   0   0]
 [  0   1   0   0  20   0   0   0   0]
 [  0   2   0   1   0  16   2   0   0]
 [  0   2   0   0   0   2  11   0   0]
 [  0   0   0   0   0   0   0 117   0]
 [  2   0   0   0   0   0   0   0   6]]
------------------------------------------------------------------------
Training for fold 5 …
Epoch 1/20
50/50 - 6s - loss: 1.0430 - accuracy: 0.7194 - val_loss: 1.4979 - val_accuracy:
0.7400 - 6s/epoch - 120ms/step
Epoch 2/20
50/50 - 3s - loss: 0.6199 - accuracy: 0.7900 - val_loss: 1.4330 - val_accuracy:
0.7325 - 3s/epoch - 60ms/step
Epoch 3/20
50/50 - 3s - loss: 0.4825 - accuracy: 0.8537 - val_loss: 1.3179 - val_accuracy:
0.7350 - 3s/epoch - 56ms/step
Epoch 4/20
```

```
50/50 - 3s - loss: 0.4083 - accuracy: 0.8725 - val_loss: 1.0859 - val_accuracy:
0.7450 - 3s/epoch - 58ms/step
Epoch 5/20
50/50 - 3s - loss: 0.3643 - accuracy: 0.8781 - val_loss: 0.8675 - val_accuracy:
0.7550 - 3s/epoch - 56ms/step
Epoch 6/20
50/50 - 3s - loss: 0.3253 - accuracy: 0.8969 - val_loss: 0.6773 - val_accuracy:
0.7775 - 3s/epoch - 54ms/step
Epoch 7/20
50/50 - 3s - loss: 0.3066 - accuracy: 0.8944 - val_loss: 0.5067 - val_accuracy:
0.8225 - 3s/epoch - 59ms/step
Epoch 8/20
50/50 - 3s - loss: 0.2896 - accuracy: 0.9044 - val_loss: 0.4407 - val_accuracy:
0.8325 - 3s/epoch - 59ms/step
Epoch 9/20
50/50 - 3s - loss: 0.2868 - accuracy: 0.9038 - val_loss: 0.3472 - val_accuracy:
0.8800 - 3s/epoch - 59ms/step
Epoch 10/20
50/50 - 3s - loss: 0.2683 - accuracy: 0.9100 - val_loss: 0.3033 - val_accuracy:
0.8925 - 3s/epoch - 57ms/step
Epoch 11/20
50/50 - 3s - loss: 0.2602 - accuracy: 0.9125 - val_loss: 0.2681 - val_accuracy:
0.8950 - 3s/epoch - 59ms/step
Epoch 12/20
50/50 - 3s - loss: 0.2642 - accuracy: 0.9112 - val_loss: 0.2489 - val_accuracy:
0.8925 - 3s/epoch - 58ms/step
Epoch 13/20
50/50 - 3s - loss: 0.2494 - accuracy: 0.9156 - val_loss: 0.2397 - val_accuracy:
0.8950 - 3s/epoch - 59ms/step
Epoch 14/20
50/50 - 3s - loss: 0.2441 - accuracy: 0.9206 - val_loss: 0.2171 - val_accuracy:
0.9150 - 3s/epoch - 55ms/step
Epoch 15/20
50/50 - 3s - loss: 0.2396 - accuracy: 0.9144 - val_loss: 0.2098 - val_accuracy:
0.9225 - 3s/epoch - 56ms/step
Epoch 16/20
50/50 - 3s - loss: 0.2488 - accuracy: 0.9125 - val_loss: 0.2159 - val_accuracy:
0.9250 - 3s/epoch - 57ms/step
Epoch 17/20
50/50 - 3s - loss: 0.2420 - accuracy: 0.9137 - val_loss: 0.2026 - val_accuracy:
0.9275 - 3s/epoch - 56ms/step
Epoch 18/20
50/50 - 3s - loss: 0.2152 - accuracy: 0.9244 - val_loss: 0.1996 - val_accuracy:
0.9325 - 3s/epoch - 52ms/step
Epoch 19/20
50/50 - 3s - loss: 0.2384 - accuracy: 0.9187 - val_loss: 0.1978 - val_accuracy:
0.9350 - 3s/epoch - 51ms/step
Epoch 20/20
```

```
50/50 - 3s - loss: 0.2316 - accuracy: 0.9206 - val_loss: 0.1966 - val_accuracy:
0.9275 - 3s/epoch - 51ms/step
```

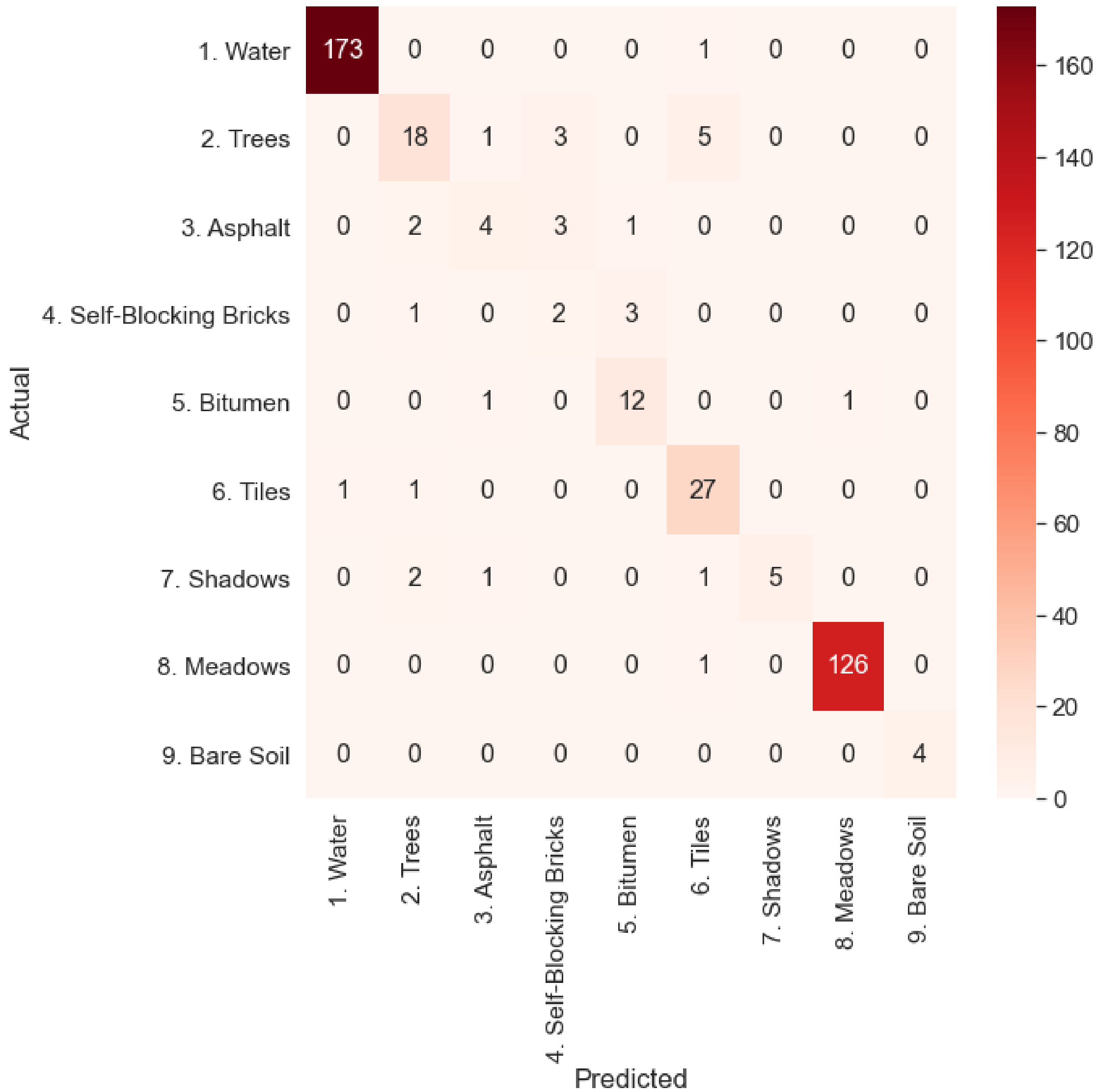
```
<Figure size 432x288 with 0 Axes>
```

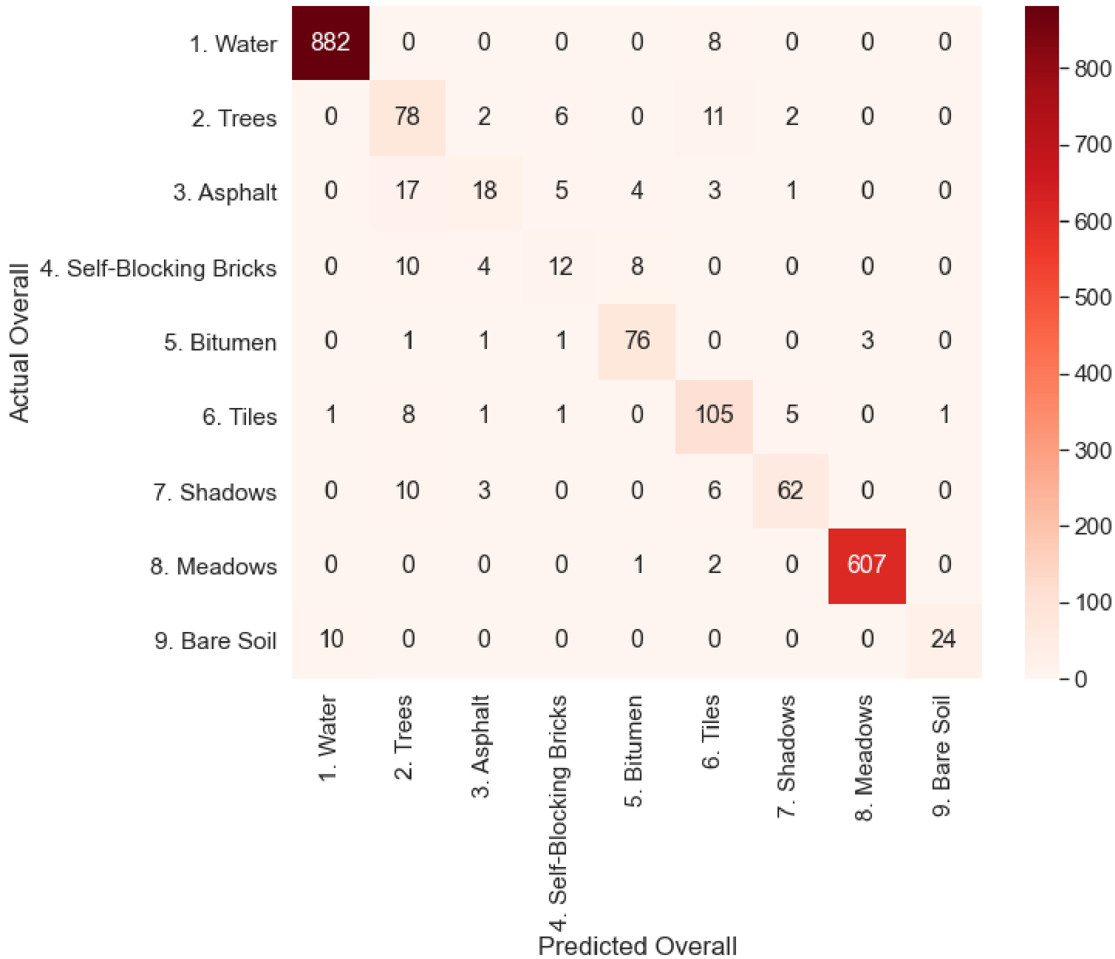





```
Score for fold 5: loss of 0.1966128796339035; accuracy of 92.75000095367432%
13/13 [==============================] - 1s 53ms/step
[[173   0   0   0   0   1   0   0   0]
 [  0  18   1   3   0   5   0   0   0]
 [  0   2   4   3   1   0   0   0   0]
 [  0   1   0   2   3   0   0   0   0]
 [  0   0   1   0  12   0   0   1   0]
 [  1   1   0   0   0  27   0   0   0]
 [  0   2   1   0   0   1   5   0   0]
 [  0   0   0   0   0   1   0 126   0]
 [  0   0   0   0   0   0   0   0   4]]
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.23945873975753784 - Accuracy: 92.5000011920929%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.19002245366573334 - Accuracy: 93.75%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.18330445885658264 - Accuracy: 93.99999976158142%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.19887861609458923 - Accuracy: 93.00000071525574%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.1966128796339035 - Accuracy: 92.75000095367432%
```

```
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 93.20000052452087 (+- 0.578791293160594)
> Loss: 0.20165542960166932
------------------------------------------------------------------------
Predicted Overall        1. Water  2. Trees  3. Asphalt  \
Actual Overall
1. Water                      882         0           0
2. Trees                        0        78           2
3. Asphalt                      0        17          18
4. Self-Blocking Bricks         0        10           4
5. Bitumen                      0         1           1
6. Tiles                        1         8           1
7. Shadows                      0        10           3
8. Meadows                      0         0           0
9. Bare Soil                   10         0           0

Predicted Overall        4. Self-Blocking Bricks  5. Bitumen  6. Tiles  \
Actual Overall
1. Water                                       0           0         8
2. Trees                                       6           0        11
3. Asphalt                                     5           4         3
4. Self-Blocking Bricks                       12           8         0
5. Bitumen                                     1          76         0
6. Tiles                                       1           0       105
7. Shadows                                     0           0         6
8. Meadows                                     0           1         2
9. Bare Soil                                   0           0         0

Predicted Overall        7. Shadows  8. Meadows  9. Bare Soil
Actual Overall
1. Water                          0           0             0
2. Trees                          2           0             0
3. Asphalt                        1           0             0
4. Self-Blocking Bricks           0           0             0
5. Bitumen                        0           3             0
6. Tiles                          5           0             1
7. Shadows                       62           0             0
8. Meadows                        0         607             0
9. Bare Soil                      0           0            24

<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
173 0 0 0 0 1 0 0 0
0 18 1 3 0 5 0 0 0
0 2 4 3 1 0 0 0 0
0 1 0 2 3 0 0 0 0
0 0 1 0 12 0 0 1 0
1 1 0 0 0 27 0 0 0
0 2 1 0 0 1 5 0 0
0 0 0 0 0 1 0 126 0
0 0 0 0 0 0 0 0 4
160
140
120
100
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
882 0 0 0 0 8 0 0 0
0 78 2 6 0 11 2 0 0
0 17 18 5 4 3 1 0 0
0 10 4 12 8 0 0 0 0
0 1 1 1 76 0 0 3 0
1 8 1 1 0 105 5 0 1
0 10 3 0 0 6 62 0 0
0 0 0 0 1 2 0 607 0
10 0 0 0 0 0 0 0 24
800
700
600
500
400
300
200
100
0