# 15 X_ResNet_Center

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_
```

[ ]:
```
# 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=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'])
```

[ ]:
```
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
```

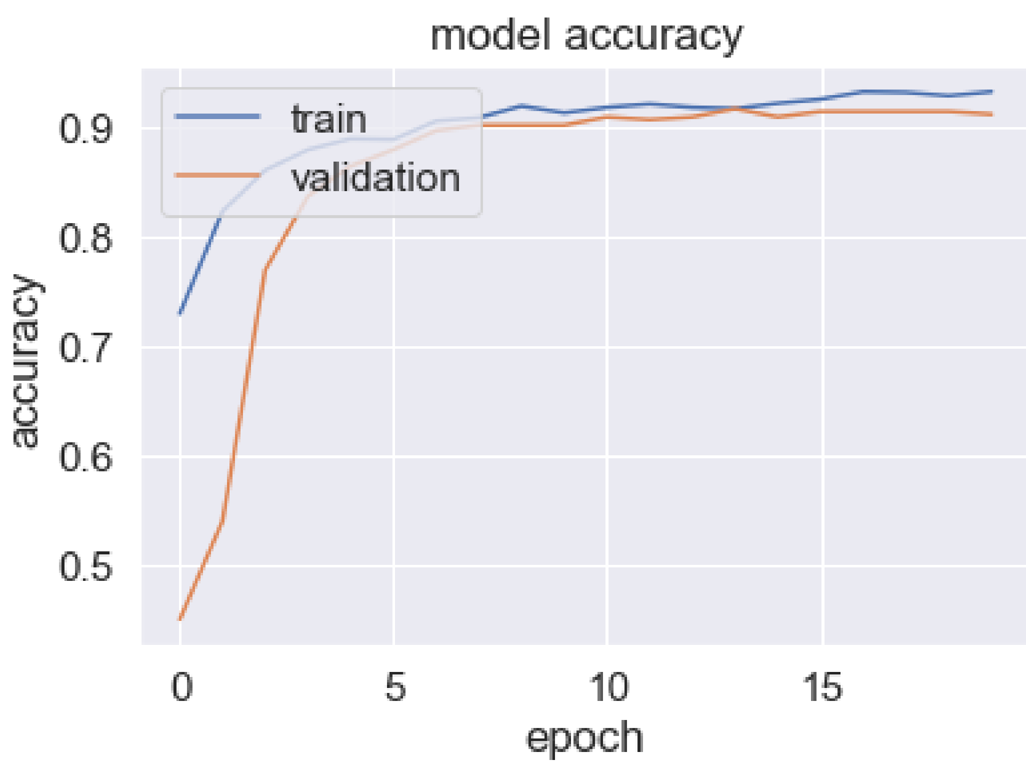
```
 # Compile the model
 #model.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
↪metrics=['accuracy'])

  # 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 - 5s - loss: 0.9231 - accuracy: 0.7300 - val_loss: 1.8590 - val_accuracy:
0.4500 - 5s/epoch - 100ms/step
Epoch 2/20
50/50 - 2s - loss: 0.5518 - accuracy: 0.8238 - val_loss: 1.1576 - val_accuracy:
0.5400 - 2s/epoch - 46ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4373 - accuracy: 0.8612 - val_loss: 0.7115 - val_accuracy:
0.7700 - 2s/epoch - 45ms/step
Epoch 4/20
50/50 - 2s - loss: 0.4033 - accuracy: 0.8800 - val_loss: 0.5136 - val_accuracy:
0.8375 - 2s/epoch - 46ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3586 - accuracy: 0.8900 - val_loss: 0.4055 - val_accuracy:
0.8650 - 2s/epoch - 45ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3448 - accuracy: 0.8894 - val_loss: 0.3427 - val_accuracy:
0.8800 - 2s/epoch - 47ms/step
Epoch 7/20
50/50 - 3s - loss: 0.3050 - accuracy: 0.9062 - val_loss: 0.3037 - val_accuracy:
0.8975 - 3s/epoch - 55ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2936 - accuracy: 0.9094 - val_loss: 0.2829 - val_accuracy:
0.9025 - 2s/epoch - 50ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2804 - accuracy: 0.9200 - val_loss: 0.2679 - val_accuracy:
0.9025 - 2s/epoch - 49ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2711 - accuracy: 0.9137 - val_loss: 0.2667 - val_accuracy:
0.9025 - 2s/epoch - 47ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2689 - accuracy: 0.9187 - val_loss: 0.2505 - val_accuracy:
0.9100 - 2s/epoch - 47ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2642 - accuracy: 0.9219 - val_loss: 0.2503 - val_accuracy:
0.9075 - 2s/epoch - 46ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2556 - accuracy: 0.9187 - val_loss: 0.2441 - val_accuracy:
0.9100 - 2s/epoch - 48ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2532 - accuracy: 0.9175 - val_loss: 0.2390 - val_accuracy:
0.9175 - 2s/epoch - 48ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2339 - accuracy: 0.9225 - val_loss: 0.2311 - val_accuracy:
```

```
0.9100 - 2s/epoch - 49ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2356 - accuracy: 0.9262 - val_loss: 0.2336 - val_accuracy:
0.9150 - 2s/epoch - 47ms/step
Epoch 17/20
50/50 - 3s - loss: 0.2271 - accuracy: 0.9331 - val_loss: 0.2195 - val_accuracy:
0.9150 - 3s/epoch - 51ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2175 - accuracy: 0.9325 - val_loss: 0.2163 - val_accuracy:
0.9150 - 2s/epoch - 49ms/step
Epoch 19/20
50/50 - 3s - loss: 0.2292 - accuracy: 0.9294 - val_loss: 0.2216 - val_accuracy:
0.9150 - 3s/epoch - 51ms/step
Epoch 20/20
50/50 - 2s - loss: 0.2081 - accuracy: 0.9331 - val_loss: 0.2225 - val_accuracy:
0.9125 - 2s/epoch - 49ms/step
```

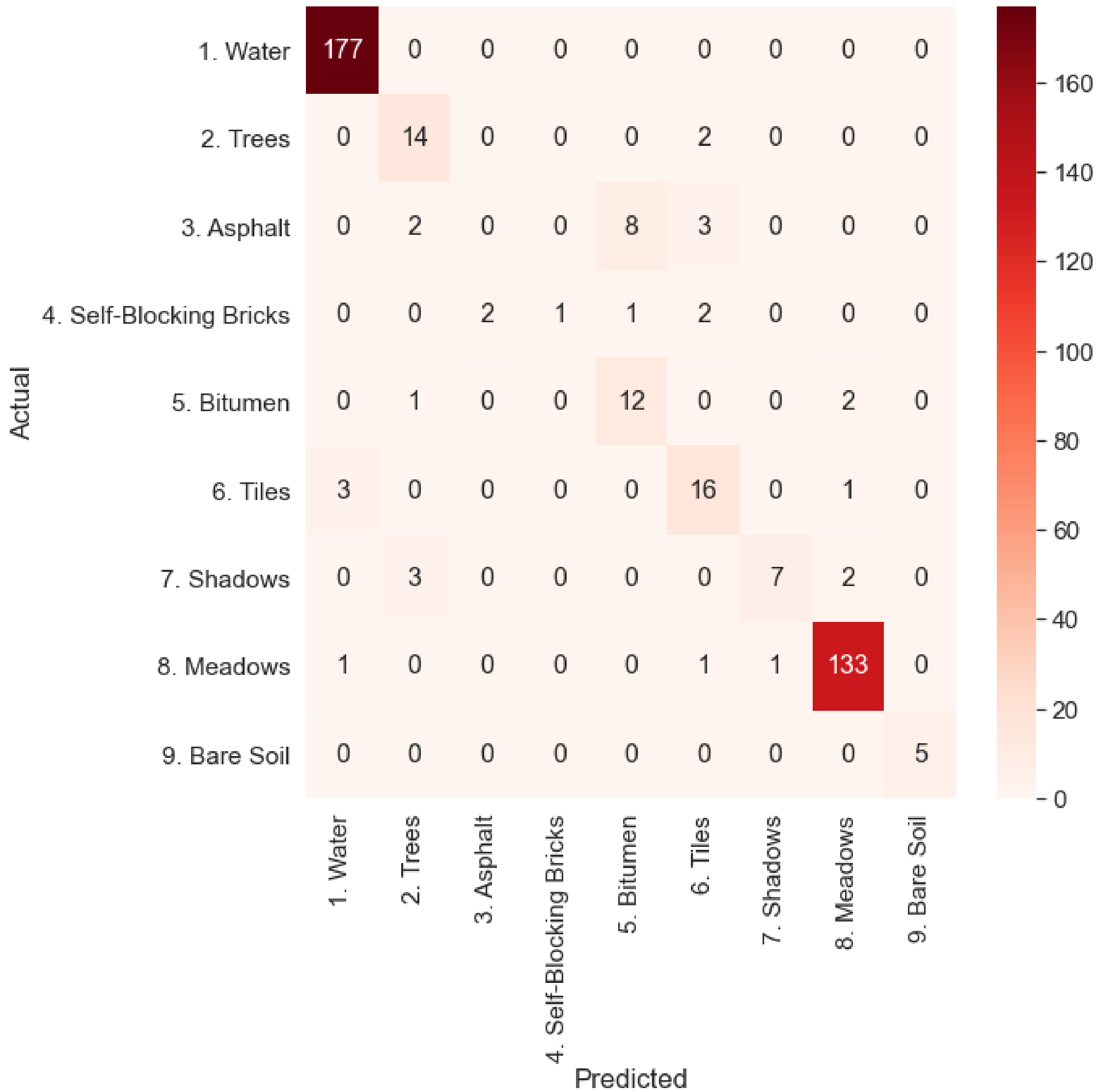



```
Score for fold 1: loss of 0.22247274219989777; accuracy of 91.25000238418579%
13/13 [==============================] - 1s 37ms/step
[[177   0   0   0   0   0   0   0   0]
 [  0  14   0   0   0   2   0   0   0]
 [  0   2   0   0   8   3   0   0   0]
 [  0   0   2   1   1   2   0   0   0]
```

```
 [  0   1   0   0  12   0   0   2   0]
 [  3   0   0   0   0  16   0   1   0]
 [  0   3   0   0   0   0   7   2   0]
 [  1   0   0   0   0   1   1 133   0]
 [  0   0   0   0   0   0   0   0   5]]
------------------------------------------------------------------------
Training for fold 2 …
Epoch 1/20
50/50 - 5s - loss: 0.9267 - accuracy: 0.7319 - val_loss: 1.9854 - val_accuracy:
0.4475 - 5s/epoch - 107ms/step
Epoch 2/20
50/50 - 2s - loss: 0.5280 - accuracy: 0.8356 - val_loss: 1.2629 - val_accuracy:
0.5350 - 2s/epoch - 42ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4511 - accuracy: 0.8581 - val_loss: 0.8076 - val_accuracy:
0.7375 - 2s/epoch - 42ms/step
Epoch 4/20
50/50 - 2s - loss: 0.3812 - accuracy: 0.8869 - val_loss: 0.5732 - val_accuracy:
0.8100 - 2s/epoch - 40ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3533 - accuracy: 0.8881 - val_loss: 0.4428 - val_accuracy:
0.8625 - 2s/epoch - 40ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3341 - accuracy: 0.8950 - val_loss: 0.3907 - val_accuracy:
0.8850 - 2s/epoch - 39ms/step
Epoch 7/20
50/50 - 2s - loss: 0.3062 - accuracy: 0.9137 - val_loss: 0.3550 - val_accuracy:
0.9000 - 2s/epoch - 37ms/step
Epoch 8/20
50/50 - 2s - loss: 0.2934 - accuracy: 0.9094 - val_loss: 0.3314 - val_accuracy:
0.9150 - 2s/epoch - 37ms/step
Epoch 9/20
50/50 - 2s - loss: 0.2805 - accuracy: 0.9062 - val_loss: 0.3199 - val_accuracy:
0.9200 - 2s/epoch - 38ms/step
Epoch 10/20
50/50 - 2s - loss: 0.2825 - accuracy: 0.9119 - val_loss: 0.3134 - val_accuracy:
0.9200 - 2s/epoch - 44ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2653 - accuracy: 0.9187 - val_loss: 0.3118 - val_accuracy:
0.9025 - 2s/epoch - 43ms/step
Epoch 12/20
50/50 - 3s - loss: 0.2477 - accuracy: 0.9312 - val_loss: 0.3099 - val_accuracy:
0.9100 - 3s/epoch - 50ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2510 - accuracy: 0.9256 - val_loss: 0.2972 - val_accuracy:
0.9175 - 2s/epoch - 47ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2394 - accuracy: 0.9262 - val_loss: 0.2928 - val_accuracy:
```

```
0.9200 - 2s/epoch - 45ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2417 - accuracy: 0.9225 - val_loss: 0.2817 - val_accuracy:
0.9300 - 2s/epoch - 49ms/step
Epoch 16/20
50/50 - 3s - loss: 0.2281 - accuracy: 0.9275 - val_loss: 0.2742 - val_accuracy:
0.9325 - 3s/epoch - 56ms/step
Epoch 17/20
50/50 - 2s - loss: 0.2197 - accuracy: 0.9337 - val_loss: 0.2770 - val_accuracy:
0.9300 - 2s/epoch - 48ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2266 - accuracy: 0.9369 - val_loss: 0.2807 - val_accuracy:
0.9200 - 2s/epoch - 42ms/step
Epoch 19/20
50/50 - 2s - loss: 0.2230 - accuracy: 0.9319 - val_loss: 0.2693 - val_accuracy:
0.9250 - 2s/epoch - 41ms/step
Epoch 20/20
50/50 - 2s - loss: 0.2011 - accuracy: 0.9312 - val_loss: 0.2713 - val_accuracy:
0.9250 - 2s/epoch - 42ms/step
```

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



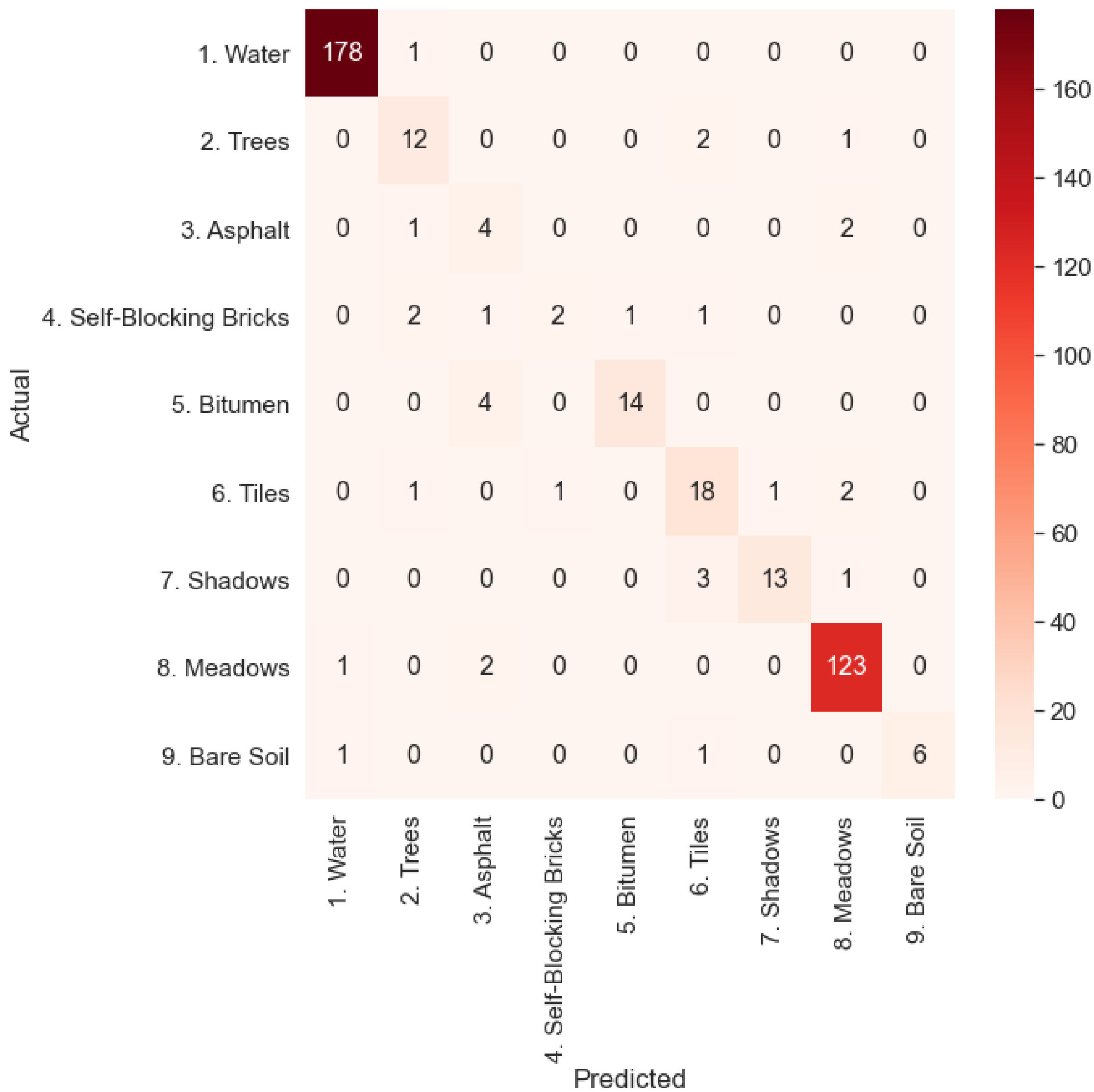
Actual
Predicted
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
177 0 0 0 0 0 0 0 0
0 14 0 0 0 2 0 0 0
0 2 0 0 8 3 0 0 0
0 0 2 1 1 2 0 0 0
0 1 0 0 12 0 0 2 0
3 0 0 0 0 16 0 1 0
0 3 0 0 0 0 7 2 0
1 0 0 0 0 1 1 133 0
0 0 0 0 0 0 0 0 5
160
140
120
100
80
60
40
20
0

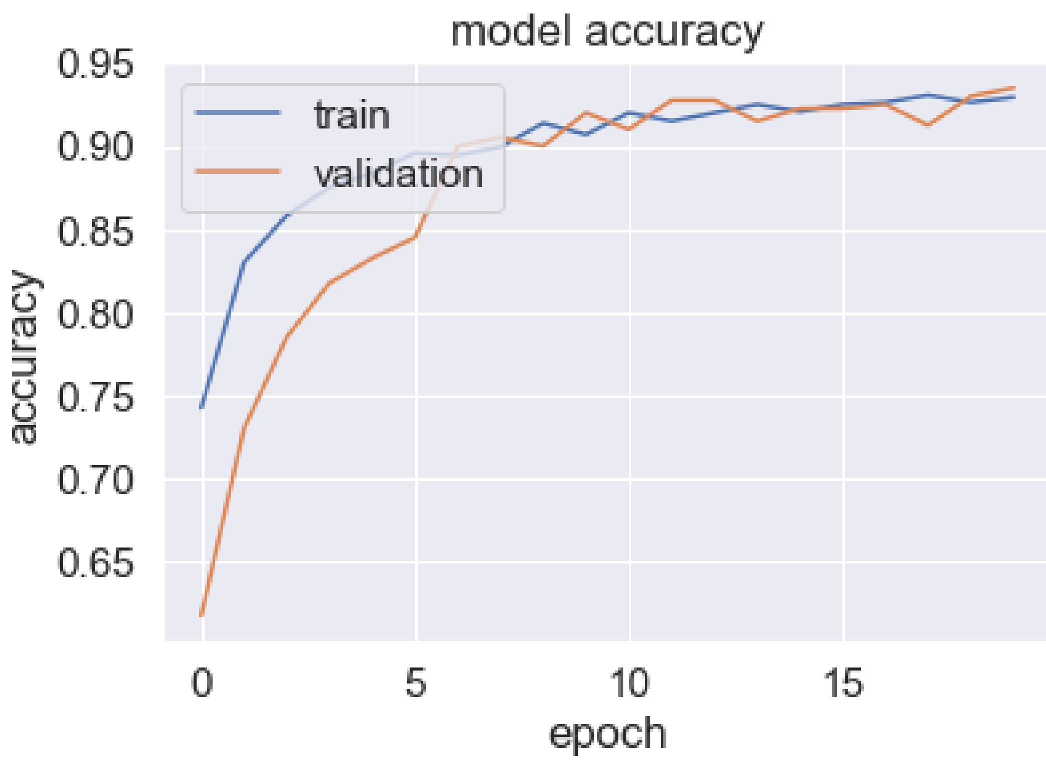

```
Score for fold 2: loss of 0.27128857374191284; accuracy of 92.5000011920929%
13/13 [==============================] - 1s 32ms/step
[[178   1   0   0   0   0   0   0   0]
 [  0  12   0   0   0   2   0   1   0]
 [  0   1   4   0   0   0   0   2   0]
 [  0   2   1   2   1   1   0   0   0]
 [  0   0   4   0  14   0   0   0   0]
 [  0   1   0   1   0  18   1   2   0]
 [  0   0   0   0   0   3  13   1   0]
 [  1   0   2   0   0   0   0 123   0]
 [  1   0   0   0   0   1   0   0   6]]
------------------------------------------------------------------------
Training for fold 3 …
Epoch 1/20
50/50 - 6s - loss: 0.9011 - accuracy: 0.7425 - val_loss: 1.1629 - val_accuracy:
0.6175 - 6s/epoch - 112ms/step
Epoch 2/20
50/50 - 3s - loss: 0.5359 - accuracy: 0.8300 - val_loss: 0.8691 - val_accuracy:
0.7300 - 3s/epoch - 53ms/step
Epoch 3/20
50/50 - 3s - loss: 0.4384 - accuracy: 0.8581 - val_loss: 0.6554 - val_accuracy:
0.7850 - 3s/epoch - 55ms/step
Epoch 4/20
```

```
50/50 - 3s - loss: 0.3893 - accuracy: 0.8750 - val_loss: 0.5072 - val_accuracy:
0.8175 - 3s/epoch - 51ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3608 - accuracy: 0.8838 - val_loss: 0.4290 - val_accuracy:
0.8325 - 2s/epoch - 50ms/step
Epoch 6/20
50/50 - 3s - loss: 0.3229 - accuracy: 0.8956 - val_loss: 0.3941 - val_accuracy:
0.8450 - 3s/epoch - 52ms/step
Epoch 7/20
50/50 - 3s - loss: 0.3285 - accuracy: 0.8944 - val_loss: 0.3403 - val_accuracy:
0.9000 - 3s/epoch - 52ms/step
Epoch 8/20
50/50 - 3s - loss: 0.3042 - accuracy: 0.8994 - val_loss: 0.3175 - val_accuracy:
0.9050 - 3s/epoch - 51ms/step
Epoch 9/20
50/50 - 3s - loss: 0.2882 - accuracy: 0.9137 - val_loss: 0.3043 - val_accuracy:
0.9000 - 3s/epoch - 51ms/step
Epoch 10/20
50/50 - 3s - loss: 0.2789 - accuracy: 0.9069 - val_loss: 0.2783 - val_accuracy:
0.9200 - 3s/epoch - 53ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2638 - accuracy: 0.9200 - val_loss: 0.2758 - val_accuracy:
0.9100 - 2s/epoch - 50ms/step
Epoch 12/20
50/50 - 3s - loss: 0.2633 - accuracy: 0.9150 - val_loss: 0.2757 - val_accuracy:
0.9275 - 3s/epoch - 50ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2516 - accuracy: 0.9200 - val_loss: 0.2568 - val_accuracy:
0.9275 - 2s/epoch - 49ms/step
Epoch 14/20
50/50 - 2s - loss: 0.2411 - accuracy: 0.9250 - val_loss: 0.2589 - val_accuracy:
0.9150 - 2s/epoch - 49ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2452 - accuracy: 0.9206 - val_loss: 0.2548 - val_accuracy:
0.9225 - 2s/epoch - 50ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2359 - accuracy: 0.9250 - val_loss: 0.2496 - val_accuracy:
0.9225 - 2s/epoch - 49ms/step
Epoch 17/20
50/50 - 2s - loss: 0.2219 - accuracy: 0.9262 - val_loss: 0.2454 - val_accuracy:
0.9250 - 2s/epoch - 50ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2249 - accuracy: 0.9306 - val_loss: 0.2413 - val_accuracy:
0.9125 - 2s/epoch - 44ms/step
Epoch 19/20
50/50 - 2s - loss: 0.2141 - accuracy: 0.9262 - val_loss: 0.2389 - val_accuracy:
0.9300 - 2s/epoch - 43ms/step
Epoch 20/20
```

```
50/50 - 2s - loss: 0.2140 - accuracy: 0.9294 - val_loss: 0.2245 - val_accuracy:
0.9350 - 2s/epoch - 44ms/step
```

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



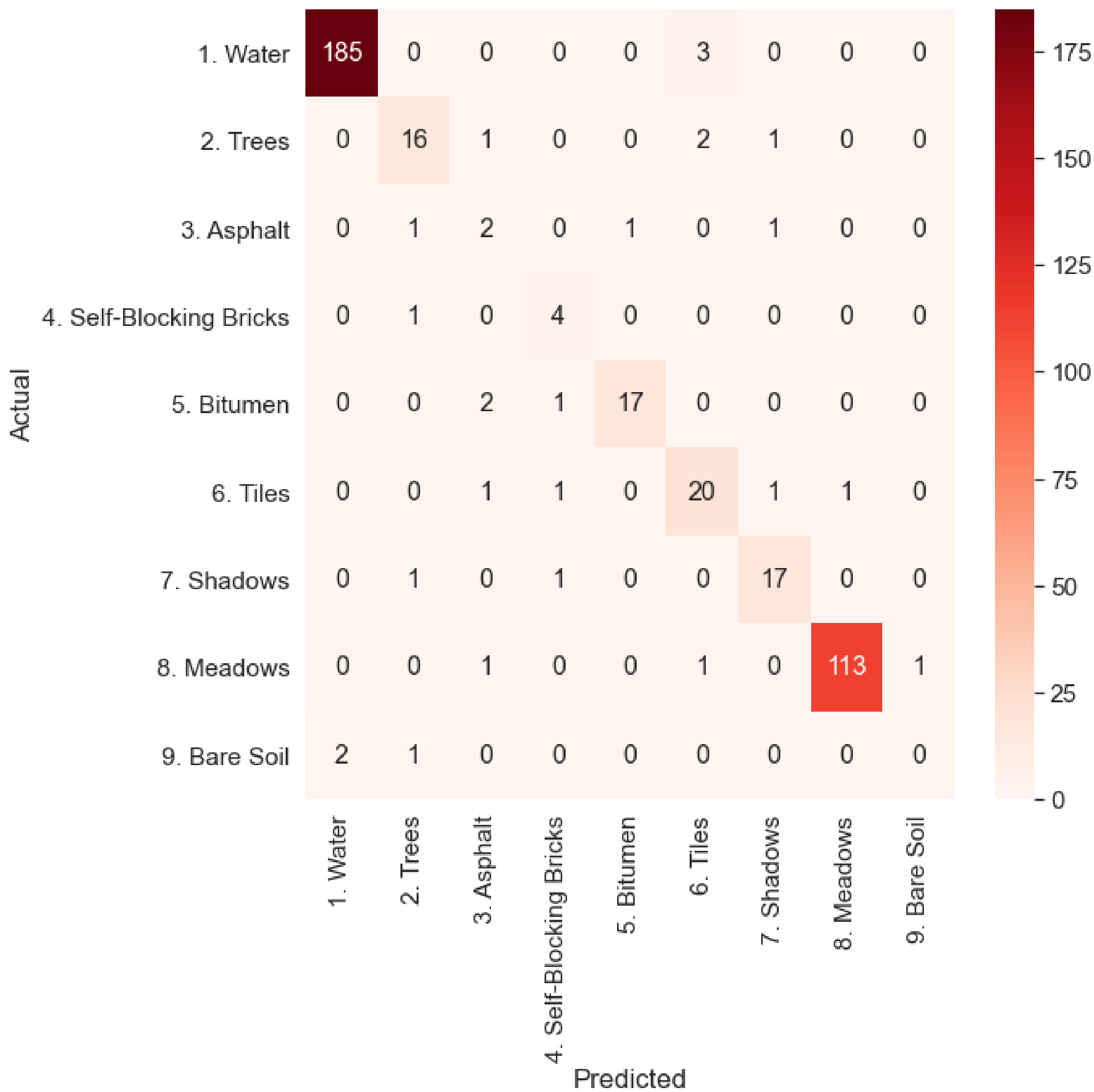


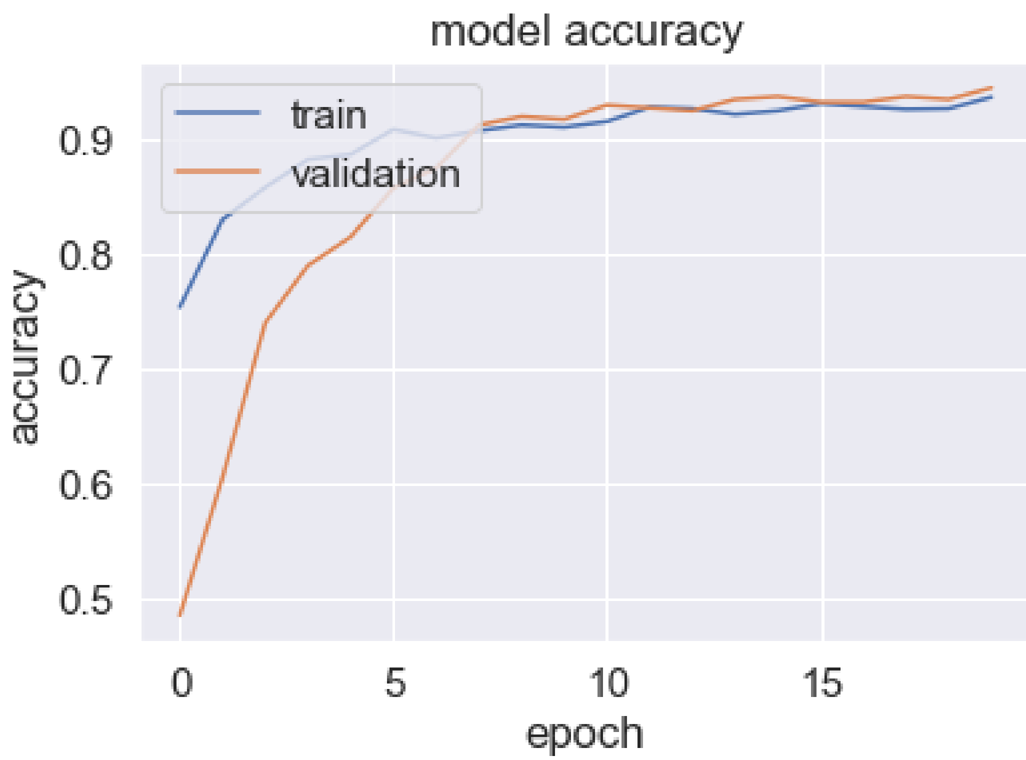

```
Score for fold 3: loss of 0.2244652956724167; accuracy of 93.50000023841858%
13/13 [==============================] - 1s 41ms/step
[[185   0   0   0   0   3   0   0   0]
 [  0  16   1   0   0   2   1   0   0]
 [  0   1   2   0   1   0   1   0   0]
 [  0   1   0   4   0   0   0   0   0]
 [  0   0   2   1  17   0   0   0   0]
 [  0   0   1   1   0  20   1   1   0]
 [  0   1   0   1   0   0  17   0   0]
 [  0   0   1   0   0   1   0 113   1]
 [  2   1   0   0   0   0   0   0   0]]
------------------------------------------------------------------------
Training for fold 4 …
Epoch 1/20
50/50 - 5s - loss: 0.9038 - accuracy: 0.7538 - val_loss: 1.5609 - val_accuracy:
0.4850 - 5s/epoch - 108ms/step
Epoch 2/20
50/50 - 3s - loss: 0.5263 - accuracy: 0.8300 - val_loss: 1.1289 - val_accuracy:
0.6050 - 3s/epoch - 54ms/step
Epoch 3/20
50/50 - 3s - loss: 0.4375 - accuracy: 0.8581 - val_loss: 0.7634 - val_accuracy:
0.7400 - 3s/epoch - 54ms/step
Epoch 4/20
```

```
50/50 - 3s - loss: 0.3932 - accuracy: 0.8825 - val_loss: 0.5843 - val_accuracy:
0.7900 - 3s/epoch - 54ms/step
Epoch 5/20
50/50 - 3s - loss: 0.3568 - accuracy: 0.8869 - val_loss: 0.4881 - val_accuracy:
0.8150 - 3s/epoch - 54ms/step
Epoch 6/20
50/50 - 3s - loss: 0.3227 - accuracy: 0.9087 - val_loss: 0.4214 - val_accuracy:
0.8575 - 3s/epoch - 54ms/step
Epoch 7/20
50/50 - 3s - loss: 0.3230 - accuracy: 0.9013 - val_loss: 0.3922 - val_accuracy:
0.8750 - 3s/epoch - 54ms/step
Epoch 8/20
50/50 - 3s - loss: 0.2940 - accuracy: 0.9075 - val_loss: 0.3656 - val_accuracy:
0.9125 - 3s/epoch - 54ms/step
Epoch 9/20
50/50 - 3s - loss: 0.2821 - accuracy: 0.9125 - val_loss: 0.3469 - val_accuracy:
0.9200 - 3s/epoch - 55ms/step
Epoch 10/20
50/50 - 3s - loss: 0.2696 - accuracy: 0.9106 - val_loss: 0.3449 - val_accuracy:
0.9175 - 3s/epoch - 55ms/step
Epoch 11/20
50/50 - 3s - loss: 0.2675 - accuracy: 0.9156 - val_loss: 0.3325 - val_accuracy:
0.9300 - 3s/epoch - 54ms/step
Epoch 12/20
50/50 - 3s - loss: 0.2496 - accuracy: 0.9281 - val_loss: 0.3351 - val_accuracy:
0.9275 - 3s/epoch - 54ms/step
Epoch 13/20
50/50 - 3s - loss: 0.2360 - accuracy: 0.9269 - val_loss: 0.3272 - val_accuracy:
0.9250 - 3s/epoch - 54ms/step
Epoch 14/20
50/50 - 3s - loss: 0.2380 - accuracy: 0.9219 - val_loss: 0.3195 - val_accuracy:
0.9350 - 3s/epoch - 54ms/step
Epoch 15/20
50/50 - 3s - loss: 0.2388 - accuracy: 0.9250 - val_loss: 0.3105 - val_accuracy:
0.9375 - 3s/epoch - 54ms/step
Epoch 16/20
50/50 - 3s - loss: 0.2182 - accuracy: 0.9312 - val_loss: 0.3306 - val_accuracy:
0.9325 - 3s/epoch - 54ms/step
Epoch 17/20
50/50 - 3s - loss: 0.2184 - accuracy: 0.9287 - val_loss: 0.3214 - val_accuracy:
0.9325 - 3s/epoch - 53ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2276 - accuracy: 0.9262 - val_loss: 0.3286 - val_accuracy:
0.9375 - 2s/epoch - 48ms/step
Epoch 19/20
50/50 - 3s - loss: 0.2217 - accuracy: 0.9269 - val_loss: 0.3437 - val_accuracy:
0.9350 - 3s/epoch - 51ms/step
Epoch 20/20
```

```
50/50 - 2s - loss: 0.2154 - accuracy: 0.9369 - val_loss: 0.3497 - val_accuracy:
0.9450 - 2s/epoch - 49ms/step
```

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



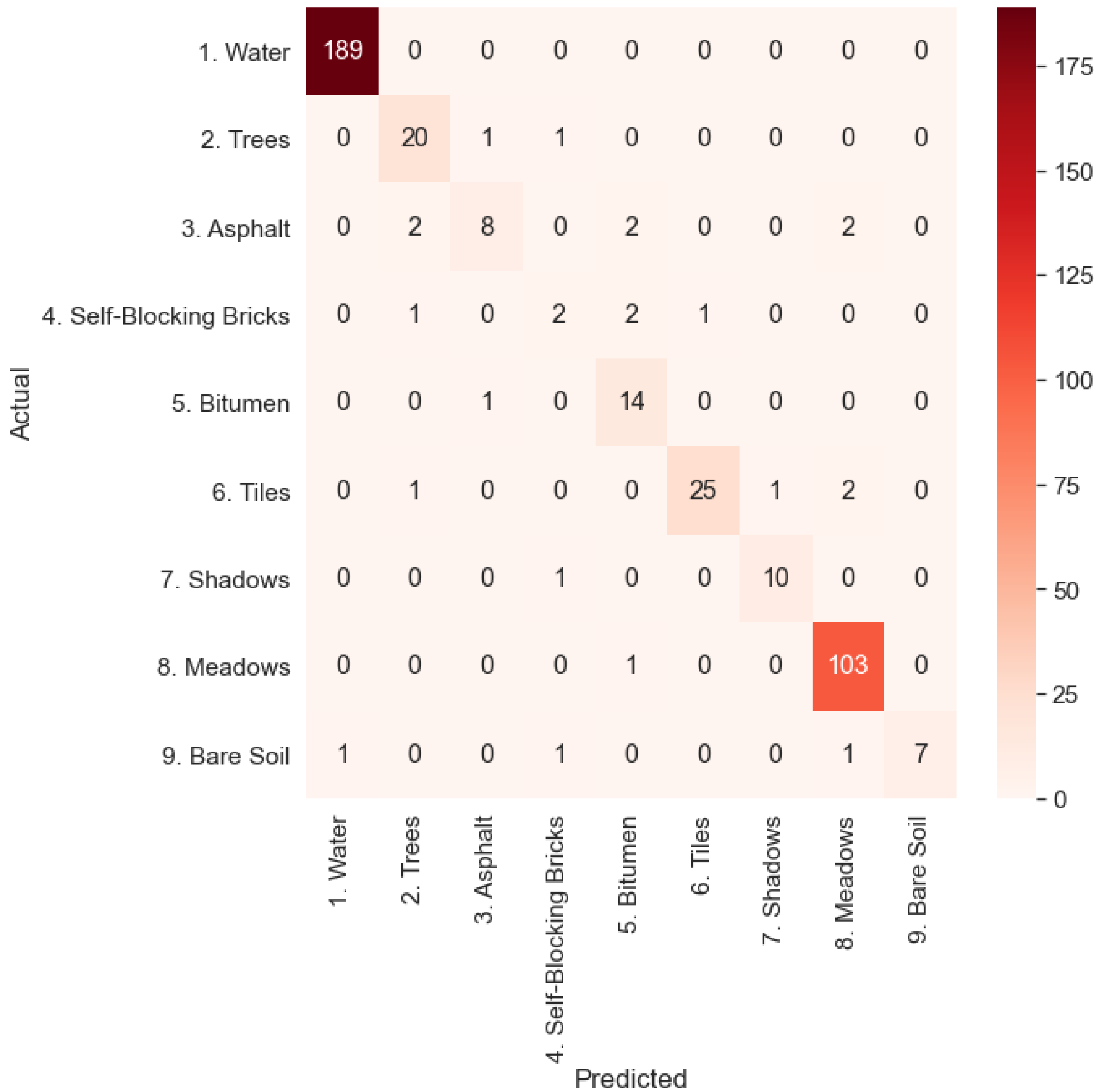


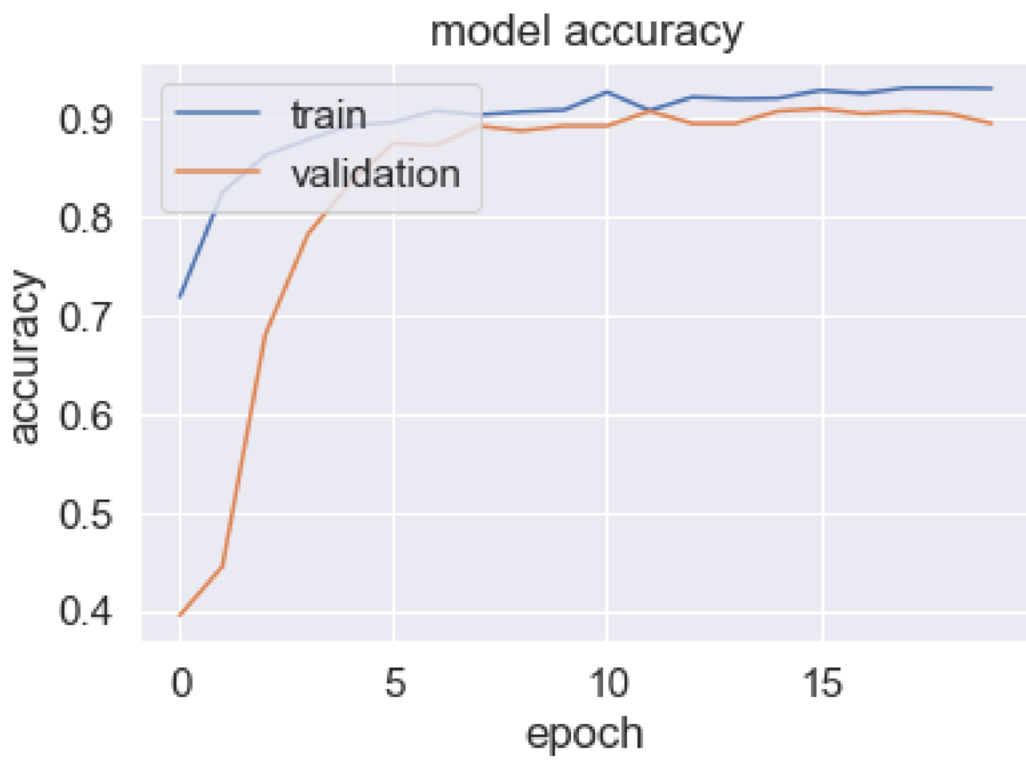

```
Score for fold 4: loss of 0.34966567158699036; accuracy of 94.49999928474426%
13/13 [==============================] - 1s 40ms/step
[[189   0   0   0   0   0   0   0   0]
 [  0  20   1   1   0   0   0   0   0]
 [  0   2   8   0   2   0   0   2   0]
 [  0   1   0   2   2   1   0   0   0]
 [  0   0   1   0  14   0   0   0   0]
 [  0   1   0   0   0  25   1   2   0]
 [  0   0   0   1   0   0  10   0   0]
 [  0   0   0   0   1   0   0 103   0]
 [  1   0   0   1   0   0   0   1   7]]
------------------------------------------------------------------------
Training for fold 5 …
Epoch 1/20
50/50 - 5s - loss: 0.9683 - accuracy: 0.7188 - val_loss: 2.5192 - val_accuracy:
0.3950 - 5s/epoch - 98ms/step
Epoch 2/20
50/50 - 2s - loss: 0.5341 - accuracy: 0.8250 - val_loss: 1.6335 - val_accuracy:
0.4450 - 2s/epoch - 49ms/step
Epoch 3/20
50/50 - 2s - loss: 0.4373 - accuracy: 0.8625 - val_loss: 0.9445 - val_accuracy:
0.6800 - 2s/epoch - 50ms/step
Epoch 4/20
```

```
50/50 - 2s - loss: 0.4019 - accuracy: 0.8788 - val_loss: 0.6524 - val_accuracy:
0.7825 - 2s/epoch - 49ms/step
Epoch 5/20
50/50 - 2s - loss: 0.3514 - accuracy: 0.8931 - val_loss: 0.5086 - val_accuracy:
0.8350 - 2s/epoch - 49ms/step
Epoch 6/20
50/50 - 2s - loss: 0.3277 - accuracy: 0.8963 - val_loss: 0.4275 - val_accuracy:
0.8750 - 2s/epoch - 49ms/step
Epoch 7/20
50/50 - 2s - loss: 0.3116 - accuracy: 0.9081 - val_loss: 0.3820 - val_accuracy:
0.8725 - 2s/epoch - 49ms/step
Epoch 8/20
50/50 - 2s - loss: 0.3078 - accuracy: 0.9038 - val_loss: 0.3372 - val_accuracy:
0.8925 - 2s/epoch - 50ms/step
Epoch 9/20
50/50 - 3s - loss: 0.2950 - accuracy: 0.9069 - val_loss: 0.3129 - val_accuracy:
0.8875 - 3s/epoch - 53ms/step
Epoch 10/20
50/50 - 3s - loss: 0.2866 - accuracy: 0.9087 - val_loss: 0.2989 - val_accuracy:
0.8925 - 3s/epoch - 52ms/step
Epoch 11/20
50/50 - 2s - loss: 0.2574 - accuracy: 0.9269 - val_loss: 0.2896 - val_accuracy:
0.8925 - 2s/epoch - 49ms/step
Epoch 12/20
50/50 - 2s - loss: 0.2634 - accuracy: 0.9081 - val_loss: 0.2740 - val_accuracy:
0.9075 - 2s/epoch - 50ms/step
Epoch 13/20
50/50 - 2s - loss: 0.2528 - accuracy: 0.9219 - val_loss: 0.2637 - val_accuracy:
0.8950 - 2s/epoch - 49ms/step
Epoch 14/20
50/50 - 3s - loss: 0.2474 - accuracy: 0.9200 - val_loss: 0.2509 - val_accuracy:
0.8950 - 3s/epoch - 50ms/step
Epoch 15/20
50/50 - 2s - loss: 0.2401 - accuracy: 0.9206 - val_loss: 0.2476 - val_accuracy:
0.9075 - 2s/epoch - 49ms/step
Epoch 16/20
50/50 - 2s - loss: 0.2377 - accuracy: 0.9287 - val_loss: 0.2451 - val_accuracy:
0.9100 - 2s/epoch - 49ms/step
Epoch 17/20
50/50 - 3s - loss: 0.2279 - accuracy: 0.9256 - val_loss: 0.2426 - val_accuracy:
0.9050 - 3s/epoch - 52ms/step
Epoch 18/20
50/50 - 2s - loss: 0.2270 - accuracy: 0.9312 - val_loss: 0.2359 - val_accuracy:
0.9075 - 2s/epoch - 47ms/step
Epoch 19/20
50/50 - 3s - loss: 0.2176 - accuracy: 0.9312 - val_loss: 0.2419 - val_accuracy:
0.9050 - 3s/epoch - 54ms/step
Epoch 20/20
```

```
50/50 - 3s - loss: 0.2181 - accuracy: 0.9306 - val_loss: 0.2424 - val_accuracy: 0.8950 - 3s/epoch - 52ms/step
```

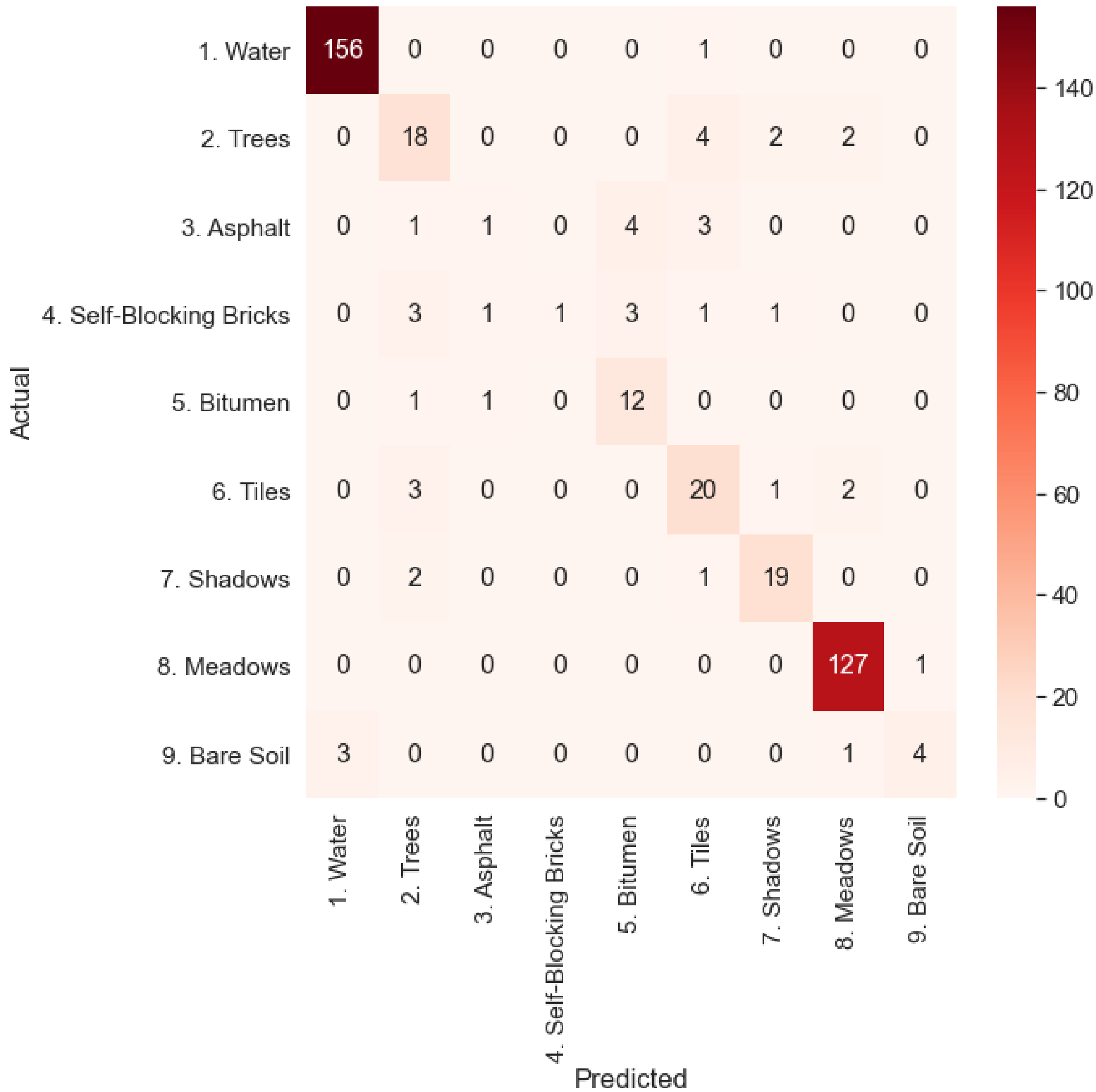
<Figure size 432x288 with 0 Axes>

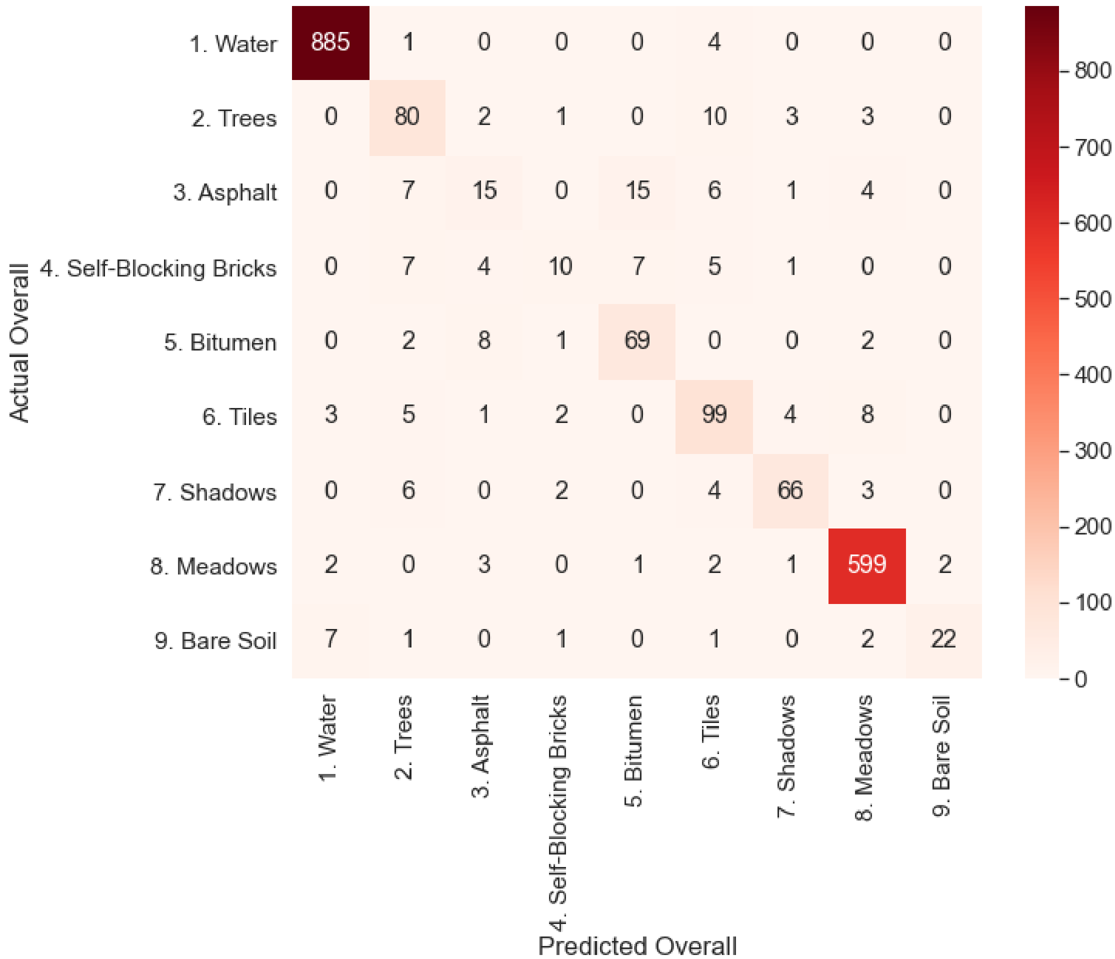





```
Score for fold 5: loss of 0.2423560470342636; accuracy of 89.49999809265137%
13/13 [==============================] - 1s 38ms/step
[[156   0   0   0   0   1   0   0   0]
 [  0  18   0   0   0   4   2   2   0]
 [  0   1   1   0   4   3   0   0   0]
 [  0   3   1   1   3   1   1   0   0]
 [  0   1   1   0  12   0   0   0   0]
 [  0   3   0   0   0  20   1   2   0]
 [  0   2   0   0   0   1  19   0   0]
 [  0   0   0   0   0   0   0 127   1]
 [  3   0   0   0   0   0   0   1   4]]
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.22247274219989777 - Accuracy: 91.25000238418579%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.27128857374191284 - Accuracy: 92.5000011920929%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.2244652956724167 - Accuracy: 93.50000023841858%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.34966567158699036 - Accuracy: 94.49999928474426%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.2423560470342636 - Accuracy: 89.49999809265137%
```

```
-----------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 92.25000023841858 (+- 1.7464251312609427)
> Loss: 0.26204966604709623
-----------------------------------------------------------------------------
Predicted Overall        1. Water  2. Trees  3. Asphalt  \
Actual Overall
1. Water                      885         1           0
2. Trees                        0        80           2
3. Asphalt                      0         7          15
4. Self-Blocking Bricks         0         7           4
5. Bitumen                      0         2           8
6. Tiles                        3         5           1
7. Shadows                      0         6           0
8. Meadows                      2         0           3
9. Bare Soil                    7         1           0

Predicted Overall        4. Self-Blocking Bricks  5. Bitumen  6. Tiles  \
Actual Overall
1. Water                                       0           0         4
2. Trees                                       1           0        10
3. Asphalt                                     0          15         6
4. Self-Blocking Bricks                       10           7         5
5. Bitumen                                     1          69         0
6. Tiles                                       2           0        99
7. Shadows                                     2           0         4
8. Meadows                                     0           1         2
9. Bare Soil                                   1           0         1

Predicted Overall        7. Shadows  8. Meadows  9. Bare Soil
Actual Overall
1. Water                          0           0             0
2. Trees                          3           3             0
3. Asphalt                        1           4             0
4. Self-Blocking Bricks           1           0             0
5. Bitumen                        0           2             0
6. Tiles                          4           8             0
7. Shadows                       66           3             0
8. Meadows                        1         599             2
9. Bare Soil                      0           2            22

<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
156 0 0 0 0 1 0 0 0
0 18 0 0 0 4 2 2 0
0 1 1 0 4 3 0 0 0
0 3 1 1 3 1 1 0 0
0 1 1 0 12 0 0 0 0
0 3 0 0 0 20 1 2 0
0 2 0 0 0 1 19 0 0
0 0 0 0 0 0 0 127 1
3 0 0 0 0 0 0 1 4
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
885 1 0 0 0 4 0 0 0
0 80 2 1 0 10 3 3 0
0 7 15 0 15 6 1 4 0
0 7 4 10 7 5 1 0 0
0 2 8 1 69 0 0 2 0
3 5 1 2 0 99 4 8 0
0 6 0 2 0 4 66 3 0
2 0 3 0 1 2 1 599 2
7 1 0 1 0 1 0 2 22
800
700
600
500
400
300
200
100
0