Full_4 X_Xception_centr

June 13, 2023

1 Date: 9 2022

2 Method: Cross_Inception

3 Data: Pavia

4 Results v.05

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

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

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

X, y = read_HSI()
```

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

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

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

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

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

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

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

[ ]:
```
test_ratio = 0.5

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

windowSize=15  # accuracy of
# Score for fold 1: loss of 0.34631192684173584; accuracy of 89.49999809265137%

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

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

input_image_size=windowSize
INPUT_IMG_SIZE=windowSize

dimReduction=3

InputShape=(windowSize, windowSize, dimReduction)

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

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

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

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

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

```
[ ]: 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
     from tensorflow.keras import layers
     from keras.layers import Dense, GlobalAveragePooling2D, Dropout, Flatten
     import tensorflow as tf

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


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

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


         # Freeze the pretrained weights
         model.trainable = False

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

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

    x = model.output


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


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

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

[ ]:

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

[ ]:

```
from tensorflow.keras.applications import EfficientNetB0
```

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


    # Freeze the pretrained weights
   #model.trainable = False

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

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

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

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

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

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

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


def plot_hist(hist):
```

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

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

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

import tensorflow as tf

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

NN=len(Xtrain)
#NN=2000
NN=5000


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

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

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

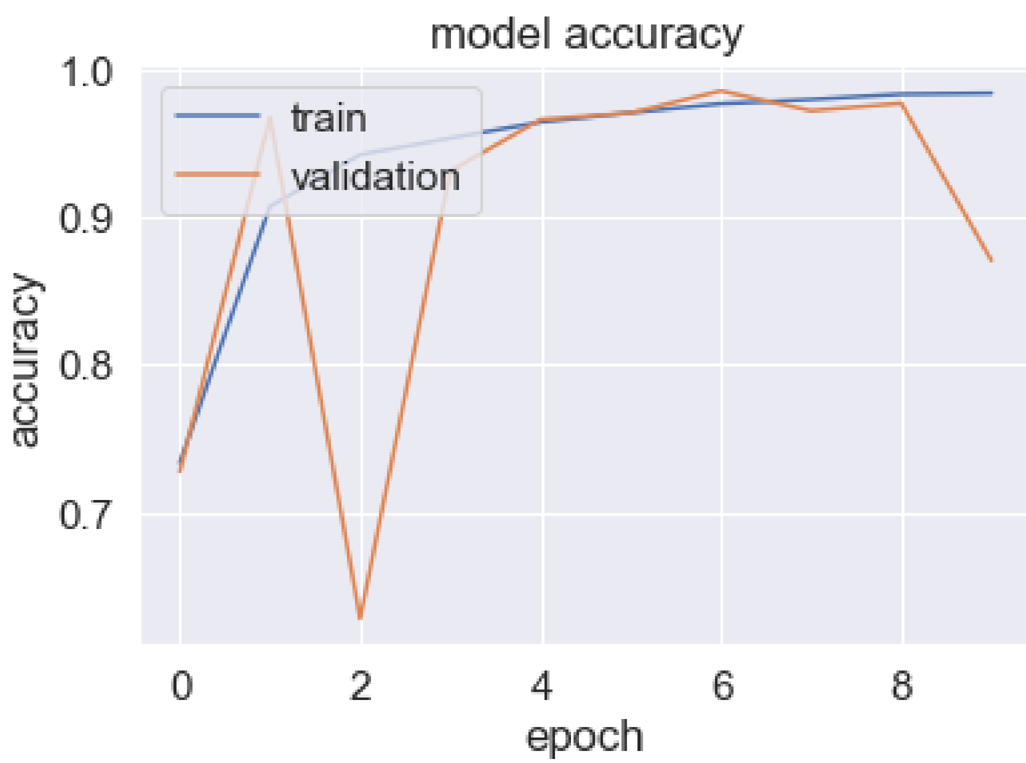
```
# Parse numbers as floats
#input_train = input_train.astype('float32')
#input_test = input_test.astype('float32')

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

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

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

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

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

  #  model architecture

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

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

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

  #model.summary()

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

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

 # Fit data to model
 #model.summary()

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



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

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

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

```
    confmat    = confmat + confusion;


    # Increase fold number
    fold_no = fold_no + 1

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

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


# Notes for next trial

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

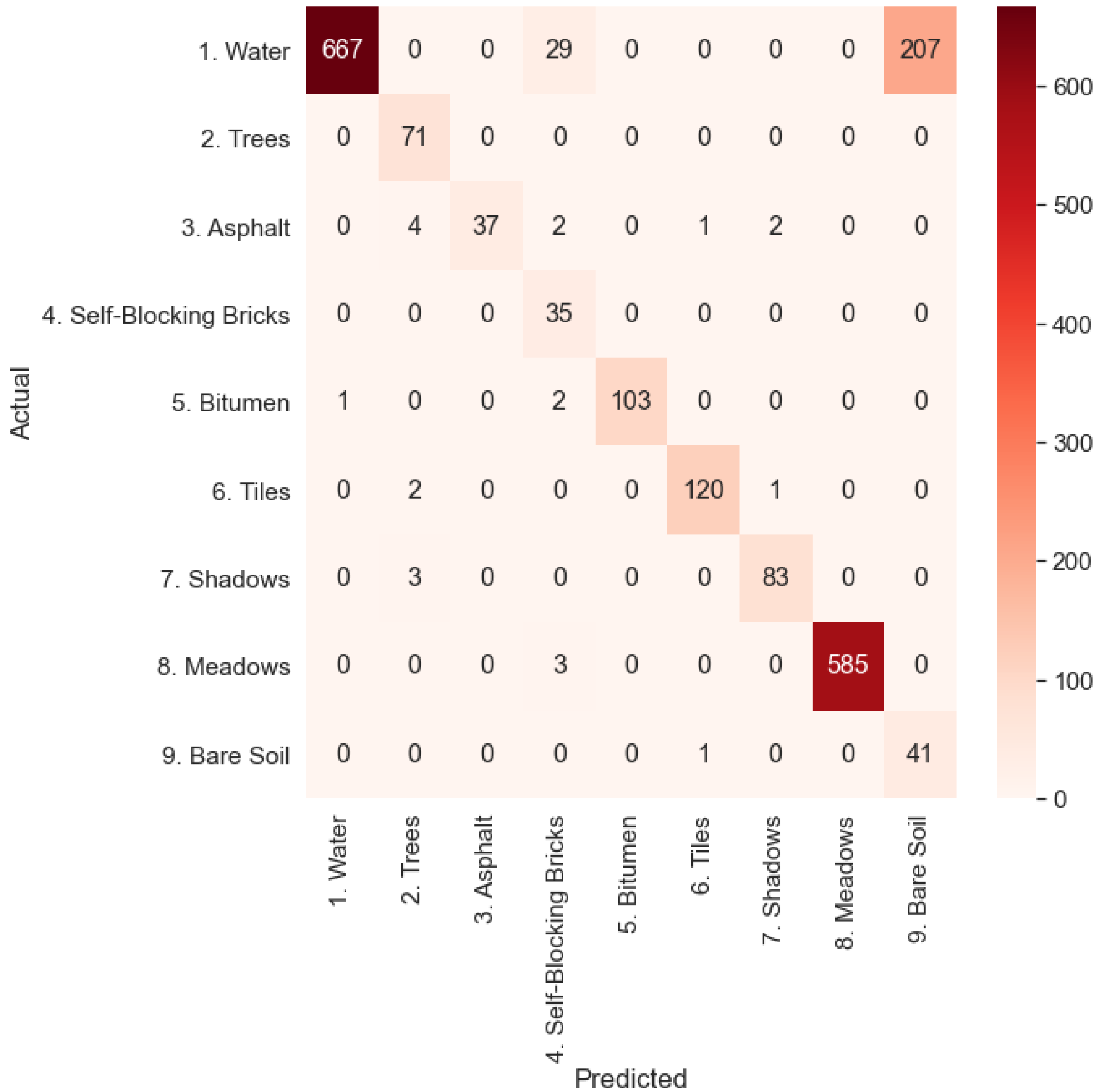
```
------------------------------------------------------------------------
Training for fold 1 …
Epoch 1/10
250/250 - 207s - loss: 0.8361 - accuracy: 0.7336 - val_loss: 0.9933 -
val_accuracy: 0.7275 - 207s/epoch - 830ms/step
Epoch 2/10
250/250 - 201s - loss: 0.3387 - accuracy: 0.9075 - val_loss: 0.1749 -
```

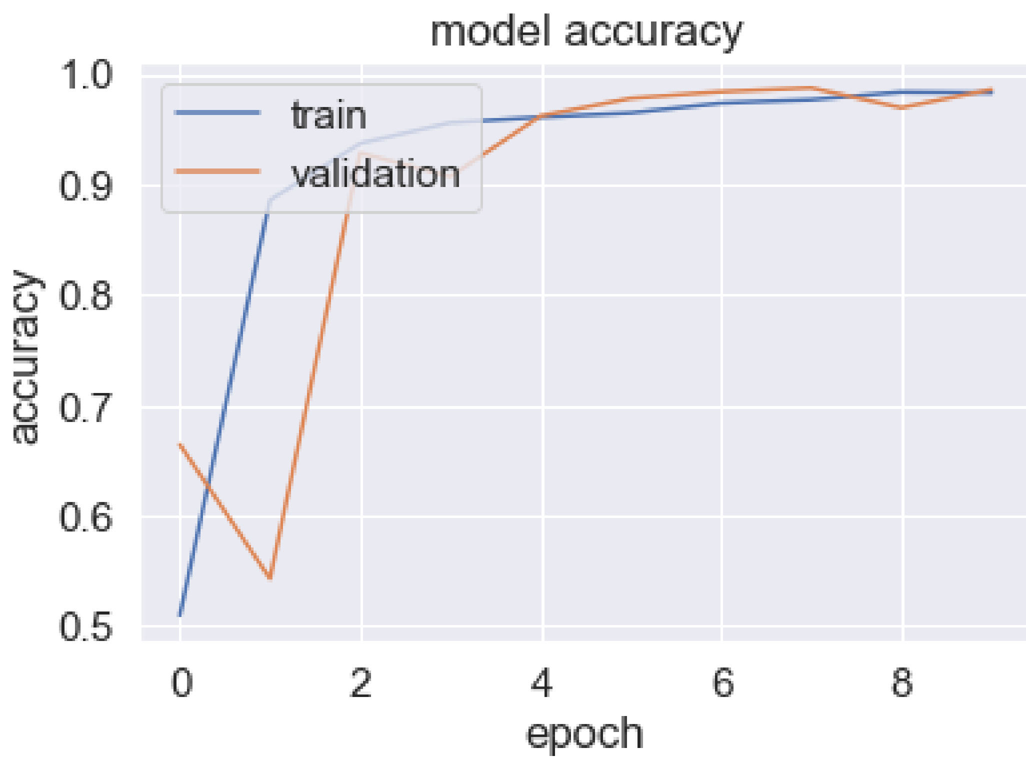
```
val_accuracy: 0.9680 - 201s/epoch - 806ms/step
Epoch 3/10
250/250 - 241s - loss: 0.2152 - accuracy: 0.9426 - val_loss: 6.2874 -
val_accuracy: 0.6280 - 241s/epoch - 962ms/step
Epoch 4/10
250/250 - 252s - loss: 0.1856 - accuracy: 0.9539 - val_loss: 1.7030 -
val_accuracy: 0.9315 - 252s/epoch - 1s/step
Epoch 5/10
250/250 - 256s - loss: 0.1201 - accuracy: 0.9647 - val_loss: 0.3575 -
val_accuracy: 0.9665 - 256s/epoch - 1s/step
Epoch 6/10
250/250 - 211s - loss: 0.0937 - accuracy: 0.9711 - val_loss: 0.1008 -
val_accuracy: 0.9710 - 211s/epoch - 844ms/step
Epoch 7/10
250/250 - 211s - loss: 0.0867 - accuracy: 0.9772 - val_loss: 0.0528 -
val_accuracy: 0.9860 - 211s/epoch - 845ms/step
Epoch 8/10
250/250 - 215s - loss: 0.0828 - accuracy: 0.9801 - val_loss: 0.1079 -
val_accuracy: 0.9725 - 215s/epoch - 860ms/step
Epoch 9/10
250/250 - 211s - loss: 0.0620 - accuracy: 0.9837 - val_loss: 0.0815 -
val_accuracy: 0.9775 - 211s/epoch - 843ms/step
Epoch 10/10
250/250 - 186s - loss: 0.0590 - accuracy: 0.9844 - val_loss: 0.9210 -
val_accuracy: 0.8710 - 186s/epoch - 743ms/step
```



```
Score for fold 1: loss of 0.9210289716720581; accuracy of 87.09999918937683%
63/63 [==============================] - 2s 28ms/step
[[667   0   0  29   0   0   0   0 207]
 [  0  71   0   0   0   0   0   0   0]
 [  0   4  37   2   0   1   2   0   0]
 [  0   0   0  35   0   0   0   0   0]
 [  1   0   0   2 103   0   0   0   0]
 [  0   2   0   0   0 120   1   0   0]
 [  0   3   0   0   0   0  83   0   0]
 [  0   0   0   3   0   0   0 585   0]
 [  0   0   0   0   0   1   0   0  41]]
------------------------------------------------------------------------
Training for fold 2 …
Epoch 1/10
250/250 - 188s - loss: 1.4509 - accuracy: 0.5096 - val_loss: 1.3032 -
val_accuracy: 0.6645 - 188s/epoch - 751ms/step
Epoch 2/10
250/250 - 172s - loss: 0.3862 - accuracy: 0.8857 - val_loss: 13.8235 -
val_accuracy: 0.5430 - 172s/epoch - 687ms/step
Epoch 3/10
250/250 - 176s - loss: 0.2133 - accuracy: 0.9371 - val_loss: 0.3055 -
val_accuracy: 0.9285 - 176s/epoch - 702ms/step
Epoch 4/10
250/250 - 161s - loss: 0.1449 - accuracy: 0.9559 - val_loss: 2.6323 -
val_accuracy: 0.9070 - 161s/epoch - 646ms/step
Epoch 5/10
250/250 - 165s - loss: 0.1310 - accuracy: 0.9609 - val_loss: 2.0419 -
val_accuracy: 0.9620 - 165s/epoch - 660ms/step
Epoch 6/10
250/250 - 167s - loss: 0.1523 - accuracy: 0.9647 - val_loss: 0.1607 -
val_accuracy: 0.9780 - 167s/epoch - 667ms/step
Epoch 7/10
250/250 - 180s - loss: 0.0952 - accuracy: 0.9736 - val_loss: 0.0548 -
val_accuracy: 0.9840 - 180s/epoch - 719ms/step
Epoch 8/10
250/250 - 209s - loss: 0.0845 - accuracy: 0.9768 - val_loss: 0.0547 -
val_accuracy: 0.9875 - 209s/epoch - 835ms/step
Epoch 9/10
250/250 - 212s - loss: 0.0653 - accuracy: 0.9836 - val_loss: 0.1220 -
val_accuracy: 0.9695 - 212s/epoch - 848ms/step
Epoch 10/10
250/250 - 203s - loss: 0.0674 - accuracy: 0.9830 - val_loss: 0.0432 -
val_accuracy: 0.9860 - 203s/epoch - 811ms/step
```

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

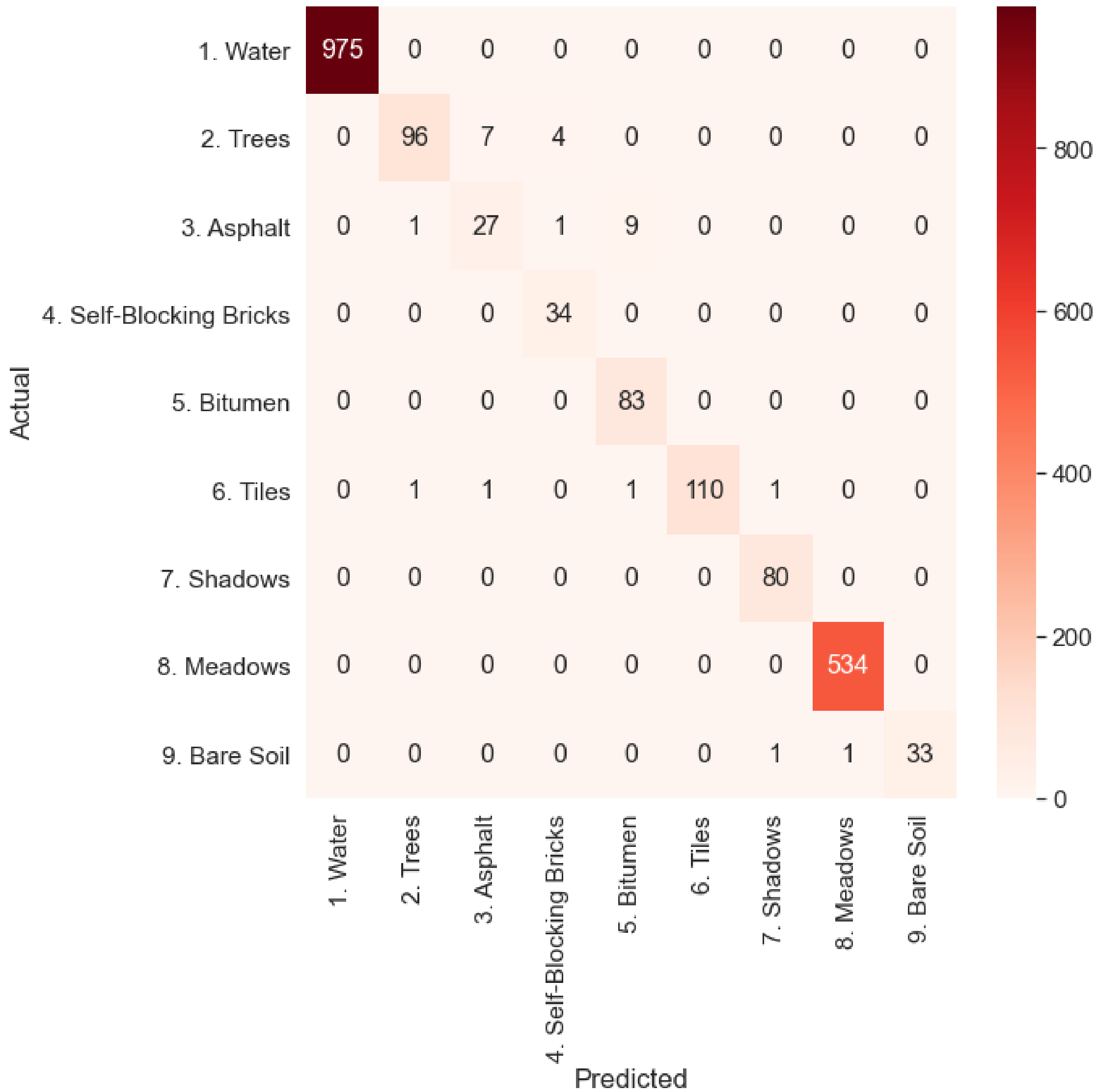

Actual
Predicted
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
667 0 0 29 0 0 0 0 207
0 71 0 0 0 0 0 0 0
0 4 37 2 0 1 2 0 0
0 0 0 35 0 0 0 0 0
1 0 0 2 103 0 0 0 0
0 2 0 0 0 120 1 0 0
0 3 0 0 0 0 83 0 0
0 0 0 3 0 0 0 585 0
0 0 0 0 0 1 0 0 41
600
500
400
300
200
100
0



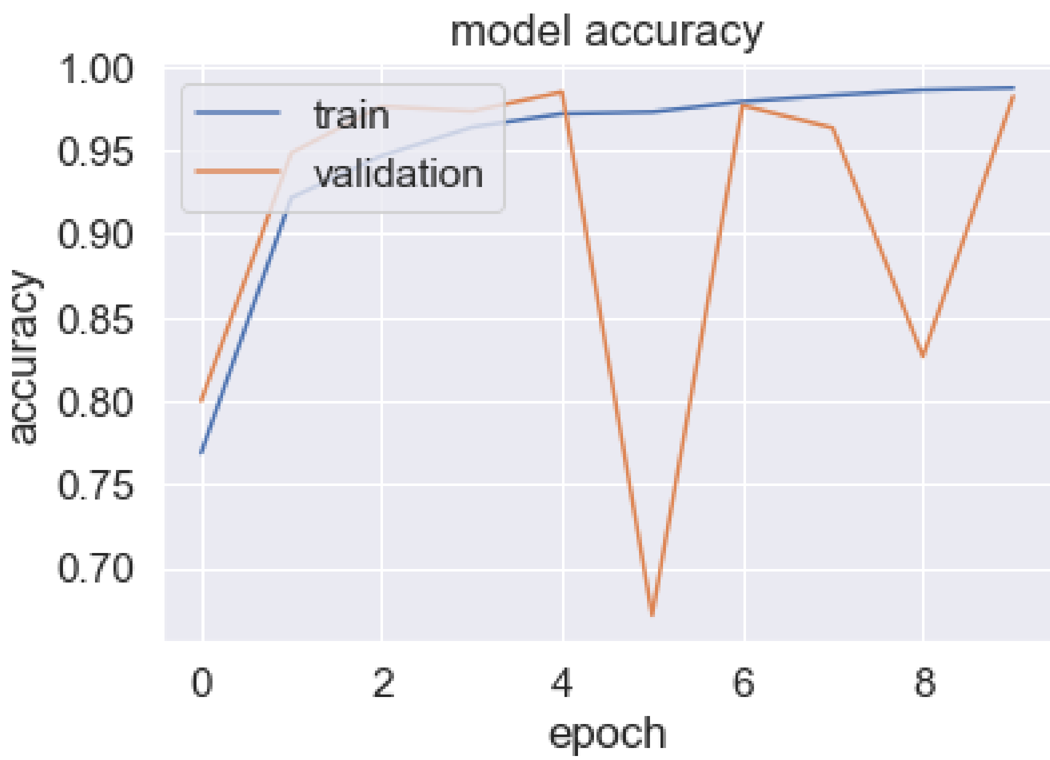

```
Score for fold 2: loss of 0.043157752603292465; accuracy of 98.60000014305115%
63/63 [==============================] - 3s 35ms/step
[[975   0   0   0   0   0   0   0   0]
 [  0  96   7   4   0   0   0   0   0]
 [  0   1  27   1   9   0   0   0   0]
 [  0   0   0  34   0   0   0   0   0]
 [  0   0   0   0  83   0   0   0   0]
 [  0   1   1   0   1 110   1   0   0]
 [  0   0   0   0   0   0  80   0   0]
 [  0   0   0   0   0   0   0 534   0]
 [  0   0   0   0   0   0   1   1  33]]
------------------------------------------------------------------------
Training for fold 3 …
Epoch 1/10
250/250 - 225s - loss: 0.7617 - accuracy: 0.7688 - val_loss: 0.7705 -
val_accuracy: 0.8000 - 225s/epoch - 900ms/step
Epoch 2/10
250/250 - 214s - loss: 0.2617 - accuracy: 0.9216 - val_loss: 0.2535 -
val_accuracy: 0.9485 - 214s/epoch - 857ms/step
Epoch 3/10
250/250 - 207s - loss: 0.1870 - accuracy: 0.9467 - val_loss: 0.0707 -
val_accuracy: 0.9765 - 207s/epoch - 828ms/step
Epoch 4/10
```

```
250/250 - 186s - loss: 0.1347 - accuracy: 0.9639 - val_loss: 0.0912 -
val_accuracy: 0.9735 - 186s/epoch - 744ms/step
Epoch 5/10
250/250 - 181s - loss: 0.0951 - accuracy: 0.9720 - val_loss: 0.0523 -
val_accuracy: 0.9850 - 181s/epoch - 722ms/step
Epoch 6/10
250/250 - 184s - loss: 0.1060 - accuracy: 0.9730 - val_loss: 6.1422 -
val_accuracy: 0.6715 - 184s/epoch - 738ms/step
Epoch 7/10
250/250 - 195s - loss: 0.0784 - accuracy: 0.9793 - val_loss: 0.0569 -
val_accuracy: 0.9770 - 195s/epoch - 781ms/step
Epoch 8/10
250/250 - 196s - loss: 0.0687 - accuracy: 0.9830 - val_loss: 0.4723 -
val_accuracy: 0.9635 - 196s/epoch - 786ms/step
Epoch 9/10
250/250 - 181s - loss: 0.0503 - accuracy: 0.9862 - val_loss: 3.6070 -
val_accuracy: 0.8265 - 181s/epoch - 725ms/step
Epoch 10/10
250/250 - 177s - loss: 0.0433 - accuracy: 0.9875 - val_loss: 0.0637 -
val_accuracy: 0.9825 - 177s/epoch - 708ms/step
```

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



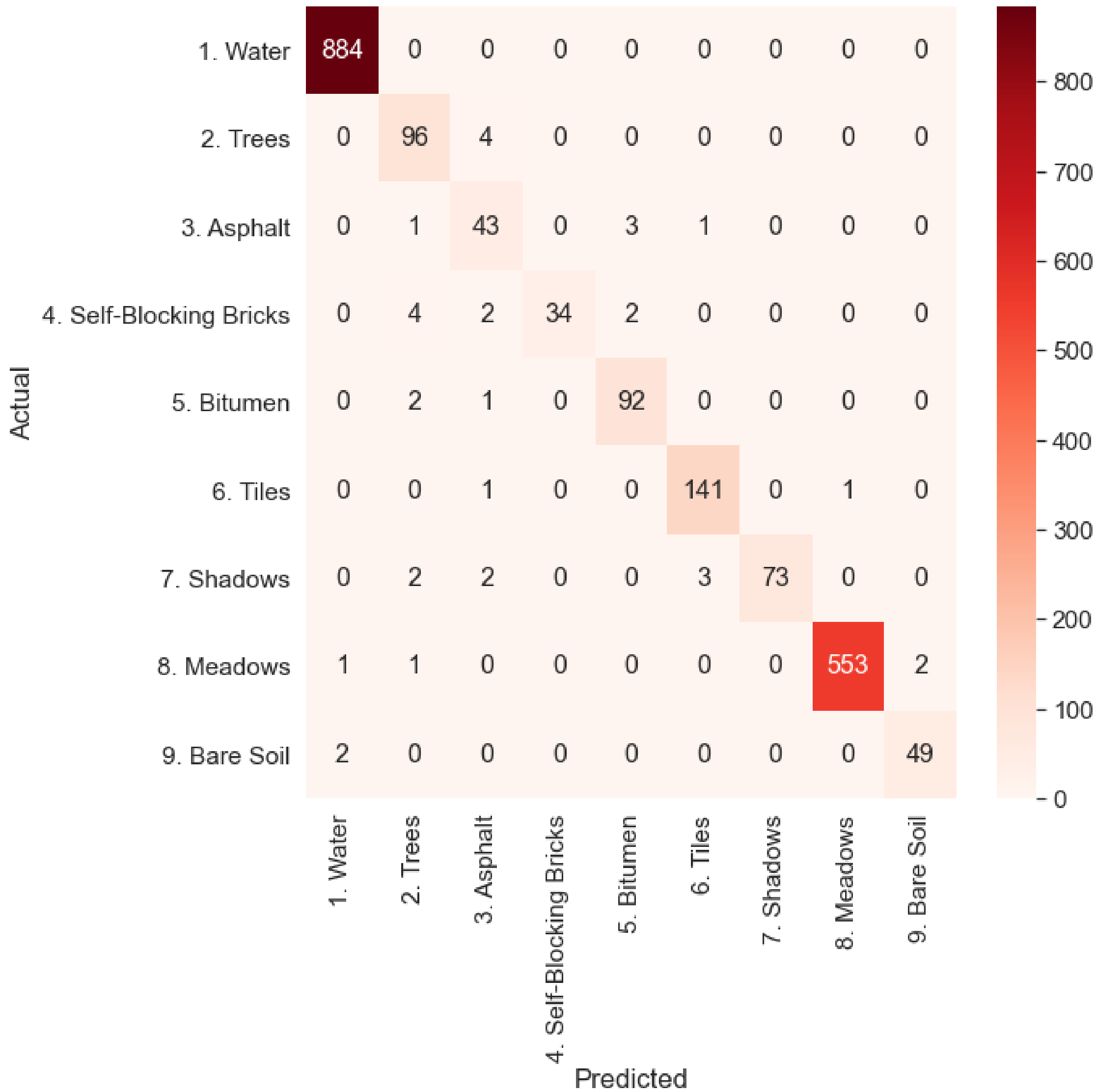
Actual
Predicted
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
975 0 0 0 0 0 0 0 0
0 96 7 4 0 0 0 0 0
0 1 27 1 9 0 0 0 0
0 0 0 34 0 0 0 0 0
0 0 0 0 83 0 0 0 0
0 1 1 0 1 110 1 0 0
0 0 0 0 0 0 80 0 0
0 0 0 0 0 0 0 534 0
0 0 0 0 0 0 1 1 33
800
600
400
200
0

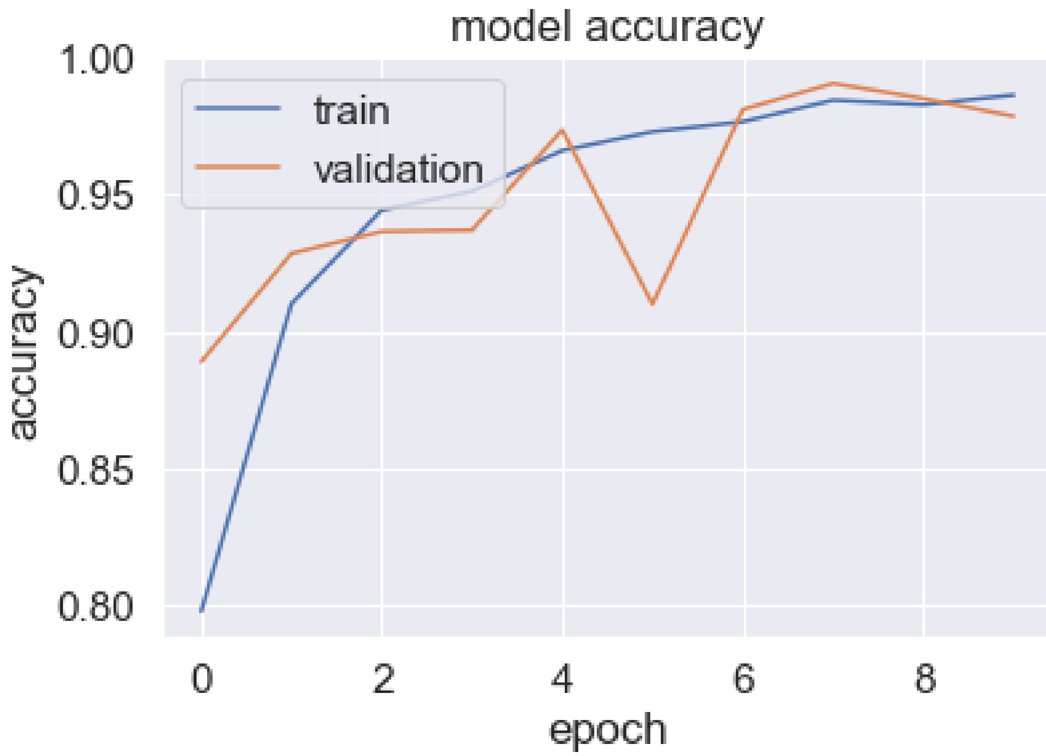



```
Score for fold 3: loss of 0.06372090429067612; accuracy of 98.25000166893005%
63/63 [==============================] - 2s 30ms/step
[[884   0   0   0   0   0   0   0   0]
 [  0  96   4   0   0   0   0   0   0]
 [  0   1  43   0   3   1   0   0   0]
 [  0   4   2  34   2   0   0   0   0]
 [  0   2   1   0  92   0   0   0   0]
 [  0   0   1   0   0 141   0   1   0]
 [  0   2   2   0   0   3  73   0   0]
 [  1   1   0   0   0   0   0 553   2]
 [  2   0   0   0   0   0   0   0  49]]
------------------------------------------------------------------------
Training for fold 4 …
Epoch 1/10
250/250 - 183s - loss: 0.7109 - accuracy: 0.7980 - val_loss: 0.4502 -
val_accuracy: 0.8890 - 183s/epoch - 733ms/step
Epoch 2/10
250/250 - 185s - loss: 0.3008 - accuracy: 0.9101 - val_loss: 13.0653 -
val_accuracy: 0.9285 - 185s/epoch - 741ms/step
Epoch 3/10
250/250 - 171s - loss: 0.1945 - accuracy: 0.9442 - val_loss: 0.7509 -
val_accuracy: 0.9365 - 171s/epoch - 683ms/step
Epoch 4/10
```

```
250/250 - 214s - loss: 0.1762 - accuracy: 0.9511 - val_loss: 0.1980 -
val_accuracy: 0.9370 - 214s/epoch - 856ms/step
Epoch 5/10
250/250 - 205s - loss: 0.1123 - accuracy: 0.9660 - val_loss: 0.0768 -
val_accuracy: 0.9735 - 205s/epoch - 821ms/step
Epoch 6/10
250/250 - 219s - loss: 0.1042 - accuracy: 0.9729 - val_loss: 1.0592 -
val_accuracy: 0.9100 - 219s/epoch - 874ms/step
Epoch 7/10
250/250 - 221s - loss: 0.0930 - accuracy: 0.9765 - val_loss: 0.0700 -
val_accuracy: 0.9810 - 221s/epoch - 884ms/step
Epoch 8/10
250/250 - 194s - loss: 0.0598 - accuracy: 0.9844 - val_loss: 0.0287 -
val_accuracy: 0.9905 - 194s/epoch - 777ms/step
Epoch 9/10
250/250 - 198s - loss: 0.0713 - accuracy: 0.9827 - val_loss: 0.0545 -
val_accuracy: 0.9850 - 198s/epoch - 794ms/step
Epoch 10/10
250/250 - 195s - loss: 0.0528 - accuracy: 0.9862 - val_loss: 0.0690 -
val_accuracy: 0.9785 - 195s/epoch - 782ms/step
```

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

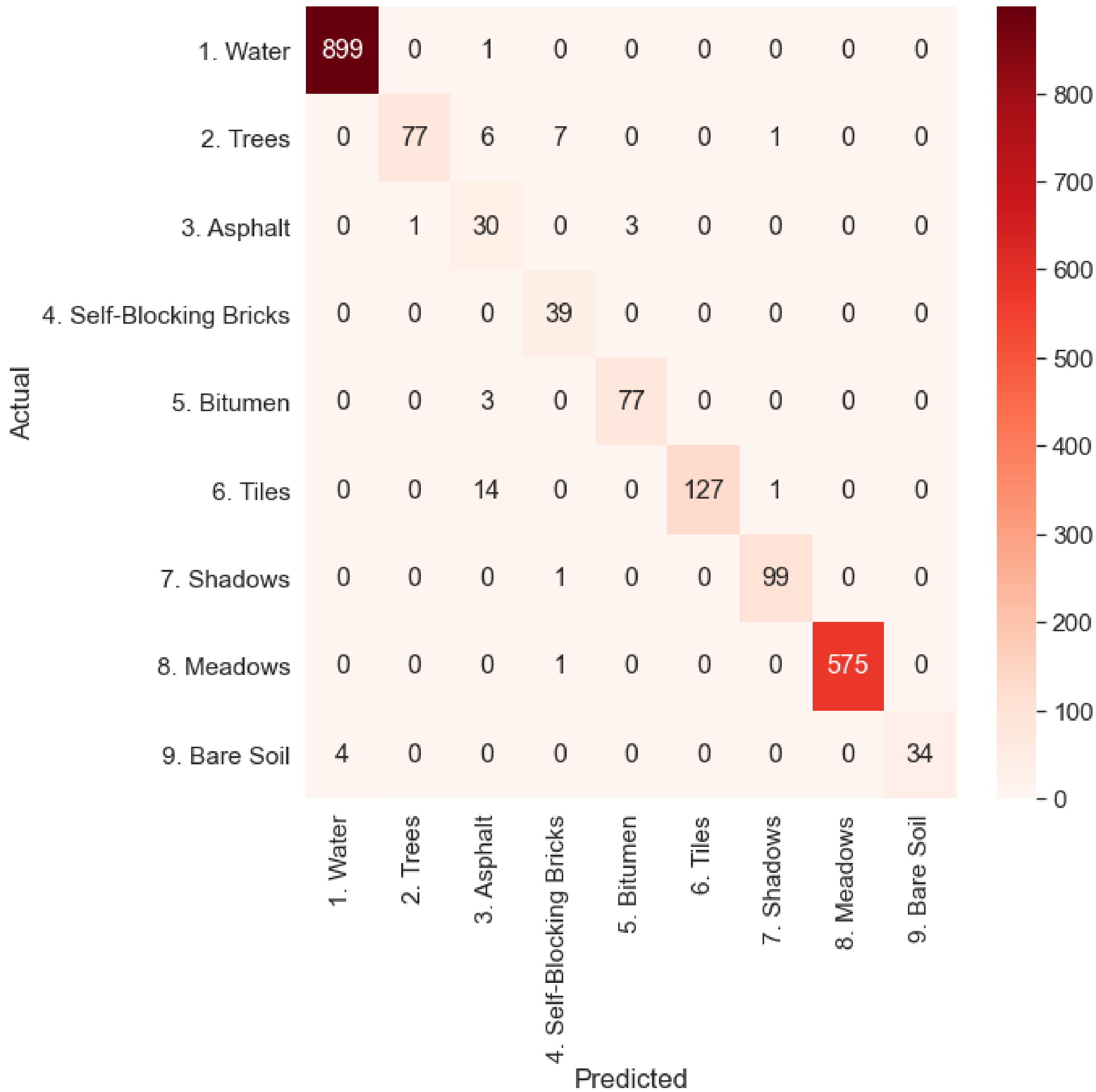

1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
884 0 0 0 0 0 0 0 0
0 96 4 0 0 0 0 0 0
0 1 43 0 3 1 0 0 0
0 4 2 34 2 0 0 0 0
0 2 1 0 92 0 0 0 0
0 0 1 0 0 141 0 1 0
0 2 2 0 0 3 73 0 0
1 1 0 0 0 0 0 553 2
2 0 0 0 0 0 0 0 49
800
700
600
500
400
300
200
100
0



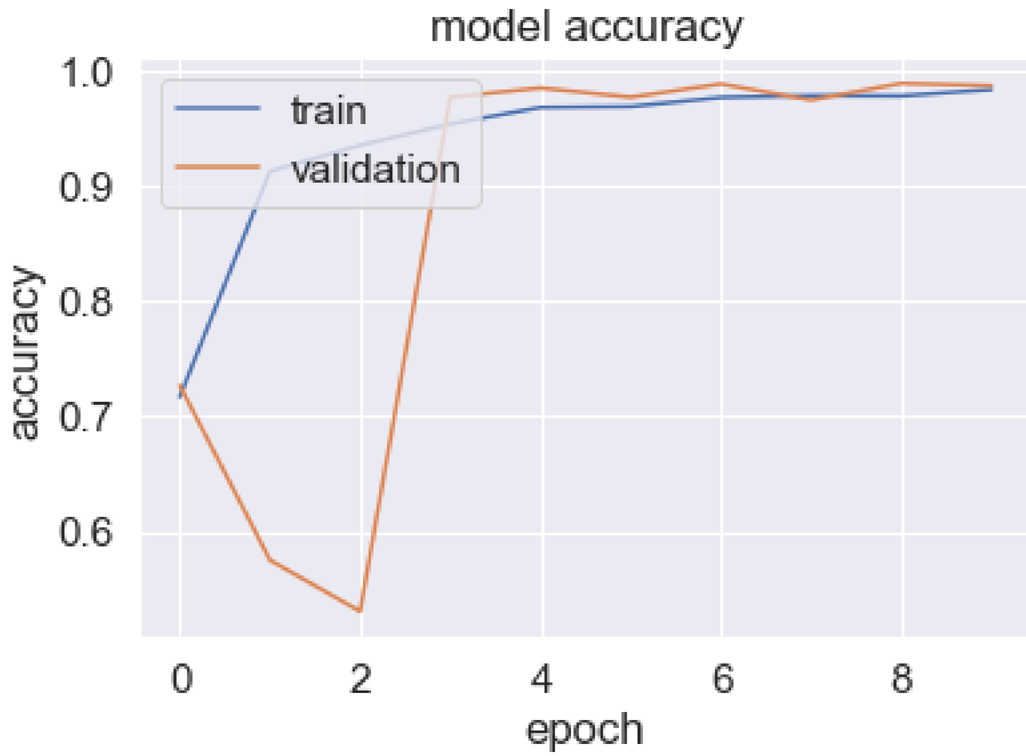

```
Score for fold 4: loss of 0.0689583271741867; accuracy of 97.85000085830688%
63/63 [==============================] - 2s 32ms/step
[[899   0   1   0   0   0   0   0   0]
 [  0  77   6   7   0   0   1   0   0]
 [  0   1  30   0   3   0   0   0   0]
 [  0   0   0  39   0   0   0   0   0]
 [  0   0   3   0  77   0   0   0   0]
 [  0   0  14   0   0 127   1   0   0]
 [  0   0   0   1   0   0  99   0   0]
 [  0   0   0   1   0   0   0 575   0]
 [  4   0   0   0   0   0   0   0  34]]
------------------------------------------------------------------------
Training for fold 5 …
Epoch 1/10
250/250 - 201s - loss: 0.8894 - accuracy: 0.7170 - val_loss: 1.2928 -
val_accuracy: 0.7270 - 201s/epoch - 805ms/step
Epoch 2/10
250/250 - 195s - loss: 0.2875 - accuracy: 0.9126 - val_loss: 4.7491 -
val_accuracy: 0.5755 - 195s/epoch - 780ms/step
Epoch 3/10
250/250 - 196s - loss: 0.2189 - accuracy: 0.9352 - val_loss: 12.2001 -
val_accuracy: 0.5310 - 196s/epoch - 782ms/step
Epoch 4/10
```

```
250/250 - 195s - loss: 0.1664 - accuracy: 0.9538 - val_loss: 0.0718 -
val_accuracy: 0.9770 - 195s/epoch - 782ms/step
Epoch 5/10
250/250 - 196s - loss: 0.1092 - accuracy: 0.9681 - val_loss: 0.0463 -
val_accuracy: 0.9850 - 196s/epoch - 784ms/step
Epoch 6/10
250/250 - 195s - loss: 0.1175 - accuracy: 0.9689 - val_loss: 0.2958 -
val_accuracy: 0.9770 - 195s/epoch - 782ms/step
Epoch 7/10
250/250 - 196s - loss: 0.0760 - accuracy: 0.9768 - val_loss: 0.0466 -
val_accuracy: 0.9885 - 196s/epoch - 783ms/step
Epoch 8/10
250/250 - 195s - loss: 0.0865 - accuracy: 0.9784 - val_loss: 0.1726 -
val_accuracy: 0.9745 - 195s/epoch - 782ms/step
Epoch 9/10
250/250 - 195s - loss: 0.0871 - accuracy: 0.9781 - val_loss: 0.0523 -
val_accuracy: 0.9890 - 195s/epoch - 781ms/step
Epoch 10/10
250/250 - 194s - loss: 0.0651 - accuracy: 0.9836 - val_loss: 0.0530 -
val_accuracy: 0.9870 - 194s/epoch - 778ms/step
```

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

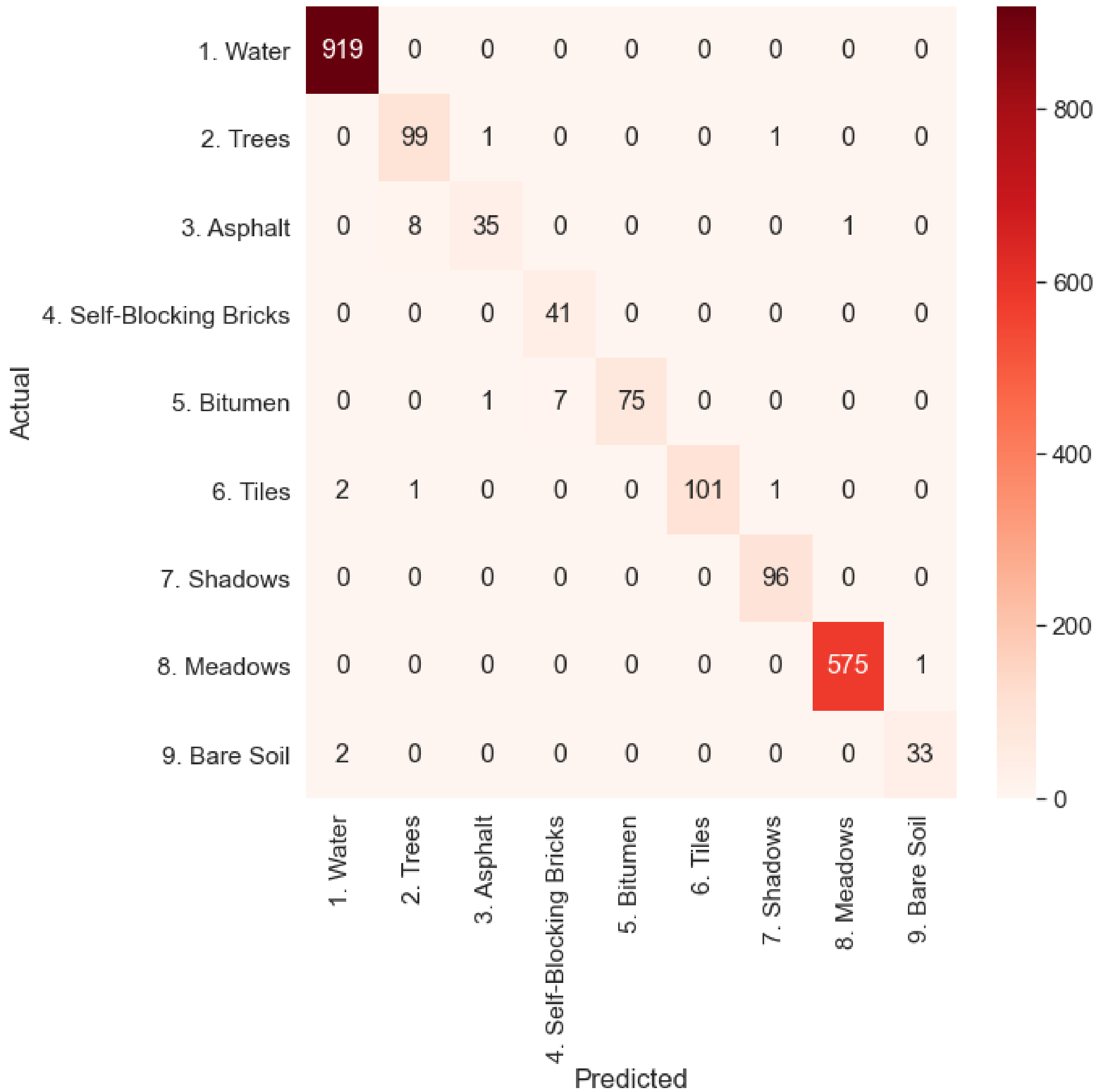


Actual
Predicted
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
899 0 1 0 0 0 0 0 0
0 77 6 7 0 0 1 0 0
0 1 30 0 3 0 0 0 0
0 0 0 39 0 0 0 0 0
0 0 3 0 77 0 0 0 0
0 0 14 0 0 127 1 0 0
0 0 0 1 0 0 99 0 0
0 0 0 1 0 0 0 575 0
4 0 0 0 0 0 0 0 34
800
700
600
500
400
300
200
100
0




```
Score for fold 5: loss of 0.0529952198266983; accuracy of 98.69999885559082%
63/63 [==============================] - 3s 34ms/step
[[919   0   0   0   0   0   0   0   0]
 [  0  99   1   0   0   0   1   0   0]
 [  0   8  35   0   0   0   0   1   0]
 [  0   0   0  41   0   0   0   0   0]
 [  0   0   1   7  75   0   0   0   0]
 [  2   1   0   0   0 101   1   0   0]
 [  0   0   0   0   0   0  96   0   0]
 [  0   0   0   0   0   0   0 575   1]
 [  2   0   0   0   0   0   0   0  33]]
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.9210289716720581 - Accuracy: 87.09999918937683%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.043157752603292465 - Accuracy: 98.60000014305115%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.06372090429067612 - Accuracy: 98.25000166893005%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.0689583271741867 - Accuracy: 97.85000085830688%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.0529952198266983 - Accuracy: 98.69999885559082%
```

```
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 96.10000014305115 (+- 4.509878480314604)
> Loss: 0.22997223511338233
------------------------------------------------------------------------
Predicted Overall        1. Water  2. Trees  3. Asphalt  \
Actual Overall
1. Water                     4344         0           1
2. Trees                        0       439          18
3. Asphalt                      0        15         172
4. Self-Blocking Bricks         0         4           2
5. Bitumen                      1         2           5
6. Tiles                        2         4          16
7. Shadows                      0         5           2
8. Meadows                      1         1           0
9. Bare Soil                    8         0           0

Predicted Overall        4. Self-Blocking Bricks  5. Bitumen  6. Tiles  \
Actual Overall
1. Water                                      29           0         0
2. Trees                                      11           0         0
3. Asphalt                                     3          15         2
4. Self-Blocking Bricks                      183           2         0
5. Bitumen                                     9         430         0
6. Tiles                                       0           1       599
7. Shadows                                     1           0         3
8. Meadows                                     4           0         0
9. Bare Soil                                   0           0         1

Predicted Overall        7. Shadows  8. Meadows  9. Bare Soil
Actual Overall
1. Water                          0           0           207
2. Trees                          2           0             0
3. Asphalt                        2           1             0
4. Self-Blocking Bricks           0           0             0
5. Bitumen                        0           0             0
6. Tiles                          4           1             0
7. Shadows                      431           0             0
8. Meadows                        0        2822             3
9. Bare Soil                      1           1           190

<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
919 0 0 0 0 0 0 0 0
0 99 1 0 0 0 1 0 0
0 8 35 0 0 0 0 1 0
0 0 0 41 0 0 0 0 0
0 0 1 7 75 0 0 0 0
2 1 0 0 0 101 1 0 0
0 0 0 0 0 0 96 0 0
0 0 0 0 0 0 0 575 1
2 0 0 0 0 0 0 0 33
800
600
400
200
0

[ ]:
```python
print(classification_report(targets[test], y_pred, target_names = names))
```

```
                      precision    recall  f1-score   support

            1. Water       1.00      1.00      1.00       919
            2. Trees       0.92      0.98      0.95       101
          3. Asphalt       0.95      0.80      0.86        44
4. Self-Blocking Bricks    0.85      1.00      0.92        41
          5. Bitumen       1.00      0.90      0.95        83
            6. Tiles       1.00      0.96      0.98       105
          7. Shadows       0.98      1.00      0.99        96
          8. Meadows       1.00      1.00      1.00       576
        9. Bare Soil       0.97      0.94      0.96        35

            accuracy                           0.99      2000
           macro avg       0.96      0.95      0.96      2000
        weighted avg       0.99      0.99      0.99      2000
```