5 X_Xception_centr

April 3, 2023

1 Date: 9 2022

2 Method: Cross_Inception

3 Data: Pavia

4 Results v.05

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

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

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

X, y = read_HSI()
```

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

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

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

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

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

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

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

[ ]:
```
test_ratio = 0.5

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

windowSize=5  # accuracy of
# Score for fold 1: loss of 0.3463119268417358 4; accuracy of 89.49999809265137%

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

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

input_image_size=windowSize
INPUT_IMG_SIZE=windowSize

dimReduction=3

InputShape=(windowSize, windowSize, dimReduction)

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

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

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

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

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

[ ]:
```python
print()

import warnings
warnings.filterwarnings("ignore")

# load libraries
from keras.initializers import VarianceScaling
from keras.regularizers import l2
from keras.models import Sequential
from keras.layers import Dense
from sklearn import datasets
from sklearn.model_selection import StratifiedKFold
import numpy as np
```

[ ]:
```python
# 9 classes names

names = ['1. Water', '2. Trees', '3. Asphalt', '4. Self-Blocking Bricks',
                 '5. Bitumen','6. Tiles', '7. Shadows',
                 '8. Meadows', '9. Bare Soil']
```

[ ]:
```python
from tensorflow.keras.applications import EfficientNetB0
from keras.applications import densenet, inception_v3, mobilenet, resnet,␣
↪vgg16, vgg19, xception
from tensorflow.keras import layers
from keras.layers import Dense, GlobalAveragePooling2D, Dropout, Flatten
import tensorflow as tf

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


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

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


    # Freeze the pretrained weights
    model.trainable = False

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

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

    x = model.output


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


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

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

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

[ ]:
```
from tensorflow.keras.applications import EfficientNetB0
```

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


    # Freeze the pretrained weights
   #model.trainable = False

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

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

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

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

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

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

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


def plot_hist(hist):
```

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

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

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

import tensorflow as tf

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

NN=len(Xtrain)
NN=700
#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)
```

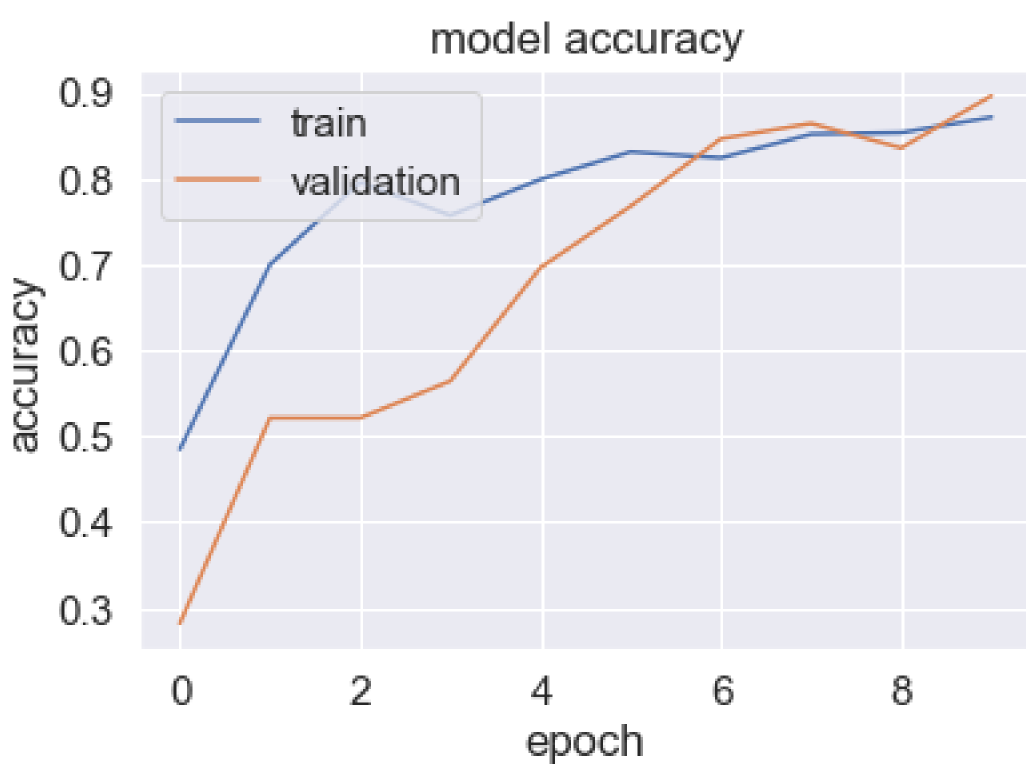
```
# Parse numbers as floats
#input_train = input_train.astype('float32')
#input_test = input_test.astype('float32')

# 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
```

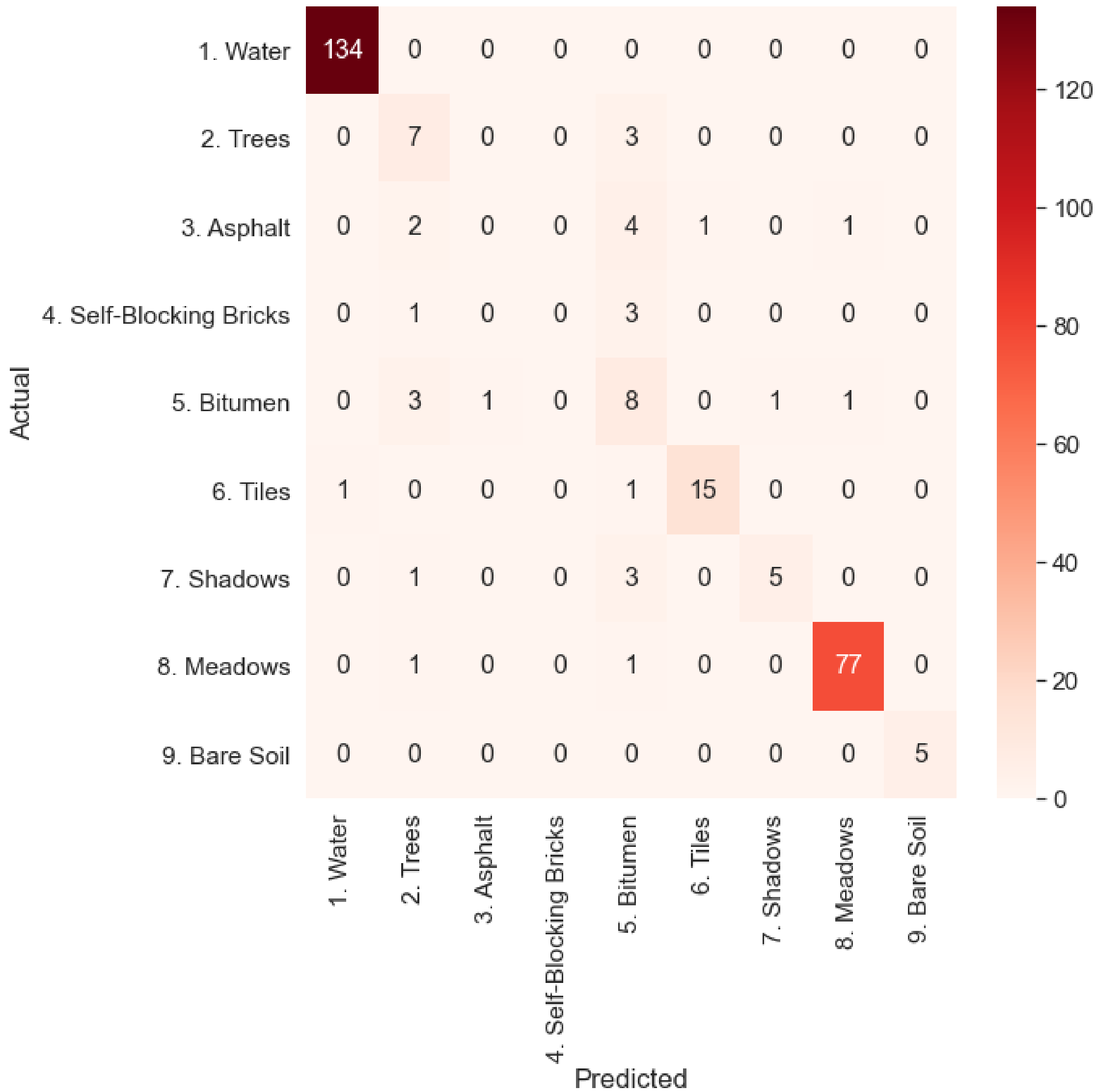
```
------------------------------------------------------------------------
Training for fold 1 …
Epoch 1/10
35/35 - 37s - loss: 1.6848 - accuracy: 0.4848 - val_loss: 1.5543 - val_accuracy:
0.2821 - 37s/epoch - 1s/step
Epoch 2/10
35/35 - 31s - loss: 0.9981 - accuracy: 0.7000 - val_loss: 1.4624 - val_accuracy:
```

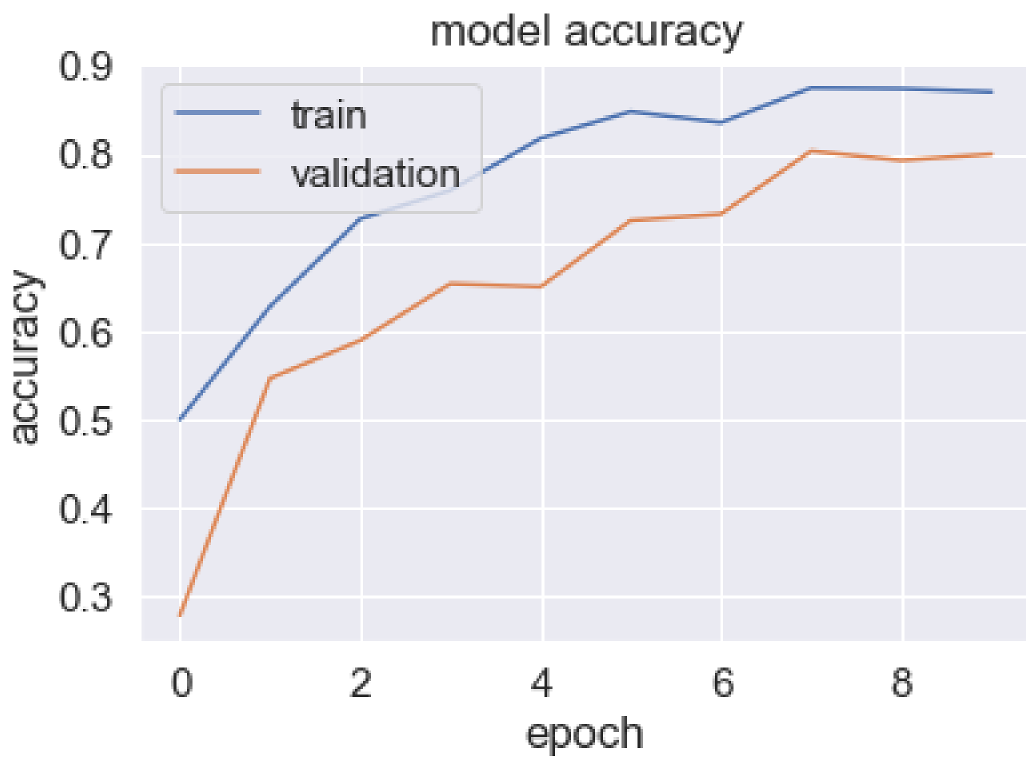
```
0.5214 - 31s/epoch - 881ms/step
Epoch 3/10
35/35 - 31s - loss: 0.6657 - accuracy: 0.7955 - val_loss: 1.4448 - val_accuracy:
0.5214 - 31s/epoch - 893ms/step
Epoch 4/10
35/35 - 30s - loss: 0.9612 - accuracy: 0.7571 - val_loss: 1.3767 - val_accuracy:
0.5643 - 30s/epoch - 857ms/step
Epoch 5/10
35/35 - 29s - loss: 0.6493 - accuracy: 0.7991 - val_loss: 1.2371 - val_accuracy:
0.6964 - 29s/epoch - 832ms/step
Epoch 6/10
35/35 - 29s - loss: 0.5050 - accuracy: 0.8313 - val_loss: 1.0850 - val_accuracy:
0.7679 - 29s/epoch - 821ms/step
Epoch 7/10
35/35 - 29s - loss: 0.5526 - accuracy: 0.8241 - val_loss: 0.5527 - val_accuracy:
0.8464 - 29s/epoch - 820ms/step
Epoch 8/10
35/35 - 29s - loss: 0.4624 - accuracy: 0.8518 - val_loss: 0.4942 - val_accuracy:
0.8643 - 29s/epoch - 824ms/step
Epoch 9/10
35/35 - 28s - loss: 0.5121 - accuracy: 0.8536 - val_loss: 5.2260 - val_accuracy:
0.8357 - 28s/epoch - 812ms/step
Epoch 10/10
35/35 - 29s - loss: 0.4555 - accuracy: 0.8714 - val_loss: 1.7005 - val_accuracy:
0.8964 - 29s/epoch - 815ms/step
```



```
Score for fold 1: loss of 1.7004808187484741; accuracy of 89.64285850524902%
9/9 [==============================] - 1s 34ms/step
[[134   0   0   0   0   0   0   0   0]
 [  0   7   0   0   3   0   0   0   0]
 [  0   2   0   0   4   1   0   1   0]
 [  0   1   0   0   3   0   0   0   0]
 [  0   3   1   0   8   0   1   1   0]
 [  1   0   0   0   1  15   0   0   0]
 [  0   1   0   0   3   0   5   0   0]
 [  0   1   0   0   1   0   0  77   0]
 [  0   0   0   0   0   0   0   0   5]]
------------------------------------------------------------------------
Training for fold 2 …
Epoch 1/10
35/35 - 36s - loss: 1.6127 - accuracy: 0.5000 - val_loss: 1.8610 - val_accuracy:
0.2786 - 36s/epoch - 1s/step
Epoch 2/10
35/35 - 31s - loss: 1.3385 - accuracy: 0.6277 - val_loss: 1.5071 - val_accuracy:
0.5464 - 31s/epoch - 895ms/step
Epoch 3/10
35/35 - 31s - loss: 0.9322 - accuracy: 0.7268 - val_loss: 1.3973 - val_accuracy:
0.5893 - 31s/epoch - 876ms/step
Epoch 4/10
35/35 - 29s - loss: 0.7721 - accuracy: 0.7589 - val_loss: 1.4485 - val_accuracy:
0.6536 - 29s/epoch - 823ms/step
Epoch 5/10
35/35 - 29s - loss: 0.6196 - accuracy: 0.8179 - val_loss: 1.4363 - val_accuracy:
0.6500 - 29s/epoch - 837ms/step
Epoch 6/10
35/35 - 29s - loss: 0.5129 - accuracy: 0.8482 - val_loss: 0.9883 - val_accuracy:
0.7250 - 29s/epoch - 837ms/step
Epoch 7/10
35/35 - 30s - loss: 0.5164 - accuracy: 0.8357 - val_loss: 0.8147 - val_accuracy:
0.7321 - 30s/epoch - 848ms/step
Epoch 8/10
35/35 - 30s - loss: 0.3752 - accuracy: 0.8750 - val_loss: 0.5641 - val_accuracy:
0.8036 - 30s/epoch - 852ms/step
Epoch 9/10
35/35 - 35s - loss: 0.3787 - accuracy: 0.8741 - val_loss: 0.5219 - val_accuracy:
0.7929 - 35s/epoch - 986ms/step
Epoch 10/10
35/35 - 35s - loss: 0.3766 - accuracy: 0.8705 - val_loss: 0.5310 - val_accuracy:
0.8000 - 35s/epoch - 1s/step
```

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

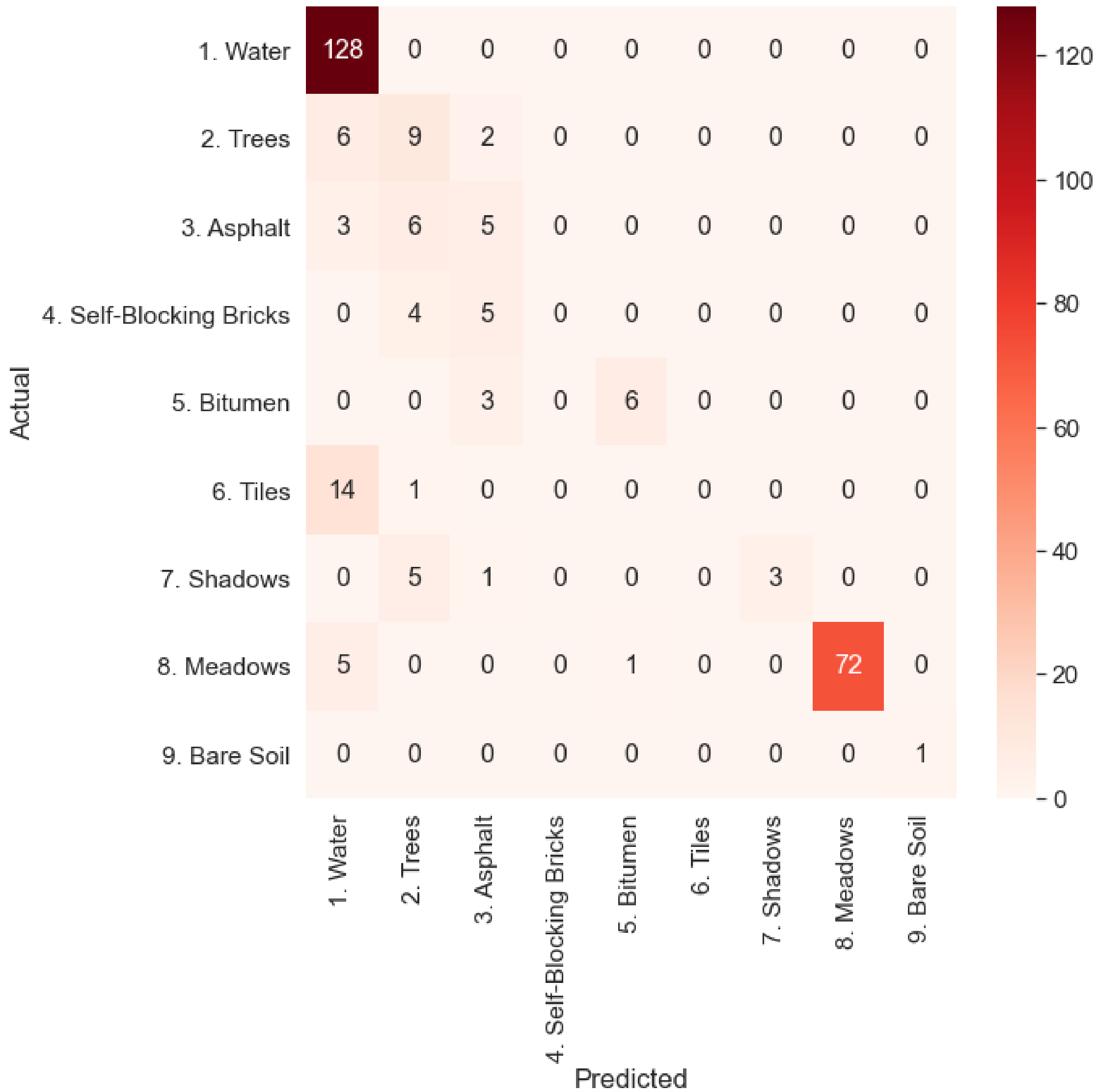

1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
134 0 0 0 0 0 0 0 0
0 7 0 0 3 0 0 0 0
0 2 0 0 4 1 0 1 0
0 1 0 0 3 0 0 0 0
0 3 1 0 8 0 1 1 0
1 0 0 0 1 15 0 0 0
0 1 0 0 3 0 5 0 0
0 1 0 0 1 0 0 77 0
0 0 0 0 0 0 0 0 5
120
100
80
60
40
20
0

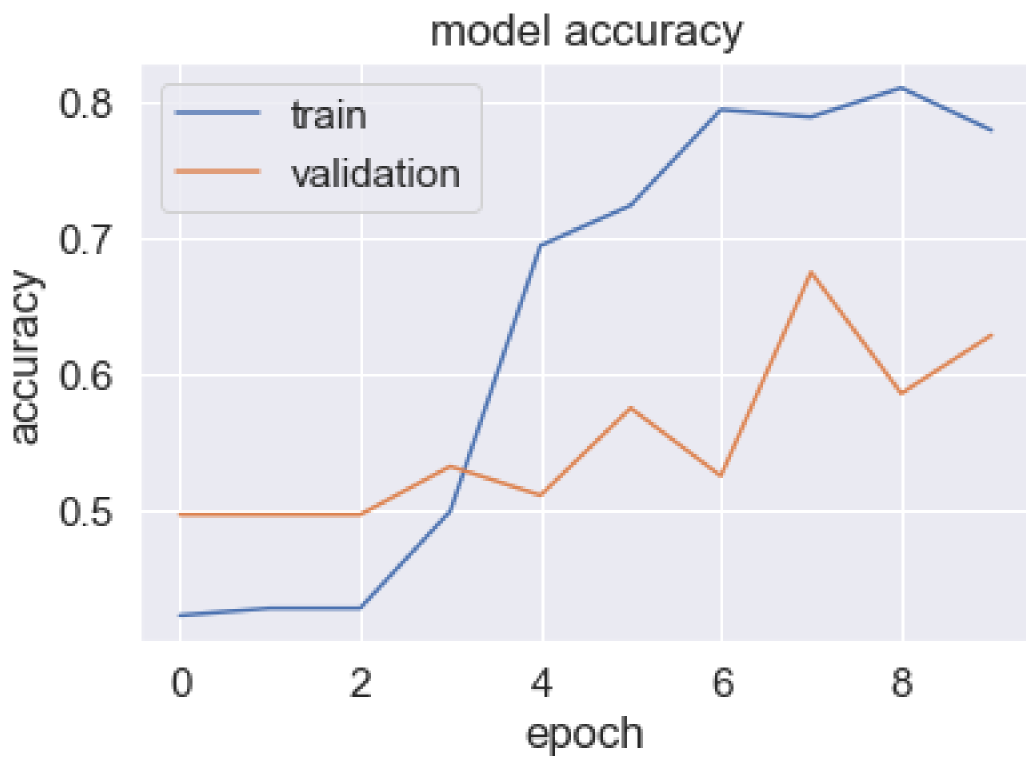



```
Score for fold 2: loss of 0.5309717059135437; accuracy of 80.0000011920929%
9/9 [==============================] - 1s 40ms/step
[[128   0   0   0   0   0   0   0   0]
 [  6   9   2   0   0   0   0   0   0]
 [  3   6   5   0   0   0   0   0   0]
 [  0   4   5   0   0   0   0   0   0]
 [  0   0   3   0   6   0   0   0   0]
 [ 14   1   0   0   0   0   0   0   0]
 [  0   5   1   0   0   0   3   0   0]
 [  5   0   0   0   1   0   0  72   0]
 [  0   0   0   0   0   0   0   0   1]]
------------------------------------------------------------------------
Training for fold 3 …
Epoch 1/10
35/35 - 41s - loss: 1.7158 - accuracy: 0.4223 - val_loss: 1.4761 - val_accuracy:
0.4964 - 41s/epoch - 1s/step
Epoch 2/10
35/35 - 32s - loss: 1.5796 - accuracy: 0.4277 - val_loss: 1.4891 - val_accuracy:
0.4964 - 32s/epoch - 916ms/step
Epoch 3/10
35/35 - 33s - loss: 1.5800 - accuracy: 0.4277 - val_loss: 1.4784 - val_accuracy:
0.4964 - 33s/epoch - 940ms/step
Epoch 4/10
```

```
35/35 - 32s - loss: 1.5194 - accuracy: 0.4991 - val_loss: 1.4372 - val_accuracy:
0.5321 - 32s/epoch - 921ms/step
Epoch 5/10
35/35 - 34s - loss: 1.0772 - accuracy: 0.6946 - val_loss: 1.4788 - val_accuracy:
0.5107 - 34s/epoch - 958ms/step
Epoch 6/10
35/35 - 35s - loss: 0.8277 - accuracy: 0.7241 - val_loss: 1.4877 - val_accuracy:
0.5750 - 35s/epoch - 1s/step
Epoch 7/10
35/35 - 31s - loss: 0.6614 - accuracy: 0.7946 - val_loss: 2.9608 - val_accuracy:
0.5250 - 31s/epoch - 895ms/step
Epoch 8/10
35/35 - 30s - loss: 0.5993 - accuracy: 0.7893 - val_loss: 5.7760 - val_accuracy:
0.6750 - 30s/epoch - 846ms/step
Epoch 9/10
35/35 - 27s - loss: 0.5942 - accuracy: 0.8107 - val_loss: 1.5897 - val_accuracy:
0.5857 - 27s/epoch - 774ms/step
Epoch 10/10
35/35 - 29s - loss: 0.6350 - accuracy: 0.7795 - val_loss: 1.0341 - val_accuracy:
0.6286 - 29s/epoch - 836ms/step
```

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

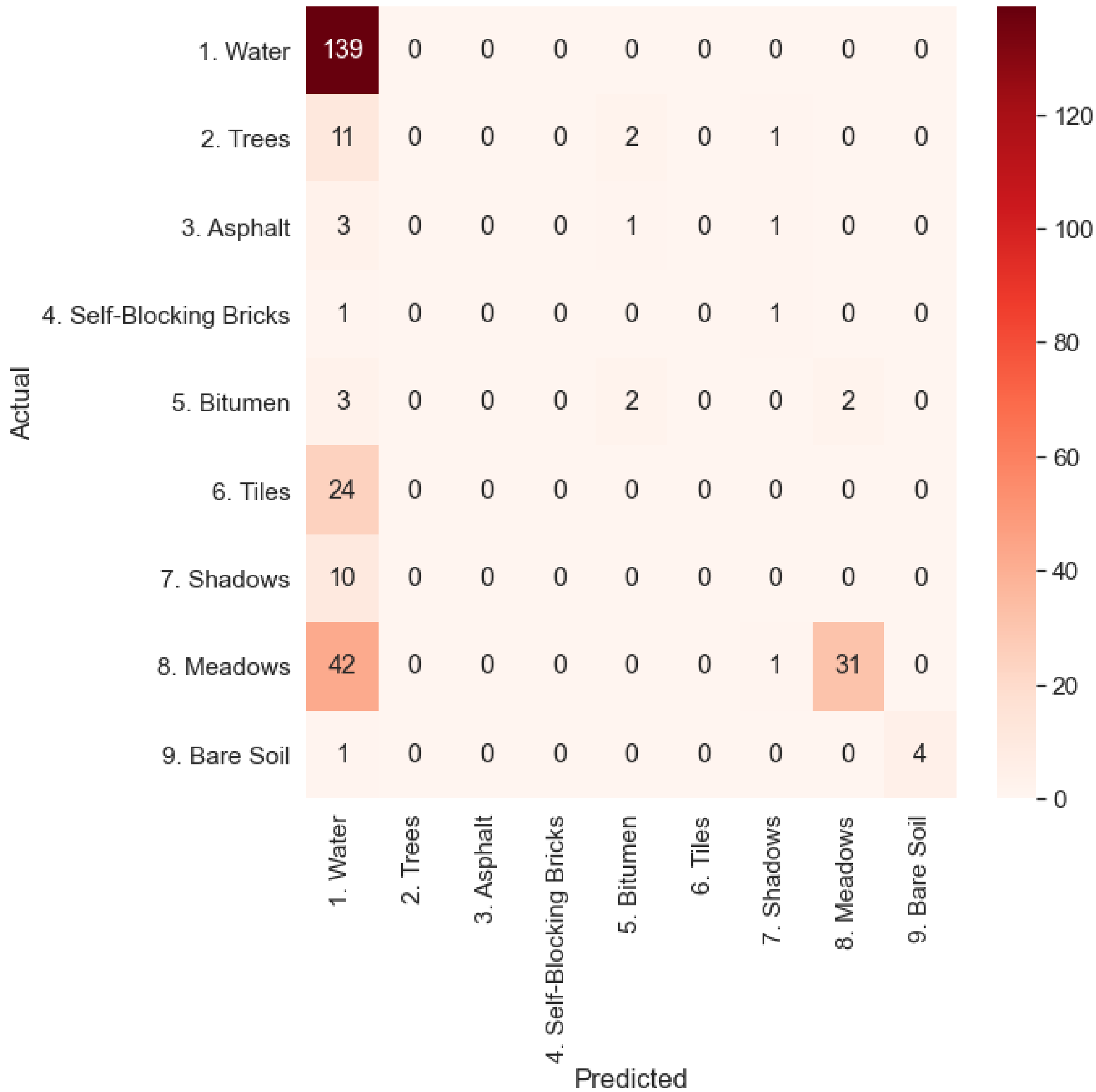

1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
128 0 0 0 0 0 0 0 0
6 9 2 0 0 0 0 0 0
3 6 5 0 0 0 0 0 0
0 4 5 0 0 0 0 0 0
0 0 3 0 6 0 0 0 0
14 1 0 0 0 0 0 0 0
0 5 1 0 0 0 3 0 0
5 0 0 0 1 0 0 72 0
0 0 0 0 0 0 0 0 1
120
100
80
60
40
20
0



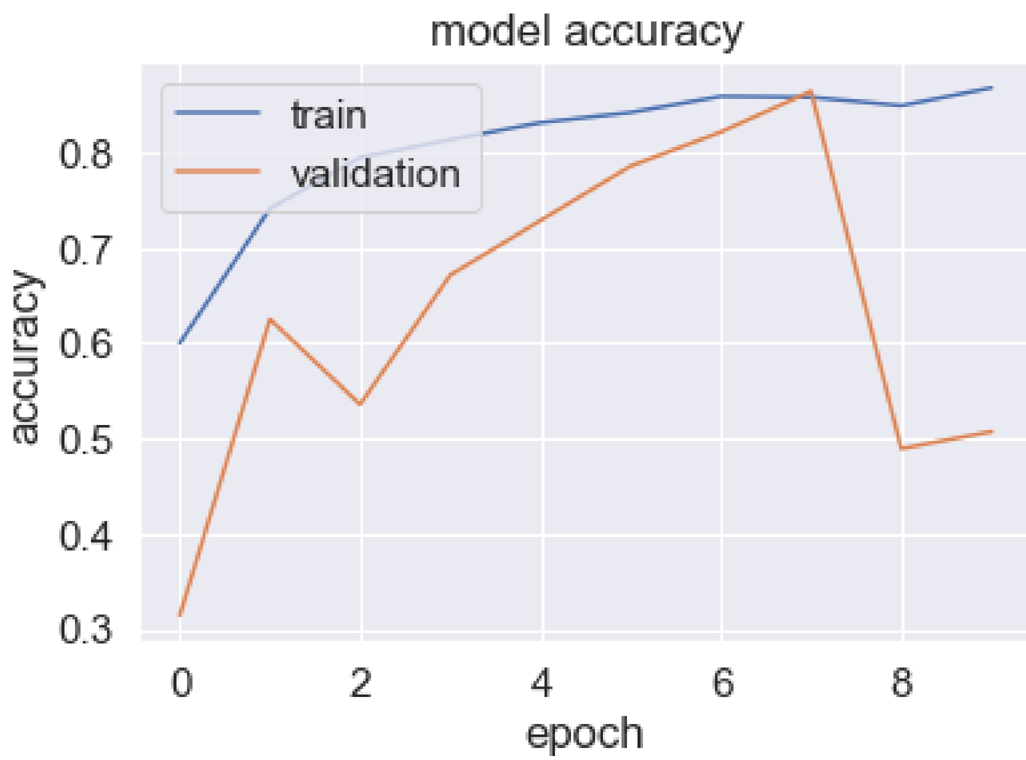

```
Score for fold 3: loss of 1.034110188484192; accuracy of 62.85714507102966%
9/9 [==============================] - 1s 30ms/step
[[139   0   0   0   0   0   0   0   0]
 [ 11   0   0   0   2   0   1   0   0]
 [  3   0   0   0   1   0   1   0   0]
 [  1   0   0   0   0   0   1   0   0]
 [  3   0   0   0   2   0   0   2   0]
 [ 24   0   0   0   0   0   0   0   0]
 [ 10   0   0   0   0   0   0   0   0]
 [ 42   0   0   0   0   0   1  31   0]
 [  1   0   0   0   0   0   0   0   4]]
------------------------------------------------------------------------
Training for fold 4 …
Epoch 1/10
35/35 - 32s - loss: 1.5064 - accuracy: 0.6000 - val_loss: 2.4409 - val_accuracy:
0.3143 - 32s/epoch - 919ms/step
Epoch 2/10
35/35 - 27s - loss: 0.8615 - accuracy: 0.7411 - val_loss: 1.4282 - val_accuracy:
0.6250 - 27s/epoch - 766ms/step
Epoch 3/10
35/35 - 26s - loss: 0.6919 - accuracy: 0.7946 - val_loss: 1.5018 - val_accuracy:
0.5357 - 26s/epoch - 744ms/step
Epoch 4/10
```

```
35/35 - 27s - loss: 0.6400 - accuracy: 0.8134 - val_loss: 1.1673 - val_accuracy:
0.6714 - 27s/epoch - 765ms/step
Epoch 5/10
35/35 - 24s - loss: 0.5821 - accuracy: 0.8313 - val_loss: 0.9015 - val_accuracy:
0.7286 - 24s/epoch - 699ms/step
Epoch 6/10
35/35 - 24s - loss: 0.5134 - accuracy: 0.8420 - val_loss: 0.7347 - val_accuracy:
0.7857 - 24s/epoch - 688ms/step
Epoch 7/10
35/35 - 25s - loss: 0.4471 - accuracy: 0.8589 - val_loss: 0.5822 - val_accuracy:
0.8214 - 25s/epoch - 710ms/step
Epoch 8/10
35/35 - 27s - loss: 0.4877 - accuracy: 0.8580 - val_loss: 0.4603 - val_accuracy:
0.8643 - 27s/epoch - 759ms/step
Epoch 9/10
35/35 - 25s - loss: 0.4765 - accuracy: 0.8491 - val_loss: 1.7734 - val_accuracy:
0.4893 - 25s/epoch - 702ms/step
Epoch 10/10
35/35 - 24s - loss: 0.4242 - accuracy: 0.8679 - val_loss: 6.3098 - val_accuracy:
0.5071 - 24s/epoch - 698ms/step
```

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



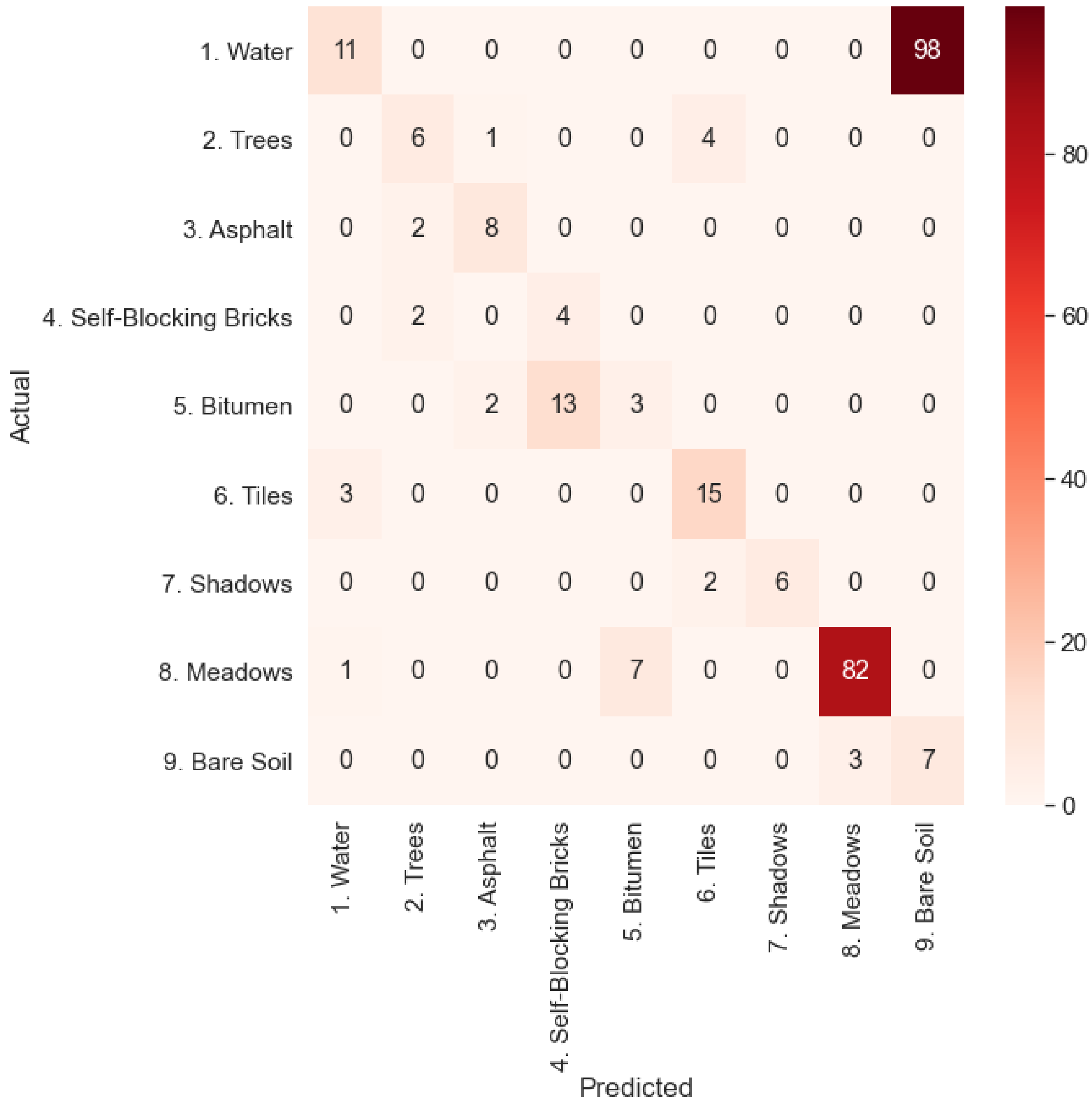
1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
139 0 0 0 0 0 0 0 0
11 0 0 0 2 0 1 0 0
3 0 0 0 1 0 1 0 0
1 0 0 0 0 0 1 0 0
3 0 0 0 2 0 0 2 0
24 0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 0 0
42 0 0 0 0 0 1 31 0
1 0 0 0 0 0 0 0 4
120
100
80
60
40
20
0

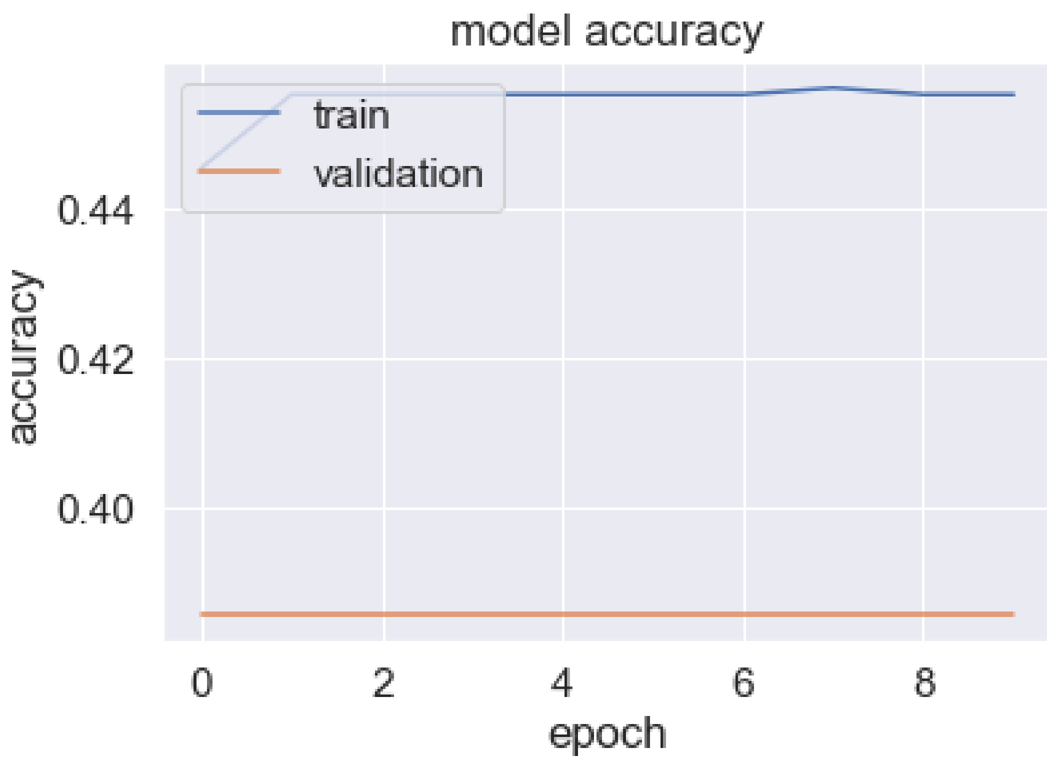



```
Score for fold 4: loss of 6.309820652008057; accuracy of 50.71428418159485%
9/9 [==============================] - 1s 31ms/step
[[11  0  0  0  0  0  0  0 98]
 [ 0  6  1  0  0  4  0  0  0]
 [ 0  2  8  0  0  0  0  0  0]
 [ 0  2  0  4  0  0  0  0  0]
 [ 0  0  2 13  3  0  0  0  0]
 [ 3  0  0  0  0 15  0  0  0]
 [ 0  0  0  0  0  2  6  0  0]
 [ 1  0  0  0  7  0  0 82  0]
 [ 0  0  0  0  0  0  0  3  7]]
------------------------------------------------------------------------
Training for fold 5 …
Epoch 1/10
35/35 - 31s - loss: 1.6524 - accuracy: 0.4455 - val_loss: 1.5890 - val_accuracy:
0.3857 - 31s/epoch - 889ms/step
Epoch 2/10
35/35 - 26s - loss: 1.5552 - accuracy: 0.4554 - val_loss: 1.6035 - val_accuracy:
0.3857 - 26s/epoch - 741ms/step
Epoch 3/10
35/35 - 27s - loss: 1.5538 - accuracy: 0.4554 - val_loss: 1.5756 - val_accuracy:
0.3857 - 27s/epoch - 777ms/step
Epoch 4/10
```

```
35/35 - 27s - loss: 1.5547 - accuracy: 0.4554 - val_loss: 1.5830 - val_accuracy:
0.3857 - 27s/epoch - 769ms/step
Epoch 5/10
35/35 - 25s - loss: 1.5551 - accuracy: 0.4554 - val_loss: 1.5815 - val_accuracy:
0.3857 - 25s/epoch - 706ms/step
Epoch 6/10
35/35 - 26s - loss: 1.5550 - accuracy: 0.4554 - val_loss: 1.5834 - val_accuracy:
0.3857 - 26s/epoch - 754ms/step
Epoch 7/10
35/35 - 28s - loss: 1.5509 - accuracy: 0.4554 - val_loss: 1.5718 - val_accuracy:
0.3857 - 28s/epoch - 795ms/step
Epoch 8/10
35/35 - 27s - loss: 1.5578 - accuracy: 0.4563 - val_loss: 1.5763 - val_accuracy:
0.3857 - 27s/epoch - 778ms/step
Epoch 9/10
35/35 - 27s - loss: 1.5544 - accuracy: 0.4554 - val_loss: 1.5793 - val_accuracy:
0.3857 - 27s/epoch - 763ms/step
Epoch 10/10
35/35 - 34s - loss: 1.5572 - accuracy: 0.4554 - val_loss: 1.5776 - val_accuracy:
0.3857 - 34s/epoch - 972ms/step
```

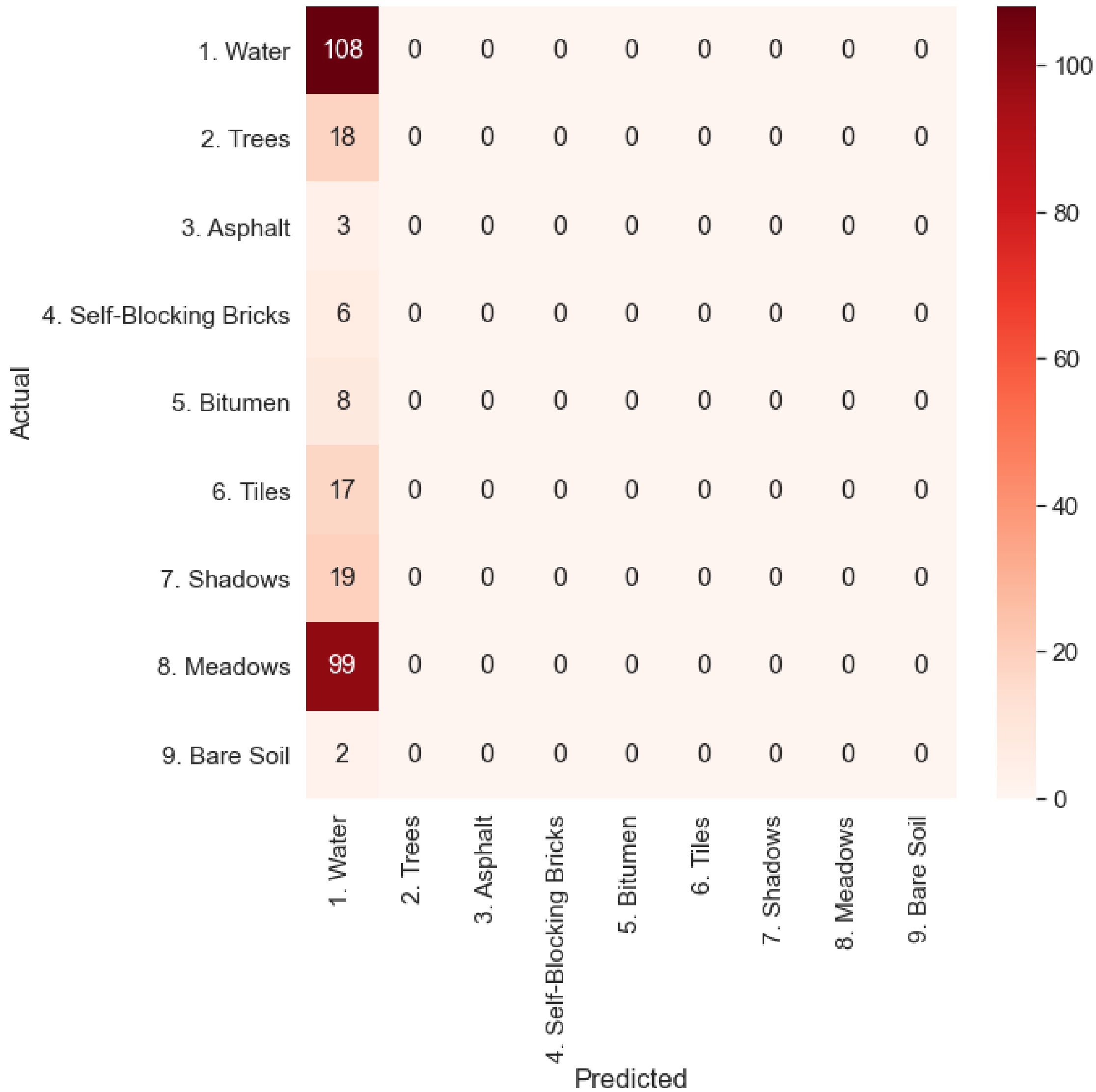
```
<Figure size 432x288 with 0 Axes>
```


1. Water
2. Trees
3. Asphalt
4. Self-Blocking Bricks
5. Bitumen
6. Tiles
7. Shadows
8. Meadows
9. Bare Soil
Actual
Predicted
11 0 0 0 0 0 0 0 98
0 6 1 0 0 4 0 0 0
0 2 8 0 0 0 0 0 0
0 2 0 4 0 0 0 0 0
0 0 2 13 3 0 0 0 0
3 0 0 0 0 15 0 0 0
0 0 0 0 0 2 6 0 0
1 0 0 0 7 0 0 82 0
0 0 0 0 0 0 0 3 7
80
60
40
20
0




```
Score for fold 5: loss of 1.5776281356811523; accuracy of 38.57142925262451%
9/9 [==============================] - 1s 43ms/step
[[108   0   0   0   0   0   0   0   0]
 [ 18   0   0   0   0   0   0   0   0]
 [  3   0   0   0   0   0   0   0   0]
 [  6   0   0   0   0   0   0   0   0]
 [  8   0   0   0   0   0   0   0   0]
 [ 17   0   0   0   0   0   0   0   0]
 [ 19   0   0   0   0   0   0   0   0]
 [ 99   0   0   0   0   0   0   0   0]
 [  2   0   0   0   0   0   0   0   0]]
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 1.7004808187484741 - Accuracy: 89.64285850524902%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5309717059135437 - Accuracy: 80.0000011920929%
------------------------------------------------------------------------
> Fold 3 - Loss: 1.034110188484192 - Accuracy: 62.85714507102966%
------------------------------------------------------------------------
> Fold 4 - Loss: 6.309820652008057 - Accuracy: 50.71428418159485%
------------------------------------------------------------------------
> Fold 5 - Loss: 1.5776281356811523 - Accuracy: 38.57142925262451%
```

```
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 64.35714364051819 (+- 18.640531349818097)
> Loss: 2.230602300167084
------------------------------------------------------------------------
Predicted Overall        1. Water  2. Trees  3. Asphalt  \
Actual Overall
1. Water                      520         0           0
2. Trees                       35        22           3
3. Asphalt                      9        10          13
4. Self-Blocking Bricks         7         7           5
5. Bitumen                     11         3           6
6. Tiles                       59         1           0
7. Shadows                     29         6           1
8. Meadows                    147         1           0
9. Bare Soil                    3         0           0

Predicted Overall        4. Self-Blocking Bricks  5. Bitumen  6. Tiles  \
Actual Overall
1. Water                                       0           0         0
2. Trees                                       0           5         4
3. Asphalt                                     0           5         1
4. Self-Blocking Bricks                        4           3         0
5. Bitumen                                    13          19         0
6. Tiles                                       0           1        30
7. Shadows                                     0           3         2
8. Meadows                                     0           9         0
9. Bare Soil                                   0           0         0

Predicted Overall        7. Shadows  8. Meadows  9. Bare Soil
Actual Overall
1. Water                          0           0            98
2. Trees                          1           0             0
3. Asphalt                        1           1             0
4. Self-Blocking Bricks           1           0             0
5. Bitumen                        1           3             0
6. Tiles                          0           0             0
7. Shadows                       14           0             0
8. Meadows                        1         262             0
9. Bare Soil                      0           3            17

<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
108
18
3
6
8
17
19
99
2
0
20
40
60
80
100

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
520 0 0 0 0 0 0 0 98
35 22 3 0 5 4 1 0 0
9 10 13 0 5 1 1 1 0
7 7 5 4 3 0 1 0 0
11 3 6 13 19 0 1 3 0
59 1 0 0 1 30 0 0 0
29 6 1 0 3 2 14 0 0
147 1 0 0 9 0 1 262 0
3 0 0 0 0 0 0 3 17
500
400
300
200
100
0