

# rueegg\_wissiak\_optimized

March 3, 2023

## 1 Optimized Model

This model does not show under- or overfitting and performs well on both, training and testing data. Afterwards, a brief description on how to tackle the challenges of an optimal model complexity.

To address underfitting, one approach is to increase the complexity of the model by adding more layers or increasing the number of filters in each layer. To address overfitting, we can try several approaches. One approach is to simplify the model by removing some layers or decreasing the number of filters in each layer. Another approach is to use less epochs for example.

Adding dropout or weight decay can help to address both of the above mentioned issues. We can also try adjusting the hyperparameters such as learning rate, batch size, or number of epochs.

```
[32]: import numpy as np
import matplotlib.pyplot as plt
```

```
[33]: from tensorflow.keras.preprocessing.image import ImageDataGenerator

img_size = 150
batch_size = 32

# TODO: add data augmentation for optimized model
train_datagen = ImageDataGenerator(
    rescale=1/255.,
    rotation_range=1,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.005,
    zoom_range=0.05,
    horizontal_flip=True,
    fill_mode='nearest'
)
test_datagen = ImageDataGenerator(rescale=1/255.)

train_generator = train_datagen.flow_from_directory(
    './dataset/seg_train/seg_train',
    target_size=(img_size, img_size),
    batch_size=batch_size,
    shuffle=True,
```

```

        class_mode='sparse'
    )
    test_generator = test_datagen.flow_from_directory(
        './dataset/seg_test/seg_test',
        target_size=(img_size, img_size),
        batch_size=batch_size,
        shuffle=True,
        class_mode='sparse'
    )
    labels = ['buildings', 'forest', 'glacier', 'mountain', 'sea']

    samples = train_generator.__next__()
    images = samples[0]
    target = samples[1]

    plt.figure(figsize = (10,10))
    for i in range(15):
        plt.subplot(5,5,i+1)
        plt.subplots_adjust(hspace=0.3,wspace=0.3)
        plt.imshow(images[i])
        plt.title(f"Class: {labels[int(target[i])]}")
        plt.axis('off')

```

Found 2666 images belonging to 5 classes.

Found 2499 images belonging to 5 classes.



## 1.1 Building the Model

Here we use the same model as the overfitting one but we add some extras to reduce the overfitting behavior.

### 1.1.1 Regularization

Regularization is used to reduce the impact of the weights. The weights then end up having less impact on the loss function which determines the error between the actual label and predicted label. This reduces complexity of the model and therefore reduces overfitting. We are adding regularization only to those layers which have the largest number of parameters according to the model summary. We are using L2 (Ridge) regularization since it is predetermined from the task.

Dropout Layers: The benefit of using dropout is no node in the network will be assigned with high parameter values, as a result the parameter values will be dispersed and the output of the current layer will not depend on a single node. E.g. Dropout(0.2) drops the input layers at a probability of 0.2.

### 1.1.2 Generalization

To improve generalization of the model, data augmentation is a useful tool. With data augmentation we can add artificial effects to the images such as shearing, stretching, flipping, rotating and translating. Through these effects, the images always appear differently each time they appear in the training step and therefore the CNN doesn't adapt to the exact images but rather learns about the relative features inside of an image.

### 1.1.3 Optimizer

For the optimized model we chose Adam over the competitors because it is the most common among SGD. We tried out SGD but it performed very poorly compared to Adam which might be due to insufficient configuration of the learning rate. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order (mean) and second-order (uncentered variance) moments. Its default implementation already provides a form of annealed learning,  $\beta_1=0.9$  for the first-order moment and  $\beta_2=0.999$  for the second-order moment.

### 1.1.4 Activation Function

The [following article](#) states that ReLU is the overall the best suited activation function so based on this we decided to use ReLU for our optimized model.

```
[34]: from tensorflow.keras.models import Sequential
      from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
      from tensorflow.keras.regularizers import l2
      from tensorflow.keras.optimizers import SGD
      from tensorflow.keras.optimizers.schedules import ExponentialDecay

      model = Sequential()
```

```

model.add(Conv2D(32, (3,3), input_shape= (img_size,img_size,3), activation = 'relu', padding = 'same')) #padding = same size output
model.add(MaxPooling2D())

model.add(Conv2D(64, (3,3), activation = 'relu', padding = 'same'))
model.add(MaxPooling2D())

model.add(Conv2D(128, (3,3), activation = 'relu', padding = 'same'))
model.add(MaxPooling2D())

model.add(Conv2D(256, (3,3), activation = 'relu', padding = 'same'))
model.add(MaxPooling2D())

model.add(Conv2D(512, (3,3), activation = 'relu', padding = 'same', kernel_regularizer=l2(1=0.0001)))
model.add(MaxPooling2D())
model.add(Dropout(0.05))

model.add(Conv2D(1024, (3,3), activation = 'relu', padding = 'same', kernel_regularizer=l2(1=0.001)))
model.add(MaxPooling2D())
model.add(Dropout(0.15))

model.add(Conv2D(512, (3,3), activation = 'relu', padding = 'same', kernel_regularizer=l2(1=0.001)))
model.add(MaxPooling2D())
model.add(Dropout(0.15))

model.add(Flatten())

model.add(Dense(256, activation='relu'))
model.add(Dense(5, activation = 'softmax'))

lr_schedule = ExponentialDecay(
    initial_learning_rate=1e-1,
    decay_rate=0.9975)
model.compile(optimizer = SGD(learning_rate=lr_schedule), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

```

[35]: `model.summary()`

Model: "sequential\_7"

Layer (type)	Output Shape	Param #
conv2d_49 (Conv2D)	(None, 150, 150, 32)	896

max_pooling2d_49 (MaxPoolin g2D)	(None, 75, 75, 32)	0
conv2d_50 (Conv2D)	(None, 75, 75, 64)	18496
max_pooling2d_50 (MaxPoolin g2D)	(None, 37, 37, 64)	0
conv2d_51 (Conv2D)	(None, 37, 37, 128)	73856
max_pooling2d_51 (MaxPoolin g2D)	(None, 18, 18, 128)	0
conv2d_52 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_52 (MaxPoolin g2D)	(None, 9, 9, 256)	0
conv2d_53 (Conv2D)	(None, 9, 9, 512)	1180160
max_pooling2d_53 (MaxPoolin g2D)	(None, 4, 4, 512)	0
dropout_21 (Dropout)	(None, 4, 4, 512)	0
conv2d_54 (Conv2D)	(None, 4, 4, 1024)	4719616
max_pooling2d_54 (MaxPoolin g2D)	(None, 2, 2, 1024)	0
dropout_22 (Dropout)	(None, 2, 2, 1024)	0
conv2d_55 (Conv2D)	(None, 2, 2, 512)	4719104
max_pooling2d_55 (MaxPoolin g2D)	(None, 1, 1, 512)	0
dropout_23 (Dropout)	(None, 1, 1, 512)	0
flatten_7 (Flatten)	(None, 512)	0
dense_14 (Dense)	(None, 256)	131328
dense_15 (Dense)	(None, 5)	1285

=====

Total params: 11,139,909

Trainable params: 11,139,909  
Non-trainable params: 0

-----

## 1.2 Training the Model

```
[36]: history = model.fit(train_generator, validation_data=test_generator, epochs=20)
```

Epoch 1/20

84/84 [=====] - 103s 1s/step - loss: 2.9643 - accuracy: 0.2442 - val\_loss: 2.8456 - val\_accuracy: 0.4822

Epoch 2/20

84/84 [=====] - 103s 1s/step - loss: 2.8566 - accuracy: 0.3398 - val\_loss: 2.8535 - val\_accuracy: 0.3842

Epoch 3/20

84/84 [=====] - 103s 1s/step - loss: 2.8815 - accuracy: 0.2412 - val\_loss: 2.8776 - val\_accuracy: 0.2613

Epoch 4/20

84/84 [=====] - 103s 1s/step - loss: 2.8196 - accuracy: 0.3016 - val\_loss: 2.8339 - val\_accuracy: 0.2721

Epoch 5/20

84/84 [=====] - 103s 1s/step - loss: 2.7020 - accuracy: 0.3567 - val\_loss: 2.4897 - val\_accuracy: 0.5354

Epoch 6/20

84/84 [=====] - 121s 1s/step - loss: 2.5957 - accuracy: 0.4010 - val\_loss: 2.3386 - val\_accuracy: 0.5614

Epoch 7/20

84/84 [=====] - 104s 1s/step - loss: 2.4488 - accuracy: 0.4527 - val\_loss: 2.2796 - val\_accuracy: 0.5454

Epoch 8/20

84/84 [=====] - 103s 1s/step - loss: 2.2671 - accuracy: 0.5364 - val\_loss: 2.1110 - val\_accuracy: 0.5874

Epoch 9/20

84/84 [=====] - 103s 1s/step - loss: 2.1310 - accuracy: 0.5709 - val\_loss: 1.9847 - val\_accuracy: 0.6363

Epoch 10/20

84/84 [=====] - 102s 1s/step - loss: 2.0894 - accuracy: 0.5833 - val\_loss: 2.0132 - val\_accuracy: 0.5534

Epoch 11/20

84/84 [=====] - 103s 1s/step - loss: 1.9831 - accuracy: 0.5949 - val\_loss: 2.0470 - val\_accuracy: 0.5642

Epoch 12/20

84/84 [=====] - 103s 1s/step - loss: 1.9244 - accuracy: 0.6035 - val\_loss: 1.8459 - val\_accuracy: 0.6355

Epoch 13/20

84/84 [=====] - 103s 1s/step - loss: 1.8550 - accuracy: 0.6309 - val\_loss: 1.8815 - val\_accuracy: 0.6230

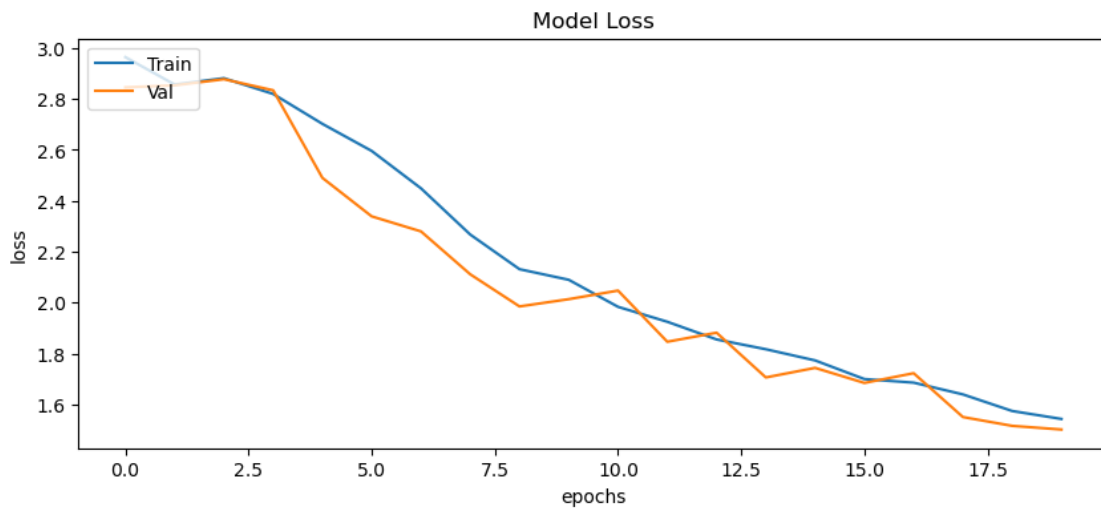
Epoch 14/20

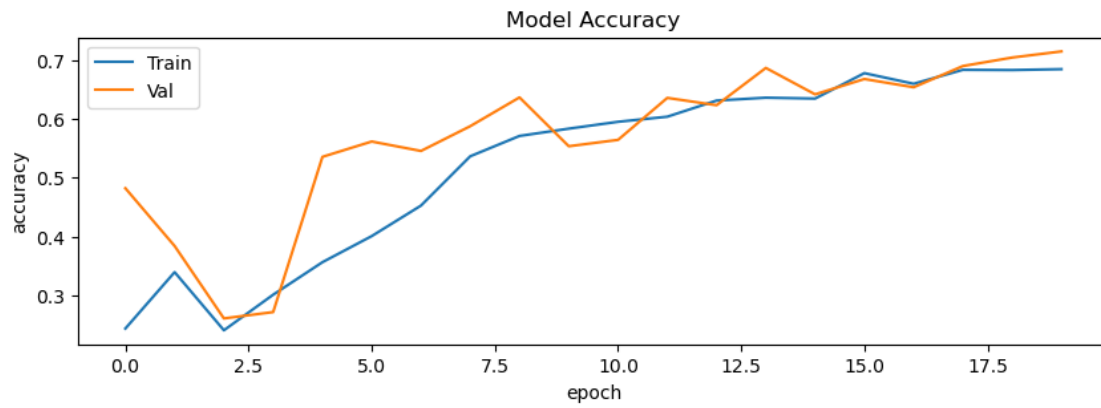
```

84/84 [=====] - 103s 1s/step - loss: 1.8165 - accuracy:
0.6358 - val_loss: 1.7057 - val_accuracy: 0.6863
Epoch 15/20
84/84 [=====] - 103s 1s/step - loss: 1.7726 - accuracy:
0.6343 - val_loss: 1.7432 - val_accuracy: 0.6415
Epoch 16/20
84/84 [=====] - 102s 1s/step - loss: 1.6993 - accuracy:
0.6774 - val_loss: 1.6841 - val_accuracy: 0.6675
Epoch 17/20
84/84 [=====] - 102s 1s/step - loss: 1.6852 - accuracy:
0.6594 - val_loss: 1.7223 - val_accuracy: 0.6535
Epoch 18/20
84/84 [=====] - 1047s 13s/step - loss: 1.6390 -
accuracy: 0.6830 - val_loss: 1.5501 - val_accuracy: 0.6895
Epoch 19/20
84/84 [=====] - 101s 1s/step - loss: 1.5738 - accuracy:
0.6827 - val_loss: 1.5154 - val_accuracy: 0.7039
Epoch 20/20
84/84 [=====] - 103s 1s/step - loss: 1.5426 - accuracy:
0.6842 - val_loss: 1.5010 - val_accuracy: 0.7143

```

```
[37]: %run rueegg_wissiak_model_visualization.ipynb
```





```
[38]: %run rueegg_wissiak_model_evaluation.ipynb
```

```
79/79 [=====] - 13s 168ms/step
Predicted classes: [2 3 1 ... 2 2 1]
True labels: [0 0 0 ... 4 4 4]
Accuracy:
0.2036814725890356
```