pokegpt

March 19, 2023

Names: Fabio Lenherr & Kaj Habegger

0.1 Install required libraries

```
[ ]: # This is simply a workaround if the pipenv is not properly loaded
     ! pip install tensorflow
     ! pip install matplotlib
     ! pip install scikit-learn
```

1 Pokegpt

1.1 Idea

Our idea was to create a classification model that will determine the pokemon type based on the image provided. We do this with the data of the first generation to keep both the image amount and type amount limited. We fully understand, that this model will not be 100% correct, as pokemon will not always have a type that is solely dependent on the color or body. In other words, with some pokemon the creators were a bit too creative with their interpretation of how a type should be represented.

1.2 Notes

- There is a problem with the pokemon that we needed to solve. Our model is designed to end up with one classification, however there are many pokemon with a dual-type. For example one pokemon might have the type Water+Ground. In this case we chose the first type for

the vast majority of pokemon, with a few exceptions where we found that the secondary type makes more sense.

- Some classes do not feature enough pokemon to be included for the assignment, these classes were simply removed. Pokemon featuring dual-types with a removed type will simply live on in the other class should that be included.

1.3 Acknowledgements

- Dataset 1 Pokemon Generation One by HarshitDwivedi, License GPL2.0
- Dataset 2 Pokemon Images, First Generation(17000 files) by Mikołaj Kolman, License GPL2.0
- Template for notebook Basic classification: Classify images of clothing by François Chollet, License Apache 2.0

1.4 ---

1.5 Setup

1.5.1 Imports

[1]:
```
# TensorFlow and tf.keras
import tensorflow as tf
import tensorflow.keras as keras
import keras.layers as layers
from sklearn.model_selection import KFold

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

# Sklearn for Confusion Matrix
from sklearn.metrics import confusion_matrix

# os for environment variables
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"]="2"


print(tf.__version__)
```

```
2.10.1
```

1.5.2 GPU Detection

[2]:
```
# Check whether GPU is available for ML or not
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

1.5.3 Set Parameters

[3]:
```
train_dir = 'datasets/dataset_training/'
test_dir = 'datasets/dataset_testing/'
batch_size = 32
img_height = 64
img_width = 64
```

1.5.4 Define function for Confusion Matrix

[4]:
```
def plt_confmatrix(y_preds, y_trues, class_names):
    conf_matrix = confusion_matrix(y_trues, y_preds, labels=class_names)
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.matshow(conf_matrix)

    ax.xaxis.set_major_locator(ticker.FixedLocator(range(-1, len(class_names))))
    ax.yaxis.set_major_locator(ticker.FixedLocator(range(-1, len(class_names))))
    ax.set_xticklabels([''] + class_names)
    ax.set_yticklabels([''] + class_names)

    for i in range(conf_matrix.shape[0]):
        for j in range(conf_matrix.shape[1]):
            ax.text(x=j, y=i, s=conf_matrix[i, j], va='center', ha='center',␣
 ↪size='xx-large')

    plt.xlabel('Predictions')
    plt.ylabel('Actuals')
    plt.title('Confusion Matrix')
    plt.show()
```

1.5.5 Load base data

[5]:
```
# We split the base data into validation and testing data.
training_ds, validation_ds = keras.utils.image_dataset_from_directory(
    train_dir,
    labels="inferred", # classes can be inferred from folders. We have this␣
 ↪structured for this specifically.
    color_mode="rgb",
    batch_size=batch_size,
    image_size=(img_height, img_width), # Some preprocessing happening here,␣
 ↪resizing the images
    subset="both",
    seed=24,
    validation_split=0.2
)
```

```
# Training set for kfold-cross validation. Here we want no split as we use␣
 ↪folds as validation data.
o_training_ds = keras.utils.image_dataset_from_directory(
    train_dir,
    labels="inferred",
    color_mode="rgb",
    batch_size=batch_size,
    image_size=(img_height, img_width), # Some preprocessing happening here,␣
 ↪resizing the images
)

# The testing set is made of a completely different dataset, however it still␣
 ↪features the 1.generation of pokemon
testing_ds = keras.utils.image_dataset_from_directory(
    test_dir,
    labels="inferred",
    color_mode="rgb",
    batch_size=batch_size,
    image_size=(img_height, img_width) # Some preprocessing happening here,␣
 ↪resizing the images
)
```

```
Found 16276 files belonging to 11 classes.
Using 13021 files for training.
Using 3255 files for validation.
Found 16276 files belonging to 11 classes.
Found 9979 files belonging to 11 classes.
```

1.5.6 Data information and visualization

[6]:
```
# Here we simply retrieve all class names that we have available
class_names = training_ds.class_names
num_classes = len(training_ds.class_names)
print(class_names)
# as shown below, some types are left out as they had too few pictures. -> Ice,␣
 ↪Ghost, etc.
```

```
['bug', 'electric', 'fire', 'fly', 'grass', 'ground', 'normal', 'poison',
'psychic', 'rock', 'water']
```

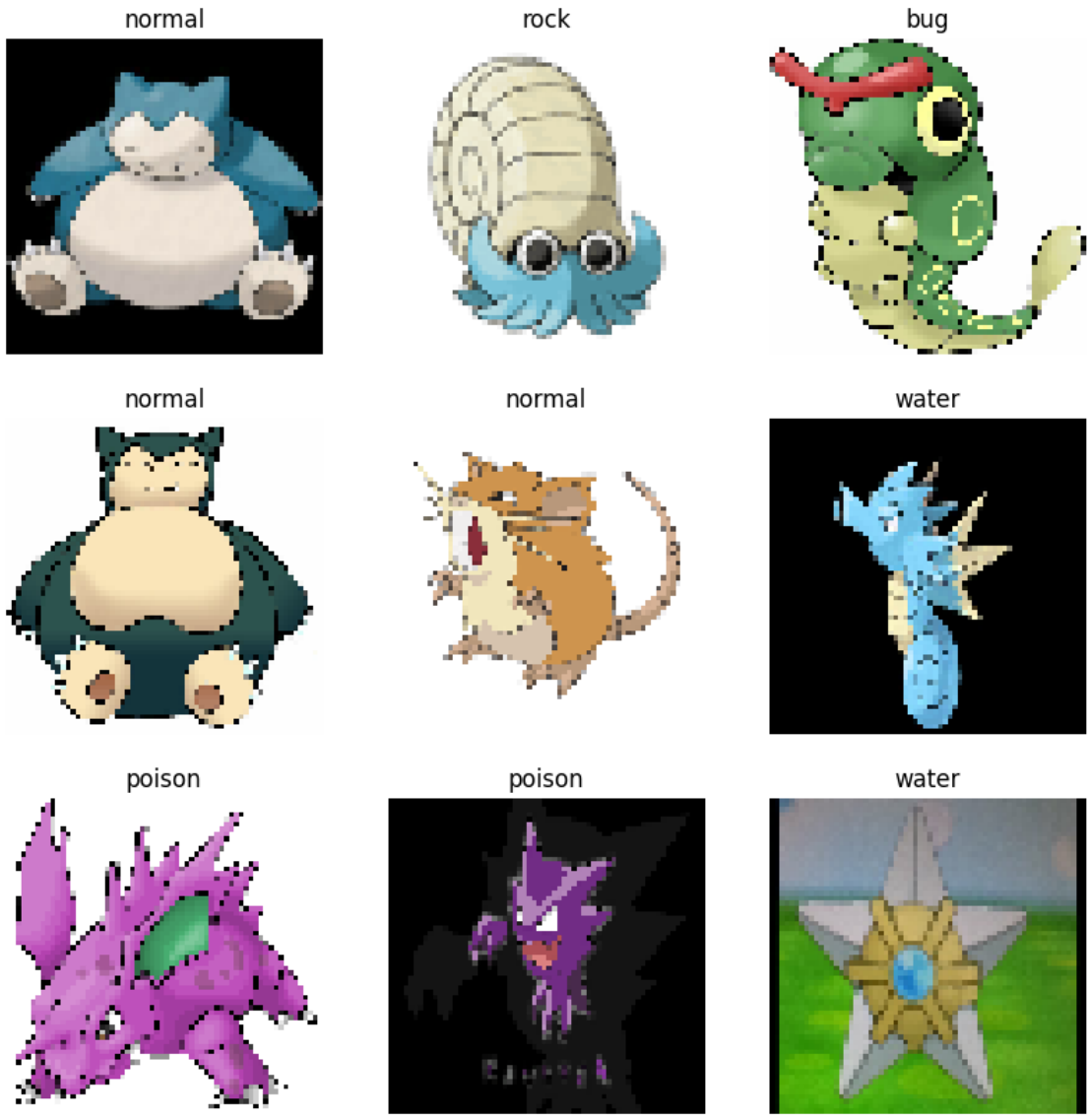
1.5.7 Configure the datasets for performance

[7]:
```
AUTOTUNE = tf.data.AUTOTUNE

training_ds = training_ds.cache().prefetch(buffer_size=AUTOTUNE)
validation_ds = validation_ds.cache().prefetch(buffer_size=AUTOTUNE)
o_training_ds = o_training_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

```
testing_ds = testing_ds.cache().prefetch(buffer_size=AUTOTUNE)

# Dataset.cache keeps the images in memory after they're loaded off disk during␣
 ↪the first epoch.
# This will ensure the dataset does not become a bottleneck while training your␣
 ↪model.
# If your dataset is too large to fit into memory, you can also use this method␣
 ↪to create a performant on-disk cache.

# Dataset.prefetch overlaps data preprocessing and model execution while␣
 ↪training
```

1.5.8 Visualization of Classes

[8]:
```
# Example figures for the classes.
plt.figure(figsize=(10, 10))
for images, labels in training_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```



1.5.9 Visualization of Tensor

[9]:
```
# Shape of each Batch
for image_batch, labels_batch in training_ds:
    print(image_batch.shape)
    print(labels_batch.shape)
    break
```

```
(32, 64, 64, 3)
(32,)
```

1.6 _____

1.7 Underfitted Model

The underfitted model features only a single convolutional layer, a max pooling and a dense layer at the end. This is model is too simple to properly learn from the information provided, this means that we will end up with bad accuracy.

1.7.1 Model Definition

[10]:
```
num_epochs = 30

underfitted_model = keras.Sequential([
    layers.Rescaling(1./63), # Some preprocessing happening here, normalizing␣
 ↪the data
    layers.Conv2D(1, 30, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(num_classes, activation='softmax')
])
```

1.7.2 Model Complilation

[11]:
```
underfitted_model.compile(
    optimizer='adam', # adam optimizer for better optimiztion
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['acc']
)
```

1.7.3 Model Fitting

[ ]:
```
underfitted_history = underfitted_model.fit(
    training_ds,
    validation_data=validation_ds,
    epochs=num_epochs,
    use_multiprocessing=True
)
```

1.7.4 Model Summary

[13]:
```
# While the parameters are still plenty compared to the minimum requirement,
# it is important to note that we 11 classes.
# This will result in more parameters due to the bigger dense layer.
underfitted_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
```

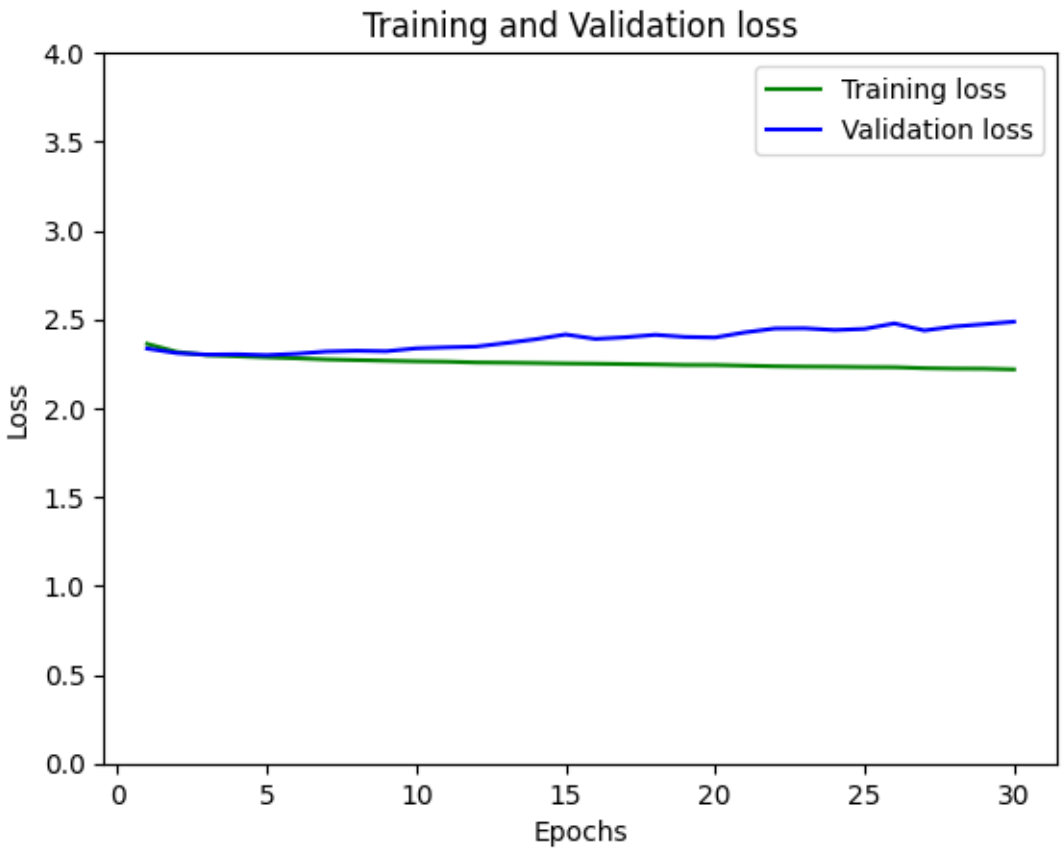
```
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling (Rescaling)       (None, 64, 64, 3)         0

 conv2d (Conv2D)             (None, 35, 35, 1)         2701

 max_pooling2d (MaxPooling2D  (None, 17, 17, 1)        0
 )

 flatten (Flatten)           (None, 289)               0

 dense (Dense)               (None, 11)                3190

=================================================================
Total params: 5,891
Trainable params: 5,891
Non-trainable params: 0
_________________________________________________________________
```

1.7.5 Loss and Accuracy calculation

[14]:
```
# Define variables used in data visualization for the underfitted model.
acc = underfitted_history.history['acc']
val_acc = underfitted_history.history['val_acc']
loss = underfitted_history.history['loss']
val_loss = underfitted_history.history['val_loss']
```

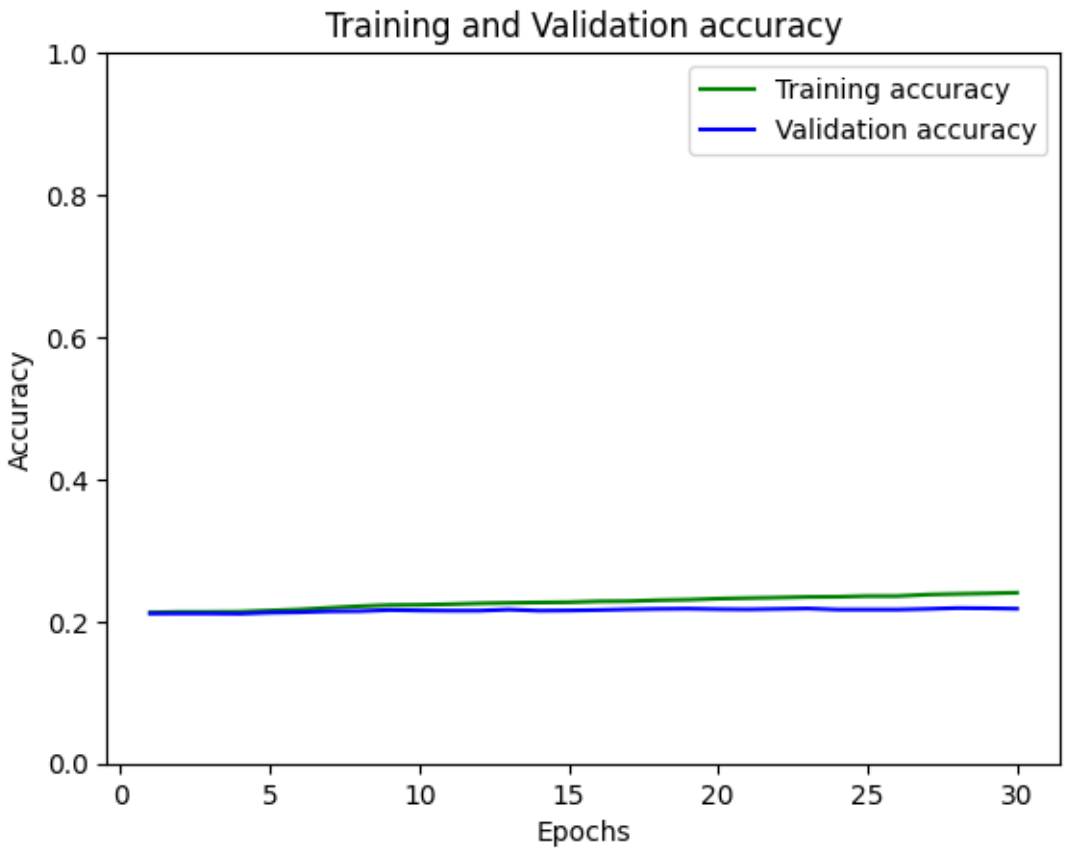
1.7.6 Loss

[15]:
```
# Plotting training and validation loss
epochs = range(1,num_epochs+1)
plt.plot(epochs, loss, 'g', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.ylim([0, 4])
plt.legend()
plt.show()
```



1.7.7 Accuracy

[16]:
```
# Plotting training and validation accuracy
plt.plot(epochs, acc, 'g', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend()
plt.show()
```

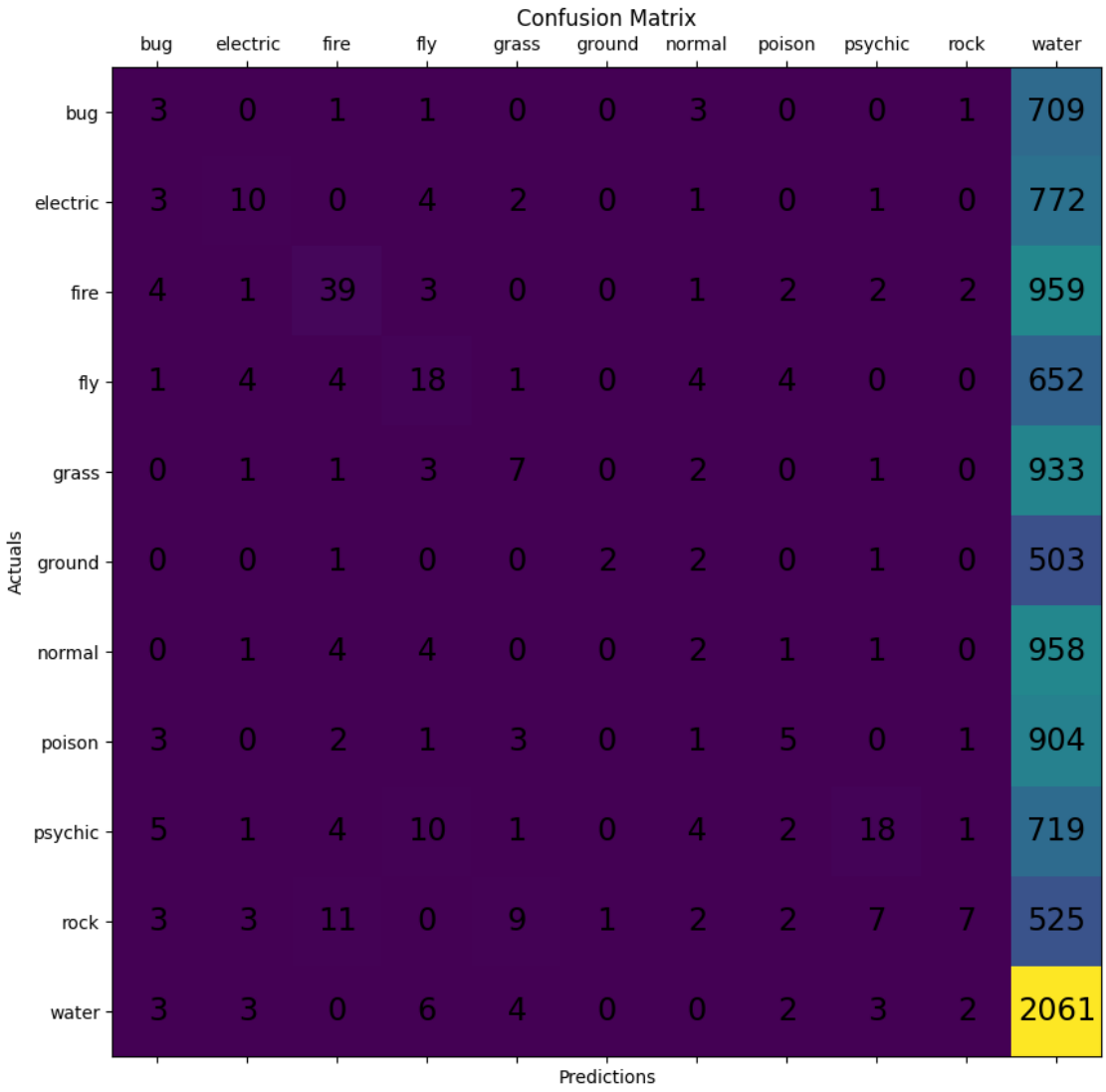


1.7.8 Calculation for confusion matrix

[ ]:
```
preds = np.empty((0, 2), dtype=int)
for images, labels in iter(testing_ds):
    pred = underfitted_model.predict(images.numpy().astype("uint8"))
    for i in range(len(pred)):
        y_pred = np.argmax(pred[i])
        preds = np.append(preds, [[class_names[labels[i]],␣
 ↪class_names[y_pred]]], axis=0)

y_preds = preds[:, 1]
y_trues = preds[:,0]
```

1.7.9 Confusion Matrix

[18]:
```
plt_confmatrix(y_preds, y_trues, class_names)
```



1.7.10 Discussion

Water gets most often predicted. That's because our dataset has a higher amount of water typed pokemons. But as we can see the model is still underfitting as obviously nothing/not enough is learned from the data.

1.8 _____

1.9 Overfitted Model

The overfitted model will feature too many convolutional layers and dense layers, this means we will likely get a really high accuracy with our training data, but our validation/testing data will have horrible results!

1.9.1 Model Definition

[19]:
```
num_epochs = 40

overfitted_model = keras.Sequential([
        layers.Rescaling(1./63), # Some preprocessing happening here,␣
 ↪normalizing the data
        layers.Conv2D(64, 2, activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(128, 2, activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(256, 2, activation='relu'),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(500),
        layers.Dense(11, activation='softmax') # Amount of neurons risen to␣
 ↪achieve ovrfitting
])
```

1.9.2 Model Compilation

[20]:
```
overfitted_model.compile(
    optimizer='adam',
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['acc']
)
```

1.9.3 Model Fitting

[ ]:
```
overfitted_history = overfitted_model.fit(
    training_ds,
    validation_data=validation_ds,
    epochs=num_epochs,
    use_multiprocessing=True
)
```

1.9.4 Model Summary

[22]:
```
# There is a significant increase in parameters compared to the underfitted␣
 ↪model.
# This is expected, as this model has more convolutional and dense layers.
overfitted_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
```

```
 rescaling_1 (Rescaling)     (None, 64, 64, 3)         0

 conv2d_1 (Conv2D)           (None, 63, 63, 64)        832

 max_pooling2d_1 (MaxPooling  (None, 31, 31, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 30, 30, 128)       32896

 max_pooling2d_2 (MaxPooling  (None, 15, 15, 128)      0
 2D)

 conv2d_3 (Conv2D)           (None, 14, 14, 256)       131328

 max_pooling2d_3 (MaxPooling  (None, 7, 7, 256)        0
 2D)

 flatten_1 (Flatten)         (None, 12544)             0

 dense_1 (Dense)             (None, 500)               6272500

 dense_2 (Dense)             (None, 11)                5511

=================================================================
Total params: 6,443,067
Trainable params: 6,443,067
Non-trainable params: 0
_________________________________________________________________
```

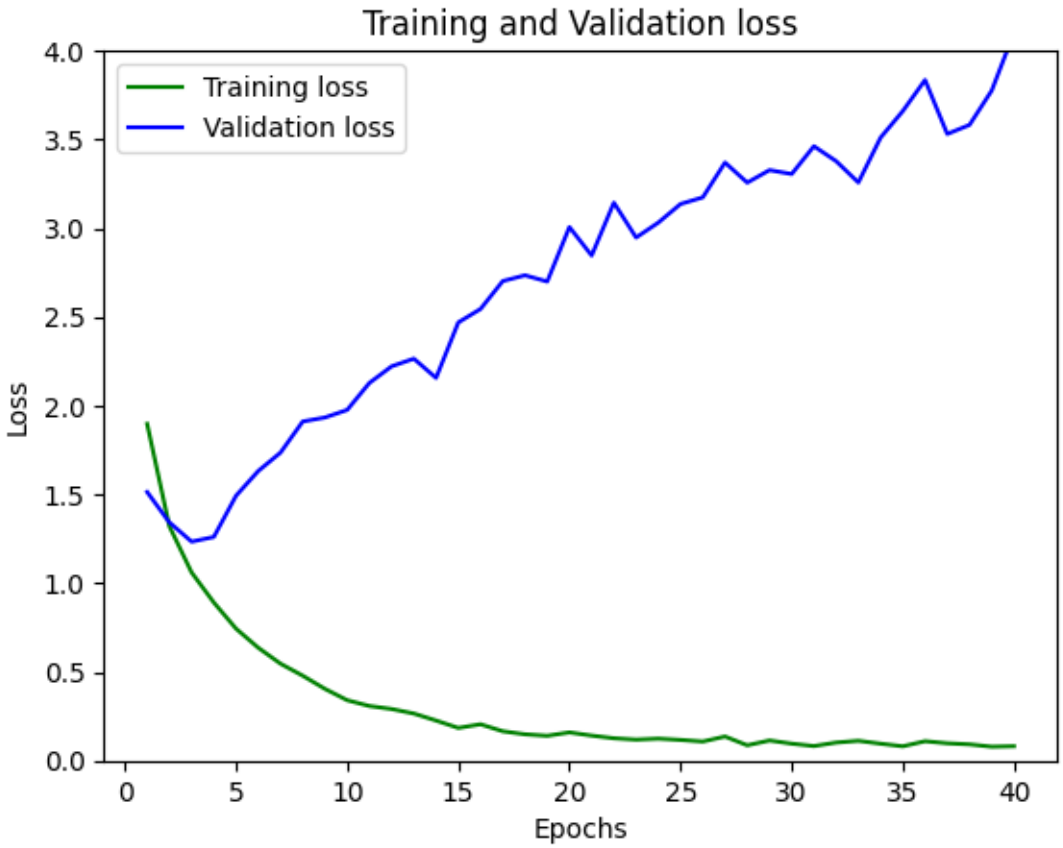
1.9.5 Loss and Accuracy calculation

[23]:
```
acc = overfitted_history.history['acc']
val_acc = overfitted_history.history['val_acc']
loss = overfitted_history.history['loss']
val_loss = overfitted_history.history['val_loss']
```

1.9.6 Loss

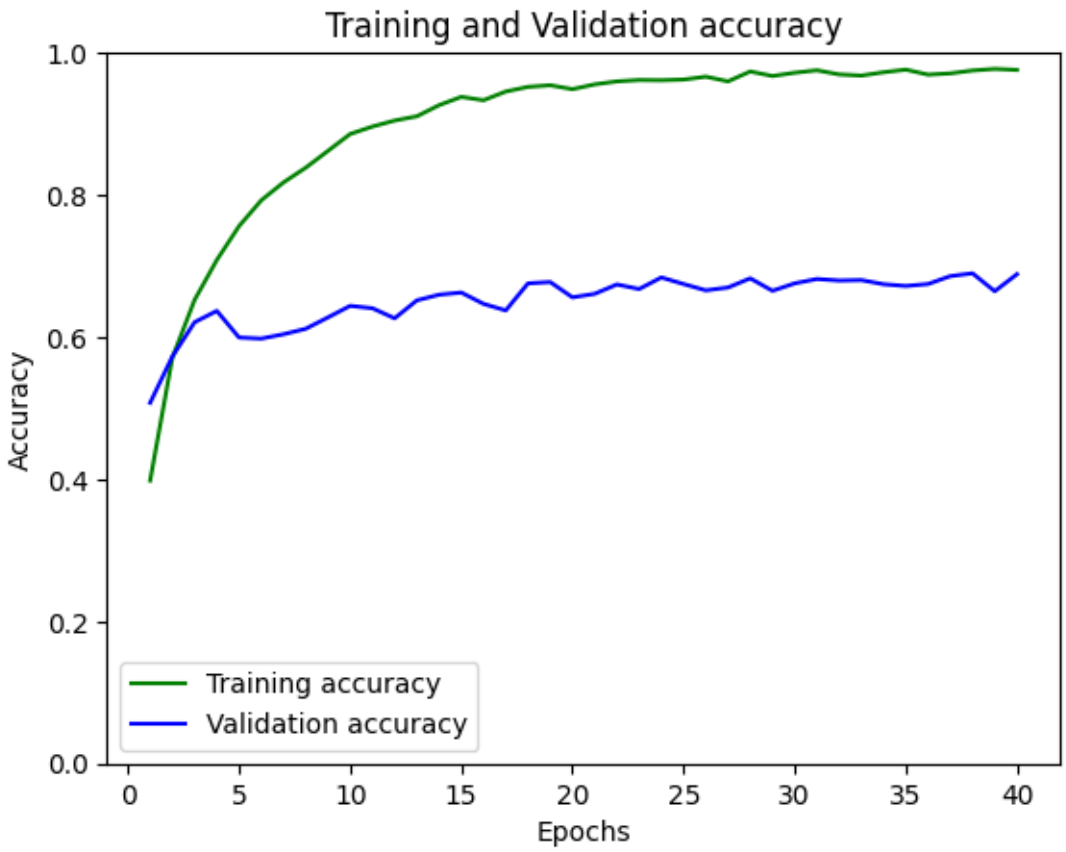
[24]:
```
# Plotting training and validation loss
# As expected, the validation loss is actually increasing, while the training␣
 ↪loss is minimalized.
epochs = range(1,num_epochs+1)
plt.plot(epochs, loss, 'g', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

```
plt.ylim([0, 4])
plt.legend()
plt.show()
```



1.9.7 Accuracy

[25]:
```
# Plotting training and validation accuracy
# The accurary is comparable to the loss above.
# While the training accuracy is good, the validation accuracy is horrible.
# It is important to note however, that the validation accuracy is somewhat␣
 ↪stable, while the loss is increasing!
plt.plot(epochs, acc, 'g', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend()
plt.show()
```

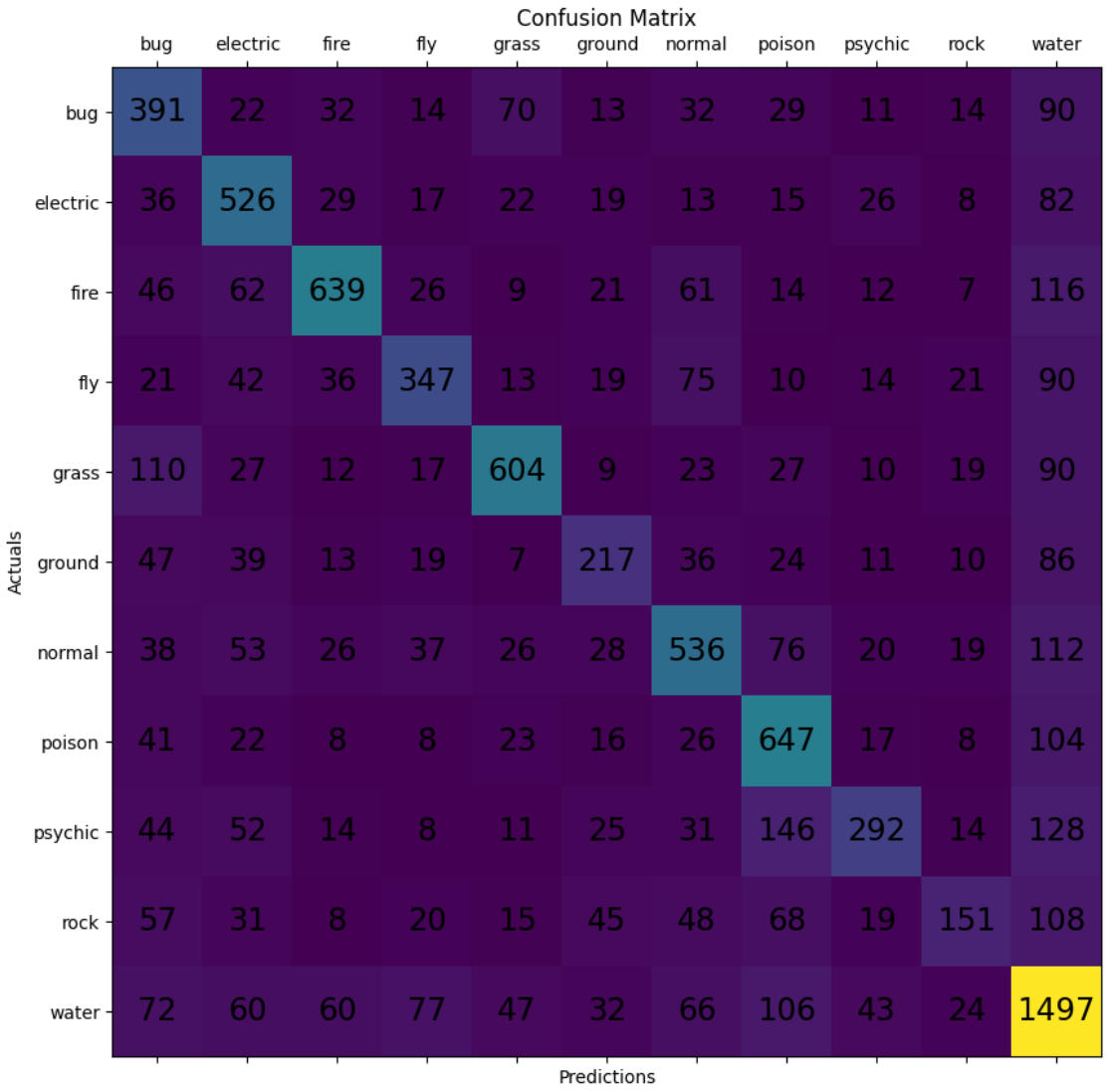


1.9.8 Calculation for Confusion Matrix

[ ]:
```
preds = np.empty((0, 2), dtype=int)
for images, labels in iter(testing_ds):
    pred = overfitted_model.predict(images.numpy().astype("uint8"))
    for i in range(len(pred)):
        y_pred = np.argmax(pred[i])
        preds = np.append(preds, [[class_names[labels[i]],␣
 ↪class_names[y_pred]]], axis=0)

y_preds = preds[:, 1]
y_trues = preds[:,0]
```

1.9.9 Confusion Matrix

[27]:
```
plt_confmatrix(y_preds, y_trues, class_names)
```



1.9.10 Discussion

As expected, the overfitted model did well on the training data, extremely well in fact! However, the validation data never got over 0.8 accuracy and the loss increased over time. This is expected in the overfitted model. What is however interesting, is that the confusion matrix that is based on the testing data is actually quite ok overall.

1.10 ———

1.11 Optimized Model

1.11.1 In here we will use kfold cross validation to ensure that our model is not overffitting or underfitting, the expectation therefore is a good score in both train/validation accuracy as well as test accuracy.

1.11.2 Reload data without the datasplit, as kfold cross validation takes care of this instead!

1.11.3 Kfold Parameters & Regularizer

[28]:
```
num_epochs = 20
splits = 5
scores = [None] * splits * 3
models = [None] * splits * 3
reg_params = [0.01, 0.001, 0.0001]
```

1.11.4 Setup for Kfold

[29]:
```
train_images = np.concatenate(list(o_training_ds.map(lambda x, y:x)))
train_labels = np.concatenate(list(o_training_ds.map(lambda x, y:y)))

inputs = train_images
targets = train_labels


kfold = KFold(n_splits=splits, shuffle=True)
iteration = 0
print("kfold setup done")
```

```
kfold setup done
```

1.11.5 KFold

[ ]:
```
print("Starting kfold model evaluation")
for train, val in kfold.split(inputs, targets):
    for reg_param in reg_params:
        regularizer = keras.regularizers.l2(reg_param)

        optimized_model = keras.Sequential([
            layers.Rescaling(1./63), # Some preprocessing happening here,␣
 ↪normalizing the data
            layers.Conv2D(64, 5, activation='relu'),
            layers.MaxPooling2D(),
            layers.Conv2D(128, 5, activation='relu',␣
 ↪kernel_regularizer=regularizer),
```

```
            layers.MaxPooling2D(),
            layers.Conv2D(256, 5, activation='relu',␣
 ↪kernel_regularizer=regularizer),
            layers.Dropout(0.5),
            layers.MaxPooling2D(),
            layers.Flatten(),
            layers.Dense(num_classes, activation='softmax')
        ])

        print("fitting model: " + f"{iteration}")

        optimized_model.compile(
            optimizer='adam',
            loss=keras.losses.SparseCategoricalCrossentropy(from_logits=False),
            metrics=['acc']
        )

        optimized_history = optimized_model.fit(
            inputs[train],
            targets[train],
            validation_data=(inputs[val], targets[val]),
            epochs=20
        )

        models[iteration] = optimized_model

        scores[iteration] = optimized_model.evaluate(testing_ds, verbose=0)
        iteration += 1

print(scores)
```

1.11.6 Retrain the best model with all data and use this as the optimized model

[ ]:
```
best_model_score = 0
best_model = 0
iteration = 0

for selected_model in scores:
    if selected_model[1] > best_model_score:
        best_model_score = selected_model[1]
        best_model = iteration
    iteration += 1

optimized_model = models[best_model]

optimized_history = optimized_model.fit(
    training_ds,
```

```
    validation_data=validation_ds,
    epochs=num_epochs
)
```

1.11.7 Model Summary

[32]:
```
# The expectation of our optimized model is that we are between the amount of␣
 ↪parameters of the underfitted and overfitted model.
optimized_model.summary()
```

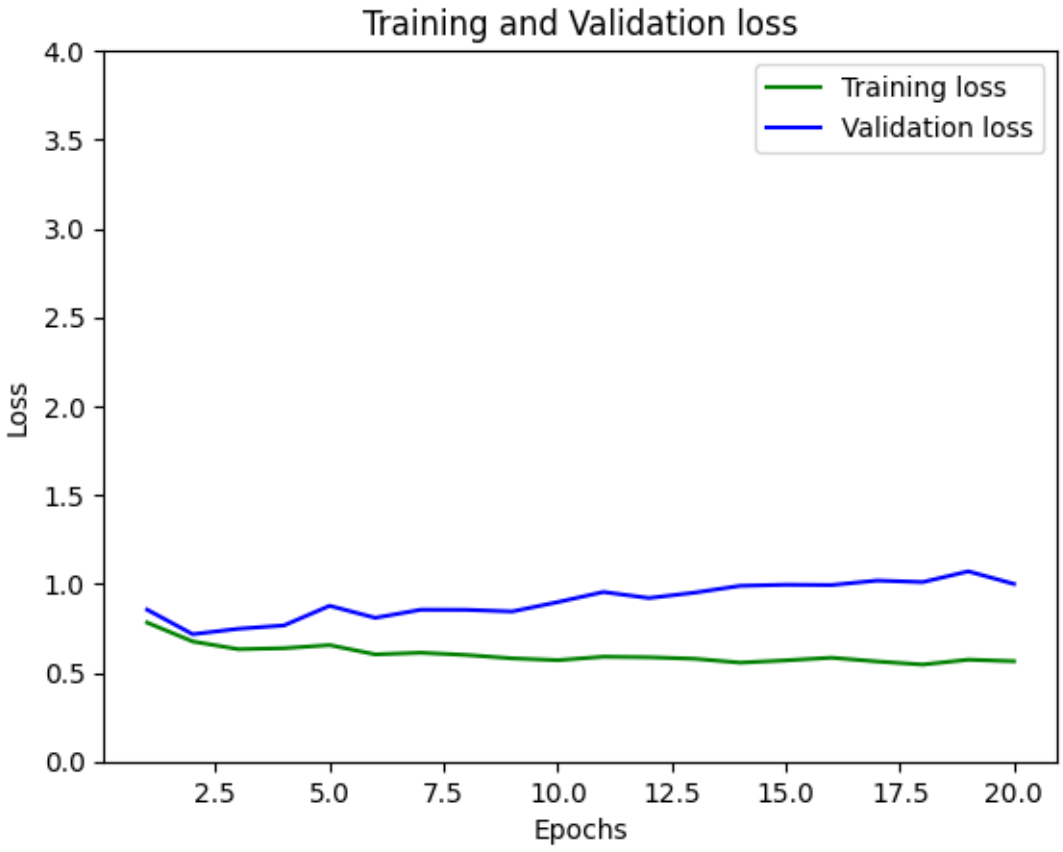
```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_12 (Rescaling)    (None, 64, 64, 3)         0

 conv2d_34 (Conv2D)          (None, 60, 60, 64)        4864

 max_pooling2d_34 (MaxPoolin  (None, 30, 30, 64)       0
 g2D)

 conv2d_35 (Conv2D)          (None, 26, 26, 128)       204928

 max_pooling2d_35 (MaxPoolin  (None, 13, 13, 128)      0
 g2D)

 conv2d_36 (Conv2D)          (None, 9, 9, 256)         819456

 dropout_10 (Dropout)        (None, 9, 9, 256)         0

 max_pooling2d_36 (MaxPoolin  (None, 4, 4, 256)        0
 g2D)

 flatten_12 (Flatten)        (None, 4096)              0

 dense_13 (Dense)            (None, 11)                45067

=================================================================
Total params: 1,074,315
Trainable params: 1,074,315
Non-trainable params: 0
_________________________________________________________________
```

1.11.8 Loss and Accuracy calculation

[33]:
```
# define variables used in visualization
acc = optimized_history.history['acc']
val_acc = optimized_history.history['val_acc']
loss = optimized_history.history['loss']
val_loss = optimized_history.history['val_loss']
```

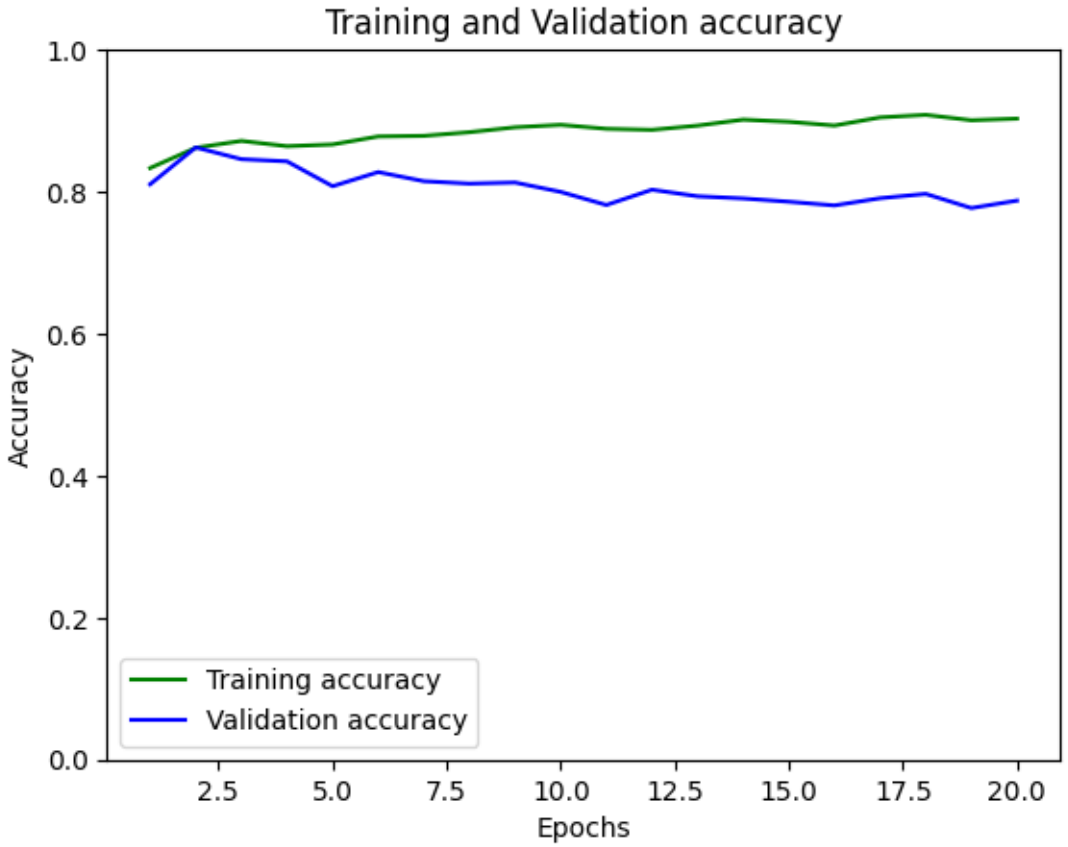
1.11.9 Loss

[34]:
```
# Plotting training and validation loss
epochs = range(1,num_epochs+1)
plt.plot(epochs, loss, 'g', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.ylim([0, 4])
plt.legend()
plt.show()
```



1.11.10 Accuracy

[35]:
```
# Plotting training and validation accuracy
plt.plot(epochs, acc, 'g', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend()
plt.show()
```



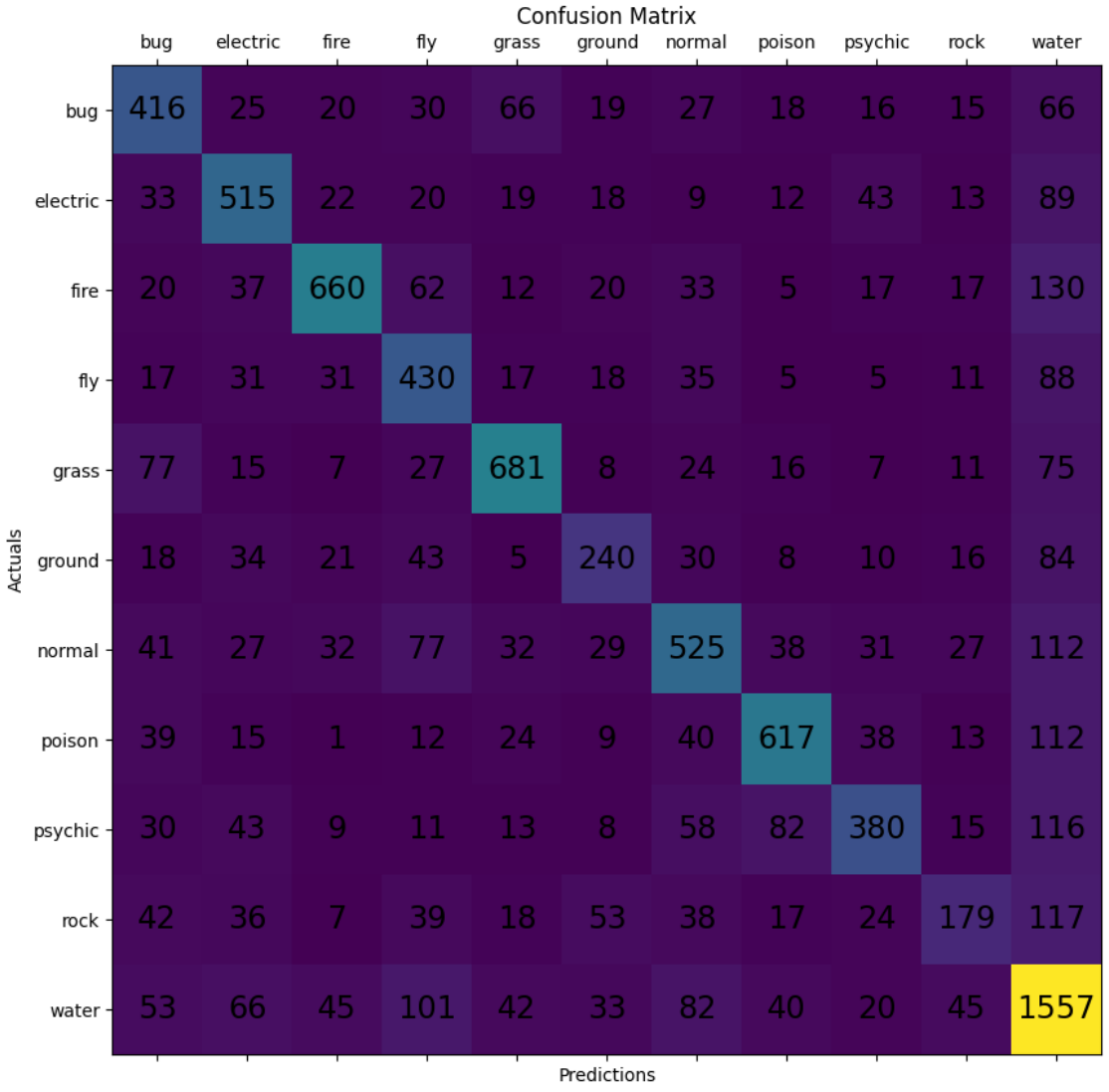
1.11.11 Calculation for Confusion Matrix

[ ]:
```
preds = np.empty((0, 2), dtype=int)
for images, labels in iter(testing_ds):
    pred = optimized_model.predict(images.numpy().astype("uint8"))
    for i in range(len(pred)):
        y_pred = np.argmax(pred[i])
```

```
        preds = np.append(preds, [[class_names[labels[i]],␣
↪class_names[y_pred]]], axis=0)

y_preds = preds[:, 1]
y_trues = preds[:, 0]
```

1.11.12 Confusion Matrix

[37]:
```
plt_confmatrix(y_preds, y_trues, class_names)
```



1.11.13 Discussion

The last model is the best overall, it is slightly worse in accuracy on the training data compared to the overfitted model, but again this is expected, as we now want to focus on all possible data, this includes validation and testing data. Depending on your usecase, meaning whether or not you will have unseen data you will want to use the overfitted model(no unseen data), or this model(unseen data exists).

1.12 ———

1.13 Prediction

To check the individual models simply change the name. We have left out individual predictions for each model to save space.

1.13.1 Define Probability Model

[38]:
```
probability_model = tf.keras.Sequential([optimized_model,
                                         tf.keras.layers.Softmax()])
```

1.13.2 Compile Probability Model

[39]:
```
predictions = probability_model.predict(testing_ds)
```

```
312/312 [==============================] - 1s 2ms/step
```

1.13.3 Prediction Array

[40]:
```
predictions[0]
```

[40]:
```
array([0.07871405, 0.07931947, 0.21229106, 0.07871065, 0.07870924,
       0.07870924, 0.07870926, 0.07870924, 0.07870924, 0.07870924,
       0.07870924], dtype=float32)
```

1.13.4 Calculatins for Prediction

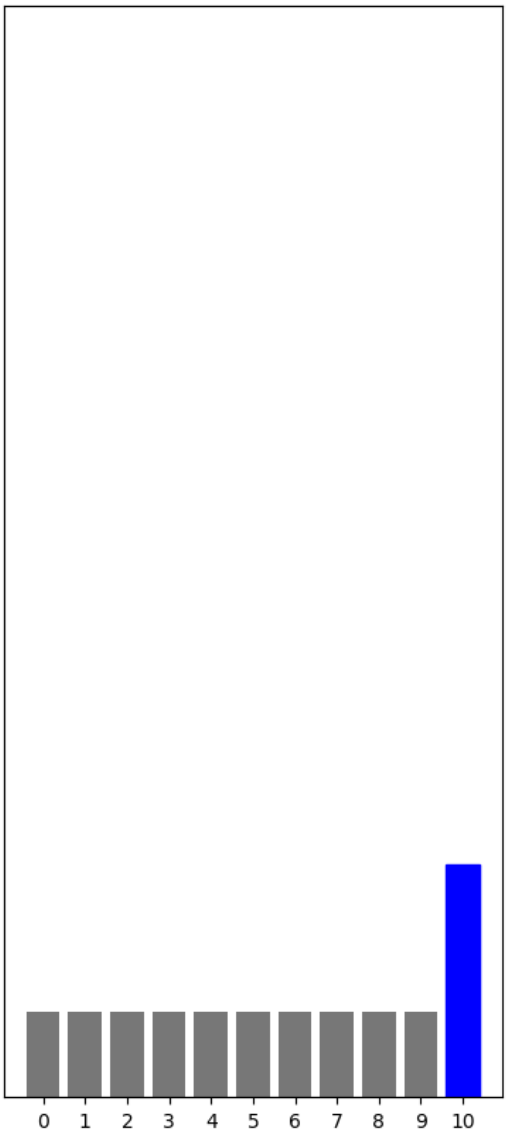
[41]:
```
def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
```

```
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(11))
  plt.yticks([])
  thisplot = plt.bar(range(11), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

1.13.5 Visualization

[43]:
```
i = 11
ds = testing_ds

plt.figure(figsize=(10, 10))
plt.subplot(1,2,1)
#for images, labels in test_ds.take(1):
iterator = iter(ds)
image, labels = next(iterator)
plot_image(i, predictions[i], labels, image)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  labels)
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



1.14 _____

2 Conclusion

2.1 Comparison

2.1.1 Underfitted Model

The underfitted model is a joke for our dataset. It essentially tries to play a philosopher, in this case Thales and predicts: “everything is water”.

However, there is a simple explanation for this, with the small amount of parameters the model can’t specialize enough and it will default to the massive water dataset. If the first generation of

pokemon would have had a more balanced distribution for types, then we would have had a model with slightly better results, despite the low amount of parameters.

For this model we used 30 epochs, while it did not have significant changes either way, there were some "bigger" ones between 15 to 30. We stopped reducing trainable parameters at around 5'800 because if we would have reduced it even more it wouldn't be a model anymore.

2.1.2 Overfitted Model

The overfitted model is special, while the accuracy on the training set and the validation is as expected, the testing data seemed to perform rather well for an overfitted model. One possible cause for this is that the data is still from the same generation of pokemon, which essentially means it's the "same" data, just slightly different pictures.

For this model we used 40 epochs because you can clearly see then that the model is overfitting.

2.1.3 Optimized Model

In terms of performance comparison it is essentially a slightly better version of the overfitted model. This also makes sense when the model definitions are compared, the optimized model is very similar with the ridiculous 500 dense layer removed.

For this model we used 20 epochs for each split of the cross-validation and afterwards 20 epochs to train the best model. We first tried training the best model with 40 epochs but it began to overfit and therefore we went back to 20 epochs.

2.2 Discussion

The initial expectation of the project was that the results would not be useful. However it was quickly apparent, that even with some very vague pokemon, a model could be found that reliably classifies pokemon of the first generation. While the problem of more modern generations with liberal interpretations of types would still apply to bigger models with multiple generations, it is apparent that machine learning is a very powerful tool, even in the hands of novices.

2.3 _____