# Project: No Kangaroos in Austria

*The complete source code can be found under https://github.com/Declaminius/GeoGuessr-AI.*

In [1]:
```
import os
os.environ['USE_PYGEOS'] = '0'
import osmnx as ox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, optimizers, losses

# Local configuration file
import config

# Local data visualization library
import data_visualization


plt.rcParams.update({"text.usetex": True})

train_ds_unbatched = config.train_ds.unbatch()
val_ds_unbatched = config.val_ds.unbatch()

train_ds32_unbatched = config.train_ds32.unbatch()
val_ds32_unbatched = config.val_ds32.unbatch()
```

```
2023-03-12 11:59:44.055536: I tensorflow/core/platform/cpu_feature_guard.cc:193] T
his TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDN
N) to use the following CPU instructions in performance-critical operations:  SSE
4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compil
er flags.
Found 25062 files belonging to 2 classes.
Using 20050 files for training.
Using 5012 files for validation.
Found 25062 files belonging to 2 classes.
Using 20050 files for training.
Using 5012 files for validation.
Found 55051 files belonging to 5 classes.
Using 44041 files for training.
Using 11010 files for validation.
Found 25062 files belonging to 2 classes.
```

## Motivation

In the game GeoGuessr the player is dropped on a random location within Google

Streetview with the goal to determine his location as precisely as possible. In the simplest case, we are not allowed to move from our starting position, pan around nor zoom in the initial image.


*Example of a game of GeoGuessr*

Hence the goal of this Machine Learning project is to map Street View images to location data. This setting naturally allows for either regression, where we aim to predict the latitude and longitude of any given image, or classification, if we ask to classify the images into pre-defined geographic regions, like countries or continents. I want to focus on the classification task, where images are classified into the countries, where they are taken in. A good starting point would then be to pick two distinct countries, like Austria and Australia to replace the dogs vs cats dataset. Once a model works well for the binary classification task I will also try to extend it to a multinomial classification problem by adding more (and more difficult) countries.

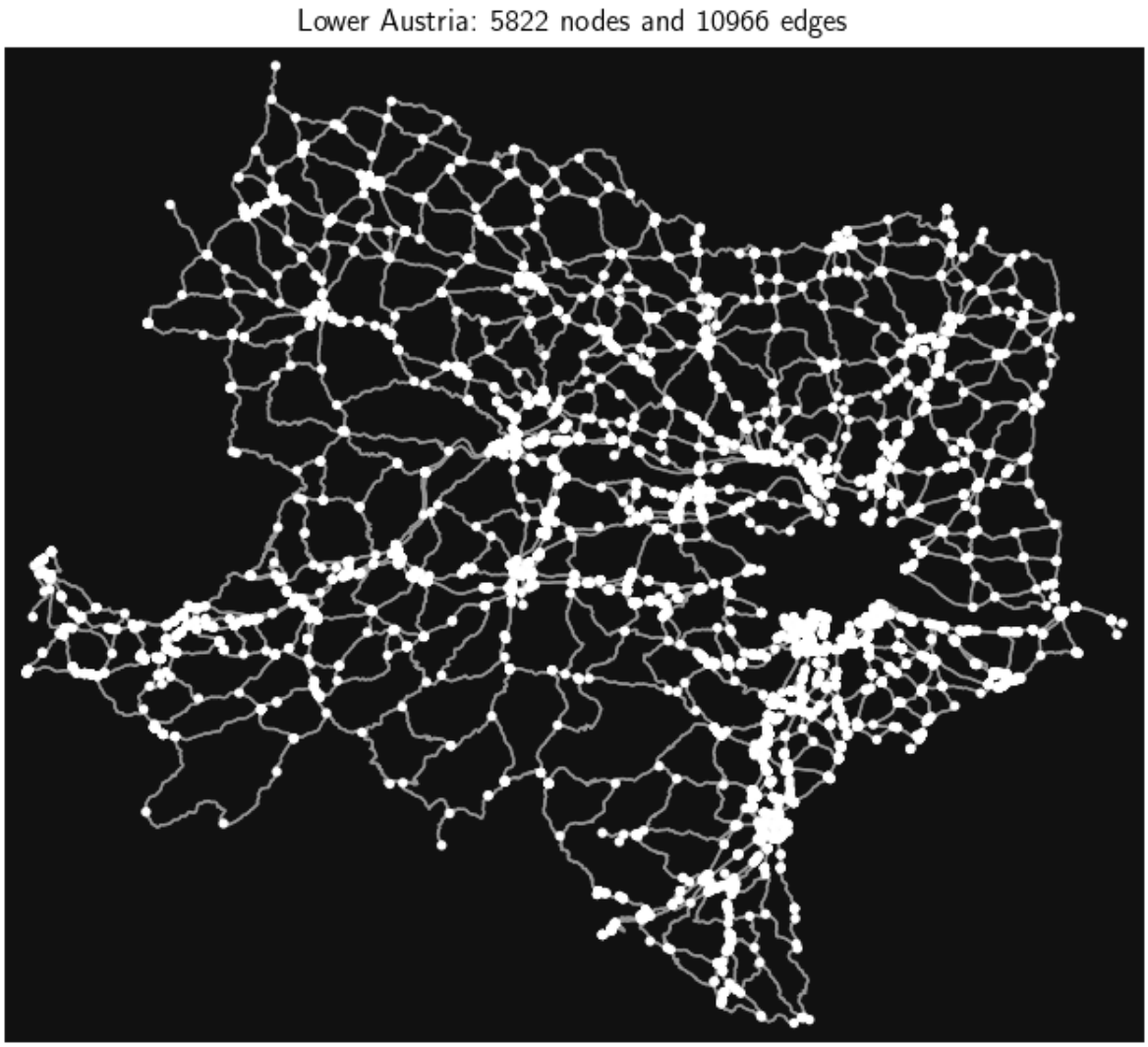
# Generating the dataset

To obtain our dataset, we sample points from street network graphs obtained via the Overpass API and enter these locations in the Google Street View Static API. The API now checks whether there is an available street view image within a fifty meter radius and if yes, returns one such image.

## Street network graphs

In order not to overwhelm the poor Overpass API, it was necessary to first split the queries for entire countries into queries of individual states, whose network size was more manageable. This approach worked well for Austrian states, which are relatively smaller than their Australian counterpart. For the bigger Australian states, however I had to resort to a different tool, where I instead downloaded the respective .osm-Files from a PlanetOSM mirror and used the handy command-line tool osmosis to filter out the relevant street network.

In [24]:
```
location = {'state': 'Lower Austria', 'country': 'Austria'}
G = ox.graph_from_place(location, network_type="drive", custom_filter = config.high
basic_stats = ox.basic_stats(G)

fig, ax = ox.plot_graph(G, show=False, close=False)
ax.set_title(f"Lower Austria: {basic_stats['n']} nodes and {basic_stats['m']} edges
plt.show()
```




## Generating images

Given a street network for a certain state, we can now use this graph to sample points along its edges and use the obtained coordinates to request street view images at this location.

In [20]:
```
from street_view_images import StreetViewImagesForState

num_images = 5

for country in ("Austria", "Australia"):
    for i in range(3):
        state = config.states_dict[country][i]
        fig = plt.figure(figsize = (25,5))
        fig.suptitle(f"Random locations in {state}, {country}", size = 20)

        street_view_images_vienna = StreetViewImagesForState(state = state, country
        sample_coordinates = street_view_images_vienna.generate_points(num_images)
        for i, location in enumerate(sample_coordinates):
            response = street_view_images_vienna.request_image(location)
            while response is None:
                location = street_view_images_vienna.generate_points(1)
```

```
            response = street_view_images_vienna.request_image(location)
        image = response[1]
        with open('image.jpg', 'wb') as file:
            file.write(image.content)
        image.close()
        plt.subplot(1,5,i+1)
        plt.axis('off')
        plt.imshow(mpimg.imread('image.jpg'))
    plt.show()
```

Random locations in Lower Austria, Austria

Random locations in Upper Austria, Austria

Random locations in Burgenland, Austria

Random locations in australian_capital_territory, Australia

Random locations in tasmania, Australia

Random locations in northern_territory, Australia

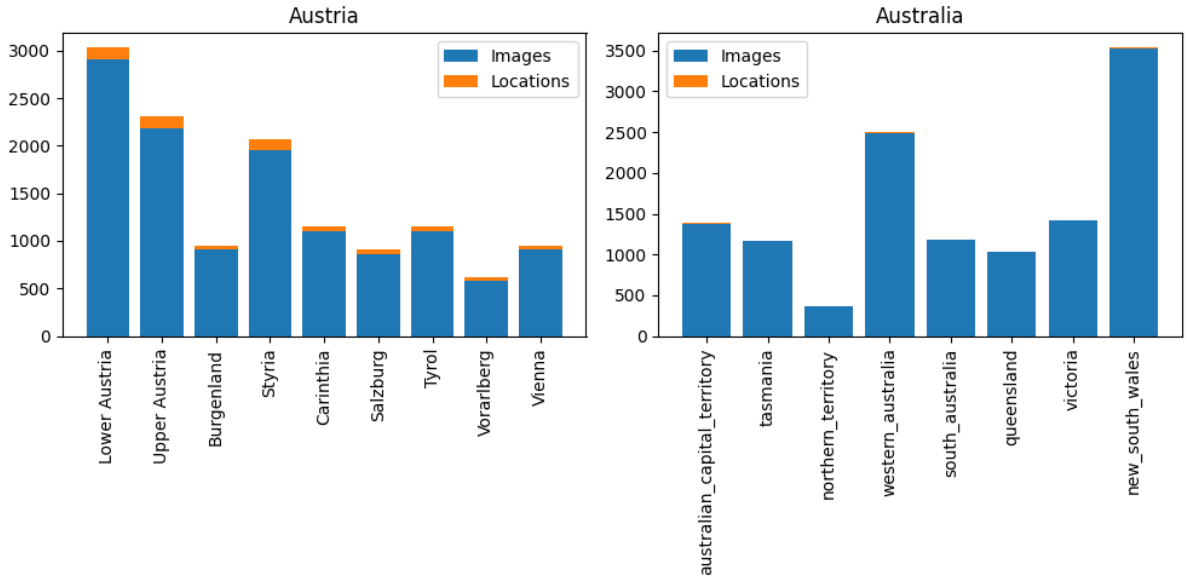
## Distribution of images

After a quick manual review of the image dataset, in which I removed a low-quality images (the main source of which where tunnel locations in Austria). In the end I aimed for roughly 12.500 images per country to match the size of the Dogs vs. Cats dataset. In the following we can see the distribution of locations across states.

In [14]:
```
for country in ("Austria", "Australia"):
    images_per_country = 0
    locations_per_country = 0
    for state in config.states_dict[country]:
        df = pd.read_csv(f"images/{country}/{state}/coordinates.csv", index_col= "i
        images_per_country += sum(df['image_exists'])
        locations_per_country += len(df)

    print(f"{images_per_country} images out of {locations_per_country} locations in
```

```
12504 images out of 13128 locations in Austria.
12560 images out of 12599 locations in Australia.
```

In [12]:
```
fig, axs = plt.subplots(ncols = 2, figsize = (10,5))
for (i,country) in enumerate(("Austria", "Australia")):
    images_per_state = []
    locations_per_state = []
    for state in config.states_dict[country]:
        df = pd.read_csv(f"images/{country}/{state}/coordinates.csv", index_col= "i
        images_per_state.append(sum(df["image_exists"]))
        locations_per_state.append(len(df))

    axs[i].bar(config.states_dict[country], images_per_state, label = "Images")
    axs[i].bar(config.states_dict[country], np.array(locations_per_state) - np.arra
    axs[i].set_title(country)
    axs[i].tick_params('x', labelrotation=90)
    axs[i].legend()
plt.tight_layout()
plt.show()
```




## Preprocessing

In [ ]:

```
from tensorflow.data import AUTOTUNE

image_dir = "images"
batch_size = 32

def standardize_image(image, label):
    mean = tf.reduce_mean(image)
    std = tf.math.reduce_std(image)
    standardized_image = tf.map_fn(lambda x: (x - mean)/std, image)
    return (standardized_image, label)

def create_standardized_dataset(image_size):
    train_ds, val_ds = keras.utils.image_dataset_from_directory(
        image_dir,
        validation_split=0.2,
        labels="inferred",
        class_names=["Austria","Australia"],
        subset="both",
        seed = 0,
        batch_size = batch_size,
        crop_to_aspect_ratio=True,
        image_size=(image_size, image_size))

    train_ds = train_ds.map(standardize_image)
    val_ds = val_ds.map(standardize_image)

    train_ds = train_ds.prefetch(buffer_size = AUTOTUNE)
    val_ds = val_ds.prefetch(buffer_size = AUTOTUNE)

    return train_ds, val_ds
```

# Ridge classificiation

The first simple model we try to train on our dataset is a Ridge regressor.

Ridge regression is an extension to least squares which introduces an additional regularization parameter $\lambda \geq 0$. As such, the loss is given by:

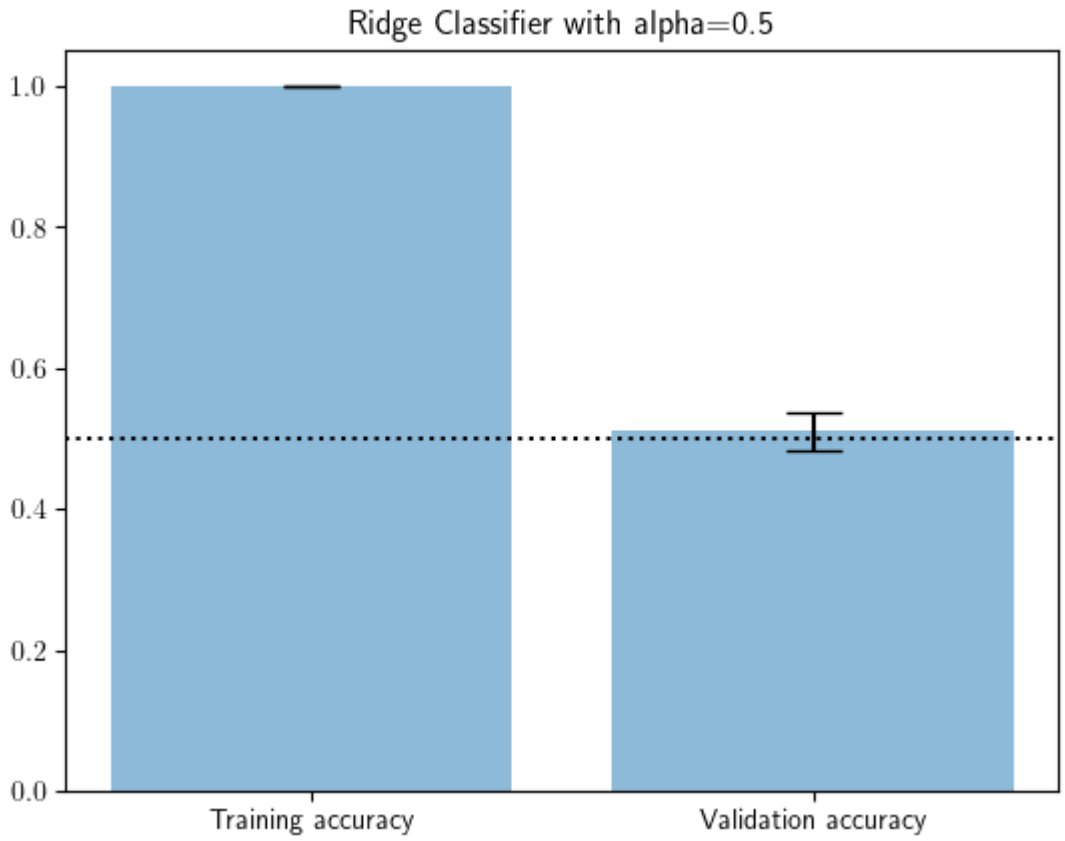
$$J_{\text{Ridge}}(\mathbf{w}) = \|X\mathbf{w} - y\|^2 + \lambda\|\mathbf{w}\|^2.$$

The regularization parameter $\lambda$ is tasked to control the *generalization* to unseen data.

We preprocess our image data by rescaling it to the interval [0,1].

In [9]:
```
from sklearn import linear_model
from ridge_classifier import evaluate_regressor

alpha = 0.5
sample_size = 500
ridge_classifier = linear_model.RidgeClassifier(alpha = alpha)
ridge_accuracy_array = np.empty((5,2))
for i in range(5):
    train_accuracy, val_accuracy = evaluate_regressor(ridge_classifier, sample_size
    ridge_accuracy_array[i] = np.array([train_accuracy, val_accuracy])
data_visualization.plot_accuracy(ridge_accuracy_array, f'Ridge Classifier with {alp
```

```
Training data:
500 out of 500 correctly classified.
Validation data:
277 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
259 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
236 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
249 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
256 out of 500 correctly classified.
```




This is a clear example of overfitting. The reason for that becomes apparent when we compare the size of the input with our sample size. Since each image consists of three channels on a grid of size 256x256, we have circa 200.000 input features compared to only 500 samples. Putting it that way makes it pretty obvious that the task of finding a separating plane in the training set should not be very hard, in fact it can be done perfectly. However, due to the sheer mulitude of possible hyperplanes, it is unlikely that this classification generalizes well to unseen data. However, to make sure we do some additional cross-validation for different values of alpha:

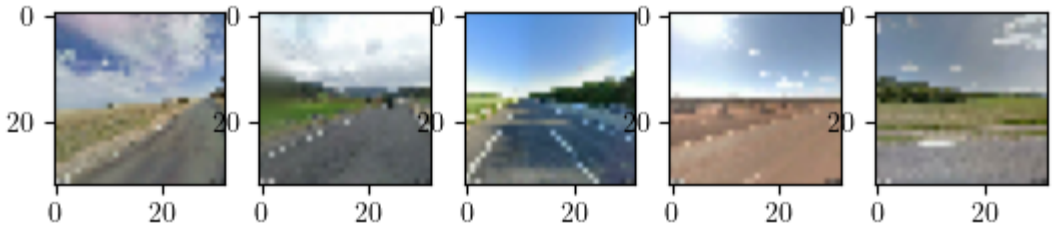
In [3]:
```
from ridge_classifier import sample_data
sample_size = 500

ridge_regressor = linear_model.RidgeClassifierCV(alphas = [0.1,1,10])
train_images, train_labels = sample_data(train_ds_unbatched, sample_size)
val_images, val_labels = sample_data(val_ds_unbatched, sample_size)

# Fit the regressor
ridge_regressor.fit(train_images, train_labels)
print(ridge_regressor.alpha_, ridge_regressor.best_score_)
print(ridge_regressor.score(train_images, train_labels))
print(ridge_regressor.score(val_images, val_labels))
```

```
10.0 -1.2237007160532294
1.0
0.454
```

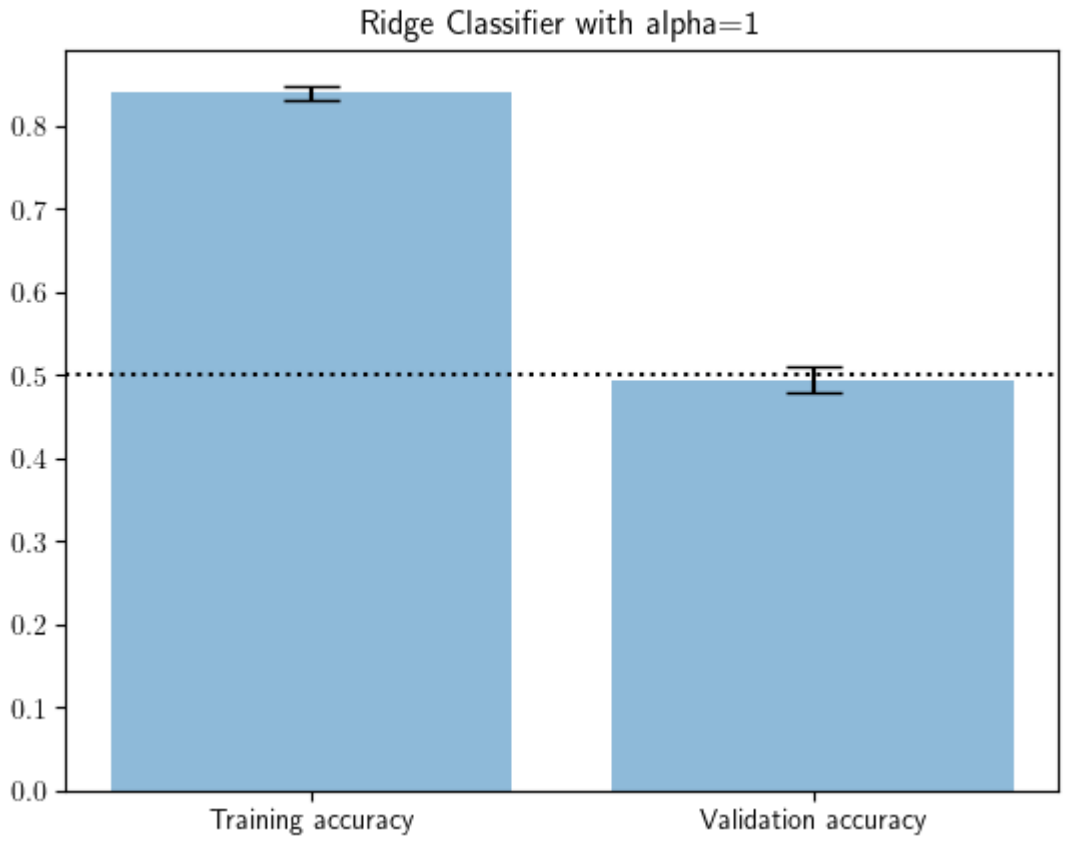
To adapt to this problem, let's try again with downscaled images in 32x32 resolution and an

increased sample size of 5.000 images.

In [8]:
```
from sklearn import linear_model
from ridge_classifier import evaluate_regressor

i = 1
for image, label in train_ds_unbatched.take(5):
    plt.subplot(1,5,i)
    plt.imshow(image.numpy().astype(np.uint8))
    i += 1

plt.show()

alpha = 1
sample_size = 5000
ridge_classifier = linear_model.RidgeClassifier(alpha = alpha)
ridge_accuracy_array = np.empty((5,2))
for i in range(5):
    train_accuracy, val_accuracy = evaluate_regressor(ridge_classifier, sample_size
    ridge_accuracy_array[i] = np.array([train_accuracy, val_accuracy])
data_visualization.plot_accuracy(ridge_accuracy_array, f'Ridge Classifier with {alp
```




```
Training data:
4168 out of 5000 correctly classified.
Validation data:
2483 out of 5000 correctly classified.
Training data:
4268 out of 5000 correctly classified.
Validation data:
2334 out of 5000 correctly classified.
Training data:
4190 out of 5000 correctly classified.
Validation data:
2470 out of 5000 correctly classified.
Training data:
4157 out of 5000 correctly classified.
Validation data:
2489 out of 5000 correctly classified.
Training data:
4214 out of 5000 correctly classified.
Validation data:
2586 out of 5000 correctly classified.
```



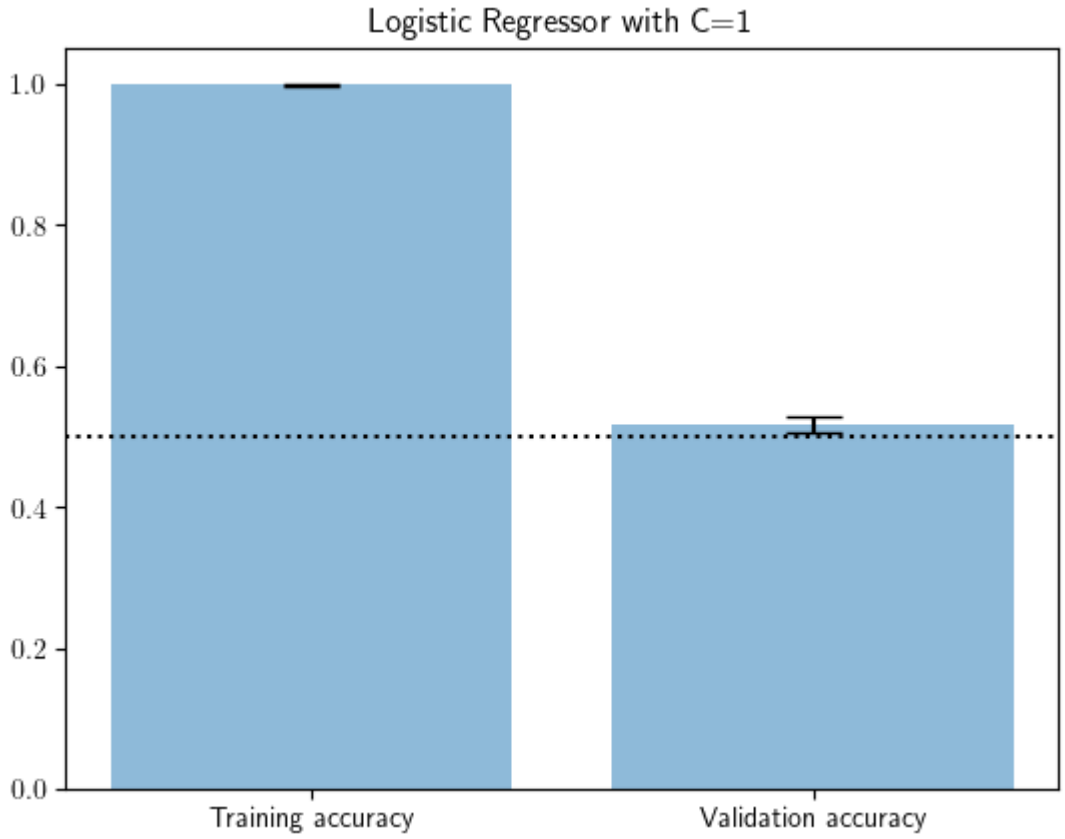

After reducing the size of the input drastically, we can observe that we no longer come close to achieving a perfect score on the training dataset. However, this change produced no statistically significant improvement of the score on the validation dataset. In fact, we are even worse off than if we would just guess at random.

## Logistic Regression

I also tried to train this regressor on the 256x256 resolution dataset, however it did not converge within 500 iterations and showed no more promising results than on the smaller 32x32 dataset.

In [11]:
```
C = 1
max_iter = 500
sample_size = 500
logistic_regressor = linear_model.LogisticRegression(C = C, max_iter = max_iter)
logistic_accuracy_array = np.empty((5,2))
for i in range(5):
    train_accuracy, val_accuracy = evaluate_regressor(logistic_regressor, sample_si
    logistic_accuracy_array[i] = np.array([train_accuracy, val_accuracy])

data_visualization.plot_accuracy(logistic_accuracy_array, f'Logistic Regressor with
```

```
Training data:
500 out of 500 correctly classified.
Validation data:
256 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
257 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
269 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
251 out of 500 correctly classified.
Training data:
499 out of 500 correctly classified.
Validation data:
260 out of 500 correctly classified.
```



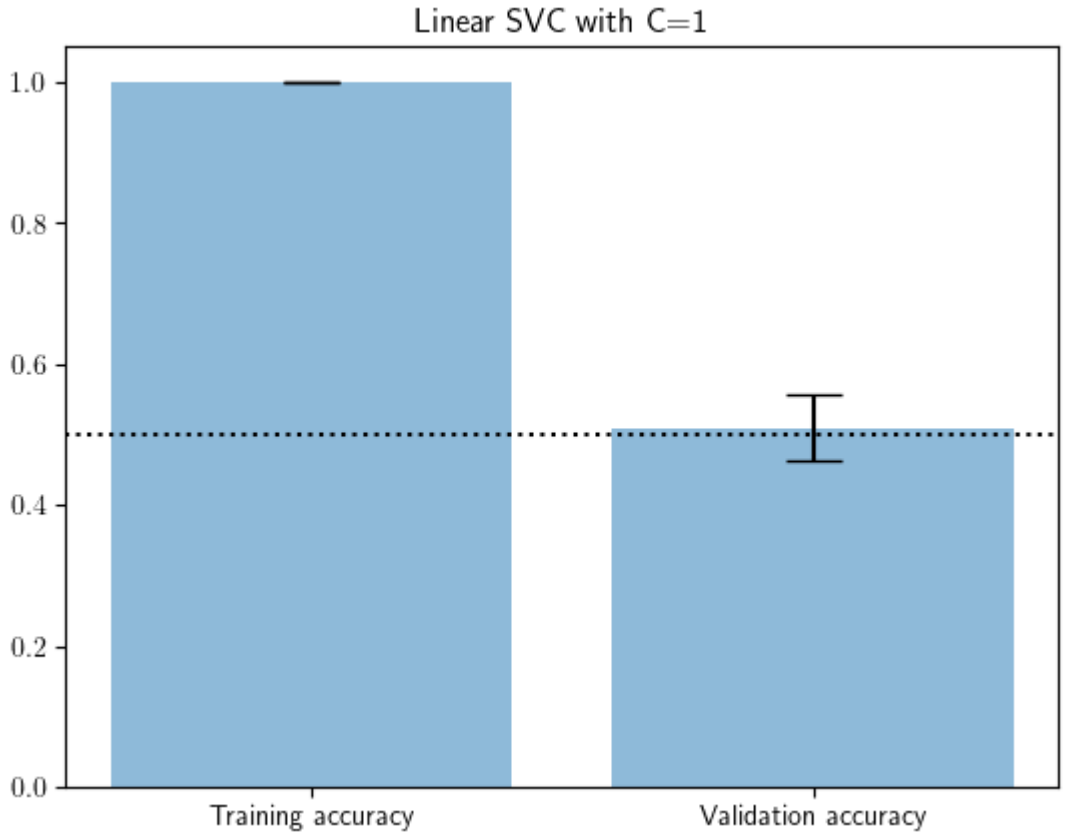

# Support vector machines

## Linear Kernel

In [13]:
```
from sklearn import svm

sample_size = 250
```

```
svc_linear = svm.SVC(kernel = "linear")
svc_linear_accuracy_array = np.empty((5,2))
for i in range(5):
    train_accuracy, val_accuracy = evaluate_regressor(svc_linear, sample_size, trai
    svc_linear_accuracy_array[i] = np.array([train_accuracy, val_accuracy])
data_visualization.plot_accuracy(svc_linear_accuracy_array, f'Linear SVC with {C=}'
plt.show()
```

```
Training data:
250 out of 250 correctly classified.
Validation data:
131 out of 250 correctly classified.
Training data:
250 out of 250 correctly classified.
Validation data:
113 out of 250 correctly classified.
Training data:
250 out of 250 correctly classified.
Validation data:
147 out of 250 correctly classified.
Training data:
250 out of 250 correctly classified.
Validation data:
126 out of 250 correctly classified.
Training data:
250 out of 250 correctly classified.
Validation data:
119 out of 250 correctly classified.
```



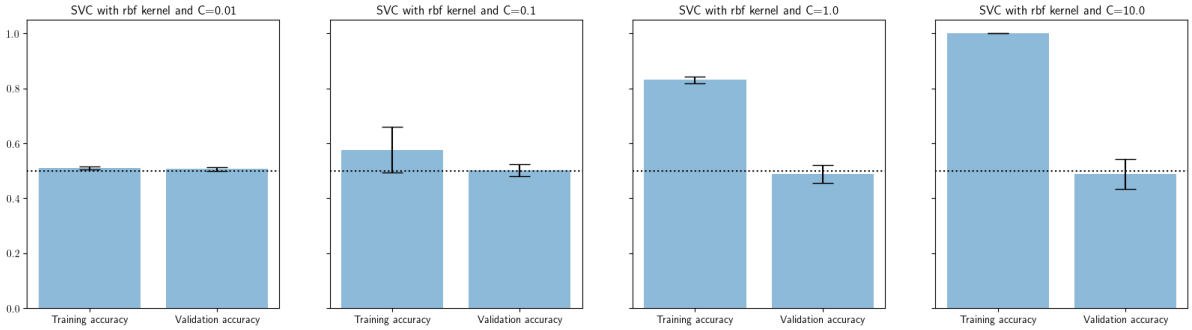

## Rbf kernel

In [6]:

```
from sklearn import svm
from ridge_classifier import evaluate_regressor

sample_size = 500
C_array = [1e-2,1e-1,1e+0,1e+1]

fig, axs = plt.subplots(nrows = 1, ncols = 4, figsize = (20,5), sharey = True)

for i, C in enumerate(C_array):
    svc_rbf = svm.SVC(C = C, kernel = "rbf")
    svc_rbf_accuracy_array = np.empty((5,2))
    for j in range(5):
        train_accuracy, val_accuracy = evaluate_regressor(svc_rbf, sample_size, tra
        svc_rbf_accuracy_array[j] = np.array([train_accuracy, val_accuracy])
    plt.sca(axs.flatten()[i])
    data_visualization.plot_accuracy(svc_rbf_accuracy_array, f'SVC with rbf kernel

plt.show()
```

```
Training data:
257 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
253 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
258 out of 500 correctly classified.
Validation data:
245 out of 500 correctly classified.
Training data:
251 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
257 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
257 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
356 out of 500 correctly classified.
Validation data:
230 out of 500 correctly classified.
Training data:
317 out of 500 correctly classified.
Validation data:
259 out of 500 correctly classified.
Training data:
259 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
252 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
411 out of 500 correctly classified.
Validation data:
253 out of 500 correctly classified.
Training data:
412 out of 500 correctly classified.
Validation data:
258 out of 500 correctly classified.
Training data:
409 out of 500 correctly classified.
Validation data:
255 out of 500 correctly classified.
Training data:
421 out of 500 correctly classified.
Validation data:
214 out of 500 correctly classified.
```

```
Training data:
424 out of 500 correctly classified.
Validation data:
241 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
242 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
236 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
213 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
235 out of 500 correctly classified.
Training data:
500 out of 500 correctly classified.
Validation data:
294 out of 500 correctly classified.
```




Since the strength of the reguralization scales inversely proportional to C, smaller values of C leads to a reduced accuracy on the training dataset. However, none of the above choices for C generalize well to unseen data.

# Neural network

## Training the model

In [3]:
```
model = keras.Sequential(
    [
    layers.Conv2D(32, (3,3), input_shape = (256,256,3), activation = "relu", paddin
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
```

```
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Flatten(),
    layers.Dense(100),
    layers.Dropout(0.1),
    layers.Dense(1, activation = 'sigmoid'),
    ])

optimizer = optimizers.Adam(learning_rate = 0.001)

model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics = ["accuracy
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 256, 256, 32)      896

 max_pooling2d_6 (MaxPooling  (None, 128, 128, 32)     0
 2D)

 conv2d_7 (Conv2D)           (None, 128, 128, 32)      9248

 max_pooling2d_7 (MaxPooling  (None, 64, 64, 32)       0
 2D)

 conv2d_8 (Conv2D)           (None, 64, 64, 32)        9248

 max_pooling2d_8 (MaxPooling  (None, 32, 32, 32)       0
 2D)

 conv2d_9 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d_9 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 conv2d_10 (Conv2D)          (None, 16, 16, 32)        9248

 max_pooling2d_10 (MaxPoolin  (None, 8, 8, 32)         0
 g2D)

 conv2d_11 (Conv2D)          (None, 8, 8, 32)          9248

 max_pooling2d_11 (MaxPoolin  (None, 4, 4, 32)         0
 g2D)

 flatten_1 (Flatten)         (None, 512)               0

 dense_2 (Dense)             (None, 100)               51300

 dropout_1 (Dropout)         (None, 100)               0

 dense_3 (Dense)             (None, 1)                 101

=================================================================
Total params: 98,537
Trainable params: 98,537
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
history = model.fit(config.train_ds, epochs = 20, validation_data = config.val_ds)
```

```
Epoch 1/20
627/627 [==============================] - 510s 811ms/step - loss: 0.3680 - accura
cy: 0.8300 - val_loss: 0.3272 - val_accuracy: 0.8733
Epoch 2/20
627/627 [==============================] - 504s 803ms/step - loss: 0.2524 - accura
cy: 0.8956 - val_loss: 0.2311 - val_accuracy: 0.9044
Epoch 3/20
627/627 [==============================] - 503s 800ms/step - loss: 0.2093 - accura
cy: 0.9158 - val_loss: 0.1807 - val_accuracy: 0.9278
Epoch 4/20
627/627 [==============================] - 511s 814ms/step - loss: 0.1740 - accura
cy: 0.9301 - val_loss: 0.1580 - val_accuracy: 0.9385
Epoch 5/20
627/627 [==============================] - 501s 798ms/step - loss: 0.1499 - accura
cy: 0.9412 - val_loss: 0.2128 - val_accuracy: 0.9226
Epoch 6/20
627/627 [==============================] - 504s 791ms/step - loss: 0.1332 - accura
cy: 0.9491 - val_loss: 0.1784 - val_accuracy: 0.9356
Epoch 7/20
627/627 [==============================] - 516s 821ms/step - loss: 0.1185 - accura
cy: 0.9544 - val_loss: 0.1699 - val_accuracy: 0.9407
Epoch 8/20
627/627 [==============================] - 516s 821ms/step - loss: 0.1047 - accura
cy: 0.9593 - val_loss: 0.2378 - val_accuracy: 0.9314
Epoch 9/20
627/627 [==============================] - 514s 818ms/step - loss: 0.0936 - accura
cy: 0.9638 - val_loss: 0.1671 - val_accuracy: 0.9469
Epoch 10/20
627/627 [==============================] - 512s 816ms/step - loss: 0.0853 - accura
cy: 0.9666 - val_loss: 0.1861 - val_accuracy: 0.9364
Epoch 11/20
627/627 [==============================] - 518s 825ms/step - loss: 0.0787 - accura
cy: 0.9688 - val_loss: 0.1818 - val_accuracy: 0.9451
Epoch 12/20
627/627 [==============================] - 522s 832ms/step - loss: 0.0733 - accura
cy: 0.9735 - val_loss: 0.2212 - val_accuracy: 0.9268
Epoch 13/20
627/627 [==============================] - 516s 822ms/step - loss: 0.0611 - accura
cy: 0.9761 - val_loss: 0.1928 - val_accuracy: 0.9417
Epoch 14/20
627/627 [==============================] - 518s 825ms/step - loss: 0.0563 - accura
cy: 0.9784 - val_loss: 0.1939 - val_accuracy: 0.9479
Epoch 15/20
627/627 [==============================] - 518s 825ms/step - loss: 0.0532 - accura
cy: 0.9799 - val_loss: 0.2735 - val_accuracy: 0.9258
Epoch 16/20
627/627 [==============================] - 524s 835ms/step - loss: 0.0444 - accura
cy: 0.9827 - val_loss: 0.2037 - val_accuracy: 0.9449
Epoch 17/20
627/627 [==============================] - 520s 828ms/step - loss: 0.0467 - accura
cy: 0.9823 - val_loss: 0.2152 - val_accuracy: 0.9441
Epoch 18/20
627/627 [==============================] - 517s 823ms/step - loss: 0.0435 - accura
cy: 0.9834 - val_loss: 0.2413 - val_accuracy: 0.9465
Epoch 19/20
627/627 [==============================] - 517s 823ms/step - loss: 0.0392 - accura
```

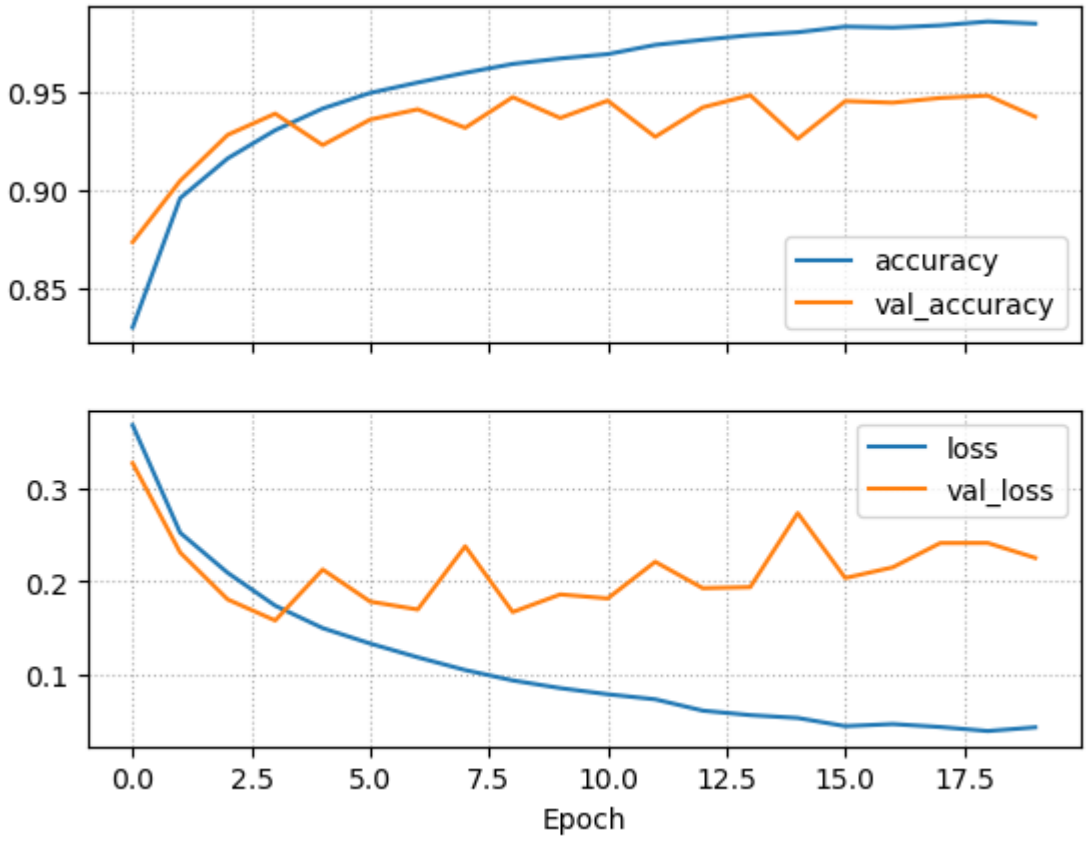
```
cy: 0.9853 - val_loss: 0.2414 - val_accuracy: 0.9477
Epoch 20/20
627/627 [==============================] - 518s 824ms/step - loss: 0.0432 - accura
cy: 0.9843 - val_loss: 0.2253 - val_accuracy: 0.9370
```

In [ ]:
```
model.save('neural_net')
```

```
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_c
ompiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_o
p, _jit_compiled_convolution_op while saving (showing 5 of 6). These functions wil
l not be directly callable after loading.
INFO:tensorflow:Assets written to: neural_net/assets
INFO:tensorflow:Assets written to: neural_net/assets
```

## Evaluation

In [ ]:
```
fig, axs = plt.subplots(2,1, sharex = True)
keys = [["accuracy", "val_accuracy"], ["loss", "val_loss"]]
for (ax, k) in zip(axs,keys):
    for key in k:
        ax.plot(history.history[key], label = key)

    ax.legend()
    ax.grid(linestyle = "dotted")

plt.xlabel("Epoch")
plt.show()
```




In [ ]:
```
predictions = model.predict(config.val_ds).reshape((5012,))
```

```
labels = np.array([x for x in config.val_ds.unbatch().map(lambda x,y: y)])

confidence = 1 - np.abs(predictions - np.round(predictions))
print(f"Average confidence: {100*np.mean(confidence):.2f}%")
print(f"Average confidence (correct labels): {100*np.mean(confidence[np.round(predi
print(f"Average confidence (wrong labels): {100*np.mean(confidence[np.round(predict
```

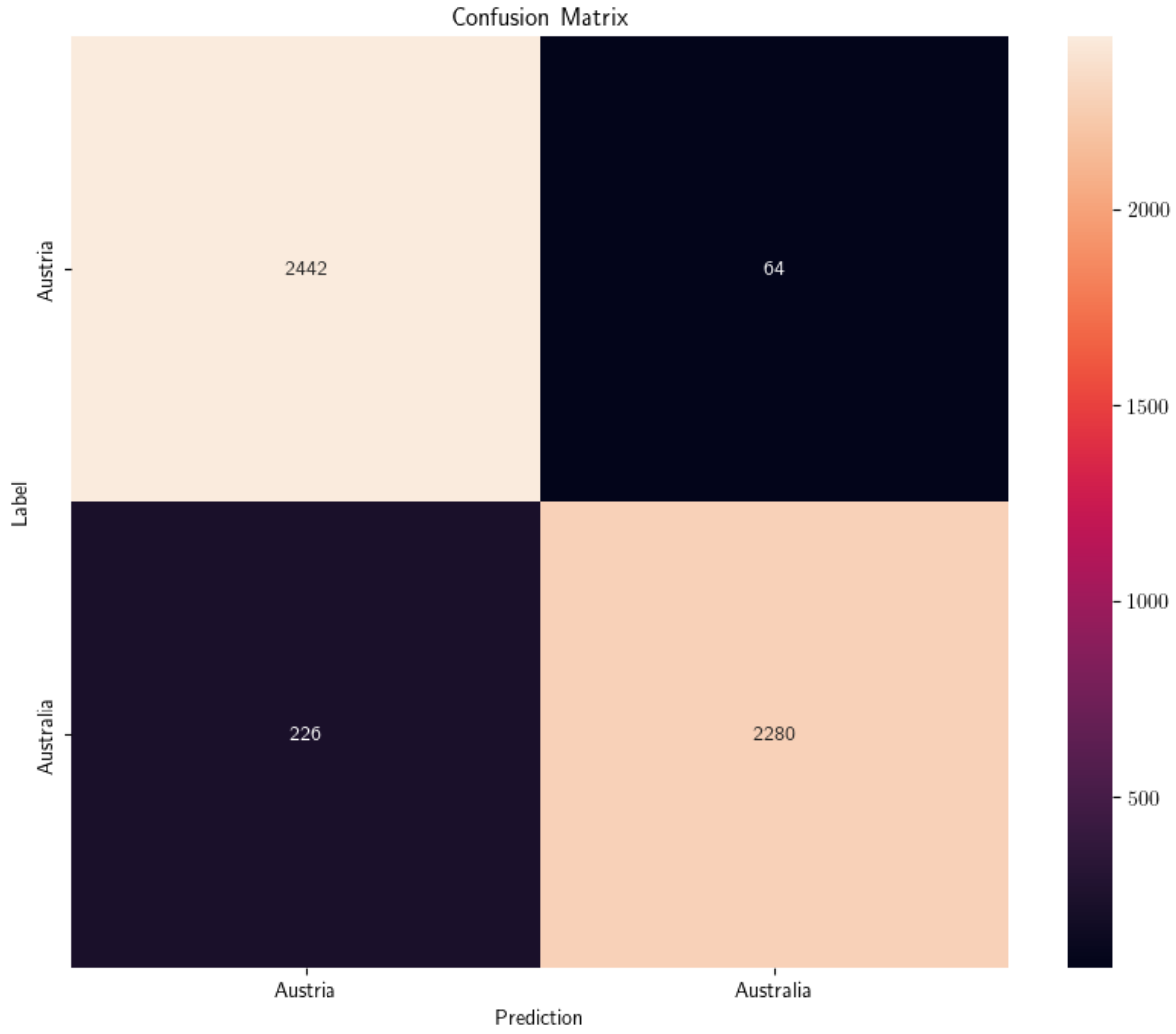
```
Average confidence: 97.32%
Average confidence (correct labels): 98.19%
Average confidence (wrong labels): 84.40%
```

From here on out, we will load the saved version of this model.

In [15]:
```
predictions = tf.squeeze(config.model.predict(config.val_ds), axis = 1)
labels = np.array([x for x in config.val_ds.unbatch().map(lambda x,y: y)])
```

```
157/157 [==============================] - 31s 197ms/step
```

In [16]:
```
confusion_matrix = tf.math.confusion_matrix(labels, np.round(predictions))
class_names = ["Austria", "Australia"]

plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix,
            xticklabels=class_names,
            yticklabels=class_names,
            annot=True, fmt='g')
plt.xlabel('Prediction')
plt.ylabel('Label')
plt.title('Confusion Matrix')
plt.show()
```




## Example images

In [8]:
```
num_images = 9

fig = plt.figure(figsize=(9, num_images))
plt.suptitle("Correctly classified images", size = 20)
i = 1
for images, labels in config.val_ds.take(10):
    predictions = config.model.predict(images).reshape((32,))
    for image, prediction, label in zip(images, predictions, labels.numpy()):
        if np.round(prediction) == label:
            ax = plt.subplot(num_images//3, 3, i)
            data_visualization.plot_image_with_confidence(image, label, prediction)
            i += 1
        if i > num_images:
            break
    if i > num_images:
        break
plt.tight_layout()
plt.show()
```

```
1/1 [==============================] - 1s 868ms/step
```

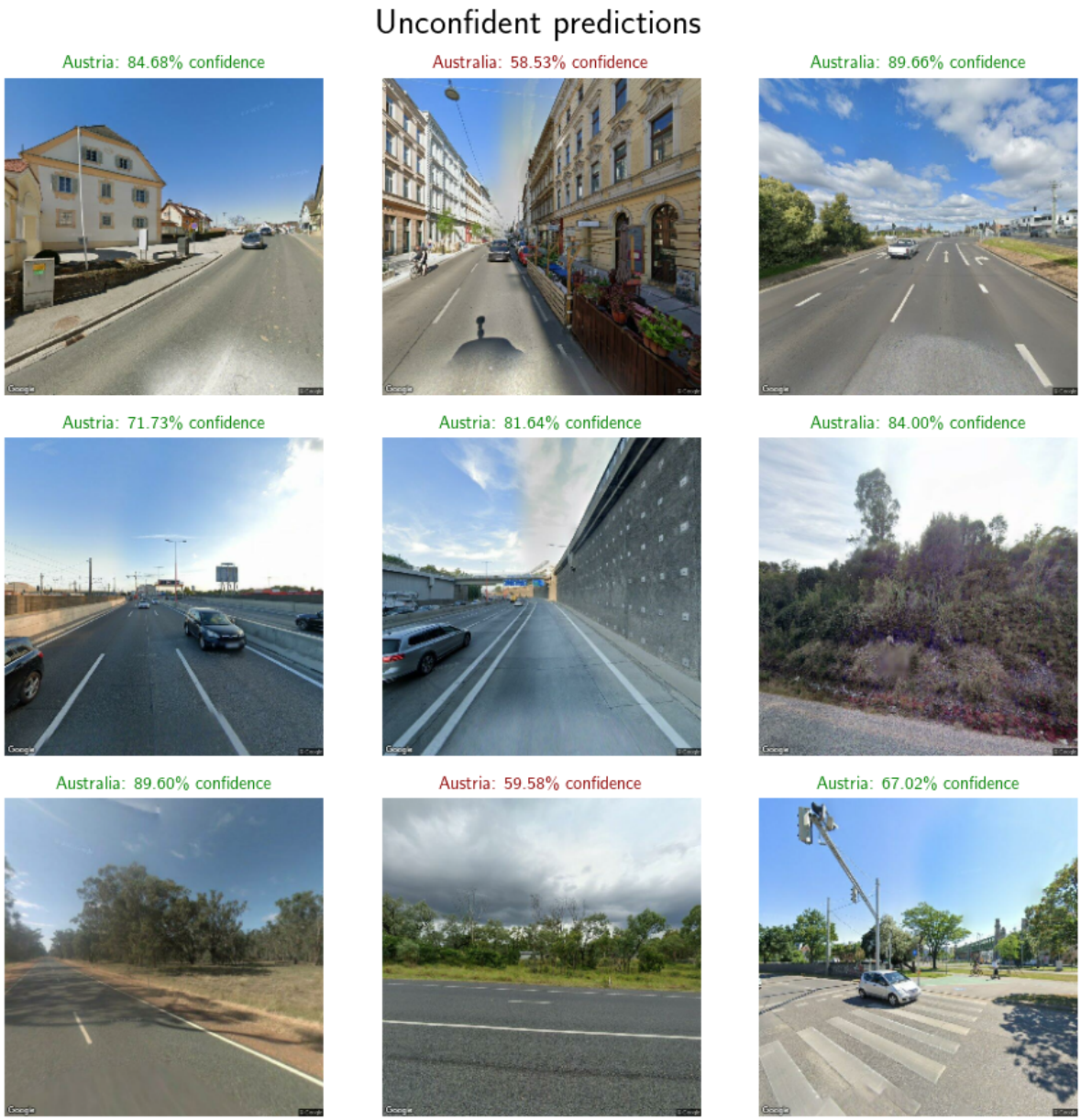

In [10]:
```
num_images = 9

fig = plt.figure(figsize=(9, num_images))
plt.suptitle("Misclassified images", size = 20)
i = 1
for images, labels in config.val_ds.take(10):
    predictions = config.model.predict(images).reshape((32,))
    for image, prediction, label in zip(images, predictions, labels.numpy()):
        if np.round(prediction) != label:
            ax = plt.subplot(num_images//3, 3, i)
            data_visualization.plot_image_with_confidence(image, label, prediction)
            i += 1
        if i > num_images:
            break
    if i > num_images:
        break
plt.tight_layout()
plt.show()
```

```
1/1 [==============================] - 1s 1s/step
1/1 [==============================] - 0s 441ms/step
1/1 [==============================] - 0s 412ms/step
1/1 [==============================] - 0s 370ms/step
1/1 [==============================] - 0s 315ms/step
```


In [11]:
```python
num_images = 9

fig = plt.figure(figsize=(9, num_images))
plt.suptitle("Unconfident predictions", size = 20)
i = 1
for images, labels in config.val_ds.take(10):
    predictions = config.model.predict(images).reshape((32,))
    for image, prediction, label in zip(images, predictions, labels.numpy()):
        if prediction > 0.1 and prediction < 0.9:
            ax = plt.subplot(num_images//3, 3, i)
            data_visualization.plot_image_with_confidence(image, label, prediction)
            i += 1
        if i > num_images:
```

```
            break
    if i > num_images:
        break
plt.tight_layout()
plt.show()
```

```
1/1 [==============================] - 1s 827ms/step
1/1 [==============================] - 0s 481ms/step
1/1 [==============================] - 0s 447ms/step
```




## Pre-trained model on imagenet dataset

While I did not expect this pre-trained model to be useful in this setting, I still wanted to try out what it might identify in my street view images.

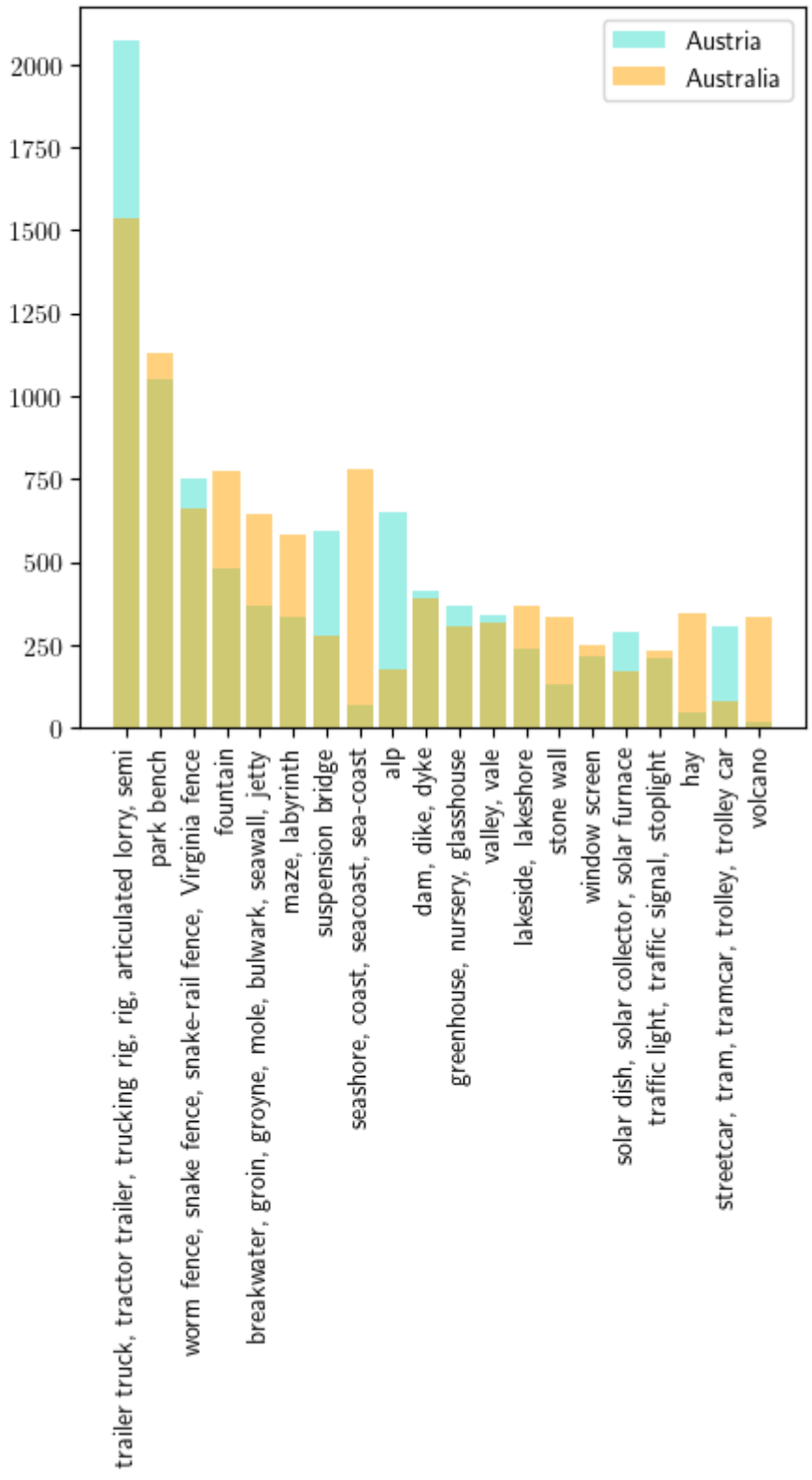
In [2]:
```
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2

imagenet_model = MobileNetV2(
```

```
    alpha = 1.0,
    include_top = True,
    weights = 'imagenet',
    input_tensor = None,
    pooling = None,
    classes = 1000,
    classifier_activation = 'softmax'
)

optimizer = optimizers.Adam()
imagenet_model.compile(optimizer = optimizer, metrics = ['accuracy'])
```

In [3]:
```
predictions = imagenet_model.predict(config.imagenet_ds)
```

```
784/784 [==============================] - 289s 368ms/step
```

In [4]:
```
import ast
fig, ax = plt.subplots(figsize = (5,5))

fig.suptitle("Top 20 class labels", size = 20)

with open('imagenet_classnames.txt') as f:
    class_names = ast.literal_eval(f.read())

labels = np.array([x for x in config.imagenet_ds.unbatch().map(lambda x,y: y)])

top_predictions = np.argmax(predictions, axis = 1)
unique, counts = np.unique(top_predictions, return_counts=True)

for j, (country,color) in enumerate(zip(("Austria", "Australia"),("turquoise", "ora
    country_counts = [np.count_nonzero(top_predictions[labels == j] == i) for i in
    ax.bar([class_names[j] for j in unique[np.argsort(-counts)]][:20], country_coun
             alpha = 0.5, label = country, color = color)

plt.xticks(rotation=90)
plt.legend()
plt.show()
```




There are of course some interesting correlations between these classes and the Austria vs. Australia distinction, like the alp category being much more prevalent in Austria and seashores being much more common in Australia. Overall however, I do not think these classes are useful enough to this task at hand to warrant using a pre-trained model like this.

In [6]:

```
num_classes = 20
fig = plt.figure(figsize=(15, 4*num_classes))
subfigs = fig.subfigures(nrows = num_classes, ncols = 1)

top_predictions = np.argmax(predictions, axis = 1)
unique, counts = np.unique(top_predictions, return_counts=True)

blank_image = np.ones((256,256,3))

axs_array = []
for row, subfig in enumerate(subfigs):
    subfig.suptitle(class_names[unique[np.argsort(-counts)][row]], size = 20)

    axs = subfig.subplots(nrows=1, ncols=5)
    for ax in axs:
        ax.imshow(blank_image)
        ax.axis("off")
    axs_array.append(axs)
confidence_values = np.zeros((num_classes,5))

for batch_number, (images, _) in enumerate(config.imagenet_ds):
    batch_predictions = predictions[32*batch_number:32*(batch_number+1)]
    for row, label in enumerate(unique[np.argsort(-counts)][:num_classes]):
        filter_array = top_predictions[32*batch_number:32*(batch_number+1)] == labe
        corresponding_images = images[filter_array]
        corresponding_predictions = batch_predictions[filter_array][:,label]
        sorted_array = sorted(zip(corresponding_predictions, corresponding_images),
        corresponding_images = [image for prediction, image in sorted_array]
        corresponding_predictions = [prediction for prediction, image in sorted_arr
        image_counter = 0
        for i in range(5):
            if image_counter >= len(corresponding_images):
                break
            prediction = corresponding_predictions[image_counter]
            if prediction > confidence_values[row, i]:
                confidence_values[row, i] = prediction
                axs_array[row][i].imshow(data_visualization.rescale_image(correspon
                axs_array[row][i].set_title(f"Confidence: {prediction:.2f}")
                image_counter += 1

plt.tight_layout()
plt.show()
```

trailer truck, tractor trailer, trucking rig, rig, articulated lorry, semi

Confidence: 0.98 Confidence: 0.96 Confidence: 0.96 Confidence: 0.95 Confidence: 0.94

park bench

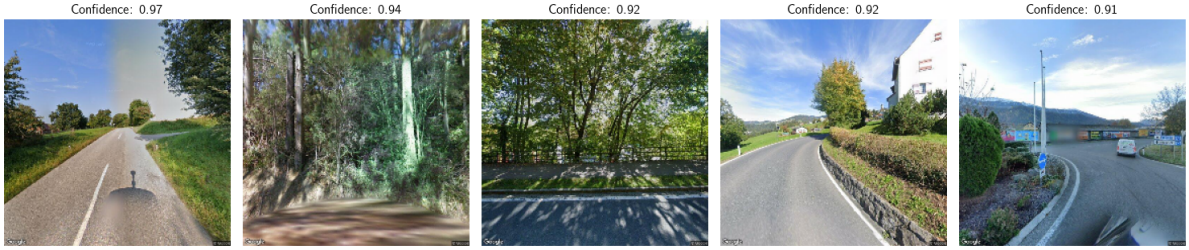
Confidence: 0.94 Confidence: 0.93 Confidence: 0.92 Confidence: 0.90 Confidence: 0.85

worm fence, snake fence, snake-rail fence, Virginia fence

Confidence: 0.99 Confidence: 0.95 Confidence: 0.88 Confidence: 0.76 Confidence: 0.76

fountain

Confidence: 0.97 Confidence: 0.94 Confidence: 0.92 Confidence: 0.92 Confidence: 0.91



breakwater, groin, groyne, mole, bulwark, seawall, jetty

Confidence: 0.89 Confidence: 0.87 Confidence: 0.82 Confidence: 0.81 Confidence: 0.80

maze, labyrinth

Confidence: 0.99 Confidence: 0.97 Confidence: 0.96 Confidence: 0.96 Confidence: 0.95

suspension bridge

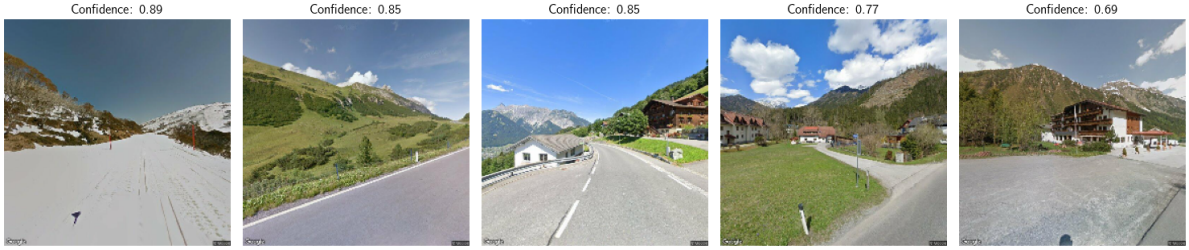
Confidence: 0.98 Confidence: 0.92 Confidence: 0.88 Confidence: 0.88 Confidence: 0.87

seashore, coast, seacoast, sea-coast

Confidence: 0.94 Confidence: 0.93 Confidence: 0.83 Confidence: 0.82 Confidence: 0.80

alp

Confidence: 0.89 Confidence: 0.85 Confidence: 0.85 Confidence: 0.77 Confidence: 0.69



dam, dike, dyke

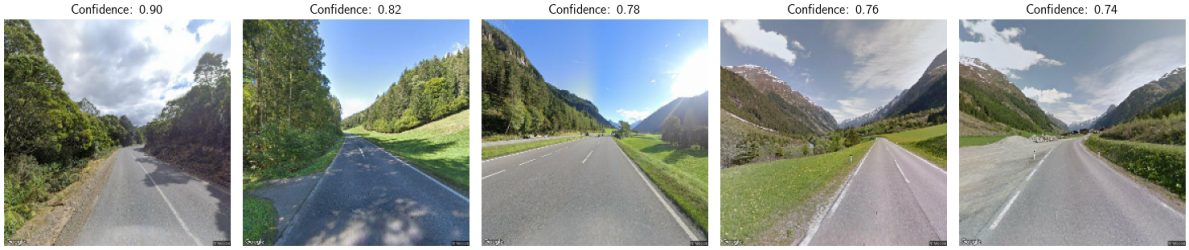
Confidence: 0.94 Confidence: 0.92 Confidence: 0.80 Confidence: 0.80 Confidence: 0.74

greenhouse, nursery, glasshouse

Confidence: 0.73 Confidence: 0.70 Confidence: 0.66 Confidence: 0.64 Confidence: 0.63

valley, vale

Confidence: 0.90 Confidence: 0.82 Confidence: 0.78 Confidence: 0.76 Confidence: 0.74



lakeside, lakeshore

Confidence: 0.93 Confidence: 0.88 Confidence: 0.81 Confidence: 0.78 Confidence: 0.76

stone wall

Confidence: 0.99 Confidence: 0.99 Confidence: 0.95 Confidence: 0.88 Confidence: 0.74

window screen

Confidence: 0.98 Confidence: 0.78 Confidence: 0.67 Confidence: 0.66 Confidence: 0.58

solar dish, solar collector, solar furnace

Confidence: 0.94 Confidence: 0.93 Confidence: 0.91 Confidence: 0.91 Confidence: 0.91

traffic light, traffic signal, stoplight

Confidence: 0.99 Confidence: 0.97 Confidence: 0.94 Confidence: 0.92 Confidence: 0.88

hay

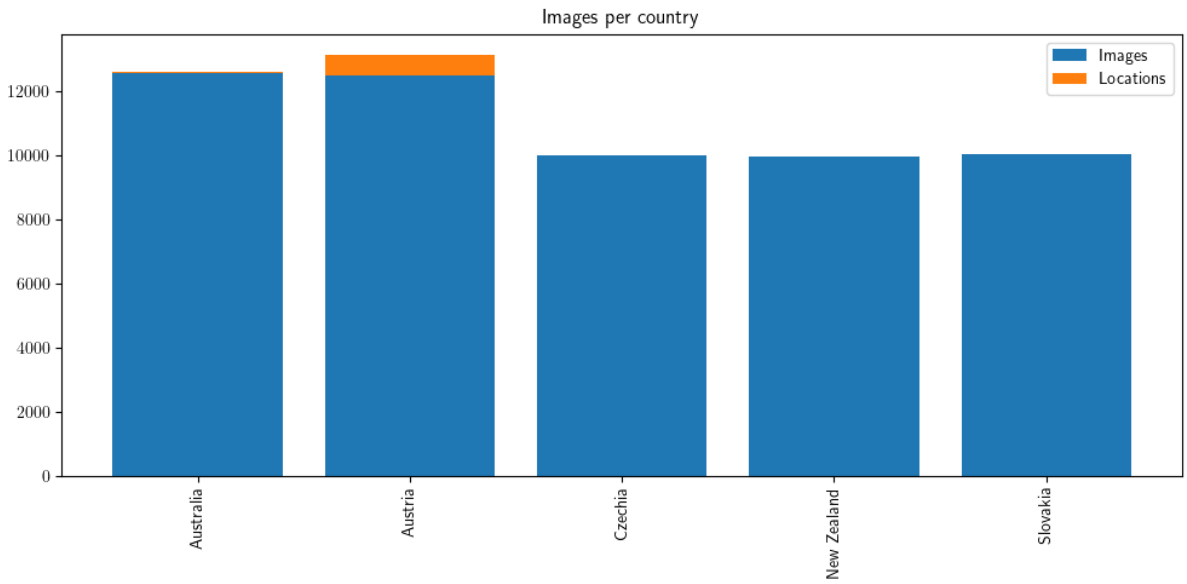
Confidence: 0.86 Confidence: 0.83 Confidence: 0.80 Confidence: 0.69 Confidence: 0.65


# Multinomial Classification

In [34]:
```
for country in ["Australia", "Austria", "Czechia", "New Zealand", "Slovakia"]:
    for state in config.states_dict[country]:
        df = pd.read_csv(f"../images_multinomial/{country}/{state}/coordinates.csv"
        image_exists = []
        for index, row in df.iterrows():
            filepath = f"../images_multinomial/{country}/{state}/{int(row['i'])}.jp
            if os.path.isfile(filepath):
                image_exists.append(True)
            else:
                image_exists.append(False)
        df["image_exists"] = image_exists
        df.to_csv(f"../images_multinomial/{country}/{state}/coordinates.csv")
```

In [37]:
```
fig, ax = plt.subplots(figsize = (10,5))
countries = ["Australia", "Austria", "Czechia", "New Zealand", "Slovakia"]

num_locations_per_country = []
num_images_per_country = []
for (i,country) in enumerate(countries):
    num_images, num_locations = 0, 0
    for state in config.states_dict[country]:
        df = pd.read_csv(f"../images_multinomial/{country}/{state}/coordinates.csv"
        num_images += np.sum(df["image_exists"])
        num_locations += len(df)
    num_images_per_country.append(num_images)
    num_locations_per_country.append(num_locations)

ax.bar(countries, num_images_per_country, label = "Images")
ax.bar(countries, np.array(num_locations_per_country) - np.array(num_images_per_cou
ax.set_title("Images per country")
ax.tick_params('x', labelrotation=45)
ax.legend()
```

```
plt.tight_layout()
plt.show()
```




In [ ]:
```
model = keras.Sequential(
    [
    layers.Conv2D(32, (3,3), input_shape = (img_height,img_width,3), activation = "
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Conv2D(32, (3,3), activation = "relu", padding="same"),
    layers.MaxPooling2D((2,2), padding="same"),
    layers.Flatten(),
    layers.Dense(100),
    layers.Dropout(0.1),
    layers.Dense(5, activation = 'softmax'),
    ])

optimizer = optimizers.Adam(learning_rate = 0.001)

model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics = ["acc
```

In [ ]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 256, 256, 32)      896

 max_pooling2d (MaxPooling2D  (None, 128, 128, 32)     0
 )

 conv2d_1 (Conv2D)           (None, 128, 128, 32)      9248

 max_pooling2d_1 (MaxPooling  (None, 64, 64, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 64, 64, 32)        9248

 max_pooling2d_2 (MaxPooling  (None, 32, 32, 32)       0
 2D)

 conv2d_3 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d_3 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 16, 16, 32)        9248

 max_pooling2d_4 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_5 (Conv2D)           (None, 8, 8, 32)          9248

 max_pooling2d_5 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 100)               51300

 dropout (Dropout)           (None, 100)               0

 dense_1 (Dense)             (None, 5)                 505

=================================================================
Total params: 98,941
Trainable params: 98,941
Non-trainable params: 0
_________________________________________________________________
```

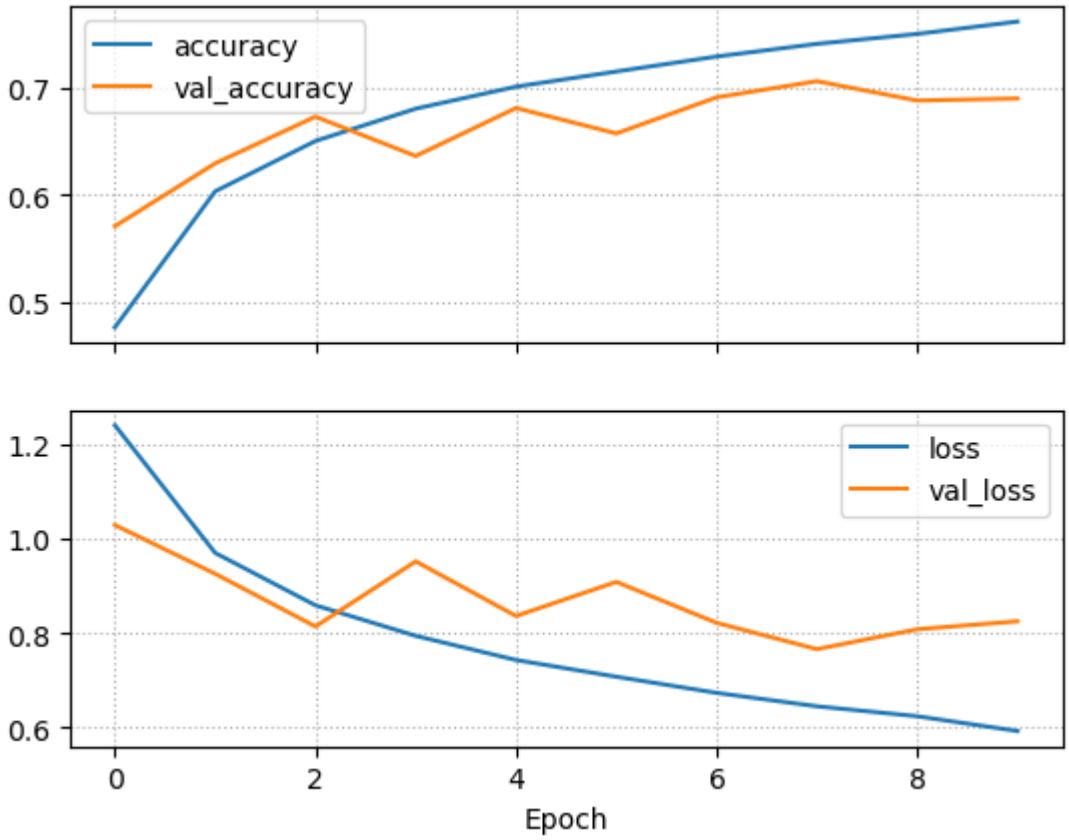
In [ ]:
```
history = model.fit(train_ds_multinomial, epochs = 10, validation_data = val_ds_mul
```

```
Epoch 1/10
1377/1377 [==============================] - 1130s 819ms/step - loss: 1.2402 - acc
uracy: 0.4758 - val_loss: 1.0288 - val_accuracy: 0.5707
Epoch 2/10
1377/1377 [==============================] - 1224s 888ms/step - loss: 0.9698 - acc
uracy: 0.6034 - val_loss: 0.9257 - val_accuracy: 0.6296
Epoch 3/10
1377/1377 [==============================] - 1271s 923ms/step - loss: 0.8589 - acc
uracy: 0.6506 - val_loss: 0.8141 - val_accuracy: 0.6735
Epoch 4/10
1377/1377 [==============================] - 1131s 820ms/step - loss: 0.7942 - acc
uracy: 0.6810 - val_loss: 0.9519 - val_accuracy: 0.6365
Epoch 5/10
1377/1377 [==============================] - 1112s 807ms/step - loss: 0.7431 - acc
uracy: 0.7016 - val_loss: 0.8363 - val_accuracy: 0.6816
Epoch 6/10
1377/1377 [==============================] - 1123s 815ms/step - loss: 0.7077 - acc
uracy: 0.7158 - val_loss: 0.9085 - val_accuracy: 0.6577
Epoch 7/10
1377/1377 [==============================] - 1118s 811ms/step - loss: 0.6736 - acc
uracy: 0.7299 - val_loss: 0.8218 - val_accuracy: 0.6916
Epoch 8/10
1377/1377 [==============================] - 1085s 787ms/step - loss: 0.6452 - acc
uracy: 0.7417 - val_loss: 0.7661 - val_accuracy: 0.7067
Epoch 9/10
1377/1377 [==============================] - 1111s 806ms/step - loss: 0.6241 - acc
uracy: 0.7511 - val_loss: 0.8082 - val_accuracy: 0.6886
Epoch 10/10
1377/1377 [==============================] - 1117s 811ms/step - loss: 0.5928 - acc
uracy: 0.7627 - val_loss: 0.8251 - val_accuracy: 0.6905
```

In [ ]:
```
model.save('multinomial_neural_net')
```

```
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_c
ompiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_o
p, _jit_compiled_convolution_op while saving (showing 5 of 6). These functions wil
l not be directly callable after loading.
INFO:tensorflow:Assets written to: multinomial_neural_net/assets
INFO:tensorflow:Assets written to: multinomial_neural_net/assets
```

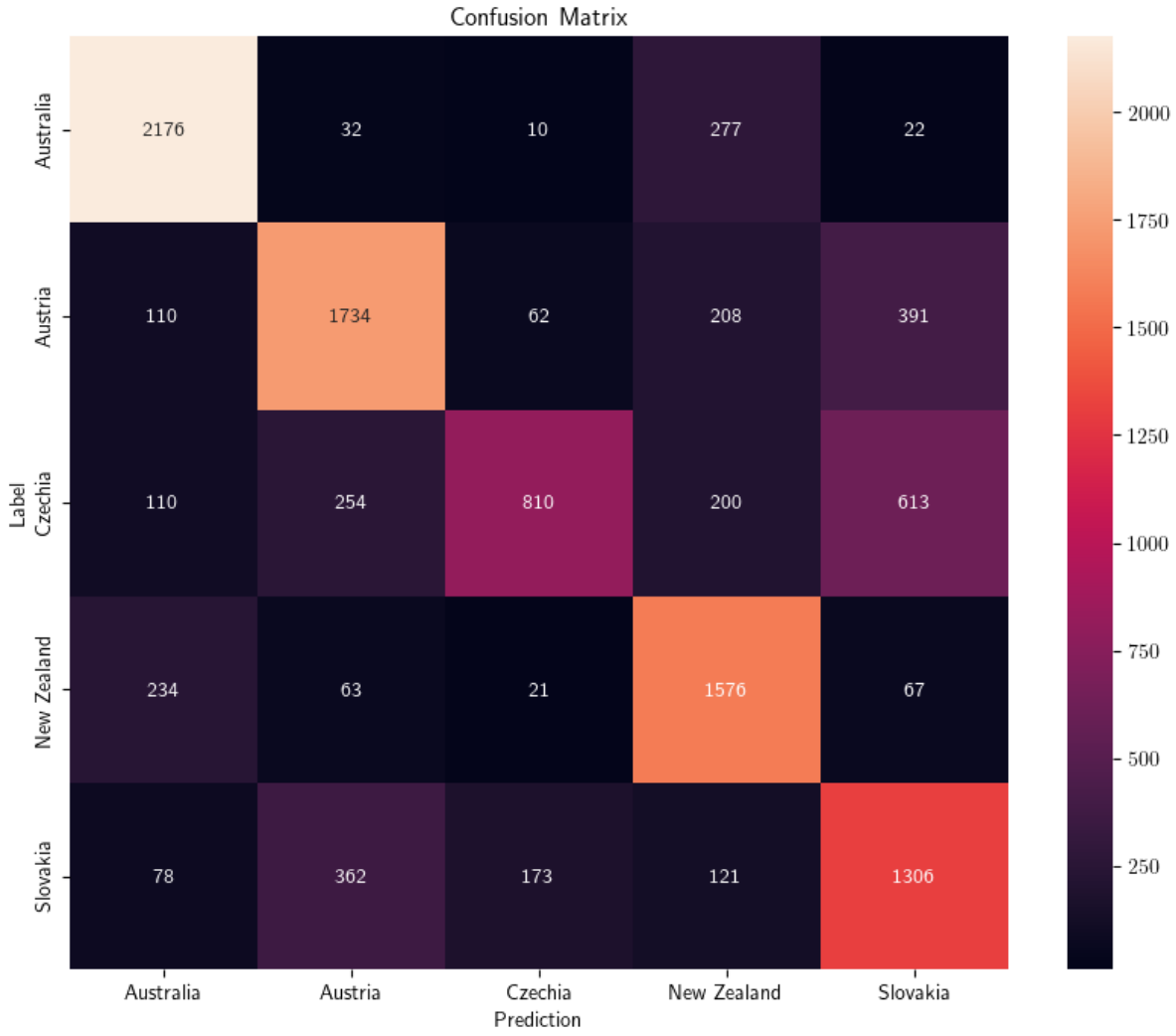
In [ ]:
```
fig, axs = plt.subplots(2,1, sharex = True)
keys = [["accuracy", "val_accuracy"], ["loss", "val_loss"]]
for (ax, k) in zip(axs,keys):
    for key in k:
        ax.plot(history.history[key], label = key)

    ax.legend()
    ax.grid(linestyle = "dotted")

plt.xlabel("Epoch")
plt.show()
```




In [4]:
```
multinomial_predictions = config.multinomial_model.predict(config.val_ds_multinomia
top_multinomial_predictions = np.argmax(predictions, axis = 1)
multinomial_labels = np.array([x for x in config.val_ds_multinomial.unbatch().map(l
```

```
345/345 [==============================] - 74s 211ms/step
```

In [6]:
```
import seaborn as sns

confusion_matrix = tf.math.confusion_matrix(labels, top_multinomial_predictions)
multinomial_class_names = ["Australia", "Austria", "Czechia", "New Zealand", "Slova

plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix,
            xticklabels = multinomial_class_names,
            yticklabels = multinomial_class_names,
            annot=True, fmt='g')
plt.xlabel('Prediction')
plt.ylabel('Label')
plt.title('Confusion Matrix')
plt.show()
```




# Adversarial Machine Learning via Signed Gradients

In [2]:
```
def adversarial_transformation(model, eps):

    def func(image, label):

        with tf.GradientTape() as g:
            g.watch(image)
            predicted_label = model(image)
            loss = losses.BinaryCrossentropy()(label, predicted_label)

            gradient = g.gradient(target = loss, sources = image)
            signed_gradient = tf.sign(gradient)
            adv_image = image + signed_gradient * eps

        return adv_image, label

    return func
```

In [5]:
```
eps_array = [0,1e-3,1e-2,1e-1,1e-0]
```

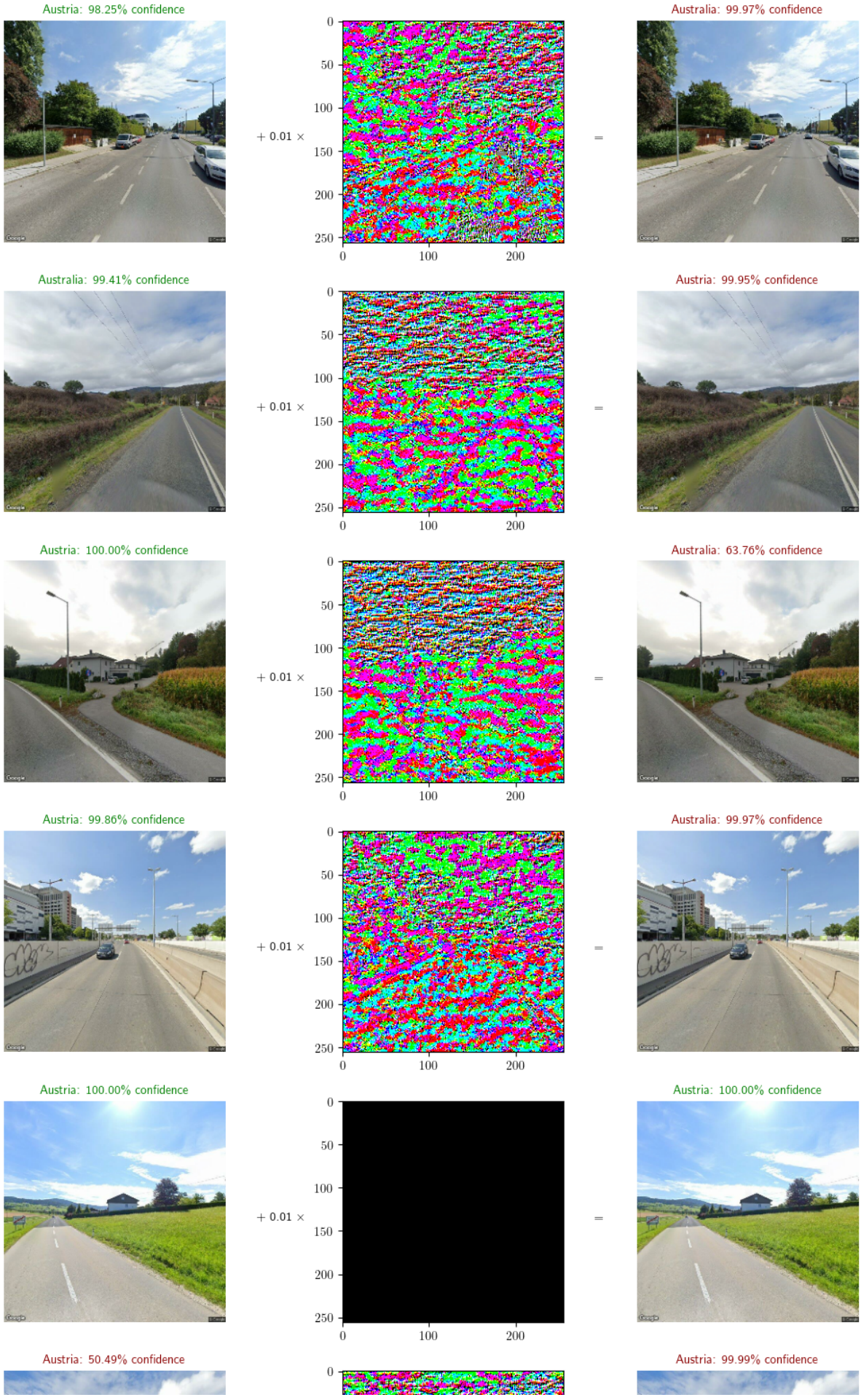
```
for eps in eps_array:
    print(f"{eps=}")
    if eps == 0:
        loss, accuracy = config.model.evaluate(config.val_ds)
    else:
        adv_ds = config.val_ds.map(adversarial_transformation(config.model, eps))
        loss, accuracy = config.model.evaluate(adv_ds)
```

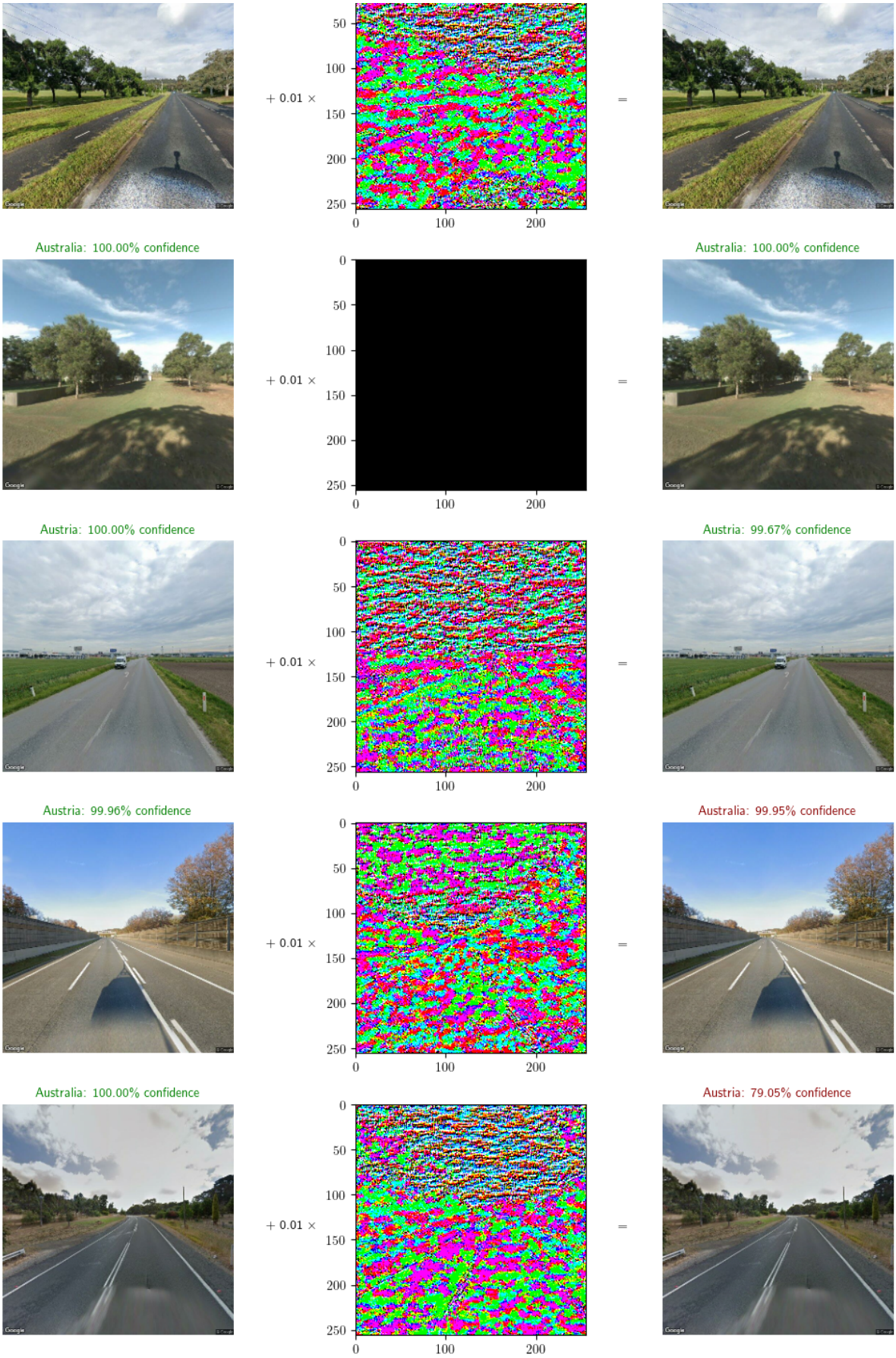
```
eps=0
157/157 [==============================] - 74s 469ms/step - loss: 0.2390 - accurac
y: 0.9342
eps=0.001
157/157 [==============================] - 513s 3s/step - loss: 0.4154 - accuracy:
0.8899
eps=0.01
157/157 [==============================] - 490s 3s/step - loss: 3.3498 - accuracy:
0.4523
eps=0.1
157/157 [==============================] - 500s 3s/step - loss: 10.6281 - accurac
y: 0.2310
eps=1.0
157/157 [==============================] - 527s 3s/step - loss: 4.8793 - accuracy:
0.4360
```

In [4]:

```
from adversarial_machine_learning import adversarial_transformation

eps = 0.01
num_images = 10
fig = plt.figure(figsize = (10, num_images*3), constrained_layout=True)
subfigs = fig.subfigures(nrows=num_images, ncols=1)

for images, labels in config.val_ds.shuffle(buffer_size = 1000).take(1):
    predictions = config.model.predict(images).reshape((32,))
    adv_images, _ = adversarial_transformation(config.model, eps)(images, labels)
    adv_predictions = config.model.predict(adv_images)
    i = 0
    for image, prediction, adv_image, adv_prediction, label in zip(images, predicti
        data_visualization.plot_adversarial_image(image, adv_image, label, predicti
        i += 1
        if i >= num_images:
            break

plt.show()
```

```
1/1 [==============================] - 1s 1s/step
1/1 [==============================] - 1s 563ms/step
```


Austria: 98.25% confidence
+ 0.01 ×
=
Australia: 99.97% confidence
Australia: 99.41% confidence
+ 0.01 ×
=
Austria: 99.95% confidence
Austria: 100.00% confidence
+ 0.01 ×
=
Australia: 63.76% confidence
Austria: 99.86% confidence
+ 0.01 ×
=
Australia: 99.97% confidence
Austria: 100.00% confidence
+ 0.01 ×
=
Austria: 100.00% confidence
Austria: 50.49% confidence
Austria: 99.99% confidence
0
50
100
150
200
250
0
100
200

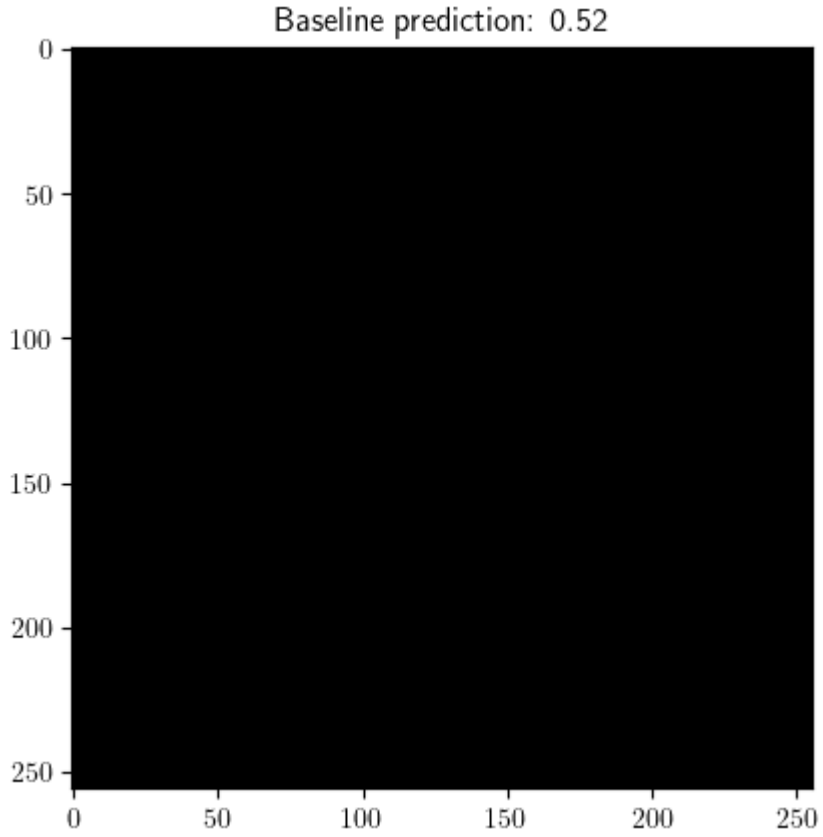



As illustrated above there are some images for which the signed gradients method fails, since the gradient is just zero. This happens, when the loss function attains a minimum at the

currently selected image. In practice, this means that the prediction is close enough to 1 (or 0) that it just gets rounded to 1.

## Model Explainability via Integrated Gradients

In [7]:
```
baseline = tf.zeros(shape = ((256,256,3)))

baseline_prediction = config.model(tf.expand_dims(baseline, axis = 0))
plt.imshow(baseline)
plt.title(f"Baseline prediction: {baseline_prediction[0,0]:.2f}")
plt.show()
```




In [11]:
```
fig = plt.figure(figsize=(20, 20))

for images, labels in config.val_ds.take(1):
    image = images[0]
    image = data_visualization.rescale_image(image, 0, 1)
    for i, alpha in enumerate(np.linspace(0,1,11)):
      plt.subplot(1, 11, i + 1)
      plt.title(f'alpha: {alpha:.1f}')
      plt.imshow(alpha*image)
      plt.axis('off')

    plt.tight_layout()
plt.show()
```

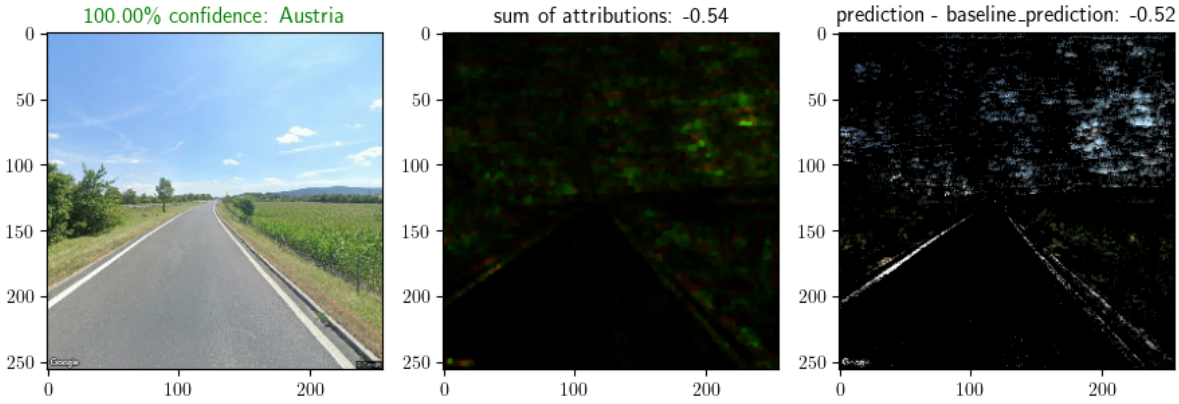

In [17]:
```
def get_attributions(image, baseline, m):
    grads = np.zeros((m+1,32, 256,256,3))
    for i in range(m+1):
        current_sample = baseline + (image - baseline)*i/m
        with tf.GradientTape() as g:
            g.watch(current_sample)
            predicted_label = config.model(current_sample)
            grads[i] = g.gradient(target = predicted_label, sources = current_sampl

    avg_grads = np.average((grads[:-1] + grads[1:]) / 2.0, axis = 0)
    return avg_grads*(image-baseline)
```

In [2]:
```
from integrated_gradients import get_attributions
from adversarial_machine_learning import adversarial_transformation

# Calculate the integrated gradients

for images, labels in config.val_ds.take(1):
    attributions = get_attributions(images, config.baseline, 50)
    predictions = config.model(images)

    adv_images, _ = adversarial_transformation(config.model, 0.05)(images, labels)
    adv_attributions = get_attributions(adv_images, config.baseline, 50)
    adv_predictions = config.model(adv_images)
```

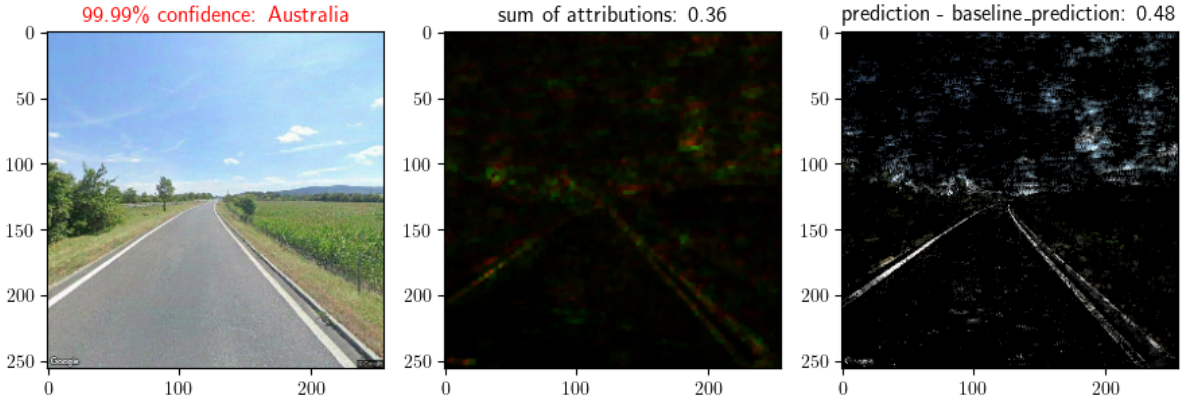
I use the visualization library https://github.com/ankurtaly/Integrated-Gradients/tree/master/VisualizationLibrary provided by Ankur Taly, one of the authors of the paper "Axiomatic Attribution for Deep Networks" to mimic the figures presented in this paper.

In [11]:
```
for images, labels in config.val_ds.take(1):
    selected_images = [7,14,15,19,20,27]
    num_images = len(selected_images)

    for i in selected_images:
        image = tf.reshape(images[i], (1,256,256,3))
        label = tf.reshape(labels[i], (1,1))
        adv_image = tf.reshape(adv_images[i], (1,256,256,3))
        prediction = predictions[i,0]

        plt.figure(figsize= (9,4))
        plt.suptitle("Original image", size = 20)
        data_visualization.visualize_integrated_gradients(image, label, predictions
        plt.show()

        plt.figure(figsize= (9,4))
        plt.suptitle("Adversarial image", size = 20)
        data_visualization.visualize_integrated_gradients(adv_image, label, adv_pre
        plt.show()
```

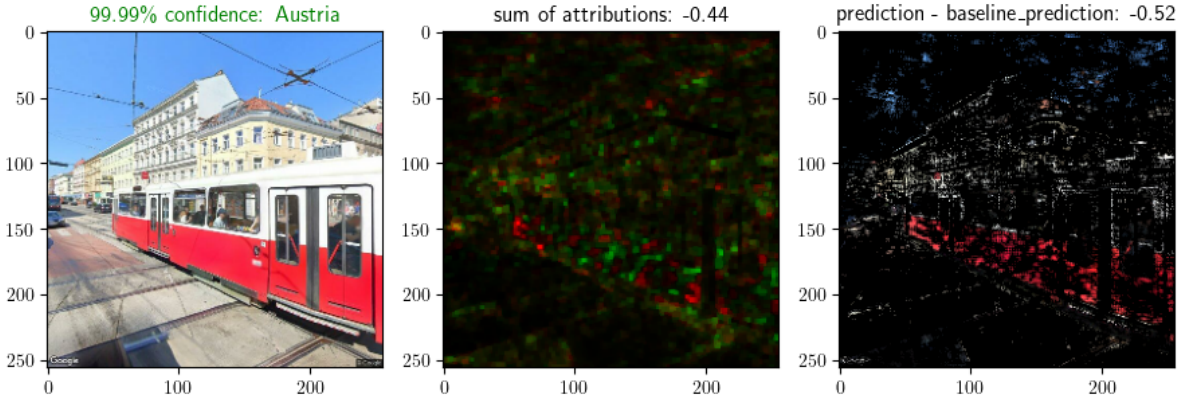
Original image



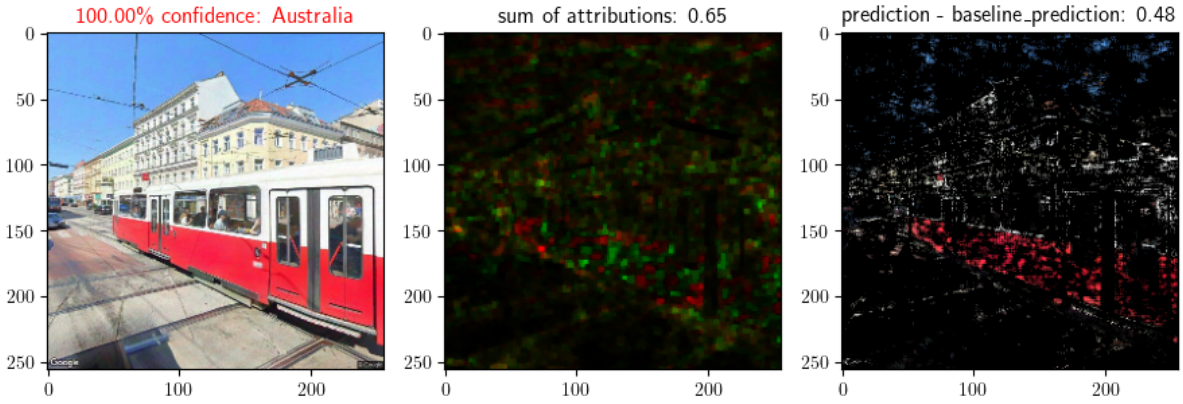

Adversarial image



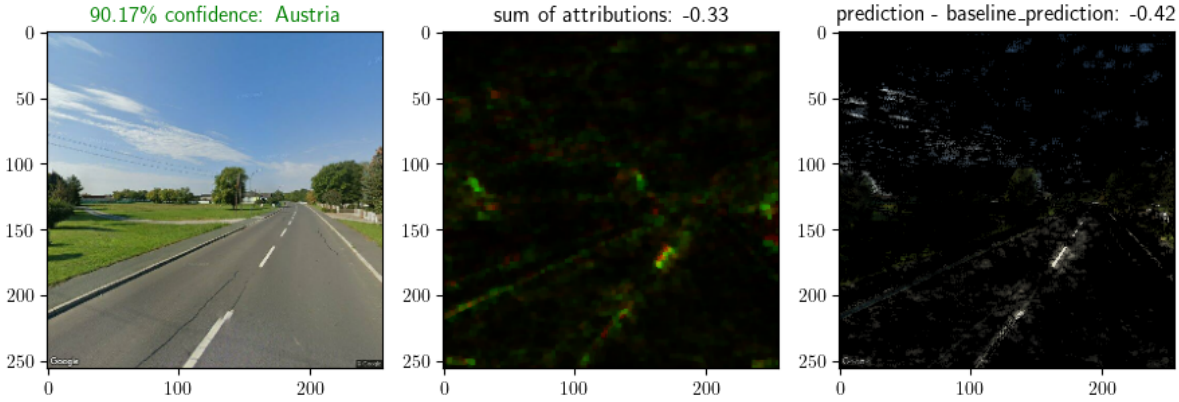

Original image



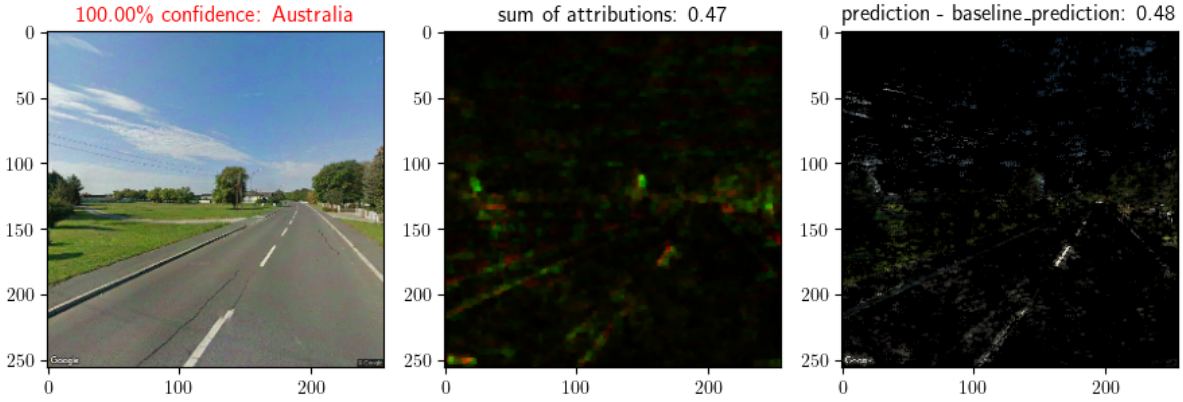
## Adversarial image



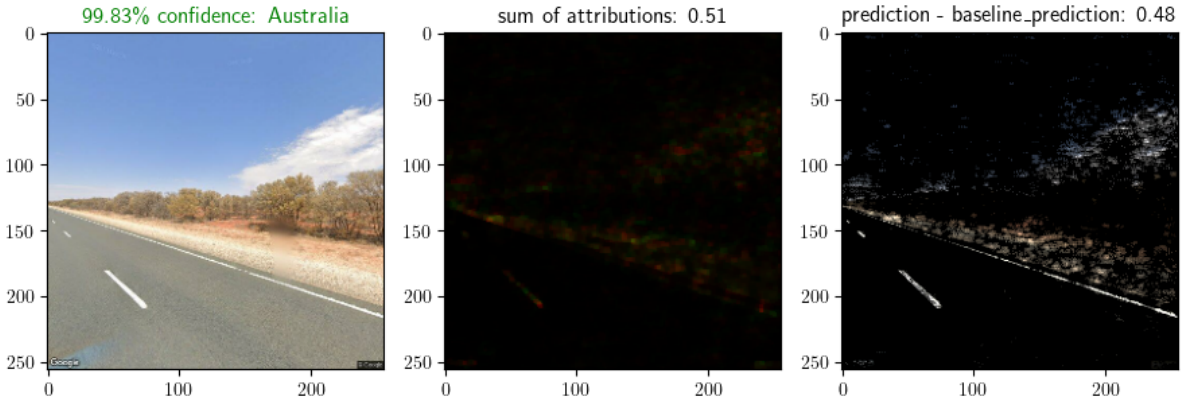

## Original image



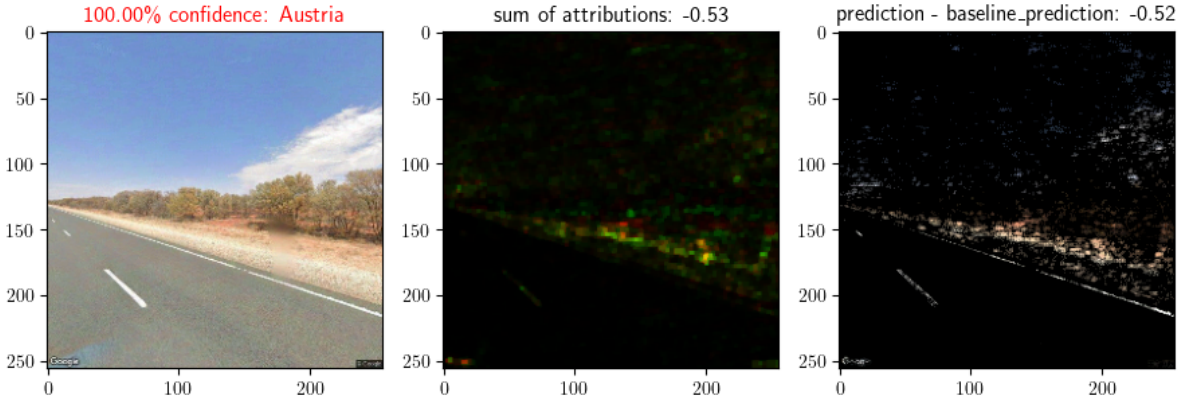

## Adversarial image



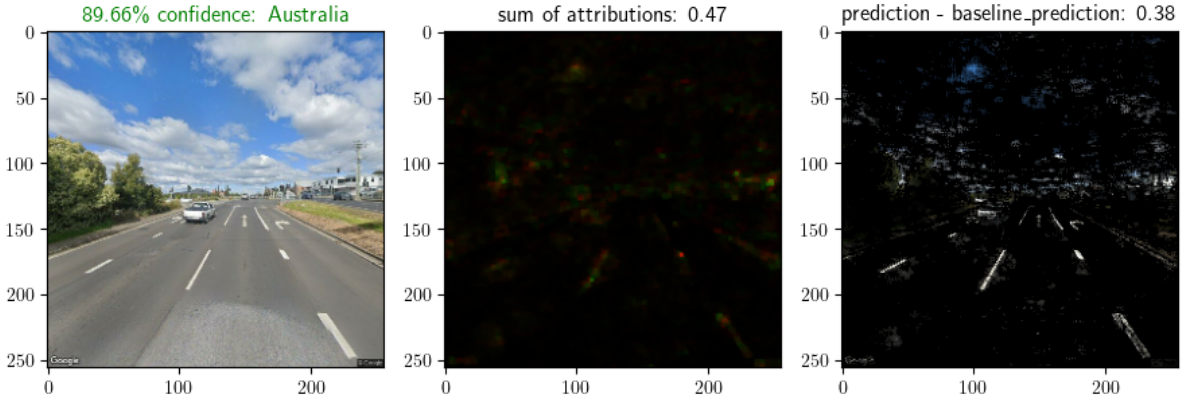
## Original image

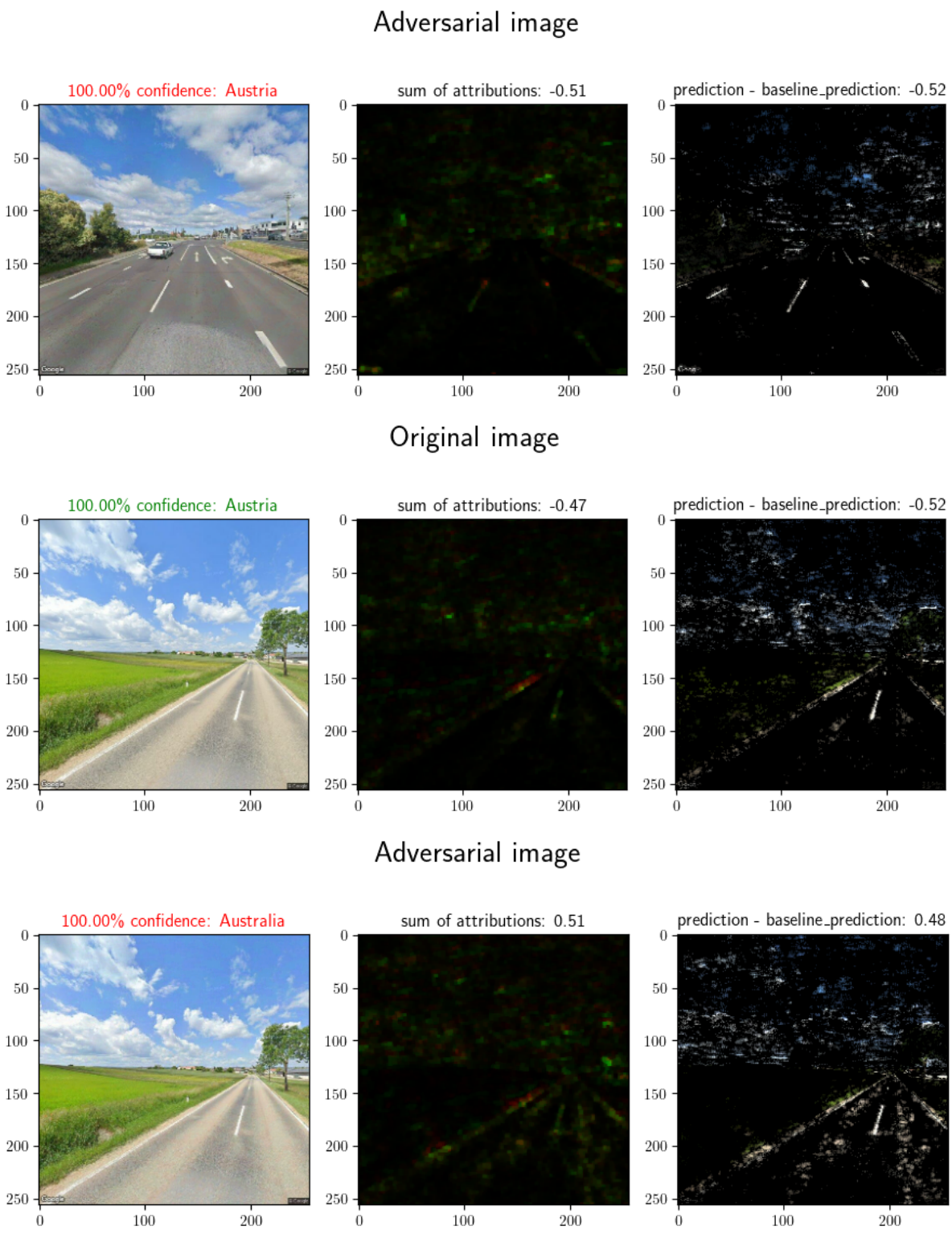



## Adversarial image




## Original image






Given that is seems harder to pinpoint, which pixels on a street view image present the biggest give-aways whether it is in Australia or Austria, then for cats and dogs, who at least have a clearly defined shape, in a lot of pictures the attributions are all over the place without an obvious interpretation. However there are still some clear patterns emerging: The most prominent feature seem to be the road lines (even though I did not even pick two countries with differing road line colors). In other images the network seems to focus much more on landscape, or sometimes even the sky.