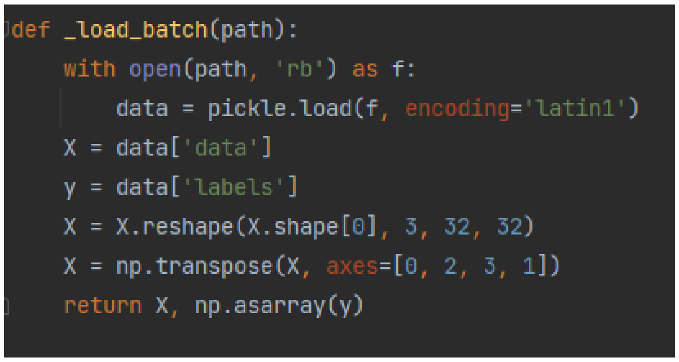
Demonstration report

Data generator

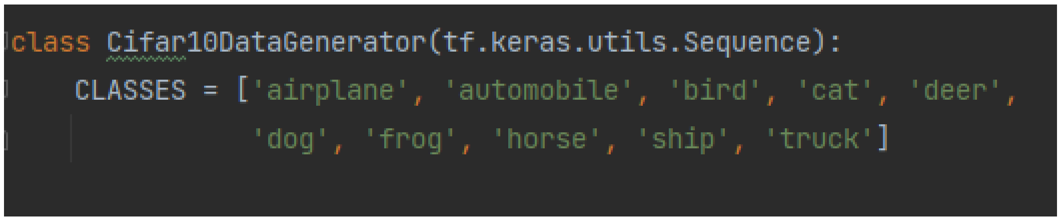
The first step is loading the data. The dataset is split into 6 batches that are loaded using the following function:



```python
def _load_batch(path):
    with open(path, 'rb') as f:
        data = pickle.load(f, encoding='latin1')
    X = data['data']
    y = data['labels']
    X = X.reshape(X.shape[0], 3, 32, 32)
    X = np.transpose(X, axes=[0, 2, 3, 1])
    return X, np.asarray(y)
```

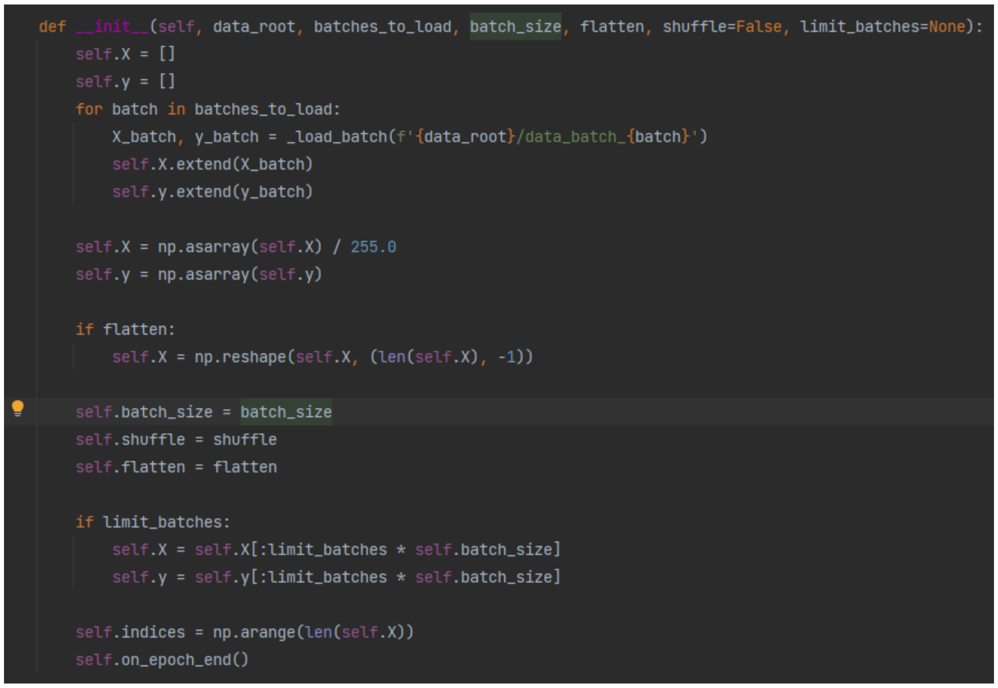
The images and labels are stored in a pickle file, and the images in the original format have the channels on the first dimension, and we convert them to images with channels on the last dimension.

In order to integrate the data generator with the Keras functionalities, we define the following class:



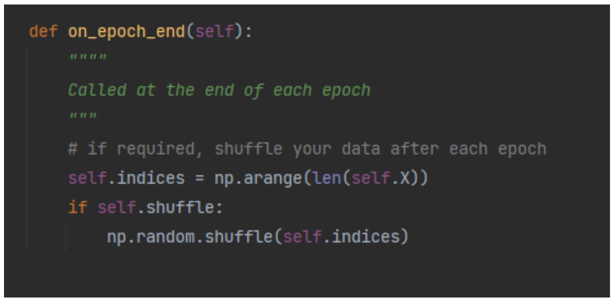
```python
class Cifar10DataGenerator(tf.keras.utils.Sequence):
    CLASSES = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
```

And in the constructor, we read the batches and normalize them in the 0-1 range by dividing with 255:



```python
def __init__(self, data_root, batches_to_load, batch_size, flatten, shuffle=False, limit_batches=None):
    self.X = []
    self.y = []
    for batch in batches_to_load:
        X_batch, y_batch = _load_batch(f'{data_root}/data_batch_{batch}')
        self.X.extend(X_batch)
        self.y.extend(y_batch)

    self.X = np.asarray(self.X) / 255.0
    self.y = np.asarray(self.y)

    if flatten:
        self.X = np.reshape(self.X, (len(self.X), -1))

    self.batch_size = batch_size
    self.shuffle = shuffle
    self.flatten = flatten

    if limit_batches:
        self.X = self.X[:limit_batches * self.batch_size]
        self.y = self.y[:limit_batches * self.batch_size]

    self.indices = np.arange(len(self.X))
    self.on_epoch_end()
```

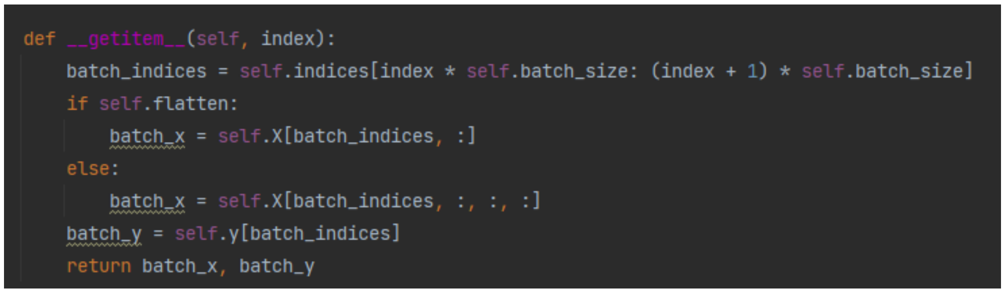
Also, we have the option to shuffle the dataset:



```
def on_epoch_end(self):
    """"
    Called at the end of each epoch
    """
    # if required, shuffle your data after each epoch
    self.indices = np.arange(len(self.X))
    if self.shuffle:
        np.random.shuffle(self.indices)
```

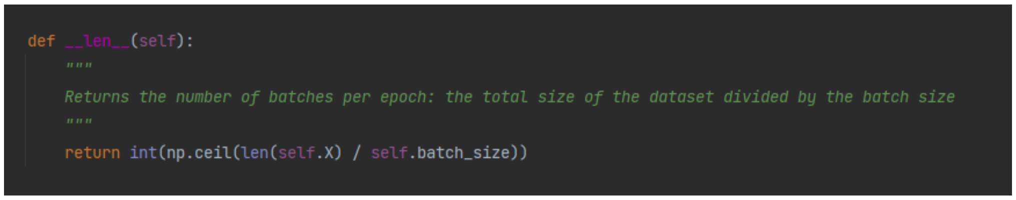
This will be called automatically by Keras after each epoch.

The data is read in batches:



```
def __getitem__(self, index):
    batch_indices = self.indices[index * self.batch_size: (index + 1) * self.batch_size]
    if self.flatten:
        batch_x = self.X[batch_indices, :]
    else:
        batch_x = self.X[batch_indices, :, :, :]
    batch_y = self.y[batch_indices]
    return batch_x, batch_y
```

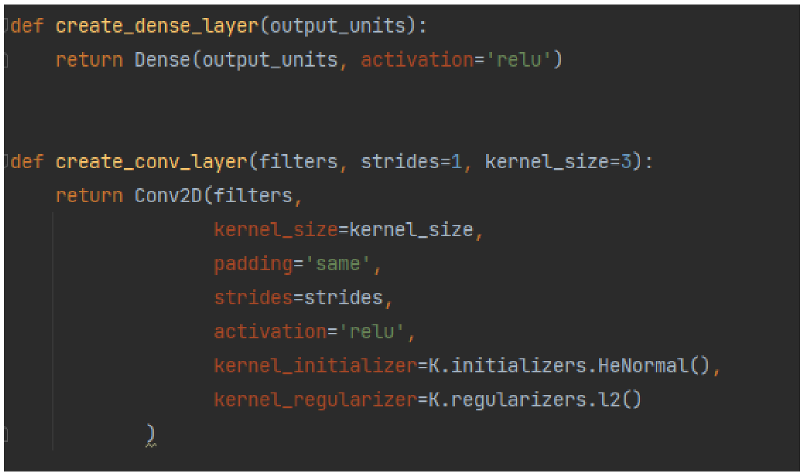
And the length of the dataset is given by the number of batches:



```
def __len__(self):
    """
    Returns the number of batches per epoch: the total size of the dataset divided by the batch size
    """
    return int(np.ceil(len(self.X) / self.batch_size))
```

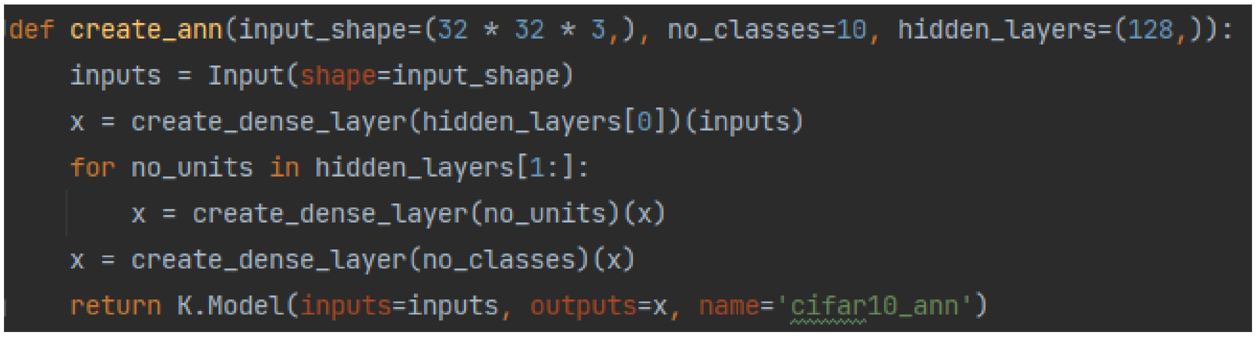
Models

In order to create the model architectures, we define the following functions for creating dense and convolutional layers:



```
def create_dense_layer(output_units):
    return Dense(output_units, activation='relu')


def create_conv_layer(filters, strides=1, kernel_size=3):
    return Conv2D(filters,
                  kernel_size=kernel_size,
                  padding='same',
                  strides=strides,
                  activation='relu',
                  kernel_initializer=K.initializers.HeNormal(),
                  kernel_regularizer=K.regularizers.l2()
           )
```

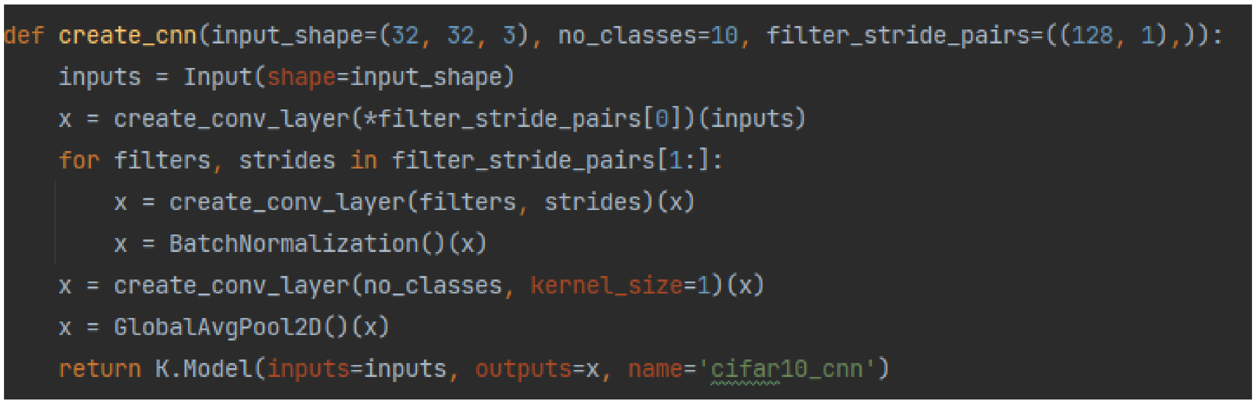
In order to create the actual models, we use a functional style of defining them. We define the ANN as:



```python
def create_ann(input_shape=(32 * 32 * 3,), no_classes=10, hidden_layers=(128,)):
    inputs = Input(shape=input_shape)
    x = create_dense_layer(hidden_layers[0])(inputs)
    for no_units in hidden_layers[1:]:
        x = create_dense_layer(no_units)(x)
    x = create_dense_layer(no_classes)(x)
    return K.Model(inputs=inputs, outputs=x, name='cifar10_ann')
```

The hidden_layers parameter controls the number of units in each hidden layer. Also, a final dense layer is needed to output the logits for the classes.

In a similar fashion, we define the CNN:

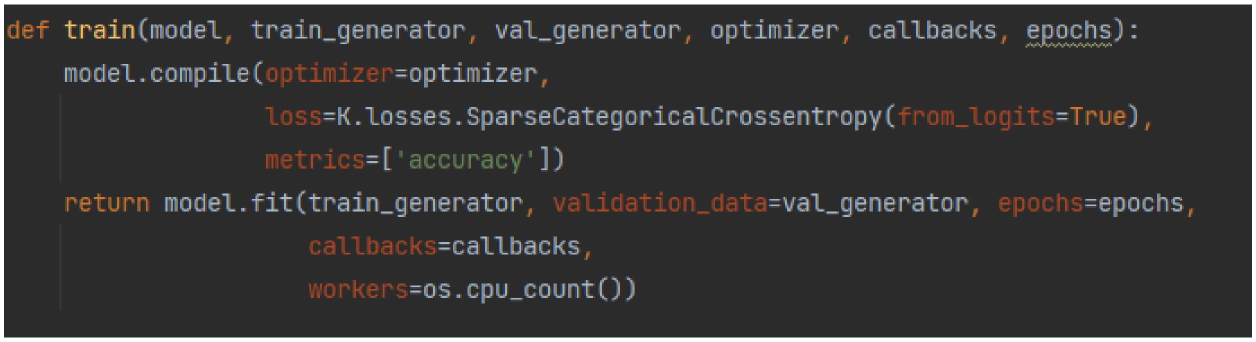


```python
def create_cnn(input_shape=(32, 32, 3), no_classes=10, filter_stride_pairs=((128, 1),)):
    inputs = Input(shape=input_shape)
    x = create_conv_layer(*filter_stride_pairs[0])(inputs)
    for filters, strides in filter_stride_pairs[1:]:
        x = create_conv_layer(filters, strides)(x)
        x = BatchNormalization()(x)
    x = create_conv_layer(no_classes, kernel_size=1)(x)
    x = GlobalAvgPool2D()(x)
    return K.Model(inputs=inputs, outputs=x, name='cifar10_cnn')
```

But here we also use BatchNormalization layers, and in order to get the desired number of outputs we use a pointwise convolution in order to control the number of channels and followed by a GlobalAvgPool in order to get the vector of logits.

Training

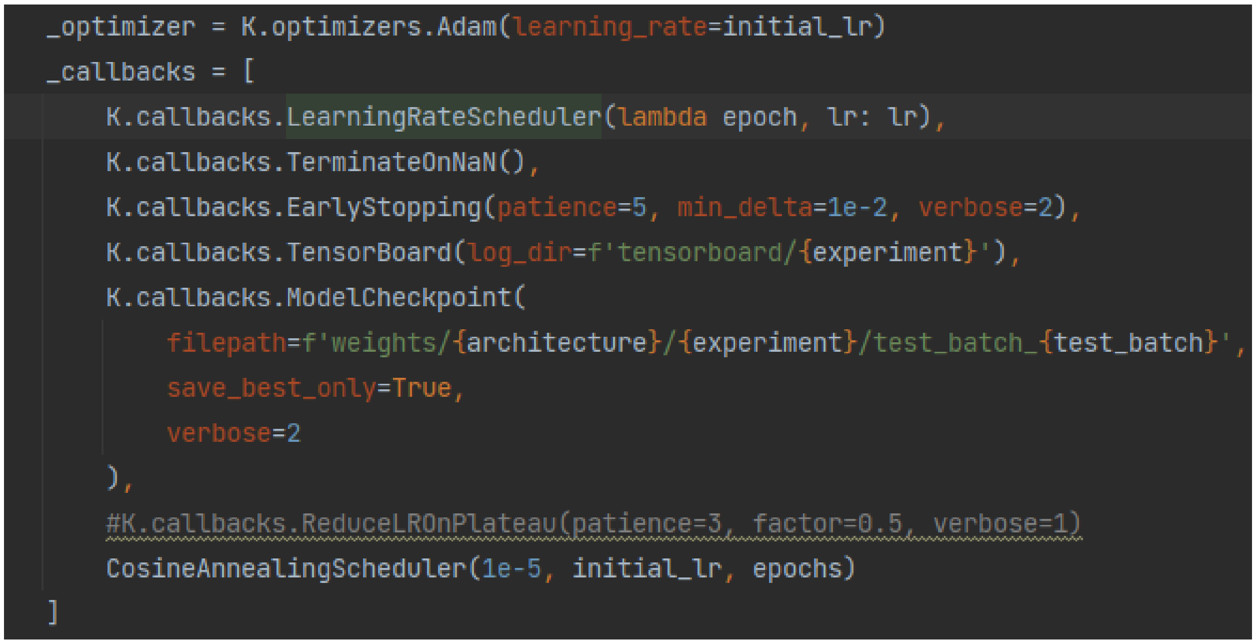
The next step is the training for which Keras handles most of the job:



```python
def train(model, train_generator, val_generator, optimizer, callbacks, epochs):
    model.compile(optimizer=optimizer,
                  loss=K.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])
    return model.fit(train_generator, validation_data=val_generator, epochs=epochs,
                     callbacks=callbacks,
                     workers=os.cpu_count())
```

The model is either the ANN or the CNN which are created using the functions described in Models.

We use the following optimizer and callbacks:

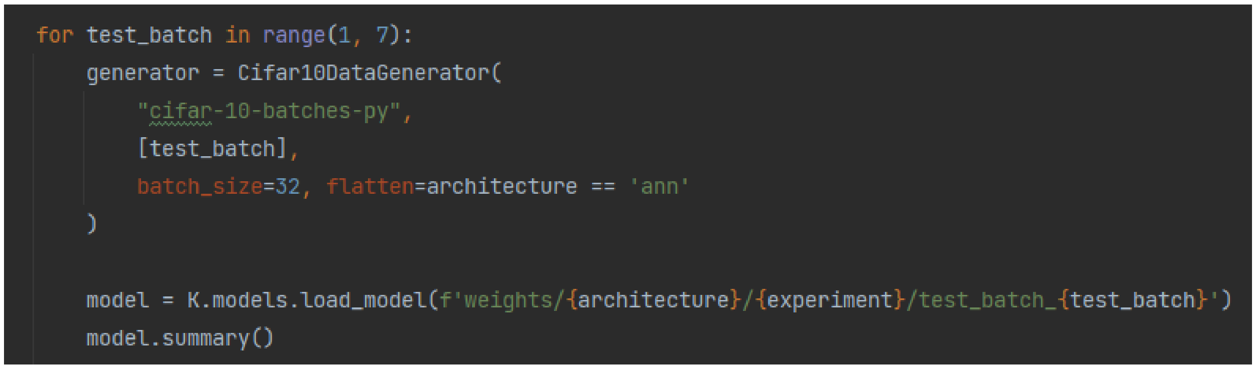


```
_optimizer = K.optimizers.Adam(learning_rate=initial_lr)
_callbacks = [
    K.callbacks.LearningRateScheduler(lambda epoch, lr: lr),
    K.callbacks.TerminateOnNaN(),
    K.callbacks.EarlyStopping(patience=5, min_delta=1e-2, verbose=2),
    K.callbacks.TensorBoard(log_dir=f'tensorboard/{experiment}'),
    K.callbacks.ModelCheckpoint(
        filepath=f'weights/{architecture}/{experiment}/test_batch_{test_batch}',
        save_best_only=True,
        verbose=2
    ),
    #K.callbacks.ReduceLROnPlateau(patience=3, factor=0.5, verbose=1)
    CosineAnnealingScheduler(1e-5, initial_lr, epochs)
]
```

Evaluation

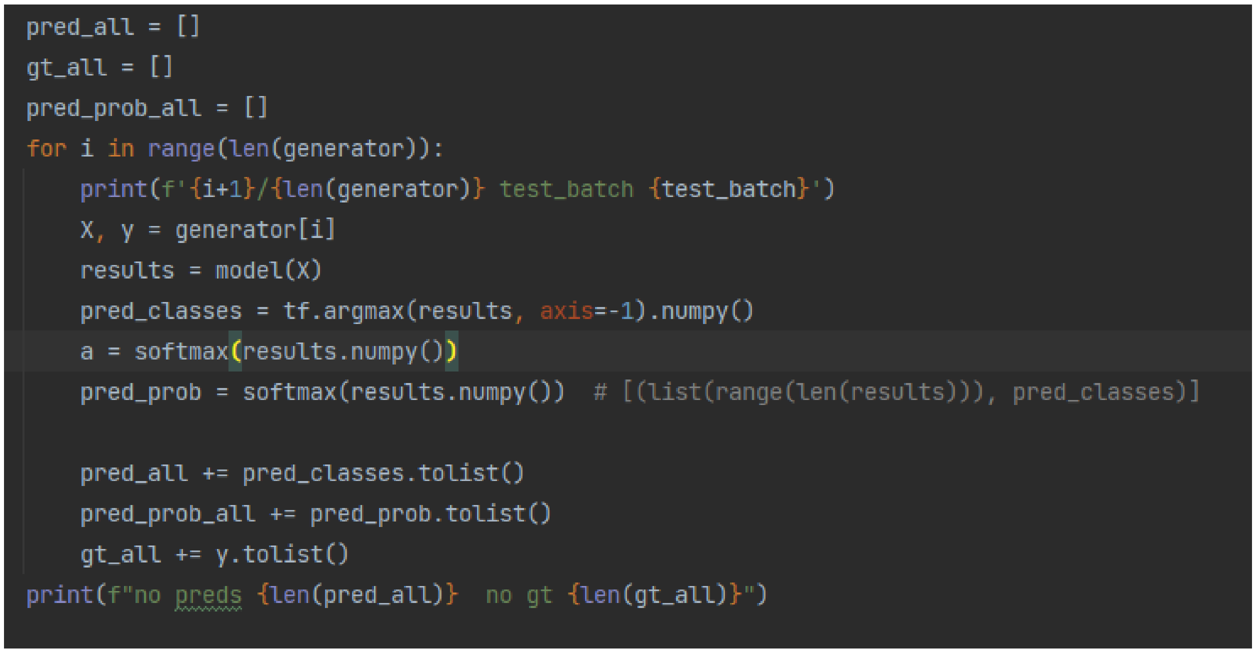
In order to perform the evaluation, we first generate a json file containing the ground truth and predicted labels, together with the probabilities for all classes.

This is done on each batch, by loading the corresponding data generator and model as follows:



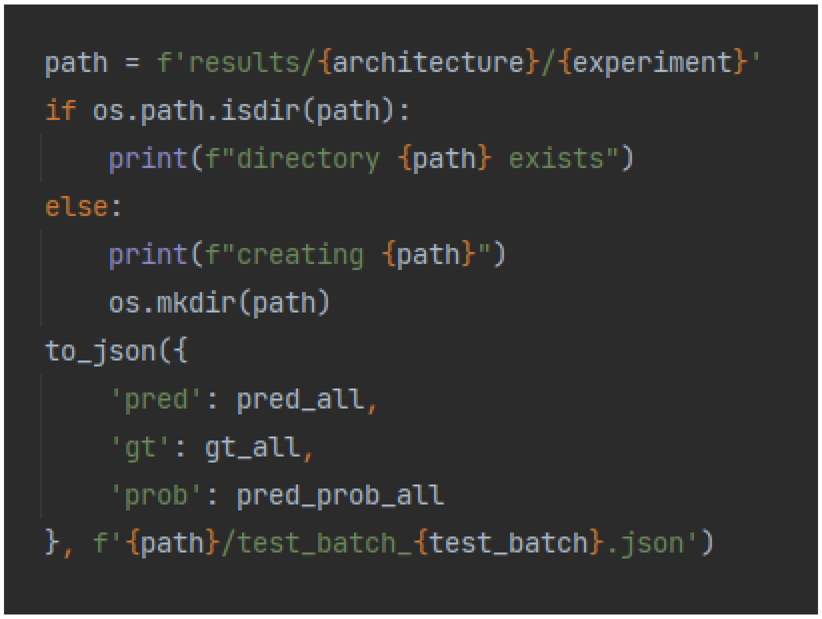
```
for test_batch in range(1, 7):
    generator = Cifar10DataGenerator(
        "cifar-10-batches-py",
        [test_batch],
        batch_size=32, flatten=architecture == 'ann'
    )

    model = K.models.load_model(f'weights/{architecture}/{experiment}/test_batch_{test_batch}')
    model.summary()
```

After this we compute the results for each image:



```python
pred_all = []
gt_all = []
pred_prob_all = []
for i in range(len(generator)):
    print(f'{i+1}/{len(generator)} test_batch {test_batch}')
    X, y = generator[i]
    results = model(X)
    pred_classes = tf.argmax(results, axis=-1).numpy()
    a = softmax(results.numpy())
    pred_prob = softmax(results.numpy())  # [(list(range(len(results))), pred_classes)]

    pred_all += pred_classes.tolist()
    pred_prob_all += pred_prob.tolist()
    gt_all += y.tolist()
print(f"no preds {len(pred_all)}  no gt {len(gt_all)}")
```

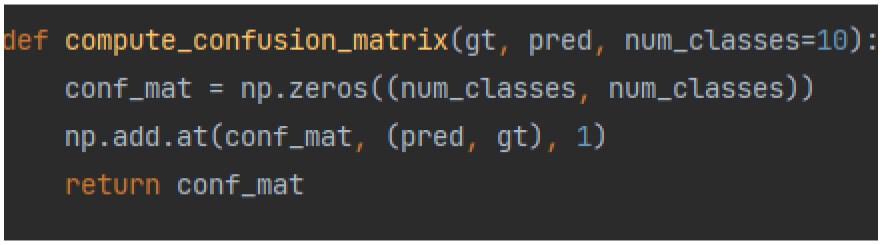
And save them as follows:



```python
path = f'results/{architecture}/{experiment}'
if os.path.isdir(path):
    print(f"directory {path} exists")
else:
    print(f"creating {path}")
    os.mkdir(path)
to_json({
    'pred': pred_all,
    'gt': gt_all,
    'prob': pred_prob_all
}, f'{path}/test_batch_{test_batch}.json')
```

We compute the following metrics: accuracy, precision, recall, f-score and AUC.

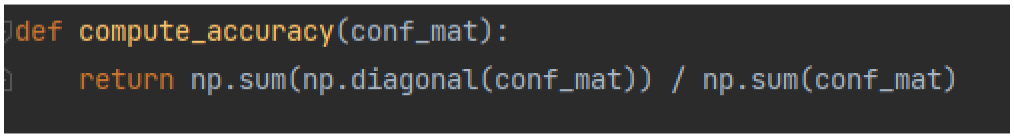
In order to compute them, first the confusion matrix must be computed:



```python
def compute_confusion_matrix(gt, pred, num_classes=10):
    conf_mat = np.zeros((num_classes, num_classes))
    np.add.at(conf_mat, (pred, gt), 1)
    return conf_mat
```

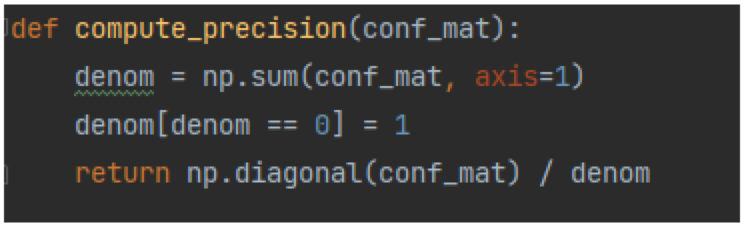
Both pred and gt are lists containing the labels, but the labels can also be seen as indices in the matrix, therefore at each position described by the pair of indexes (pred, gt) we add 1.

Accuracy:



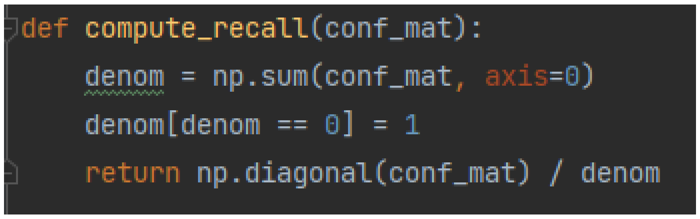
```python
def compute_accuracy(conf_mat):
    return np.sum(np.diagonal(conf_mat)) / np.sum(conf_mat)
```

Precision:



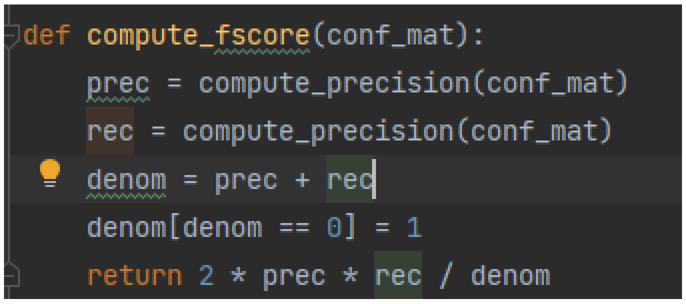
```python
def compute_precision(conf_mat):
    denom = np.sum(conf_mat, axis=1)
    denom[denom == 0] = 1
    return np.diagonal(conf_mat) / denom
```

Recall:



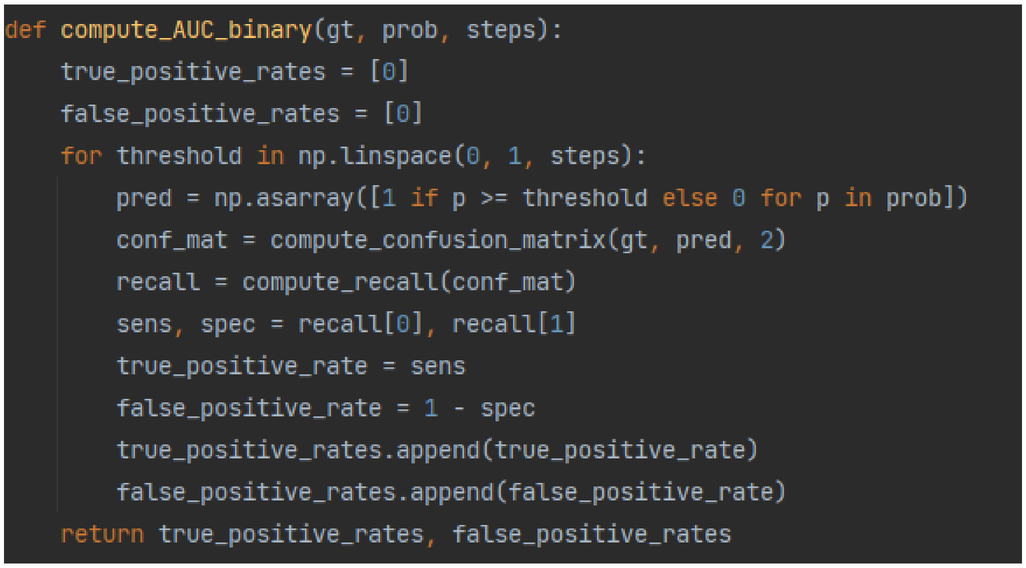
```python
def compute_recall(conf_mat):
    denom = np.sum(conf_mat, axis=0)
    denom[denom == 0] = 1
    return np.diagonal(conf_mat) / denom
```

F-score:



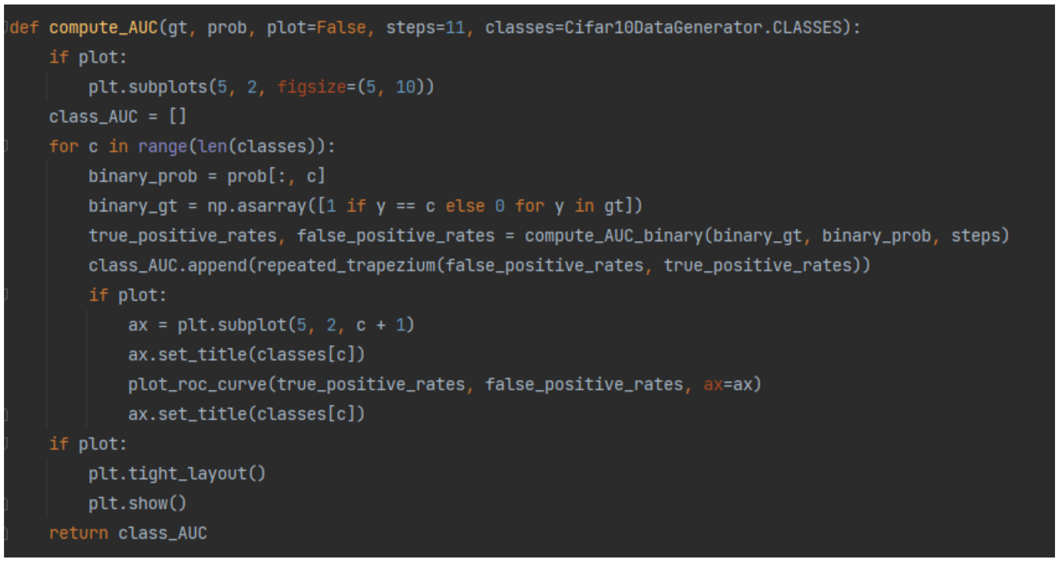
```python
def compute_fscore(conf_mat):
    prec = compute_precision(conf_mat)
    rec = compute_precision(conf_mat)
    denom = prec + rec
    denom[denom == 0] = 1
    return 2 * prec * rec / denom
```

AUC is a binary metric, therefore in order to compute we implement a one vs all approach by taking each class and considering it positive, and the rest negative. First, we implement binary AUC:



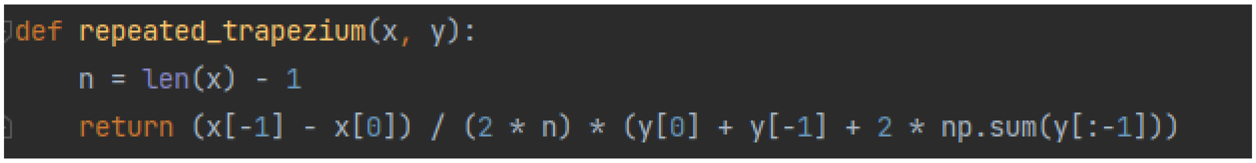
```python
def compute_AUC_binary(gt, prob, steps):
    true_positive_rates = [0]
    false_positive_rates = [0]
    for threshold in np.linspace(0, 1, steps):
        pred = np.asarray([1 if p >= threshold else 0 for p in prob])
        conf_mat = compute_confusion_matrix(gt, pred, 2)
        recall = compute_recall(conf_mat)
        sens, spec = recall[0], recall[1]
        true_positive_rate = sens
        false_positive_rate = 1 - spec
        true_positive_rates.append(true_positive_rate)
        false_positive_rates.append(false_positive_rate)
    return true_positive_rates, false_positive_rates
```

Then for all classes:



```python
def compute_AUC(gt, prob, plot=False, steps=11, classes=Cifar10DataGenerator.CLASSES):
    if plot:
        plt.subplots(5, 2, figsize=(5, 10))
    class_AUC = []
    for c in range(len(classes)):
        binary_prob = prob[:, c]
        binary_gt = np.asarray([1 if y == c else 0 for y in gt])
        true_positive_rates, false_positive_rates = compute_AUC_binary(binary_gt, binary_prob, steps)
        class_AUC.append(repeated_trapezium(false_positive_rates, true_positive_rates))
        if plot:
            ax = plt.subplot(5, 2, c + 1)
            ax.set_title(classes[c])
            plot_roc_curve(true_positive_rates, false_positive_rates, ax=ax)
            ax.set_title(classes[c])
    if plot:
        plt.tight_layout()
        plt.show()
    return class_AUC
```

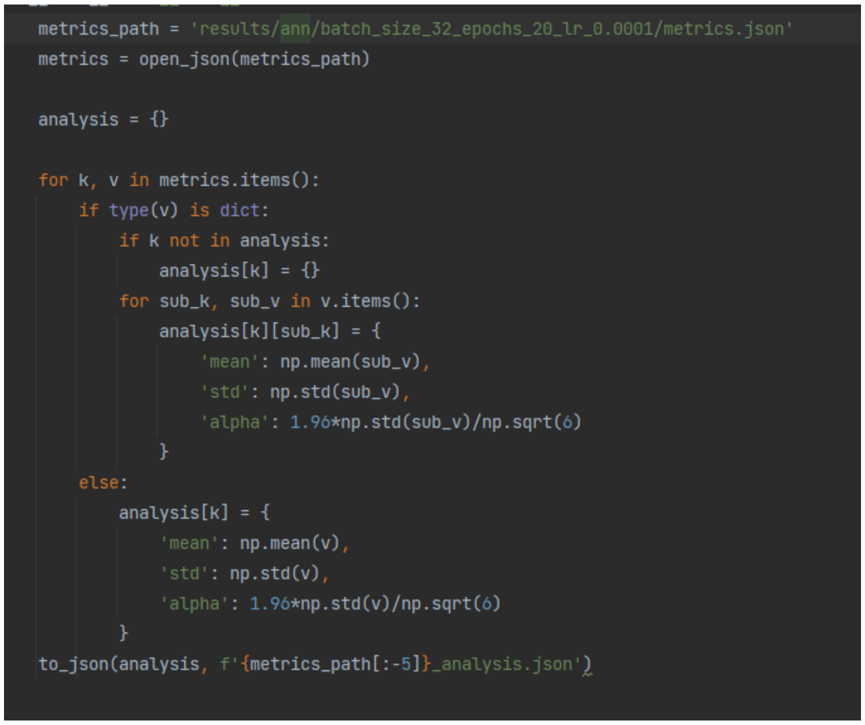
The actual AUC is an integral and we use the repeated trapezium formula to compute it:



```python
def repeated_trapezium(x, y):
    n = len(x) - 1
    return (x[-1] - x[0]) / (2 * n) * (y[0] + y[-1] + 2 * np.sum(y[:-1]))
```

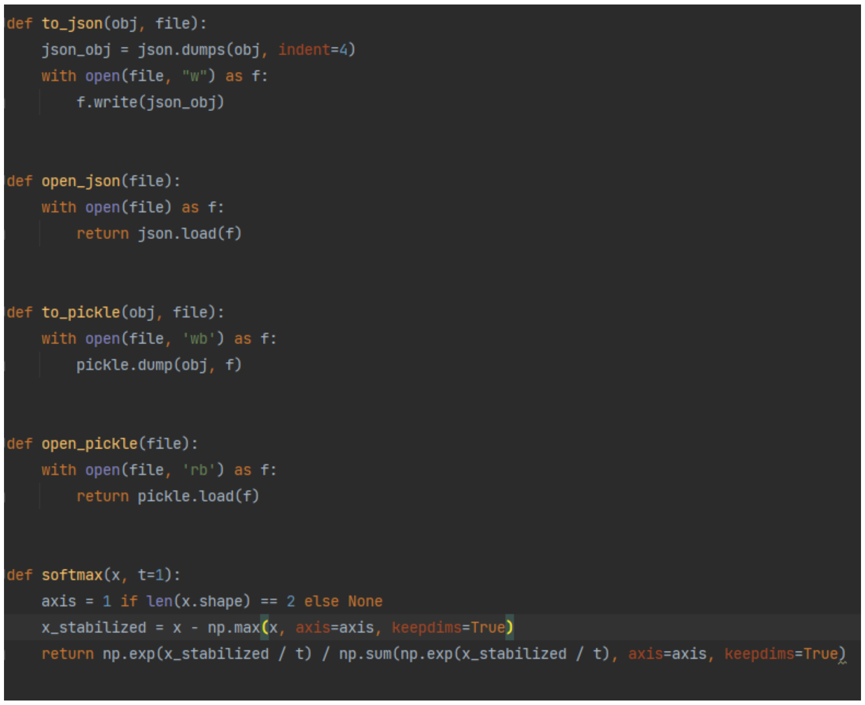
In the end, for all metrics, except for accuracy, we record the values for each class and the mean in a json file.

This is done in a cross-validation manner, by computing them on each batch. Finally, we compute the means and confidence intervals batch wise:


```
metrics_path = 'results/ann/batch_size_32_epochs_20_lr_0.0001/metrics.json'
metrics = open_json(metrics_path)

analysis = {}

for k, v in metrics.items():
    if type(v) is dict:
        if k not in analysis:
            analysis[k] = {}
        for sub_k, sub_v in v.items():
            analysis[k][sub_k] = {
                'mean': np.mean(sub_v),
                'std': np.std(sub_v),
                'alpha': 1.96*np.std(sub_v)/np.sqrt(6)
            }
    else:
        analysis[k] = {
            'mean': np.mean(v),
            'std': np.std(v),
            'alpha': 1.96*np.std(v)/np.sqrt(6)
        }
to_json(analysis, f'{metrics_path[:-5]}_analysis.json')
```

Utils

Useful functions:


```
def to_json(obj, file):
    json_obj = json.dumps(obj, indent=4)
    with open(file, "w") as f:
        f.write(json_obj)


def open_json(file):
    with open(file) as f:
        return json.load(f)


def to_pickle(obj, file):
    with open(file, 'wb') as f:
        pickle.dump(obj, f)


def open_pickle(file):
    with open(file, 'rb') as f:
        return pickle.load(f)


def softmax(x, t=1):
    axis = 1 if len(x.shape) == 2 else None
    x_stabilized = x - np.max(x, axis=axis, keepdims=True)
    return np.exp(x_stabilized / t) / np.sum(np.exp(x_stabilized / t), axis=axis, keepdims=True)
```