PROG2051 - Project

Autumn 2021

Sindre Eiklid - (sineik@stud.ntnu.no)
Rickard Loland - (rickarl@stud.ntnu.no)

Table of contents

Resources

Project Repository: GitHub
Dataset: http://www.josiahwang.com/dataset/leedsbutterfly/

Introduction

For our project in this course, PROG2051, we decided to work with the 'Image classification' case. For this work, we take the Leeds butterfly dataset and write an image classification algorithm using masking and a Convolutional Neural Network. The resulting classification, as well as confusion matrix and classification report for the model, is presented with our thoughts on how the model performed for this task, what improvements could be made to it, and any preprocessing we might want to consider doing (or skipping) and how that might change the results.

We also decided to convert our dataset to grayscale and run it through the same process to compare our performance between the two versions, in order to more thoroughly test our model.

For large parts of this project, the two of us worked together - one person would share the screen and write code, and we would discuss how to approach each problem, what might solve it, etc. Both of us would use the various resources at our disposal to find solutions.

Sindre wrote our initial code for datasetParser.py, downloading, reading, and parsing, and finally segmenting our images.

Rickard wrote the initial code for model.py, preparing our data for training/testing, creating a model, and plotting the results.

However, both of these needed a lot of further work, fixing up and bug fixing. For this, both of us would often work together, and at other times one would fix up one thing - for example, tune our CNN model - while the other might clean up the code for how we load our dataset or refactor our repo. Additionally, we both intermittently edited and added to this report.

Towards the end of the project when doing tuning and minor touchups, Sindre did most coding as his much faster PC would run through the fitting process far more quickly than even google Colaboratory - with both of us discussing, searching, and looking up improvements and fixes, etc. During this part of our process, Rickard took the initiative to write up large parts of this report while the code was being finalized. In the end, we finished our report together by filling in data and examples from finalized runs and writing the result section.

Dataset Parsing and Preprocessing

For this project case, the usage of the ‘Leeds butterfly’ dataset was specified. This is a fairly simple dataset consisting of butterflies against natural backgrounds, and in addition to descriptions, there are also masks already provided for the dataset. This significantly simplifies things for us, as we do not have to write the code for creating masks from these images ourselves.

Our first task is to access this dataset. We wrote a function ‘initializeDataset’ for this purpose. All it does is check if the dataset exists locally at the path we access, and if not, it accesses the URL where it is hosted and downloads it to this path.

We decided to do this download directly in our code as it has the advantage of not needing to worry about storing the dataset and distributing it ourselves, but the drawback is that we do not have full control of the dataset - if the target dataset at the URL is changed, or the URL itself changes, our code will not work properly. Since this is an ephemeral project for the course, and the dataset used is well-known and thus likely to stay up, we decided the advantage was worth the drawback in this case.

```
def initilizeDataset():
    print('{:<40s}'.format("Initializing dataset..."), end = "", flush = True)
    if (path.exists("./dataset/leedsbutterfly") == False):
        print(FAILED)
        print('{:<40s}'.format("Requesting compressed data..."), end = "", flush = True)
        url = 'http://www.josiahwang.com/dataset/leedsbutterfly/leedsbutterfly_dataset_v1.0.zip'
        r = get(url, allow_redirects = True)
        open('compressedData.zip', 'wb').write(r.content)
        print(DONE)
        print('{:<40s}'.format("Uncompressing data..."), end = "", flush = True)
        with ZipFile("compressedData.zip", 'r') as file:
            file.extractall("./dataset")
        print(DONE)
        print('{:<40s}'.format("Deleting temporary files..."), end = "", flush = True)
        remove('compressedData.zip')
        print(DONE)
    else:
        print(SUCCESS)
```

Once we had our dataset, we needed to parse it. We need images and masks to construct our segmented dataset, as well as labels we can use for our actual classification. We do this by going through the image and mask folders and loading them into lists, then using the filename of the image to label it as each image starts with ‘001...’, ‘002…’ and so on. The labels are then converted to integers and subtracted by 1 so that the classes start at 0. This is done for the to_categorical function that is needed for the sequential model.

```python
def parseDataset():
    images = masks = labels = []
    print('{:<40s}'.format("Parsing dataset..."), end = "", flush = True)
    for filename in listdir("dataset/leedsbutterfly/images"):
        img = imread(path.join("dataset/leedsbutterfly/images", filename))
        mask = imread(path.join("dataset/leedsbutterfly/segmentations", filename[:-4] + "_seg0.png"), 0)
        if img is not None and mask is not None:
            images.append(img[:,:,::-1])
            masks.append(mask)
            labels.append(int(filename[:3]) - 1)
    print(DONE)
    return images, masks, np.array(labels)
```

We store our labels in a NumPy array, as we don't need to do further preprocessing for these, unlike our images and masks.

Now that all our images and masks are loaded into memory, we can start with our segmentation. We want to apply our masks to their accompanying images so that we remove the background noise from them - we don't want our CNN to be training or testing on the inconsequential shrubbery around the butterflies.

Applying mask segmentation is a fairly simple process. All we need to do is multiply each image with its accompanying mask. Since the black areas of the mask have a value of 0, these areas of the image will also become 0 afterward. Meanwhile, the white areas have max value, and thus the areas of the image matching the white mask will remain unchanged.

Initially, we wrote our own function to perform this multiplication. It worked fine - but was very inefficient, as processing each image took a couple of seconds! Luckily for us, the cv2 library we use to read images has a built-in function 'bitwise_and' which performs this part of preprocessing for us in a very quick and much more efficient way.

Once we have performed segmentation, we resize our images in order to have a consistent size for our CNN. Since the dataset uses images of hugely varying sizes and shapes, this is not the ideal solution - but we needed to perform resizing in order to both make it easier to construct our CNN, and to improve the efficiency and speed of our program, as it was significantly slower when working with larger images. We initially landed on 100 x 100 as a compromise between data and efficiency, but during testing, we discovered that 50 x 50 gave us better results for our color image tests. We kept 100 x 100 when working with the grayscale imageset.

Once this is done, we store our preprocessed images in a NumPy array:

```python
def segmentData(images, masks):
    print('{:<40s}'.format("Segmenting images..."), end = "", flush = True)
    for image in range(len(images)):
        images[image] = bitwise_and(images[image], images[image], mask = masks[image])
    print(DONE)
    return images
```

At this point, we want to cache our dataset - while cv2.bitwise_and improves performance a lot, running it and then resizing for all our 832 images takes a good minute, and we don't want to have to do that every time we want to run our program. We create four helper function for this purpose - one that stores our new preprocessed data as well as their labels to file, one that loads this data from the file, one verifying stored data, and one that checks if the data is stored in the first place and calls the load function if not. Our verification function allows us to change parameters of our images (such as resizing) without manually deleting cached datasets, which was very convenient during development:

```
def isCached(filename):
    print('{:<40s}'.format("Checking if " + filename + " is cached..."), end = "", flush = True)
    if path.exists(path.join("dataset", filename + '.npy')):
        print(SUCCESS)
        return True
    else:
        print(FAILED)
        return False
```

```
def cacheData(data, filename):
    print('{:<40s}'.format("Caching " + filename + "..."), end = "", flush = True)
    np.save(path.join("dataset", filename + '.npy'), data)
    print(DONE)
```

```
def loadCachedData(filename):
    print('{:<40s}'.format("Loading cached " + filename + "..."), end = "", flush = True)
    data = np.load(path.join("dataset", filename + '.npy'))
    print(DONE)
    return data
```

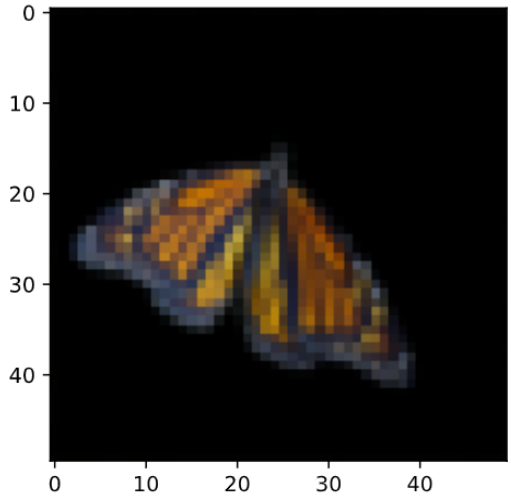
```
def verifyCachedData(imageSize, filename):
    print('{:<40s}'.format("Verifying cached " + filename + "..."), end = "", flush = True)
    data = np.load(path.join("dataset", filename + '.npy'))
    if data.shape[1] == imageSize and data.shape[2] == imageSize:
        print(SUCCESS)
        return True
    else:
        print(FAILED)
        return False
```

Humorously, this decision to use caching caused us quite a bit of issue later on during our project, as we changed how we read the labels but forgot we had cached them as well - causing us to waste a good half an hour trying to figure out why they wouldn't store as classes 0 - 9 instead of 1 - 10!

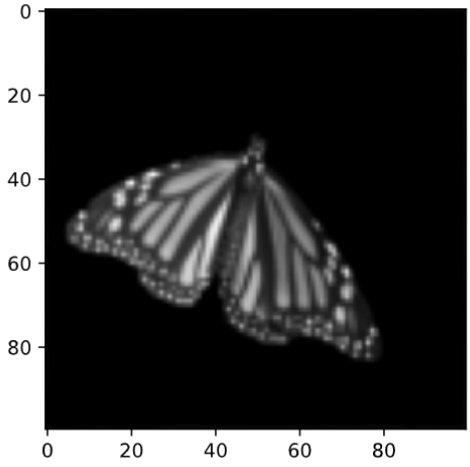
As you can see, we tried to keep our functions fairly generic and modular where possible. The primary exception to this is in our parse function, which is largely based on how the Leeds butterfly dataset specifically is constructed. In several places, we need to specify the path or URL for Leeds butterfly, rather than a more generic alternative. However, we still feel this code can be fairly easily rewritten to perform the same work on other datasets, by changing these parameters.

At this point, we have our processed butterfly dataset to work on.

Color example



Grayscale example



Sequential Modelling

As a way to differentiate between the grayscale model and the color model, we made a dictionary file with the different parameters.

```
LABEL_FILENAME = 'labels'
LABEL_NAMES = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
# status - wether to run these during runtime
CROSS_VALIDATION_STATUS = True
CONVERT_TO_GRAYSCALE = False
# set parameters for RGB color data
COLOR_DATA_FILENAME = 'color-data'
COLOR_IMAGE_FILENAME = 'modelColor.png'
COLOR_IMAGE_SIZE = 50
COLOR_IMAGE_SPLITS = 5
COLOR_IMAGE_TRAIN_SIZE = 0.80
COLOR_IMAGE_EPOCHS = 50
COLOR_IMAGE_BATCH_SIZE = 256
# set parameters for gray color data
GRAY_DATA_FILENAME = 'gray-data'
GRAY_IMAGE_FILENAME = 'modelGray.png'
GRAY_IMAGE_SIZE = 100
GRAY_IMAGE_SPLITS = 4
GRAY_IMAGE_TRAIN_SIZE = 0.70
GRAY_IMAGE_EPOCHS = 50
GRAY_IMAGE_BATCH_SIZE = 256
# status messages
DONE = 'DONE'
SUCCESS = 'SUCCESS'
FAILED = 'FAILED'
```

The CROSS_VALIDATION boolean decides whether we should run cross-validation on the model, while the CONVERT_TO_GRAYSCALE decides which models to run. During the development of the grayscale model, we weren't always interested in running the color model and this was an easy way to *flick* the switch on these models. Here we also decide the image sizes, amount of splits for cross-validation, training size, epochs, and batch size. This could be changed with ease and made the development of the models efficient.

Now we can start creating our CNN. We start by preparing our data with a simple train-test-split.

```
def prepareData(data, labels, trainSize):
    xTrain, xTest, yTrain, yTest = train_test_split(data, labels, train_size = trainSize, random_state = 0)
    yTrain = to_categorical(yTrain, num_classes = 10)
    yTest = to_categorical(yTest, num_classes = 10)
    return xTrain, xTest, yTrain, yTest
```

For the color model, we ended up with a split of 80/20. A greater testing size would result in overfitting, while a lesser testing size would result in worse accuracy. The 80/20 split also made the cross-validation give a more representable image of how well our model performs on the dataset. This is because we could choose a split of 5 which would result in a 20% training size for each iteration.

For the grayscale model, we found that 80/20 gave too much overfitting and ended up with a 70/30 split. This seems to give the best accuracy as anything lower would give worse results. We had to modify our cross-validation to account for this changed split - curiously, 4-fold validation seems to give more matching results than 3-fold validation for this split.

For the next step, we want to construct our actual model. This is - naturally - often the most challenging part of this overall process, as so many different elements go into the choices made here. First, there's simple know-how - making sure to use Conv2D layers for CNNs that work with images, constructing the correct input_shape, deciding the loss algorithm, etc.

Second, comes the detail work that can be a lot vaguer - exactly what layers we want and how to construct them. Should we add Dropout layers, and if so, how aggressively should we drop? Do we want a kernel of 2x2 or 3x3 when passing over our images? How should we utilize Flatten and MaxPooling layers for the best effect?

Last but certainly not least, is the issue of efficiency. While we all dream of making intricate, perfectly tuned models, at some points we have to face the reality that regular PCs are not built to work on overly elaborate models at any kind of speed, even if we're not exactly constructing deep learning networks here. In our experience, once we started creating models with nodes in the millions, things were slowing down significantly, even using Google Colaboratory.

For example with 13.5 million dense nodes, it was taking a good 40 - 50 seconds to start fitting the model, and a further 6 - 7 seconds per epochs. Given the size of our images, it did not take a particularly large number of layers or nodes per layer to reach this number of total nodes in our CNN. We quickly realized that using Conv2D layers and applying max-pooling layers in between them was not an optional part of our model design, as the work max-pooling did to reduce our data was vital to making manageable models we could actually run. Improving the performance of our model was also a nice bonus!

We also ran into serious issues where our CNN would start returning ‘nan’ for loss and accuracy numbers, which stumped us quite a bit. Looking at online resources, we were able to rule out a lot of perpetrators here, such as issues with invalid data, normalization, learning rate, optimization, etc. Eventually, we realized the issue was actually with memory - for some reason, corrupted data in memory was being reused, and only completely closing the IDE fixed this.

The CNN we ended up with looks like this:

```
def defineModelColor():
    model = Sequential([
        Conv2D(32, (3, 3), input_shape = (COLOR_IMAGE_SIZE, COLOR_IMAGE_SIZE, 3), activation = 'relu'),
        MaxPooling2D(pool_size = (2, 2)),

        Conv2D(32, (3, 3), activation = 'relu'),
        MaxPooling2D(pool_size = (2, 2)),

        Conv2D(64, (3, 3), activation = 'relu'),
        MaxPooling2D(pool_size = (2, 2)),

        Flatten(),

        Dense(64, activation = 'relu'),
        Dropout(0.5),

        Dense(10, activation = 'sigmoid')
    ])
    model.compile(
        loss = 'categorical_crossentropy',
        optimizer = Adam(learning_rate = 0.01),
        metrics = ['accuracy']
    )
    return model
```

Our grayscale model is identical except for the input_shape parameters - we wanted to consolidate them into one model and pass these parameters into our function, but we realized we were unfortunately not able to use functions with parameters in cross-validation. As such, we decided to stick with two separate but nearly identical model constructors.

For this Neural Network, we spent a good amount of time tuning the layers to get the construction - and results - we desired. We spent a lot of time getting our loss values and other metrics looking good, as well as fighting against overfitting which was a frequent problem with our initial models.

It took us a long while to get a good model going due to all these issues, but eventually, we stumbled upon this resource from the official Keras blog, which was a huge help. Simply trying to run their code on our dataset (modified with our 10-class categorical_crossentropy classification) vastly improved our model in many ways - fewer nodes, faster runs, and much better results! We, therefore, used this model as a basis for further development of our own.

Since we naturally did not want to just take their example layers, we spent many, many hours trying to optimize this design for our dataset. And to our chagrin, we could not find one that outperformed it! Both in results or in speed, the 32/32/64/64 layer split with 3x3 kernels, default (1,1) stride, and ReLu activation were seemingly unbeatable - increasing nodes significantly (to for example 80/160) resulted in worse performance, while any lower simply made it less accurate. More layers, fewer layers, all taunted us with poor results. Eventually, we decided to simply accept that the professionals know best and that our time improving the project was better spent elsewhere.

As we want our final layer to output all 10 classes, we are using categorical-crossentropy as our loss function. We found that diverging from a learning rate close to 0.01 for our Adam optimizer gave us much worse results, in terms of overfitting if increased and inaccuracy if reduced.

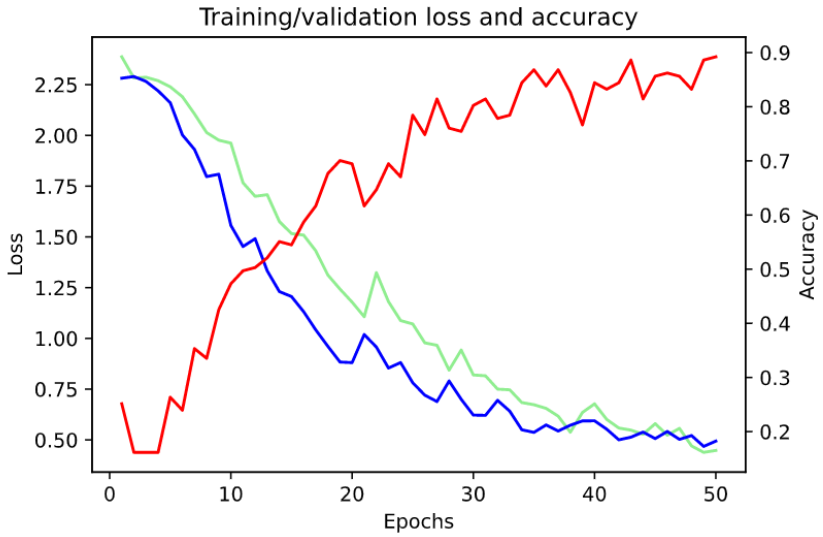
We tried using both Softmax and Sigmoid activations for our final layer - from our testing, we were surprised to find that Sigmoid activation consistently outperformed Softmax for our CNN, despite Softmax being the activation algorithm of choice for multiclass datasets like this one with no overlap in classes (only one class per image). Sigmoid not only returned better results but also worked faster than our Softmax-activated CNN. We are not sure what the reason for this discrepancy is, as a lot of google searches did not turn up anyone recommending sigmoid activation for multiclass scenarios like this one.

For a better look at the design of our model, check out the modelColor.png and modelGray.png images in our repo - they are simply too big to make sense to put in this report (we tried!) but they provide a more detailed overview of the inputs and outputs for each of our model layers from tensorflow.keras.utils.plot_model.

Results and conclusions

Color

Since we claim to have finished tuning our model, let's take a look at the results we were able to glean from it. First, let's check a plot of our loss values and accuracy as the model does its work.



As we can see, the model improves layer by layer as it gets a handle on our testing data. Overall, we were able to make a lot of improvements to our model. We struggled significantly with overfitting issues when designing our model, which meant we were not able to make full use of the processing power at our disposal. This overfitting would tend to materialize around epoch 30 - 35. Validation accuracy would at this point plateau at around 80% accuracy, with validation loss either staying stable or starting to climb again while training loss continues improving.

However, we were able to sort out most of these issues, as can be seen in our graph above! Overfitting does not become an issue in this graph, with our loss values following very closely. Accuracy reaches about 90%, staying stable in the mid-80s.

Once we had performed our fitting and evaluated the results, we could perform our predictions using model.predict. At this point, we could start our work on constructing the classification report and confusion matrix.

	precision	recall	f1-score	support
1	0.84	0.94	0.89	17
2	0.93	0.93	0.93	14
3	1.00	0.82	0.90	11
4	0.78	0.86	0.82	21
5	0.91	0.77	0.83	13
6	0.96	0.93	0.94	27
7	0.93	0.87	0.90	15
8	1.00	1.00	1.00	18
9	0.86	0.75	0.80	16
10	0.68	0.87	0.76	15
accuracy			0.88	167
macro avg	0.89	0.87	0.88	167
weighted avg	0.89	0.88	0.88	167

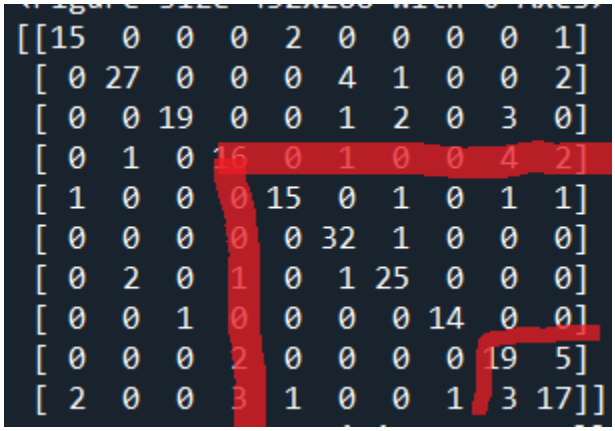
Here is an example of the classification report our model produces - as we can see, the results are quite good, with some outlier classes which we will get back to.

The accompanying confusion matrix for reference:

```
[[16  0  0  0  0  0  0  0  0  1]
 [ 0 13  0  0  0  0  0  0  0  1]
 [ 0  0  9  0  0  1  1  0  0  0]
 [ 0  0  0 18  1  0  0  0  0  2]
 [ 2  0  0  0 10  0  0  0  0  1]
 [ 0  0  0  1  0 25  0  0  1  0]
 [ 0  1  0  0  0  0 13  0  1  0]
 [ 0  0  0  0  0  0  0 18  0  0]
 [ 0  0  0  3  0  0  0  0 12  1]
 [ 1  0  0  1  0  0  0  0  0 13]]
```

As we ran our model many times in the process of fine-tuning it, we could see what areas it had issues with. We were able to identify two groups of classes that our CNN had trouble identifying as consistently or accurately as the rest of our dataset.

The first and most troublesome group was classes 4, 9, and 10. These are all brownish species of butterflies with similar shapes, albeit different patterns. It's clear from looking at the trouble our CNN has with these, that the combination of image resizing and max-pooling has made it difficult for our neural network to classify butterflies based on small details like these patterns, and instead relies on the general shape and color of the butterflies to perform classification. There are traces of these issues in our examples above, but we can see it even more clearly in this example from an earlier, less optimized version of our model:



Excuse the poorly edited lines! But here we can see the consistent issues our model had with mixing up classes 4, 9, and 10 - indeed, all these classes were misidentified for each other several times. Since this is a less optimized model, we also see some outlier issues such as mistakenly identifying 6 as 2 several times, which has not been a consistent problem in our modeling.

The second group, which only occasionally caused problems, consisted of classes 1 and 5. These again are very similar in their bright orange colors, but class 5's lower wings are a lot more distinctive in terms of color being large brown splotches, which seems to have made a pretty big difference compared to the previous group - our current CNN is often able to identify these groups quite well with >85% accuracy, with only occasional runs resulting in mixups between these classes. In our examples above we can see that there is some trace of this, but nothing that stands out compared to the various other outliers and it has been mostly fixed in our latest models.

For our other 5 classes, we noticed no consistent or frequent issues with the ability of our CNN to accurately identify them. For the most part, these classes are relatively more distinct from the rest of our dataset which helps explain why our CNN is better able to classify these.

In our opinion, one way to improve our classification process could be less aggressive resizing of our images in the preprocessing stage. There are primarily two problems standing in our way of accomplishing this; first, the vast variation in image sizes in the original set makes it difficult to pick a good size - with some images below 150 pixels in height or width, resizing to larger sizes would actually involve increasing the size of some images, which also loses data. Second, larger images make the fitting take a very long time indeed, which our PCs are not designed to handle very well.

Another preprocessing step we feel might significantly improve performance would be to apply scaling to the color values of our segmented images. Currently, all values are between 0 and 1, as we have already scaled our images from 0 - 255 to 0 - 1. However, we have not performed

any normalization on the distribution of values within that range. Our median color values for our segmented images are only around 0.2, indicating that a significant chunk of the colored pixels on our images fall in a relatively small part of that 0 - 1 range.

We could normalize our images to expand their color ranges so they made better use of the entire range of color intensity, which would give our CNN better data to work with during classification. We expect this would help reduce the instances of classes being mixed up (such as our two groups mentioned above) by enhancing subtler differences in color between the classes as well as making the butterfly patterns more detectable and giving them more weight in our CNN.

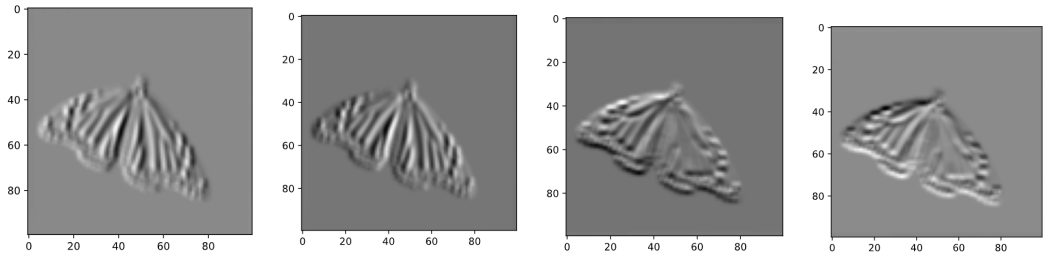
A large issue we had during our work was with splitting our dataset. With 10 classes but only 832 total images, we do not have quite as many samples per class as we would ideally want, which is compounded by the previous two issues mentioned. We noticed our model seemed very sensitive to how our dataset is split during train_test_split and decided to perform some 3-fold cross-validation in order to get a better idea of the variation we were experiencing. However, we did not see any significant issues in this process:
Cross validation results: [0.91616768 0.90419161 0.92168677 0.86144578 0.8493976]
0.89 accuracy with a standard deviation of 0.03

As we can see, there is some variation in performance - here between our first three runs and the final two. That said, we did not find anything extreme to match our previous experiences, and the standard deviation is very reasonable. Nevertheless, another measure to improve our model could be using alternate methods of train_test_split such as one that shuffles inside each class and then splitting all classes exactly equally before shuffling again inside train and test sets might also do this job. We decided to keep on using our standard TTS construction for this project.

Grayscale

When we started working on our grayscale images to compare, we figured we would probably get worse results than with our original dataset - since color seems to be such a key part of identification for us. There are a lot of reasons for this - primary among them is the fact that besides the subtle butterfly patterns, the shapes for all our classes are very similar, making it tough to use shapes only for classification. The other reason is that as mentioned, the dataset is fairly small overall, especially for a full 10 classes.

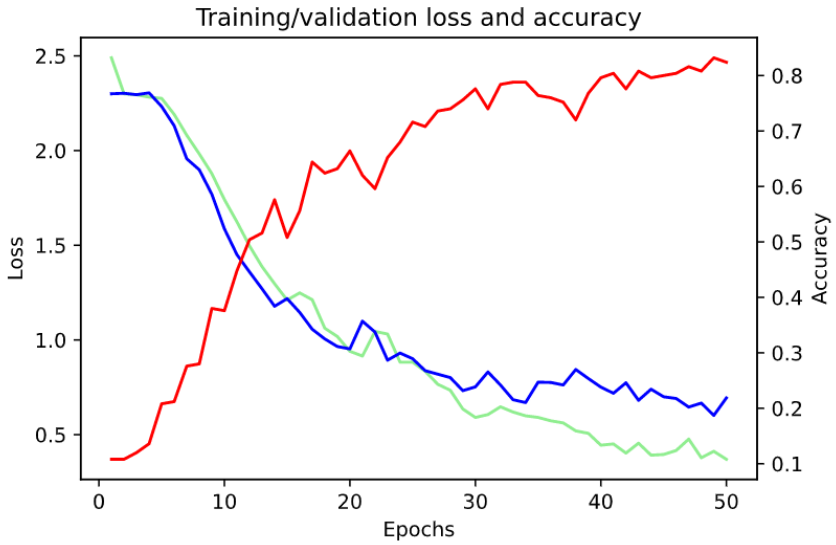
We were interested in seeing whether our assumption would hold, and exactly how much worse our grayscale dataset would perform. We tried a variety of different filters in order to see if we could improve our accuracy decently well:



However, none of the various filters we tried improved our model's performance compared to the basic grayscale-converted images, indeed all coming in slightly worse than the unfiltered images.

With our initial tests, we tried evaluating this dataset without using Conv2D layers in our model, but this gave us exceptionally poor results, with accuracy rarely getting far beyond 40%. While we expected worse performance, this was clearly down to simply a very poor model.

Instead, we changed our model to be very similar to our color model and expanded our grayscale images by an axis in order to fit this dataset into the model. With this approach, our accuracy immediately improved to 60%, and by tinkering with our parameters we managed to further improve it by quite a bit! In our finalized grayscale model, we managed to reach validation accuracy of around 80% and very similar loss values to our color image performances for our best runs:
loss: 0.6938 - accuracy: 0.8240



In the above run, we can see that arguably the model did not even reach optimum performance! Accuracy continued slowly improving even through epoch 50, with validation less also consistently trending very slightly down. However, from repeated runs with these settings, we

believe this to be somewhat of a fluke - most of our runs just about crossed 80% and generally started heavy overfitting at 45 epochs or so. Nevertheless, a significant improvement from our early attempts at grayscale classification.

	precision	recall	f1-score	support
1	0.87	0.87	0.87	30
2	0.89	0.89	0.89	19
3	0.91	0.67	0.77	15
4	0.84	0.64	0.72	33
5	0.87	0.90	0.89	30
6	0.79	0.97	0.87	35
7	0.78	0.90	0.84	20
8	0.76	0.95	0.84	20
9	0.68	0.62	0.65	21
10	0.83	0.74	0.78	27
accuracy			0.82	250
macro avg	0.82	0.81	0.81	250
weighted avg	0.82	0.82	0.82	250

Similar to our color image classification, the three classes 4, 9, and 10 are again performing worse than the rest - this time dragging class 3 along for the race by guessing other classes as class 3. Looking at our confusion matrix, we can see that all these classes are doing poorly in the recall department, while 6 - 9 have poor precision.

```
[[26  0  0  0  2  0  0  0  0  2]
 [ 0 17  0  0  0  0  2  0  0  0]
 [ 0  0 10  0  0  1  1  0  3  0]
 [ 1  0  1 21  2  2  0  2  3  1]
 [ 1  0  0  1 27  0  0  1  0  0]
 [ 0  0  0  0  0 34  1  0  0  0]
 [ 0  2  0  0  0  0 18  0  0  0]
 [ 0  0  0  1  0  0  0 19  0  0]
 [ 0  0  0  0  0  6  1  0 13  1]
 [ 2  0  0  2  0  0  0  3  0 20]]
```

We ran cross-validation to compare some runs:
Cross validation results: [0.73557693 0.78846157 0.8125 0.09615385]
0.61 accuracy with a standard deviation of 0.30

What an outlier! It seems the sensitivity to TTS which we noticed during color modeling has materialized during cross-validation here. For some reason, it seems that

sklearn.model_selection.train_test_split will occasionally fail to properly split our dataset, resulting in a complete breakdown as we can see for our fourth run of cross-validation here. We are not sure how we would go about fixing this, but thankfully it is very easy to tell when this is occurring so we can at least identify faulty fits and handle them accordingly.

Besides this, the results are hovering in the mid-70s to low-80s, which is pretty close to our observed results. This was also done using 4-fold cross-validation with 75% train set size, which performed worse than a 70/30 split for our grayscale images during manual testing. As such, we find the estimation made by our cross-validation to be in line with our expectations.

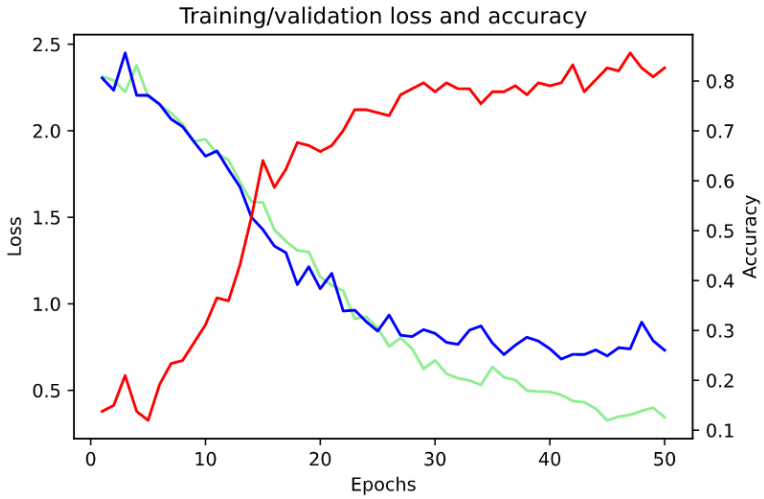
All that said, to achieve this result with our grayscale dataset, we needed to use 100 x 100 images (which was the sweet spot in terms of image size for grayscale), making the fitting take nearly twice as long compared to our color dataset. Despite this, the results were still noticeably worse. As such, we are forced to conclude that - unsurprisingly - it does not make sense to use grayscale images to classify this dataset, as we lose too much data when removing color from our dataset. A better designed model with more power behind it might be able to compensate for these issues and use the grayscale dataset better, however for our purposes using the color dataset clearly seems like a better idea. Another approach could be to pair the grayscale and color images and use both during classification, which we believe could further improve our accuracy into the 90s, but would also take extra processing power. Overall we are very happy to manage test accuracy this high with a grayscale dataset.

We also performed tests using the original unmasked image set, as we were curious how our model would perform on the dataset with backgrounds. We expected the results to be noticeably worse than both our masked datasets, as the model would use the irrelevant background information during the classification process, reducing accuracy significantly.

We were surprised to see the results were still decent, with accuracy reaching around 80% comparable to our grayscale model. Before running the model, we were predicting it to go down to somewhere in the 50s range. We believe that this is caused by the clear and consistent shapes of the butterflies, similar to the classification performance of the grayscale model itself. We are not sure why the noise from the backgrounds did not affect classification accuracy more than it did, but we theorize at least that our rescaling of the images to very small 50 x 50 size has had an impact here, as the details of the background are lost - and since the images all come from the same biome, there is a lot of similarities in colors for the backgrounds as well.

Original dataset results:

Cross validation results: [0.79041916 0.76646709 0.84337348 0.07228915 0.78313255]
0.65 accuracy with a standard deviation of 0.29




```
[[14  0  1  0  1  0  0  0  0  1]
 [ 0 11  1  0  1  0  0  0  1  0]
 [ 0  3  8  0  0  0  0  0  0  0]
 [ 0  0  0 17  0  2  0  0  0  2]
 [ 0  0  0  0 10  1  0  0  1  1]
 [ 0  0  0  2  0 25  0  0  0  0]
 [ 0  4  2  0  0  0  9  0  0  0]
 [ 0  1  0  0  1  0  0 16  0  0]
 [ 0  0  2  0  1  0  0  0 13  0]
 [ 0  0  0  1  0  0  0  0  0 14]]
```

	precision	recall	f1-score	support
1	1.00	0.82	0.90	17
2	0.58	0.79	0.67	14
3	0.57	0.73	0.64	11
4	0.85	0.81	0.83	21
5	0.71	0.77	0.74	13
6	0.89	0.93	0.91	27
7	1.00	0.60	0.75	15
8	1.00	0.89	0.94	18
9	0.87	0.81	0.84	16
10	0.78	0.93	0.85	15
accuracy			0.82	167
macro avg	0.83	0.81	0.81	167
weighted avg	0.85	0.82	0.82	167

It is interesting to see which classes perform worse in this dataset, as classes 2 and 3 were consistently doing poorly in our minimally processed dataset tests. Looking at the dataset we don't see any obvious reasons why this should be the case - the butterflies have fairly distinct shapes and colors, the backgrounds don't seem to differ from those of the other classes, and they don't stand out in our results with processed models. We can also see classes 5 and 7 have poor recall scores relative to the field. As for our grayscale cross-validation, this dataset's validation also displays the issue of poor TTS which happens occasionally.

Overall, our model performed about as well on this dataset as it did on our grayscale dataset - but despite using 50 x 50 images, it still took more time to complete than the grayscale set did. As such, we believe the grayscale set would still make more sense to use for classification than the dataset we have only resized. And certainly our masked color dataset outperforms both by quite a bit.

One thing that surprised us during this process was that increasing image fidelity and resolution did not improve the performance of our models - for neither color nor grayscale! Granted grayscale liked larger images than our color model did, but resizing to 150 x 150 gave us noticeably worse results than 100 x 100. Even though a small number of the images in the original dataset may have been upscaled a little to reach 150 x 150, we were still expecting these larger images to have more data to work with and therefore give even more accurate results, though taking a lot more time to process.

Overall, we learned a lot about creating image classification models and programs during this project, and we had a lot of inspiration throughout with how we could change and improve our approach - some of which made it into our finished project, and some of which we did not implement but have mentioned throughout the report.