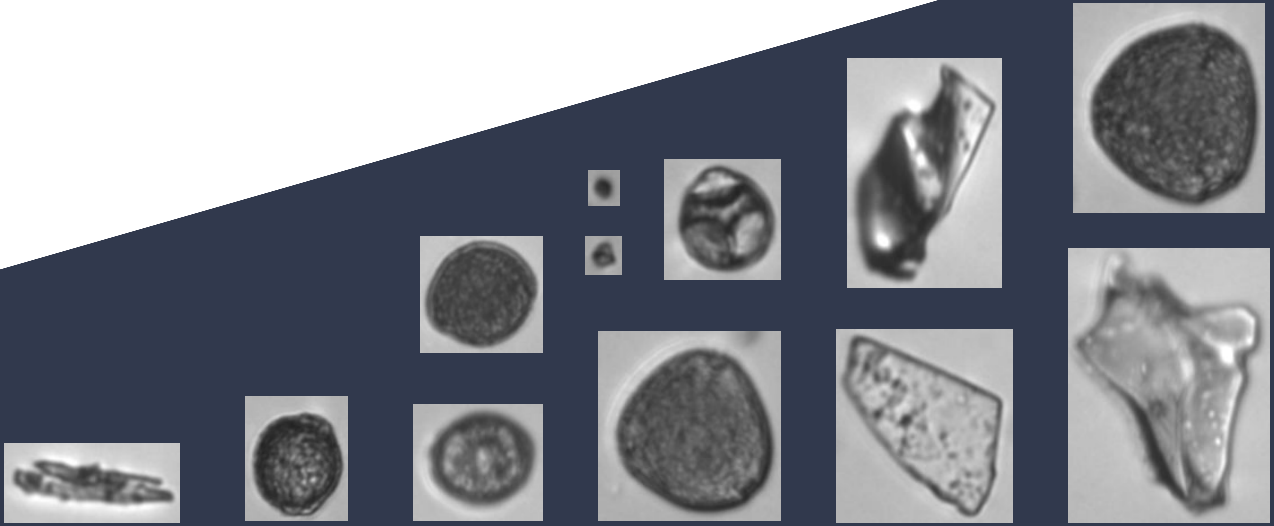
Classification of Insolubles

Data from Niccolo Maffezzoli

Applied Machine Learning, University of Copenhagen, 2021

Kian Gao
Kian Kirchhof
Tobias Særkjær
Mia-Louise Nielsen

(All participants contributed evenly)

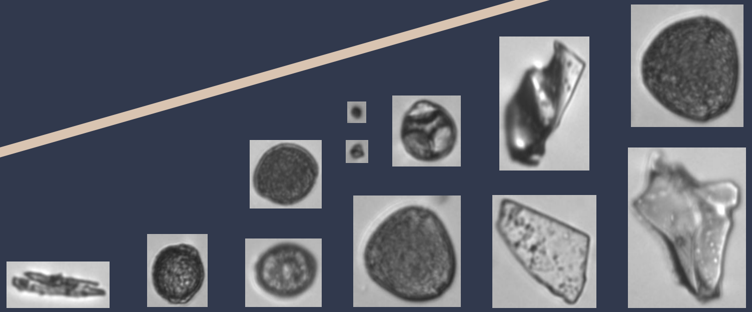


The training data

- ~135'000 images in 6 folders
- 6 .csv tables with metadata

Images depict *insolubles*, particles extracted from a Greenland ice core
- "simulated" data, images gathered

All metadata is automatically collected from analysis of ice cores - can be considered as a joint dataset (images+tables)



Preliminary plan for types of approaches
5000 entries vs. all entries

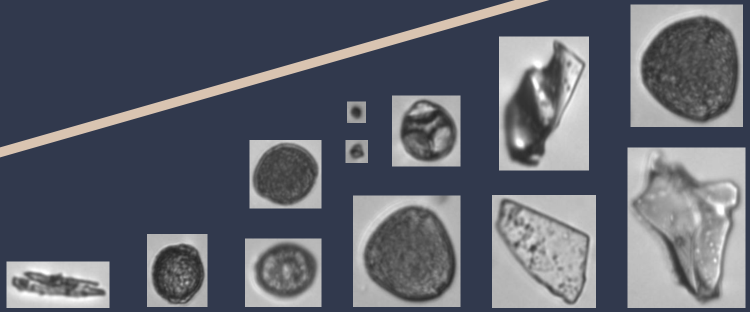
1. Gradient Boost, Pytorch and Tensorflow multi-classifications based only on data in .csv tables
2. Convolutional Neural Networks for multi-classification based only on images (simple NN, ResNet50, ...)
3. Hybrid CNN-based with .csv-data injected into a layer of the image-based model (another reason for including ResNet50)

The training data

The 6 classes are:

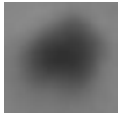
- campanian (~44.41%)
- corylus (~5.75%)
- dust (~22.61%)
- grimsvotn (~10.76%)
- qrobur (~8.05%)
- qsuber (~8.42%)

Actual test set will contain other particles as well, e.g. fibres from gloves, unknowns etc. - and probably no pollen

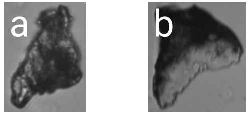


Classes and subclasses (images not to size):

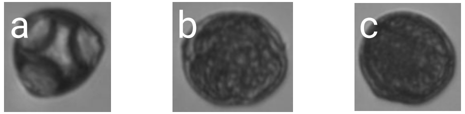
- Dust (~22.61%)



- Ash (~55.17%)
    a. campanian
    b. grimsvotn



- Pollen (~22.22%)
    a. corylus
    b. qrobur
    c. qsuber



Additionally we want to detect anomalies

This is interesting both for general purposes but also because the actual test data will contain other particles than the "simulated" test data

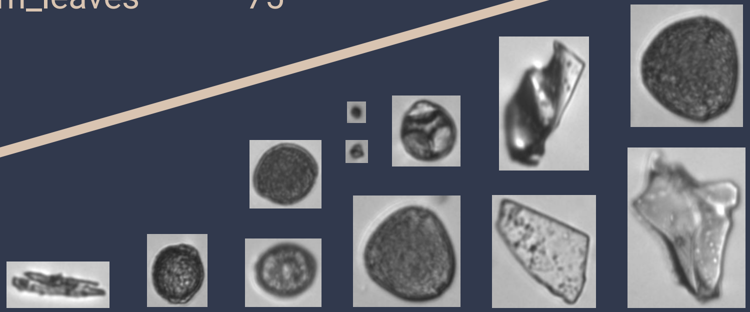
LightGBM model

Gradient boosted decision tree
using numeric data (.csv) only

Simple initial model

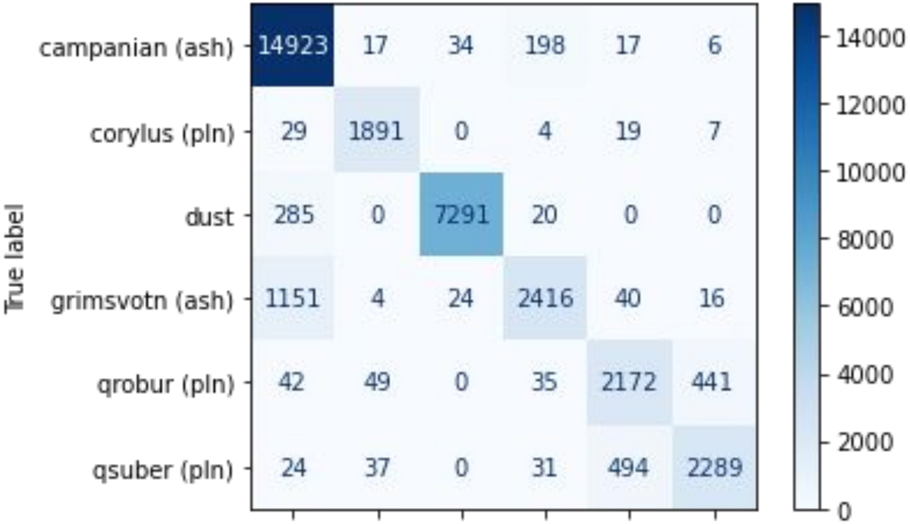
RandomizedSearchCV:

learning_rate	0.0555 (?)
max_depth	15
min_data	46
num_leaves	75



Gradient boost performed on 5000 / all particles from each class.

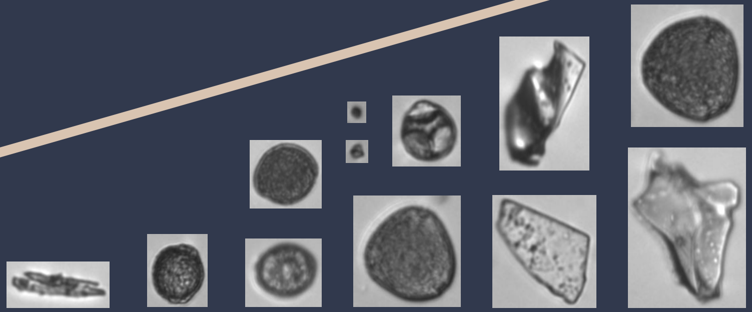
validation acc. 84.53% / 91.11% test-train split (25%)
83.12% / 90.31% cross-valid. (3-fold)



Neural network

Dense neural network (Keras) using numeric data only

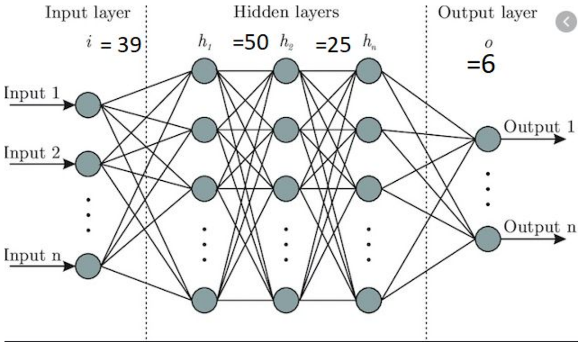
Setting up the network



Dense Neural Network on 5000 / all data particles from each class.

- Input layer: 39, all variables except the proposed drop outs .
- First layer: 50, relu activation
- Second layer: 25, relu activation
- Multi-Output: 6, softmax activation.

Overall hyperparameters: Learning rate = 0.001, epochs = 30,

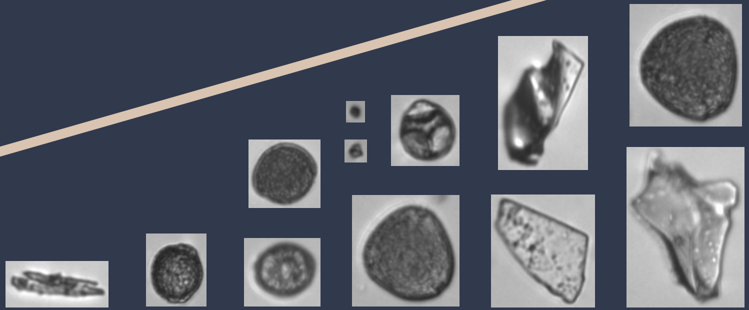


Hyperparameters found by using gridsearch between 25 and 1000 for amount of nodes, epochs between 10 and 100.

Neural network

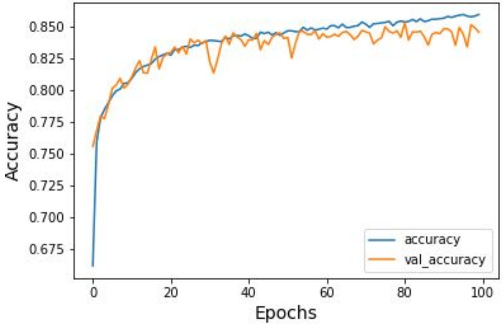
Dense neural network (Keras) using numeric data only

Results

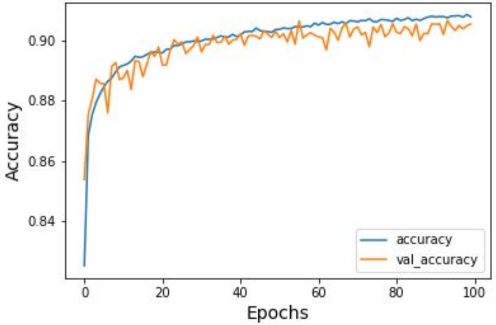


Results - validation accuracy

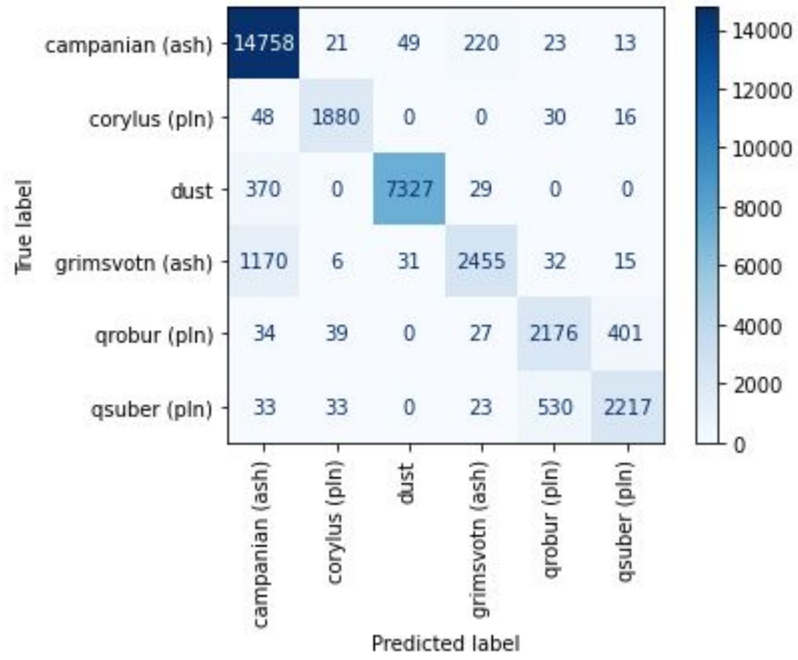
Validation accuracy for 5000 particles from each class: 84.72%

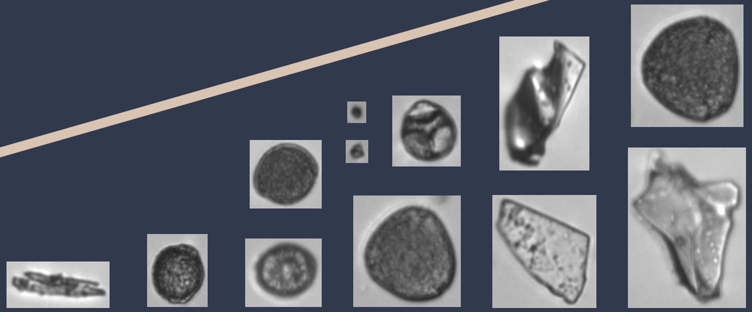


Validation accuracy for all particles from each class: 90.56%



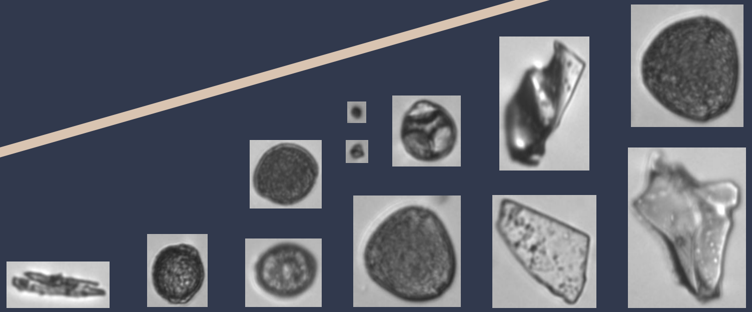
Neural Network





Scaling then padding

Target dimensions chosen from histograms of image dimensions

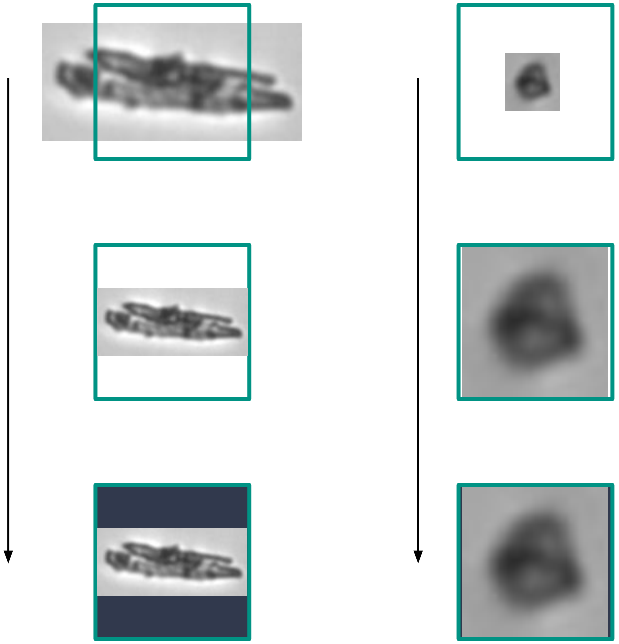


Example scaling then padding for target size:

1) Original

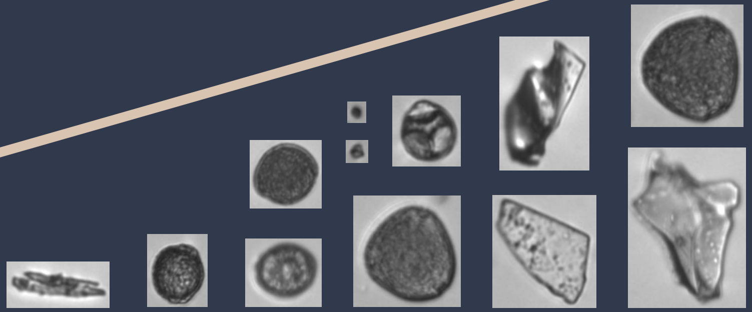
2) Scale

3) Pad



Neural network

Convolutional neural network (Keras) using images only



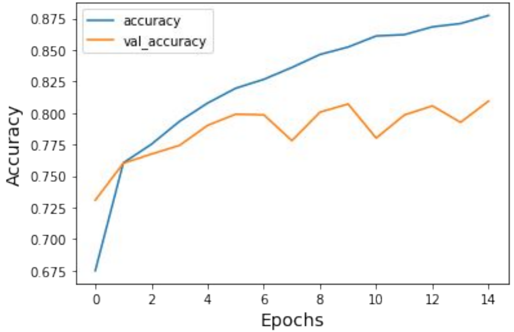
Network architecture:

- Convolution (filters=112, ReLu, dropout=0.268)
- Pooling
- Convolution (filters=57, ReLu, dropout=0.268)
- Pooling
- Flatten
- Dense (units=194, ReLu)
- Dense (units=6, softmax)

Optimized categorical cross entropy using Adam with learning rate=0.00349

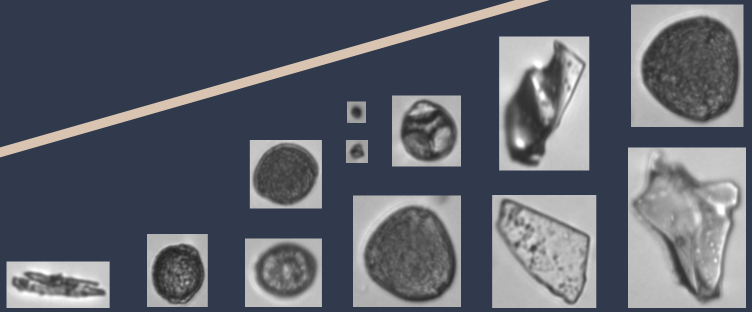
Hyper parameters were optimized using randomized search.

Validation accuracy after 15 epochs: 80.95% with 5000 Images for each class



Residual Neural network

Illustration of residual neural network architecture



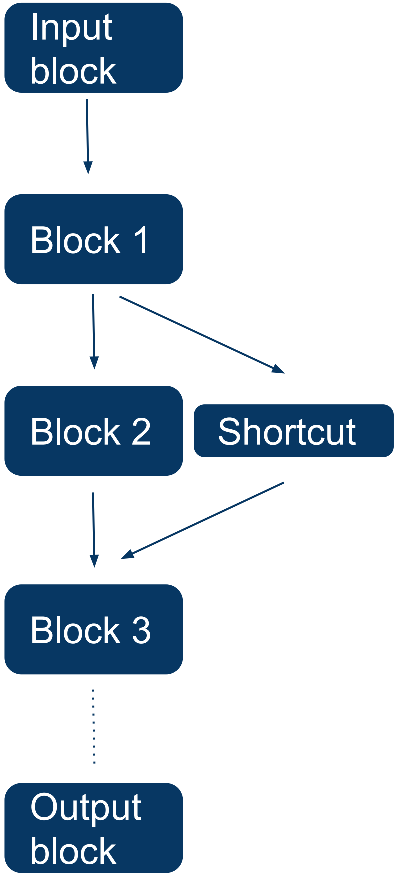
Regular blocks contain multiple layers including convolutions, batch normalization and activations (ReLu).

Shortcuts/skip connection contain fewer layer.

The model can be very deep and contain many blocks and manu shortcut layers.

The output block contains dense layers - the last one activated with a softmax.

Images stacked to get 3 channels for the RESNET50 model.

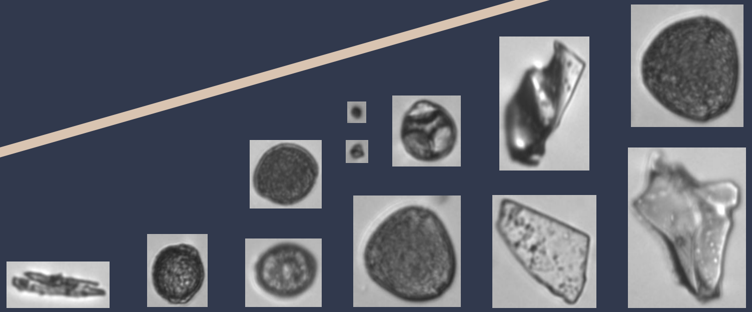


Residual Neural network

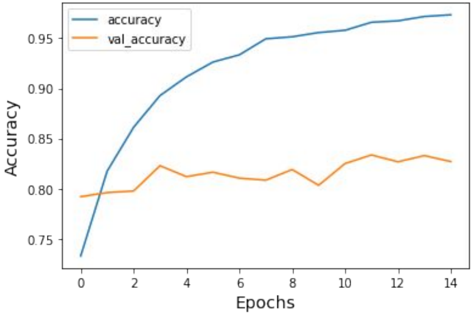
Own implementation of residual neural network inspired by Google's RESNET architecture

Same architecture with and without numeric data

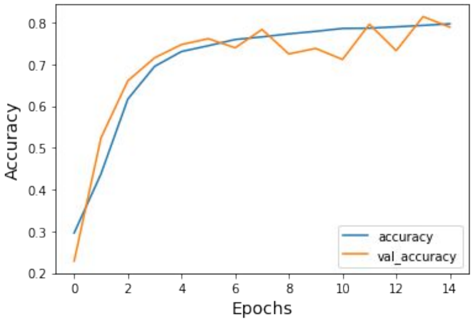
Numeric data concatenated in the next to last dense layer



We experimented with the different layers and used randomized search to choose numbers of filters in the convolutional layers, in order to optimize performance.



Similar accuracy without and with numeric data (82.70% and 83.09%, respectively)

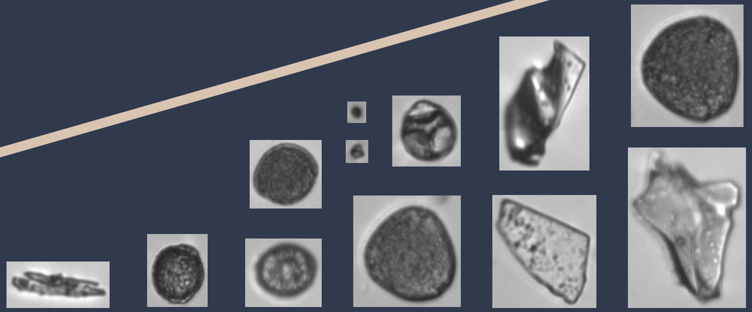


Residual Neural network

RESNET50 with pretrained weights

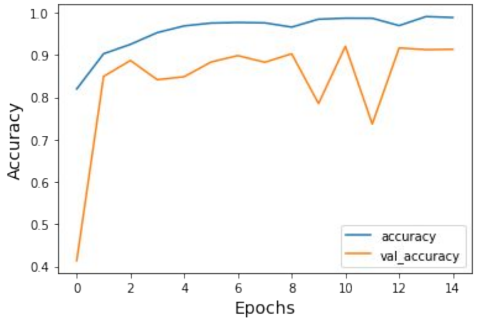
Same architecture without and with numeric data.

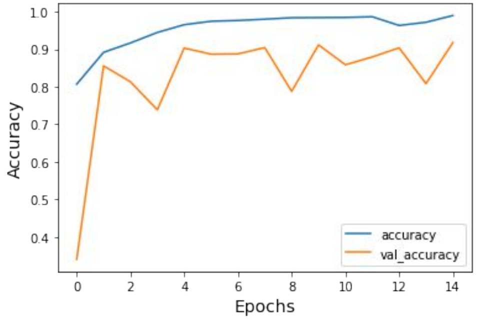
Pretrained weights are loaded and fine-tuned on our own images.



Numeric data concatenated with output from RESNET50 in the final dense layer.

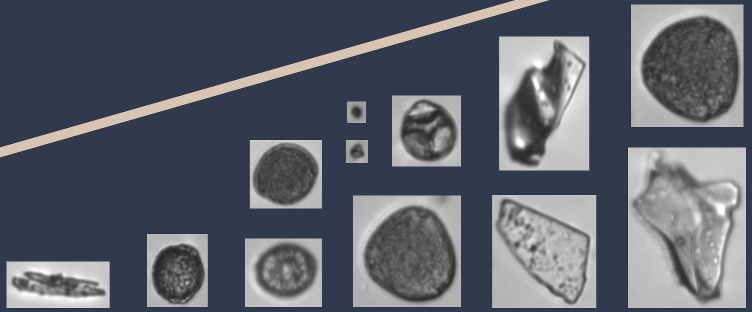
Similar accuracy without and with numeric data (91.38% and 91.77%, respectively)





Comparison

Comparison of results from different models



Model	Accuracy
Gradient boost 5000 / all	83.12% / 90.31%
DNN 5000 / all	84.72% / 90.56%
RF 5000 / all	-% / 78.33%
Simple CNN 5000 from each class	80.95%
Residual CNN 5000 from each class	82.70%
Residual CNN 5000 from each class	83.09%
RESNET50 5000 from each class	91.38%
RESNET50 5000 from each class	91.77%

The test data

Complete set is ~3'000'000 particles across 4 .csv files of metadata + images

Otherwise same metadata available as for the training set

No labels on particles
No pollen expected

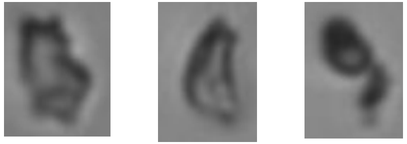
➡ UNSUPERVISED

10'000 first lines from each file

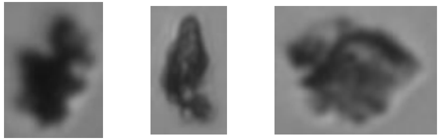
Test data images by quick visual inspection:

A lot of this:

A bit of this:



And a bit of this:



The test data

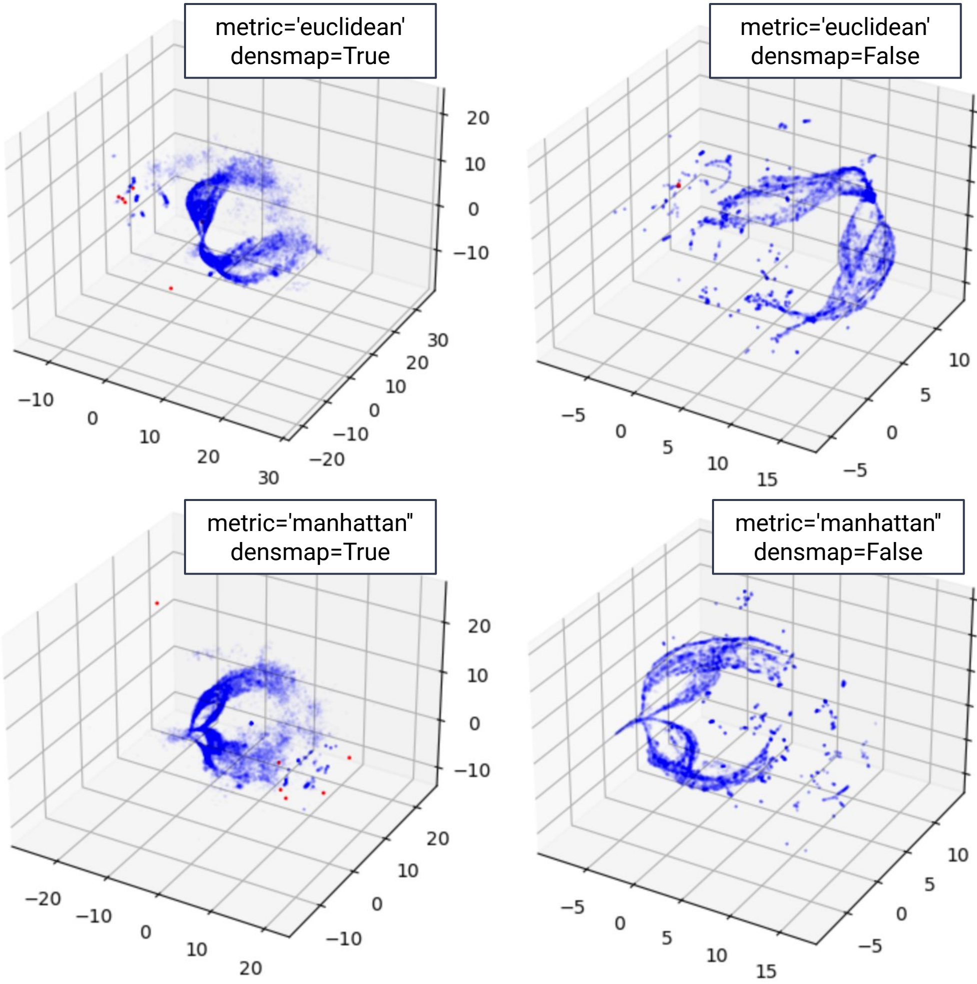
Illustrating HP impact:

UMAP main HP choices:

> metric='euclidean' / 'manhattan'
> n_neighbors$_{UMAP}$=15
> densmap=True / False

LocalOutlierFactor (sklearn) HP choice:

> n_neighbors$_{LOLF}$=50



6 outliers

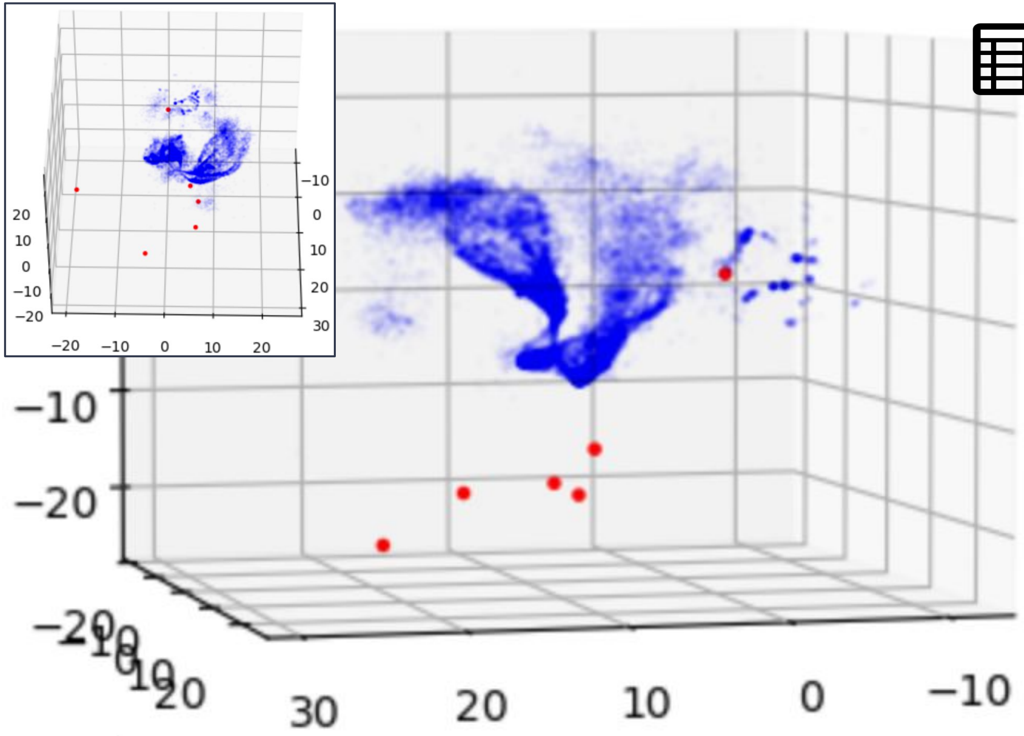
Real images chosen as most outlying after UMAP

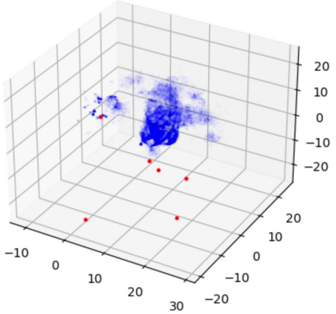
UMAP main HP choices:

metric='euclidean'
$n\_neighbors_{UMAP}$=10
densmap=True

LocalOutlierFactor (sklearn) HP choice:

$n\_neighbors_{LOLF}$=50

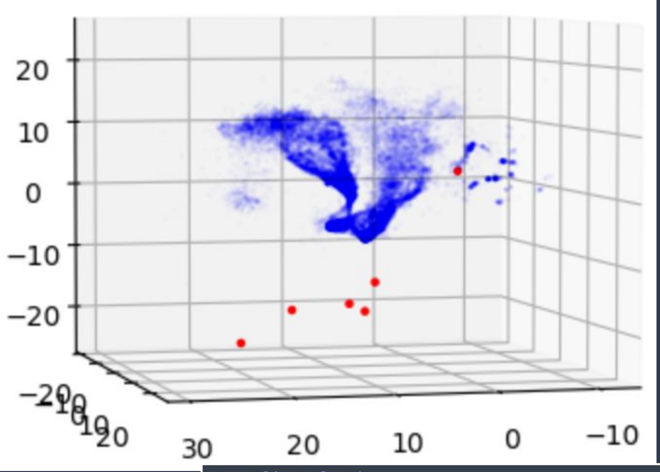





```
6 outliers found:

['...GRIP_3046_40_55_3/GRIP_3046_40_55_3_25738.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31865.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31922.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_2984.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_5811.png'
 '...GRIP_3306_40_55_3/GRIP_3306_40_55_3_168115.png']
```

6 outliers

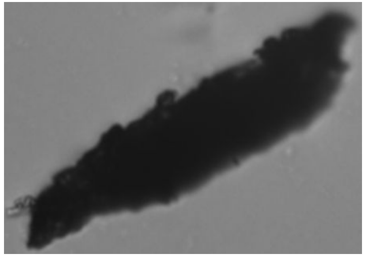
Real images chosen as most outlying after UMAP



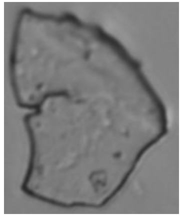

```
6 outliers found:

['...GRIP_3046_40_55_3/GRIP_3046_40_55_3_25738.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31865.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31922.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_2984.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_5811.png'
 '...GRIP_3306_40_55_3/GRIP_3306_40_55_3_168115.png']
```

(size)

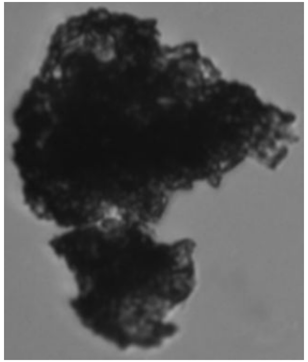
25738.png



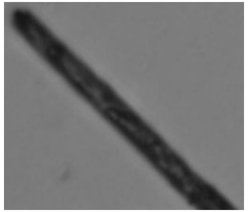
31865.png



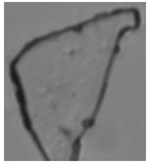
31922.png



2984.png

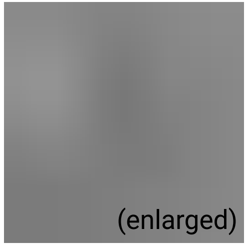


5811.png



("actual size")

168115.png



Alternate approaches

Prioritizing large amounts of test data

"Manual" pre-sorting

- Sort by image size…

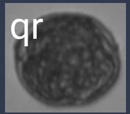
ML-assisted pre-sorting

- Train model on training data (quick exec.), predict and discard dust in test data
- Sort by important features (roughness, intensity) from e.g. LightGBM
- Predict pollen in test data or just non-dust

Test data

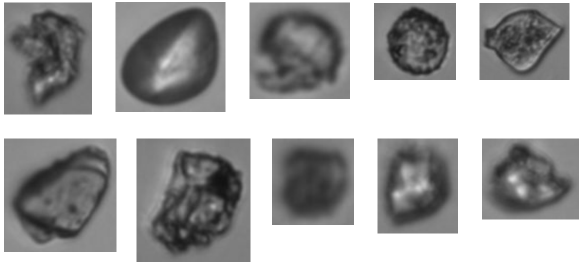
No pollen type (corylus, qrobur, qsuber) particles expected in test data

Predicting pollen in test data from model trained on training data might yield interesting results



Model	# pollen predicted
Gradient boost all	348 (0.011%)
DNN all	381 (0.012%)
Pytorch NN all	12144 (0.4%)

corylus test predictions from LightGBM:



Thanks for listening!

Any questions?

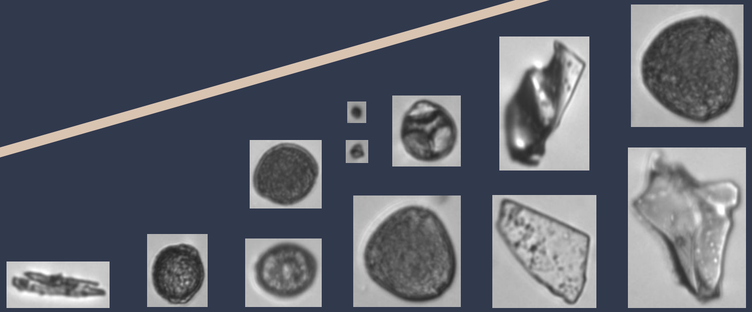


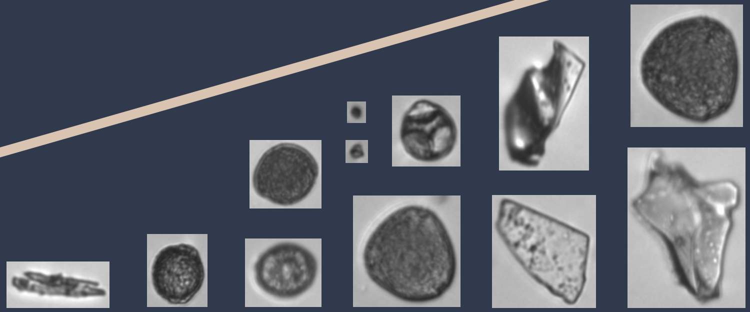
Image preparation

Images are different shapes and sizes

$6 \text{ px} \leqq \text{width} \leqq 889 \text{ px}$

$6 \text{ px} \leqq \text{height} \leqq 859 \text{ px}$

For consistent treatment we need uniform image sizes



Immediate options for image preparation

- Direct zero-padding
- Stretching
- Scaling then padding

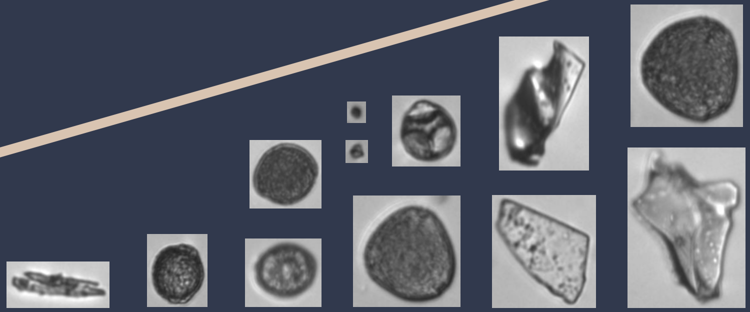
Direct zero padding would keep the most information intact, but also results in very large images; every image becomes at least 889 * 859.
For comparison the MNIST data is 28 * 28.

Stretching is undesirable, since shape information is skewed, and from visual inspection of the first few images it is apparent that shape is important.

Discarding images with at least one dimension smaller/larger than x_c will preferentially discard from the smaller/larger classes which is a problem.

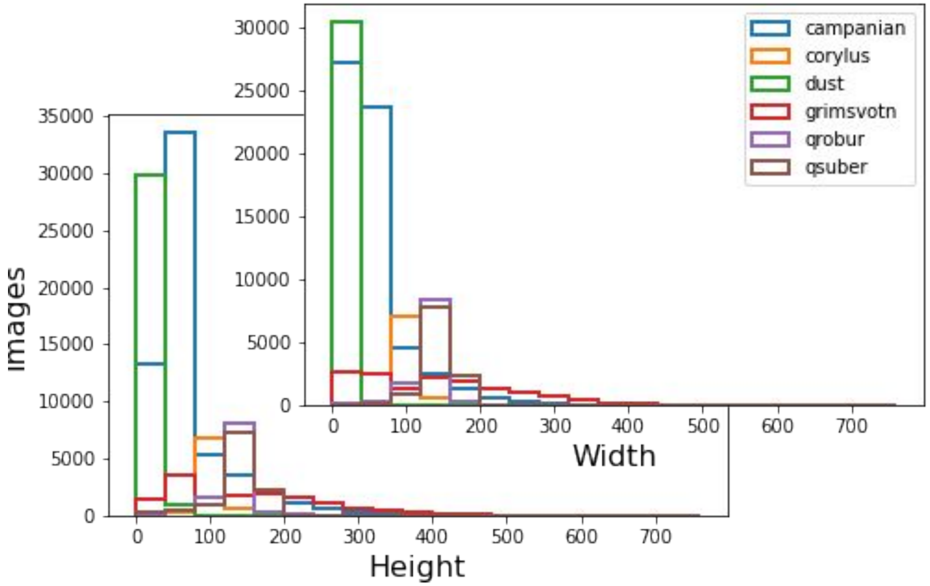
Scaling then padding

Target dimensions chosen from histograms of image dimensions



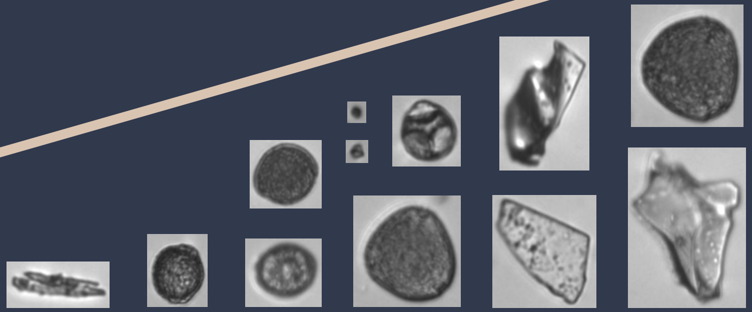
We generated histograms showing distributions of widths and heights of images.

Only a very few images have at least one dimension larger than 400 px:



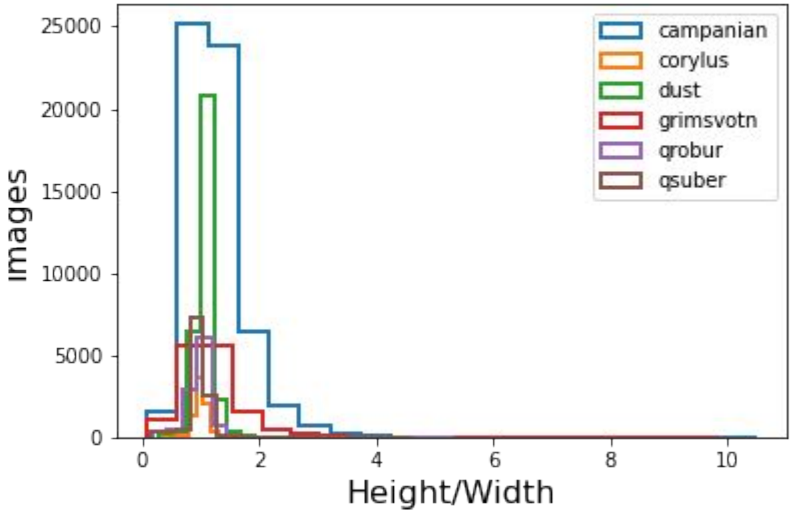
Scaling then padding

Target dimensions chosen from histograms of image dimensions



Another look at the distribution of aspect ratios lets us decide on scaling then padding to square dimensions:

This leaves us with just one tunable parameter for image size.



Simple neural network

Our simple neural network for image classification. Hyperparameters are optimized within the specified distributions.

```python
def create_model(filters1=32, filters2=64, units1=128, dp=0.4, lr=0.00005):
    model = Sequential(name='model')
    model.add(Conv2D(filters=filters1,
                     kernel_size=3,
                     strides=1,
                     padding='same',
                     activation='relu',
                     input_shape=(100, 100, 1)))
    model.add(Dropout(rate=dp))
    model.add(MaxPooling2D(pool_size=2, strides=None))
    model.add(Conv2D(filters=filters2,
                     kernel_size=3,
                     strides=1,
                     padding='same',
                     activation='relu'))
    model.add(Dropout(rate=dp))
    model.add(MaxPooling2D(pool_size=2, strides=None))
    model.add(Flatten())
    model.add(Dense(units=units1, activation='relu'))
    model.add(Dense(units=num_classes, activation='softmax'))
    opt = Adam(learning_rate=lr)
    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer=opt,
                  metrics=['accuracy'])
    return model
```

```python
distributions = dict(filters1=randint(low=8, high=128),
                     filters2 = randint(low=16, high=256),
                     units1=randint(low=32, high=256),
                     dp=uniform(loc=0.1, scale=0.4),
                     lr=uniform(loc=0.00005, scale=0.01))
```

Own RESNET inspired model

Our own implementation of a residual neural network inspired by RESNET50, but with fewer layers. We include four shortcuts/skip connections and four residual blocks in this model in addition to the input and output blocks.

Left: input block for model with both images and numeric data as inputs.
Right: Output block for model with both images and numeric data as inputs.
We input the numeric data in the next to last dense layer.

```
img_inputs = keras.Input(shape=(100, 100, 1))
num_inputs = keras.Input(shape=(num_features))

conv1 = Conv2D(filters=filters1, kernel_size=(3,3), strides=(1,1))(img_inputs)
bn1 = BatchNormalization()(conv1)
relu1 = Activation('relu')(bn1)
pool1 = MaxPooling2D(pool_size=(2,2), strides=(2,2))(relu1)
```

```
av_pool = AveragePooling2D(pool_size=(2,2), strides=(2,2))(add4)
flat = Flatten()(av_pool)
dense1 = Dense(units=filters4)(flat)
combinedInput = Concatenate(axis=1)([dense1, num_inputs])
dense2 = Dense(units=filters5)(combinedInput)
output = Dense(units=num_classes, activation='softmax')(dense2)

model = Model(inputs=[img_inputs, num_inputs], outputs=output)
```

(See next slide for example of residual and shortcut block)

Own RESNET inspired model 2

Below is an example of a convolution/residual block, a shortcut block and the layer where the results are added together. This structure is repeated four times in our model between the input and output block shown on the previous slide.

We experimented with different layers in order to optimize performance.

```
# convolution block
conv2 = Conv2D(filters=filters1, kernel_size=(3,3), strides=(1,1), padding='same')(pool1)
bn2 = BatchNormalization()(conv2)
relu2 = Activation('relu')(bn2)
conv3 = Conv2D(filters=filters1, kernel_size=(3,3), strides=(1,1), padding='valid')(relu2)
bn3 = BatchNormalization()(conv3)
relu3 = Activation('relu')(bn3)

# shortcut
conv4 = Conv2D(filters=filters1, kernel_size=(3,3), strides=(1,1), padding='valid')(pool1)
relu4 = Activation('relu')(conv4)
bn3 = BatchNormalization()(relu4)

# add results together
add1 = Add()([relu3, bn3])
```

RESNET50

RESNET50 with pre-trained weights with images as inputs. The weights in the layers are fine-tuned based on our images.

```
model = Sequential()
model.add(ResNet50(weights='imagenet', input_tensor=input, include_top=False))
model.add(Flatten())
model.add(Dense(6, activation = 'softmax'))
opt = Adam(learning_rate=0.00005)
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'], optimizer=opt)
history = model.fit(X_train_larger, y_train, epochs=15, validation_data=(X_test_larger, y_test))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Functional)        (None, 4, 4, 2048)        23587712
_________________________________________________________________
flatten_1 (Flatten)          (None, 32768)             0
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 196614
=================================================================
Total params: 23,784,326
Trainable params: 23,731,206
Non-trainable params: 53,120
```

RESNET50 2

RESNET50 with pre-trained weights with both images and numeric data as inputs. The weights in the layers are fine-tuned based on our data.

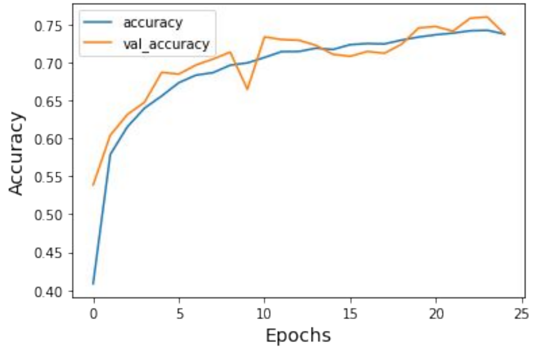
```
def create_model():
    img_inputs = keras.Input(shape=(100, 100, 3))
    num_inputs = keras.Input(shape=(num_features))
    resnet = ResNet50(weights='imagenet', input_tensor=img_inputs, include_top=False)(img_inputs)
    flat = Flatten()(resnet)
    combinedInput = Concatenate(axis=1)([flat, num_inputs])
    output = Dense(6, activation = 'softmax')(combinedInput)
    model = Model(inputs=[img_inputs, num_inputs], outputs=output)
    return model
model = create_model()
opt = Adam(learning_rate=0.00005)
model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'], optimizer=opt)
history = model.fit([X_train_larger, num_train], y_train, epochs=15, validation_data=([X_test_larger, num_test], y_test))
```

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_9 (InputLayer)            [(None, 100, 100, 3) 0

resnet50 (Functional)           (None, 4, 4, 2048)   23587712    input_9[0][0]

flatten_1 (Flatten)             (None, 32768)        0           resnet50[0][0]

input_10 (InputLayer)           [(None, 39)]         0

concatenate_1 (Concatenate)     (None, 32807)        0           flatten_1[0][0]
                                                                 input_10[0][0]

dense (Dense)                   (None, 6)            196848      concatenate_1[0][0]
==================================================================================================
Total params: 23,784,560
Trainable params: 23,731,440
Non-trainable params: 53,120
```

RESNET50 3

RESNET50 with pre-trained weights. The weights in the RESNET50 layers are frozen (made non-trainable), and the added dense layers after the flatten layer are trained on our images.

The model performs very poorly (even for more epochs) compared to the same model without frozen weights (where all layers are trainable - from the main slides)



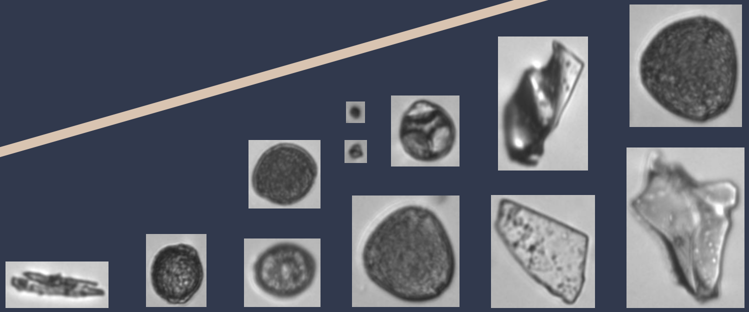

```
model = Sequential()
model.add(ResNet50(weights='imagenet', input_tensor=input, include_top=False))
model.add(Flatten())
model.add(Dense(128, activation='tanh'))
model.add(Dense(16, activation='tanh'))
model.add(Dense(6, activation = 'softmax'))

model.layers[0].trainable = False
model.layers[1].trainable = False
```

Layer (type)	Output Shape	Param #
resnet50 (Functional)	(None, 4, 4, 2048)	23587712
flatten_1 (Flatten)	(None, 32768)	0
dense_3 (Dense)	(None, 128)	4194432
dense_4 (Dense)	(None, 16)	2064
dense_5 (Dense)	(None, 6)	102

Total params: 27,784,310
Trainable params: 4,196,598
Non-trainable params: 23,587,712

Random Forest

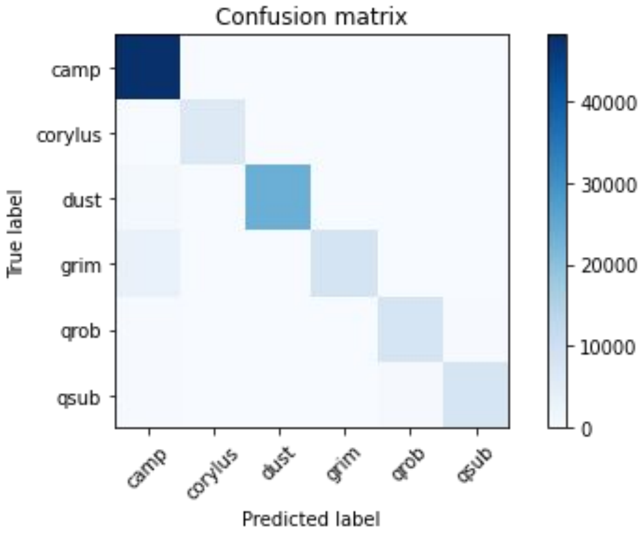
Deep NN model



Quantile transform

Random Forest + Random HP optimization

0.78 accuracy



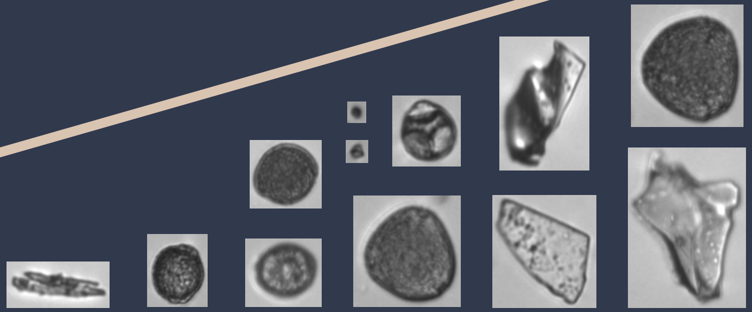
NN model (4 Dense layers and 1 drop layer)

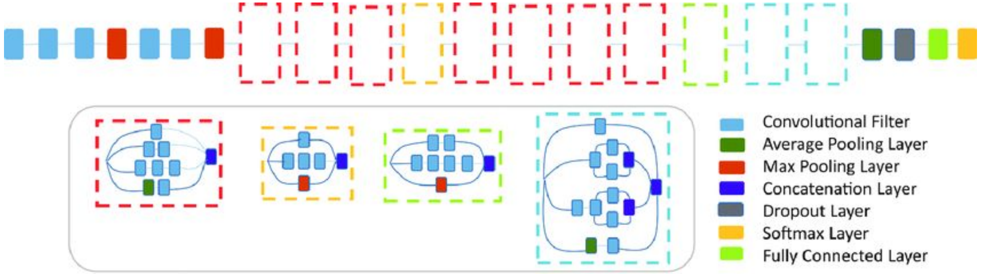
0.91 accuracy

Inception Model

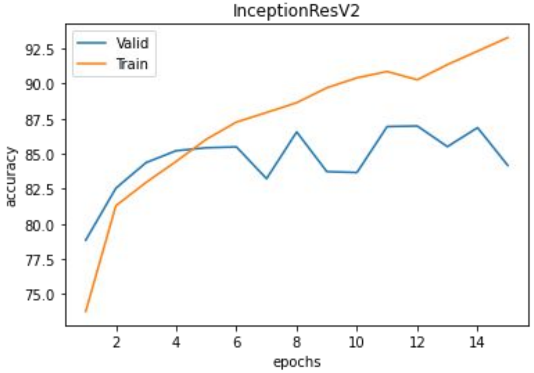
Made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concats, dropouts, and fully connected layers.

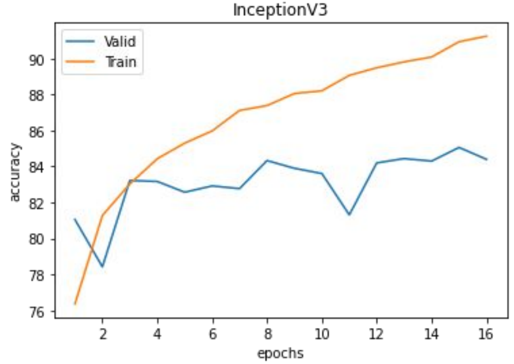
Test using InceptionResV2, InceptionV3 pretrained model.

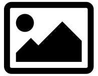




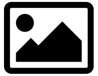
Inception Modules







Add average pooling
dense layer
At the end

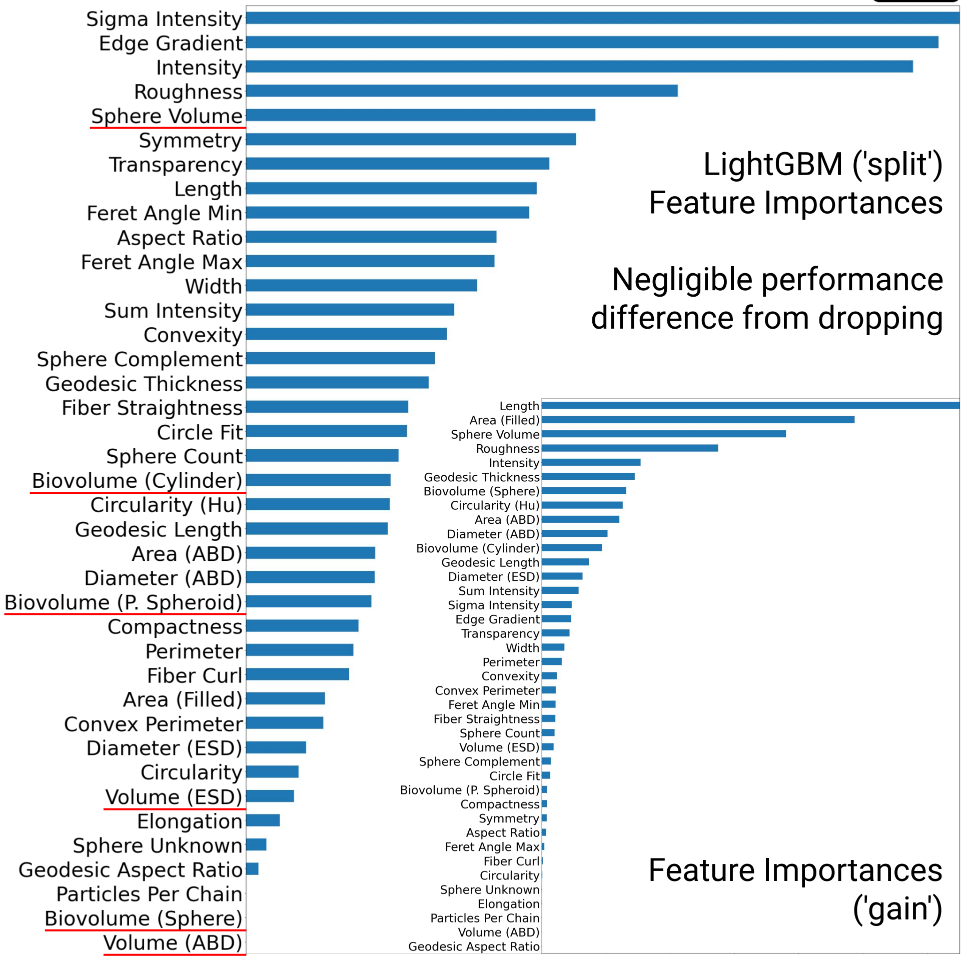


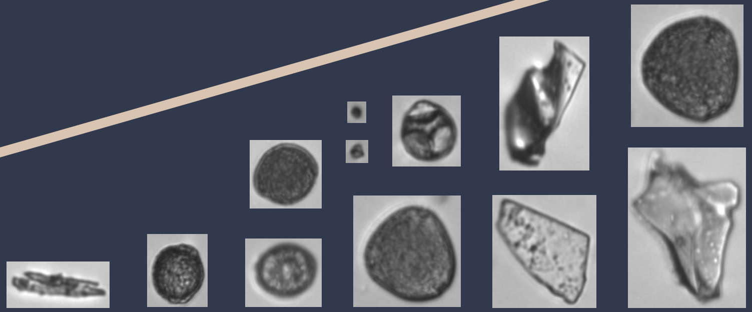
InceptionV2Res:88%
InceptionV3: 85%

Appendix LightGBM

Gradient boost (LightGBM) using numeric data (.csv) only

Feature importances



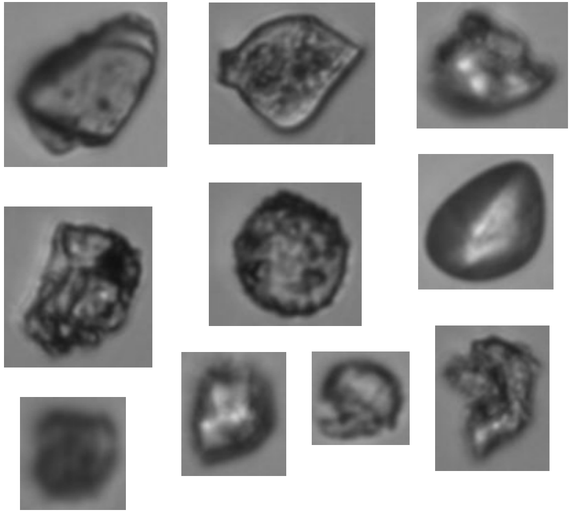


Appendix LightGBM

LightGBM test set example pollen predictions - 348 total (~0.011%)

5 first and 5 last corylus type

```
corylus particles predicted in test data:
1357           ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_1358.png
22955         ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_22956.png
75225       ...GRIP_3046_20_40_1/GRIP_3046_20_40_1_12103.png
102350      ...GRIP_3046_20_40_1/GRIP_3046_20_40_1_39228.png
141346       ...GRIP_3046_40_55_1/GRIP_3046_40_55_1_1759.png
                                ...
914562      ...GRIP_3306_20_40_3/GRIP_3306_20_40_3_42774.png
926024      ...GRIP_3306_20_40_3/GRIP_3306_20_40_3_54236.png
1131733     ...GRIP_3306_40_55_1/GRIP_3306_40_55_1_85539.png
1390926     ...GRIP_3306_40_55_3/GRIP_3306_40_55_3_17411.png
1398845     ...GRIP_3306_40_55_3/GRIP_3306_40_55_3_25330.png
Name: imgpaths, Length: 84, dtype: object
```

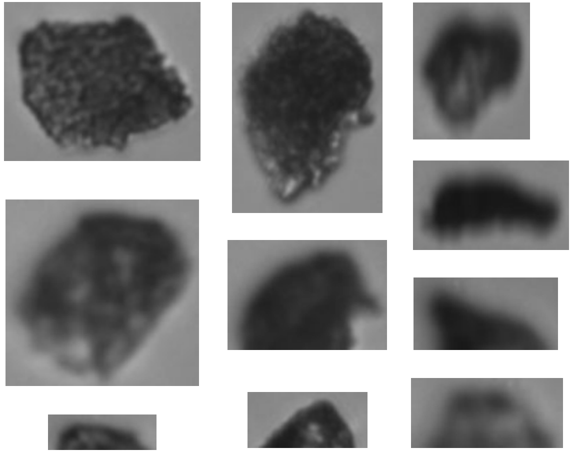


Appendix LightGBM

LightGBM test set example ash predictions - 348 total (~0.011%)

5 first and 5 last qrobur type

```
qrobur particles predicted in test data:
617              ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_618.png
2816            ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_2817.png
21950          ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_21951.png
22009          ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_22010.png
28556          ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_28557.png
                              ...
807095       ...GRIP_3306_20_40_2/GRIP_3306_20_40_2_76033.png
872332         ...GRIP_3306_20_40_3/GRIP_3306_20_40_3_544.png
950847       ...GRIP_3306_20_40_3/GRIP_3306_20_40_3_79059.png
1032002     ...GRIP_3306_20_40_3/GRIP_3306_20_40_3_160214.png
1046294        ...GRIP_3306_40_55_1/GRIP_3306_40_55_1_100.png
Name: imgpaths, Length: 103, dtype: object
```

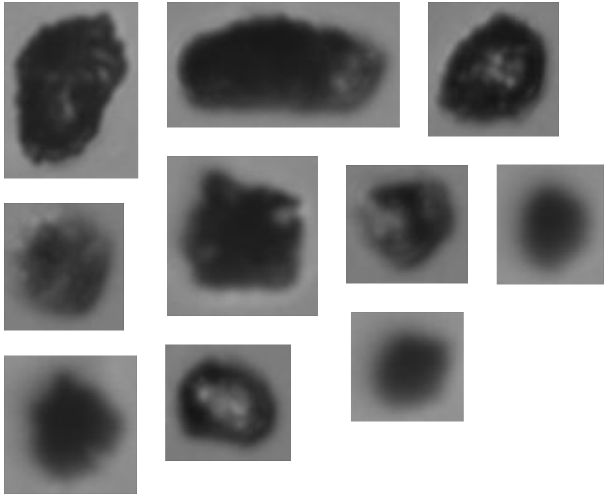


Appendix LightGBM

LightGBM test set example pollen predictions - 348 total (~0.011%)

5 first and 5 last qsuber type

```
qsuber particles predicted in test data:
1525           ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_1526.png
2755           ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_2756.png
11341         ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_11342.png
16142         ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_16143.png
23027         ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_23028.png
                                ...
1070576     ...GRIP_3306_40_55_1/GRIP_3306_40_55_1_24382.png
1131984     ...GRIP_3306_40_55_1/GRIP_3306_40_55_1_85790.png
1201002    ...GRIP_3306_40_55_1/GRIP_3306_40_55_1_154808.png
1259754     ...GRIP_3306_40_55_2/GRIP_3306_40_55_2_52453.png
1314964    ...GRIP_3306_40_55_2/GRIP_3306_40_55_2_107663.png
Name: imgpaths, Length: 161, dtype: object
```

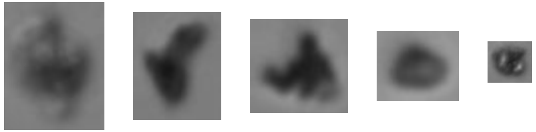


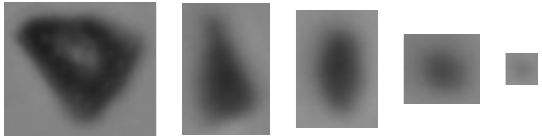
Appendix LightGBM

LightGBM test set example ash predictions - 133'196 total (~4.317%)
117'328 campanian (~3.803%) + 15'868 grimsvotn (~0.514%)

5 first campanian and 5 grimsvotn types

```
campanian particles predicted in test data:
42                ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_43.png
51                ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_52.png
60                ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_61.png
66                ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_67.png
71                ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_72.png
                                ...
grimsvotn particles predicted in test data:
12                ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_13.png
13                ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_14.png
183              ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_184.png
316              ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_317.png
377              ...GRIP_3046_0_20_1/GRIP_3046_0_20_1_378.png
                                ...
```





Appendix UMAP

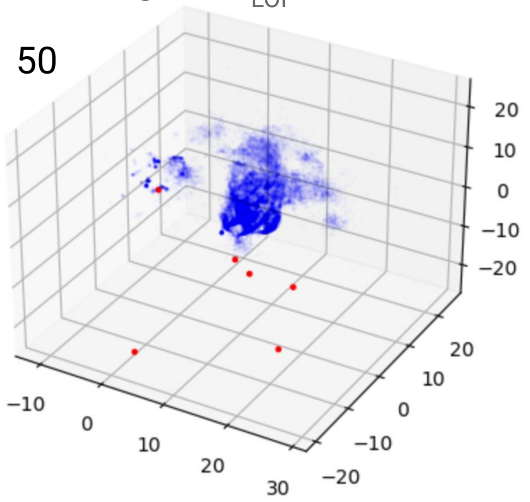
```
6 outliers found:                                        50

['...GRIP_3046_40_55_3/GRIP_3046_40_55_3_25738.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31865.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31922.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_2984.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_5811.png'
 '...GRIP_3306_40_55_3/GRIP_3306_40_55_3_168115.png']
```

```
6 outliers found:                                        100

['...GRIP_3046_40_55_3/GRIP_3046_40_55_3_25521.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_25738.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31479.png'
 '...GRIP_3046_40_55_3/GRIP_3046_40_55_3_31922.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_2984.png'
 '...GRIP_3136_40_55_3/GRIP_3136_40_55_3_5811.png']
```

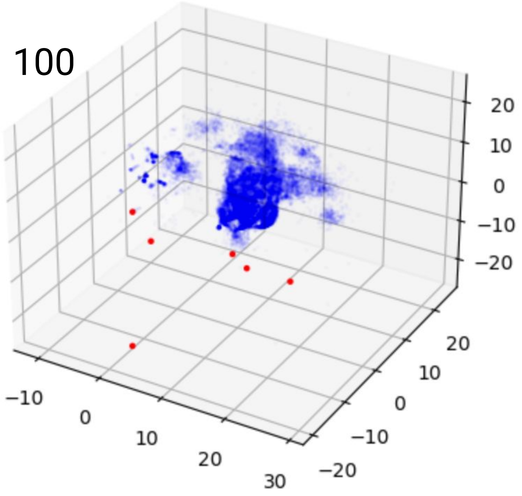
UMAP for $\text{n\_neighbors}_{\text{UMAP}}$=15, densmap=True, metric='euclidean'
10000 first entries from each .csv file
6 outliers with dot size increased for visibility

$\text{n\_neighbors}_{\text{LOF}}$ set as either 50 or 100

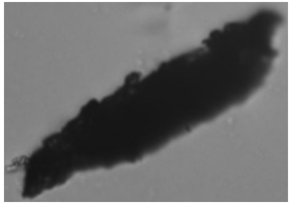
50



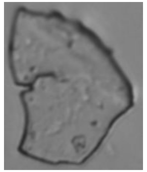
100



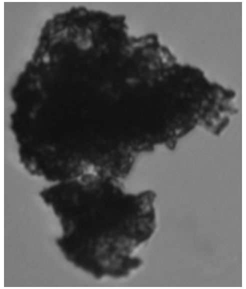
25738.png



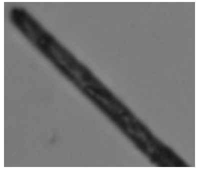
31865.png



31922.png

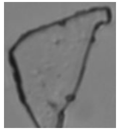


2984.png



31479.png

5811.png



+ dust

Keras Dense Neural Network

```
model = Sequential([
    Dense(39,activation='relu',name='input_layer'),
    Dense(500,activation='relu',name='hidden_layer1'),
    Dense(250,activation='relu',name='hidden_layer2'),
    Dense(6, activation='softmax', name='output')])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=["CategoricalAccuracy"])




history = model.fit(x = np.asarray(X_train).astype('float32'), y = np.asarray(y_train).astype('float32'),
                    validation_data=(np.asarray(X_test).astype('float32'),
                                     np.asarray(y_test).astype('float32')),batch_size=9, epochs= 20)
```

Keras Dense Neural Network

Searching in nodes. Very best is 1000/1000, however the simpler 500/250 is almost as good and chosen for speed/simplicity. It also had a low std on the crossvalidation results.

```
Best: 0.894048 using {'LA': 'softmax', 'Node1': 1000, 'Node2': 1000, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.887167 (0.002290) with: {'LA': 'softmax', 'Node1': 50, 'Node2': 50, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.891578 (0.004078) with: {'LA': 'softmax', 'Node1': 50, 'Node2': 250, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.883256 (0.004191) with: {'LA': 'softmax', 'Node1': 50, 'Node2': 500, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.889343 (0.004523) with: {'LA': 'softmax', 'Node1': 50, 'Node2': 1000, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.891931 (0.002074) with: {'LA': 'softmax', 'Node1': 250, 'Node2': 50, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.891166 (0.002507) with: {'LA': 'softmax', 'Node1': 250, 'Node2': 250, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.887402 (0.005839) with: {'LA': 'softmax', 'Node1': 250, 'Node2': 500, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.890843 (0.001340) with: {'LA': 'softmax', 'Node1': 250, 'Node2': 1000, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.889490 (0.005163) with: {'LA': 'softmax', 'Node1': 500, 'Node2': 50, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.892637 (0.001558) with: {'LA': 'softmax', 'Node1': 500, 'Node2': 250, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.889197 (0.009438) with: {'LA': 'softmax', 'Node1': 500, 'Node2': 500, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.887961 (0.003316) with: {'LA': 'softmax', 'Node1': 500, 'Node2': 1000, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.890196 (0.003808) with: {'LA': 'softmax', 'Node1': 1000, 'Node2': 50, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.890902 (0.002855) with: {'LA': 'softmax', 'Node1': 1000, 'Node2': 250, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.892548 (0.003027) with: {'LA': 'softmax', 'Node1': 1000, 'Node2': 500, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
0.894048 (0.001775) with: {'LA': 'softmax', 'Node1': 1000, 'Node2': 1000, 'batch_size': 9, 'epochs': 20, 'optimizer': 'adam'}
```