

Assignment No. 3

Name: Hagit Shaposhnik
ID: 037650009

Part 1 – Aim: ANN from Scratch

The following “compare changes” presents the changes that was made to add a 2nd layer to the supplied code.

Helper functions

Modification in *compute_mse_and_acc*

10	10		<code>for i, (features, targets) in enumerate(minibatch_gen):</code>
11		-	<code>_, probas = nnet.forward(features)</code>
	11	+	<code>_, _, probas = nnet.forward(features)</code>
12	12		<code>predicted_labels = np.argmax(probas, axis=1)</code>

Line 11. The `_` variable represents the output layer (`_, _` represents 2 layers).

Modification in *Train*

72	-	<code>a_h, a_out = model.forward(X_train_mini)</code>	
72	+	<code>a_h, a_h2, a_out = model.forward(X_train_mini)</code>	
73	73		
74	74	<code>#### Compute gradients ####</code>	
75	-	<code>d_loss__d_w_out, d_loss__d_b_out, d_loss__d_w_h, d_loss__d_b_h = \</code>	
76	-	<code>model.backward(X_train_mini, a_h, a_out, y_train_mini)</code>	
	75	+	<code>d_loss__d_w_out, d_loss__d_b_out, \</code>
	76	+	<code>d_loss__d_w_h, d_loss__d_b_h, \</code>
	77	+	<code>d_loss__d_w_h2, d_loss__d_b_h2 = \</code>
	78	+	<code>model.backward(X_train_mini, a_h, a_h2, a_out, y_train_mini)</code>
77	79		
78	80		<code>#### Update weights ####</code>
79	81		<code>model.weight_h -= learning_rate * d_loss__d_w_h</code>
80	82		<code>model.bias_h -= learning_rate * d_loss__d_b_h</code>
	83	+	<code>model.weight_h2 -= learning_rate * d_loss__d_w_h2</code>
	84	+	<code>model.bias_h2 -= learning_rate * d_loss__d_b_h2</code>

Line 72. The *ah* variable represents the output layer (*ah*, *ah* _ represents 2 layers).

Lines 75-78. The backward function returns the loss also for the 2nd layer.

Line 83-84. Adjust the weight and bias of the 2nd layer by using the 2nd layer loss value.

NeuralNetMLP Class

Modification in *class init*

```
117 + # 2nd layer weights (size * 2 than the first layer)
118 + rng = np.random.RandomState(random_seed)
119 +
120 + self.weight_h2 = rng.normal(
121 +     loc=0.0, scale=0.1, size=(num_hidden * 2, num_hidden))
122 + self.bias_h2 = np.zeros(num_hidden * 2)

112 123
113 124 # output
114 125 self.weight_out = rng.normal(
115 -     loc=0.0, scale=0.1, size=(num_classes, num_hidden))
126 +     loc=0.0, scale=0.1, size=(num_classes, num_hidden * 2))
```

Line 117-122. Adding 2nd layer weight and bias initiate with random values, the 2nd layer size is twice the size of the 1st layer.

Line 126. Changing the output weight, the input of the last layer switched to the output of the 2nd layer. Thus, the input size is twice the size it was before.

Modification in *foward*

```
135 + # the second layer
136 + z_h2 = np.dot(a_h, self.weight_h2.T) + self.bias_h2
137 + a_h2 = sigmoid(z_h2)

124 138 +
125 139 # Output layer
126 - # input dim: [n_examples, n_hidden] dot [n_classes, n_hidden].T
140 + # input dim: [n_examples, n_hidden * 2] dot [n_classes, n_hidden * 2].T
127 141 # output dim: [n_examples, n_classes]
128 - z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
142 + z_out = np.dot(a_h2, self.weight_out.T) + self.bias_out
143 + # z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
129 144 a_out = sigmoid(z_out)
130 - return a_h, a_out
145 + return a_h, a_h2, a_out
```

Lines 136-137. The output of the 1st layer goes through the 2nd layer.

Lines 140-142. The input of the output layer is the 2nd layer output.

Line 145. Return the output of the 2nd layer.

Modification in *backward*

```
156 - # [n_examples, n_hidden]
157 - d_z_out_dw_out = a_h
171 + # [n_examples, n_hidden * 2]
172 + # the last layer before out is now layer 2 (size = n_hidden * 2)
173 + d_z_out_dw_out = a_h2
```

Line 173. The input of the output layer is the 2nd layer output. (Set variable to 2nd layer output)

```
168 - # [n_classes, n_hidden]
169 - d_z_out_a_h = self.weight_out
184 + # [n_classes, n_hidden * 2]
185 + d_z_out_a_h2 = self.weight_out
170 186
171 - # output dim: [n_examples, n_hidden]
172 - d_loss_a_h = np.dot(delta_out, d_z_out_a_h)
187 + # output dim: [n_examples, n_hidden * 2]
188 + # delta_out is for the final layer loss, d_z_out_a_h are the final layer weights
189 + d_loss_a_h2 = np.dot(delta_out, d_z_out_a_h2)
190 +
191 + # [n_examples, n_hidden * 2]
192 + d_a_h_d_z_h2 = a_h2 * (1. - a_h2) # sigmoid derivative
193 +
194 + # [n_examples, n_hidden]
195 + d_z_h_d_w_h2 = a_h1
```

Line 185. The input of the output layer is the 2nd layer output. (Set variable to output weight).

Line 189. Calculates the loss result from the 2nd layer.

Line 192. Calculate the sigmoid on the 2nd layer.


Line 195. Set variable to 1st layer output

```
197 + # output dim: [n_hidden * 2, n_features]
198 + delta_out_h2 = d_loss_a_h2 * d_a_h_d_z_h2
199 + d_loss_d_w_h2 = np.dot(delta_out_h2.T, d_z_h_d_w_h2)
200 + d_loss_d_b_h2 = np.sum(delta_out_h2, axis=0)
201 +
202 + # [n_classes, n_hidden * 2]
203 + d_z_out_a_h = self.weight_h2
204 +
205 + # output dim: [n_examples, n_hidden * 2]
206 + # delta_out is for the final layer loss, d_z_out_a_h are the final layer weights
207 + d_loss_a_h = np.dot(delta_out_h2, d_z_out_a_h)
```

Lines 198-200. Calculates the loss for the 2nd layer.

Line 203. Set variable to the weight of the 2nd layer (The layer after the 1st layer)

Line 207. Calculate the loss result from the 1st layer.

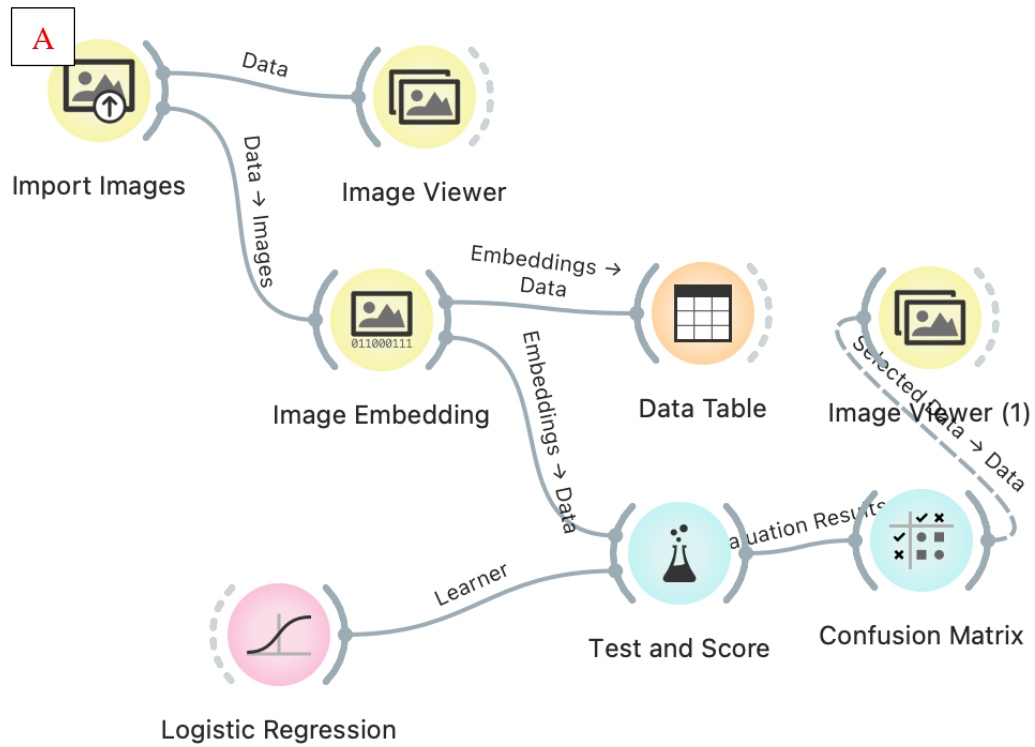
175	-	<code>d_a_h__d_z_h = a_h * (1. - a_h) # sigmoid derivative</code>
210	+	<code>d_a_h__d_z_h = a_h1 * (1. - a_h1)</code>
176	211	
177	212	<code># [n_examples, n_features]</code>
178	213	<code>d_z_h__d_w_h = x</code>
		<code>@@ -182,4 +217,5 @@ def backward(self, x, a_h, a_out, y):</code>
182	217	<code>d_loss__d_b_h = np.sum((d_loss__a_h * d_a_h__d_z_h), axis=0)</code>
183	218	
184	219	<code>return (d_loss__dw_out, d_loss__db_out,</code>
185	-	<code>d_loss__d_w_h, d_loss__d_b_h) ⊖</code>
220	+	<code>d_loss__d_w_h, d_loss__d_b_h,</code>
221	+	<code>d_loss__d_w_h2, d_loss__d_b_h2)</code>

Line 210. Calculates the sigmoid on the 1st layer.

Lines 220-221. Returns the loss for the 2nd layer.

Part 2- Aim: Practice the usage of CNN (Convolutional Neural Network).

To understand the data, I used the Orange3™



Orange Embedding

The screenshot shows two blocks in the Orange3 interface. The 'Image Embedding' block on the left is configured with 'image' as the attribute and 'SqueezeNet (local)' as the embedder. It shows 1155 instances. The 'Data Table' block on the right displays the resulting data table with 1155 instances, 1000 features, and 5 meta-attributes. The table has columns for 'hidden origin type', 'category', 'image name', and 'im'. The 'category' column contains values like 20, 20, 20, etc. The 'image name' column contains values like 04948, 04949, 04928, etc. The 'im' column contains values like 20/049, 20/049, 20/049, etc.

hidden origin type	category	image name	im
1	20	04948	20/049
2	20	04949	20/049
3	20	04928	20/049
4	20	04900	20/049
5	20	04914	20/049
6	20	04915	20/049
7	20	04901	20/049
8	20	04929	20/049
9	20	04917	20/049
10	20	04903	20/049
11	20	04902	20/049
12	20	04916	20/049

Block A is an import images block that reads the images into the orange framework. Orange default emending is a pre-trained network, SqueezeNet. The emending block converts the images into a table containing 1000 features (B). Each feature receives a probability that affiliates to a particular group of images.

Test and Score

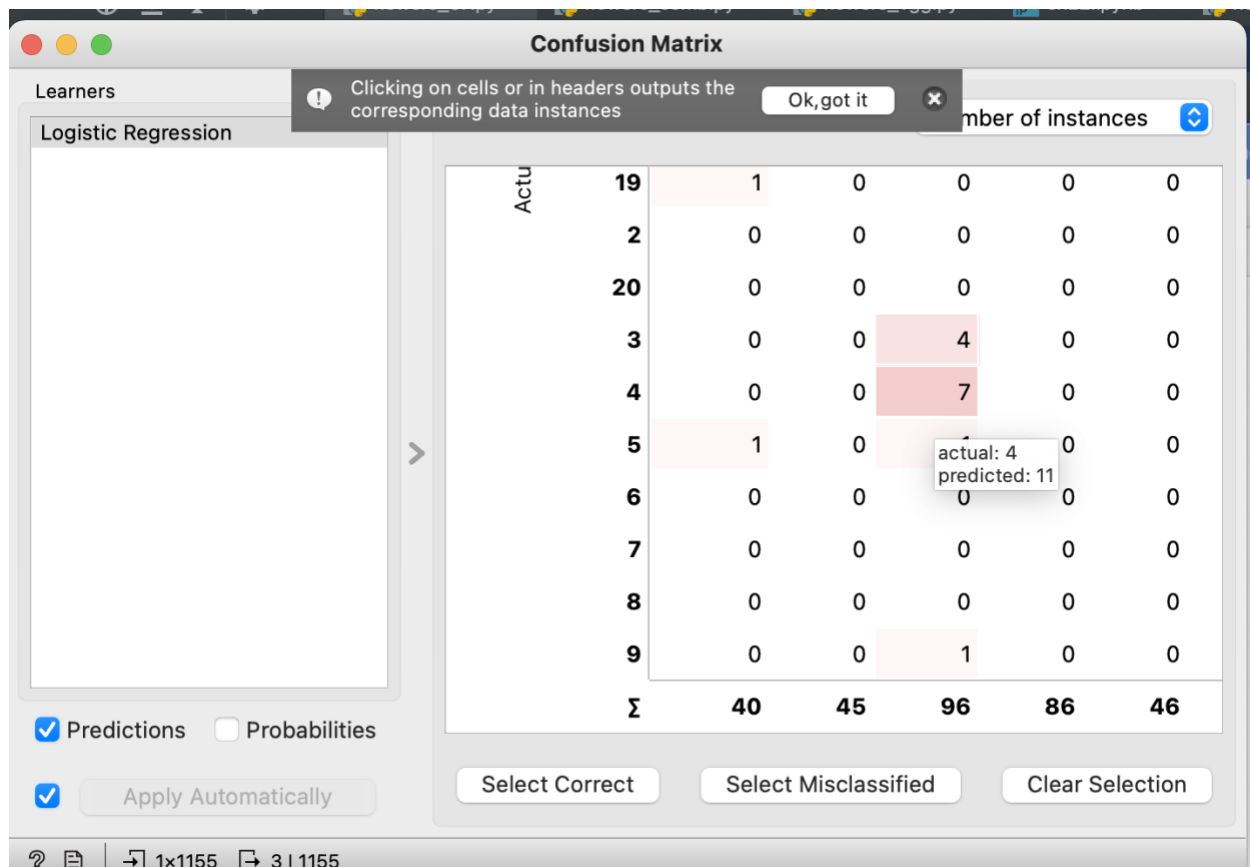
The screenshot shows the 'Test and Score' block in Orange3. The 'Cross validation' section is selected, with 'Number of folds' set to 5, 'Stratified' checked, and 'Repeat train/test' set to 10. The 'Evaluation results for target 1' table shows the following metrics for Logistic Regression:

Model	AUC	CA	F1	Precision	Recall
Logistic Regression	0.999	0.997	0.950	0.950	0.950

The 'Compare models by' section is set to 'Area under ROC' with a 'Negligible diff.' of 0.1. The table shows probabilities that the score for the model in the row is higher than that of the model in the column. Small numbers show the probability that the difference is negligible.

The results are quite high since I used a small number of Images (1155 images). Yet, it provides a good indication of the SqueezeNet.

Confusion Matrix



The following example shows that a group of 7 flowers mistakenly belong to category 11, although they belong to category 4.

Category 4 flowers



Category 11 flowers



11



11



11



11



11



11



11



11



11



11

Confused Flowers (4 -> 11)



4



4



4



4



4



4

Confusion Analysis

For example, the circled flower (in red) is a bit blur. Applying a blur augmentation on the dataset might deal with this issue.

Train, validate and test the whole dataset (102 oxford flowers)

Two pre-trained network were selected – SqueezeNet and MobileNet.
The pre-trained layer were set to non-trainable.

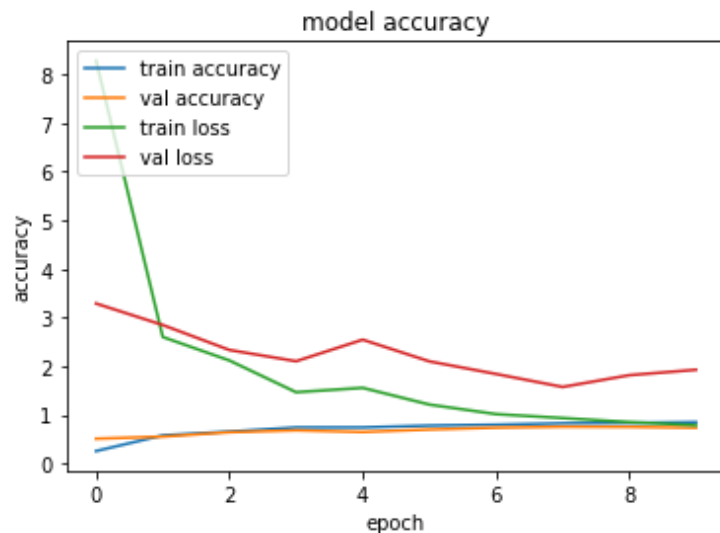
All images were imported and preprocessed using Keras image processing.

The dataset was randomly divided into training (50%), validation (25%) for hyperparameter tuning, and test sets (25%). This random split was repeated twice.

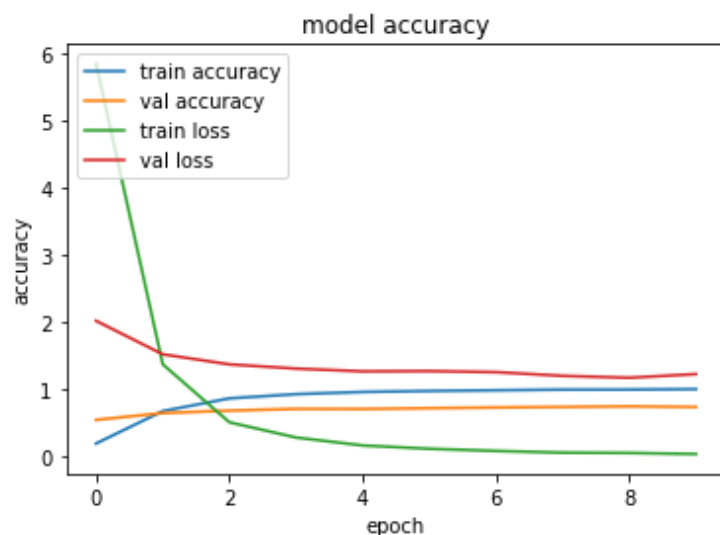
The last layer of each model is a Dense layer at a size of 102 (number of categories in the dataset)

Each fitting set consists of 10 epochs.

SqueezeNet Learning Graph



MobileNet Learning Graph



Results

SqueezeNet

1st run - accuracy of 73.4%

2nd run - - accuracy of 78.6%

MobileNet

1st run - - accuracy of 73.09%

2nd run - - accuracy of 74.46%