CSC321: Assignment 3

Due on Monday, March 21, 2016

Davi Frossard

March 29, 2016

Part 1

To find good hyperparameters for the model, we randomize the number of hidden units from 200 to 850, the activation function between ReLU and Tanh, and λ (the L2 regularization factor) from $9 \cdot 10^{-3}$ to 0, all models have dropout with a keep probability of 50% and are trained using Adam Optimizer with a learning rate of 0.0005. We also keep track of the validation accuracy throughout the training, therefore we are able to only save the model that better fits the data according to this metric. We also halt the training if the difference in cost between 5 epochs is less than $1 \cdot 10^{-5}$.

To prevent dead neurons in ReLU units, we initialize the weights randomly with values drawn from a truncated normal distribution with mean 0 and standard deviation equal to 0.01; the biases are initialized with all values set to 0.1, thus guaranteeing we have positive and negative values with small magnitudes and that the input to ReLU units do not kill them right at the beginning. We also normalize the inputs by dividing it by 255, resize it to (100,100), convert to grayscale and flatten the matrix, which guarantees values in the range of 0 to 1 while still maintaining color data, which might be useful in identifying the actors.

In order to come up with these *hyper-hyperparameters* we keep trying values smaller than the minimum or bigger than the maximum until it no longer produces a reasonable model. With the boundaries in hand we just mix them randomly in order to find the best models.

We make the training set with 510 random images, the validation set with 162 and the remaining goes to the test set (around 160 images). With these parameters we find the models listed in Table 1.

	Hyperparameters			**Cost**			**Accuracy (%)**		
ID	**Units**	**Function**	**Regularization**	**Train**	**Validation**	**Test**	**Train**	**Validation**	**Test**
1	[380]	['tanh']	0.0006	0.0773	0.7897	0.8785	99.8000	82.1000	79.2500
2	[355]	['tanh']	1×10^{-5}	0.0660	0.6351	0.6074	99.2200	79.6300	79.2500
3	[269]	['tanh']	3×10^{-5}	0.0425	0.7179	0.6416	99.4100	80.8600	77.3600
4	[610]	['tanh']	0.0000	0.1104	0.6160	0.6319	99.0200	80.2500	78.6200
5	[431]	['tanh']	6×10^{-5}	0.0308	0.6646	0.6672	100.0000	82.7200	81.7600
6	[261]	['tanh']	0.0002	0.0777	0.6617	0.6745	99.4100	80.8600	79.8700
7	[700]	['tanh']	7×10^{-5}	0.0944	0.6424	0.6575	99.0200	80.2500	77.9900
8	[315]	['tanh']	0.0005	0.0712	0.7789	0.7250	100.0000	78.4000	76.7300
9	[782]	['tanh']	3×10^{-5}	0.0265	0.8421	0.8952	99.4100	80.8600	76.7300
10	[763]	['relu']	7×10^{-5}	0.0563	0.9298	1.0601	99.6100	83.3300	77.3600
11	[242]	['tanh']	3×10^{-5}	0.0163	0.8342	0.8124	100.0000	80.2500	77.3600
12	[810]	['tanh']	0.0000	0.0350	0.7465	0.7098	99.6100	78.4000	75.4700
13	[795]	['relu']	0.0008	0.1852	0.9260	0.8935	99.0200	82.7200	77.3600
14	[504]	['relu']	0.0008	0.1326	0.9341	1.1180	99.8000	82.7200	76.1000
15	[513]	['tanh']	0.0001	0.0316	0.8619	0.8405	100.0000	80.2500	76.7300
16	[739]	['tanh']	0.0003	0.0795	0.7252	0.7715	99.6100	80.8600	75.4700
17	[692]	['relu']	6×10^{-5}	0.0398	0.7697	0.7616	100.0000	82.1000	81.1300
18	[836]	['relu']	2×10^{-5}	0.0820	0.6484	0.6801	98.8200	80.8600	77.9900
19	[251]	['tanh']	1×10^{-5}	0.0110	0.7309	0.5789	100.0000	80.8600	83.6500
20	[206]	['tanh']	0.0080	0.4734	0.8670	0.8367	94.7100	80.2500	75.4700
21	[588]	['relu']	0.0060	0.5725	0.9807	0.9966	95.1000	80.2500	76.7300
22	[337]	['tanh']	0.0005	0.1075	0.7053	0.8157	99.6100	81.4800	77.9900
23	[578]	['tanh']	5×10^{-5}	0.0389	0.7906	0.7747	99.6100	79.6300	77.3600
24	[615]	['relu']	3×10^{-5}	0.0227	0.9805	1.1125	100.0000	82.7200	79.8700
25	[350]	['tanh']	0.0005	0.0596	0.7537	0.7861	99.8000	82.1000	79.2500

Table 1: Hyperparameter search.

From Table 1, we find the best model according to validation accuracy, which gives us Table 2.

Hyperparameters			Cost			Accuracy (%)		
Units	**Function**	**Regularization**	**Train**	**Validation**	**Test**	**Train**	**Validation**	**Test**
[763]	['relu']	7×10^{-5}	0.0563	0.9298	1.0601	99.6100	83.3300	77.3600

Table 2: Best model from Table 1.

For this model, we have the training curves shown in Figure 1 and Figure 2.

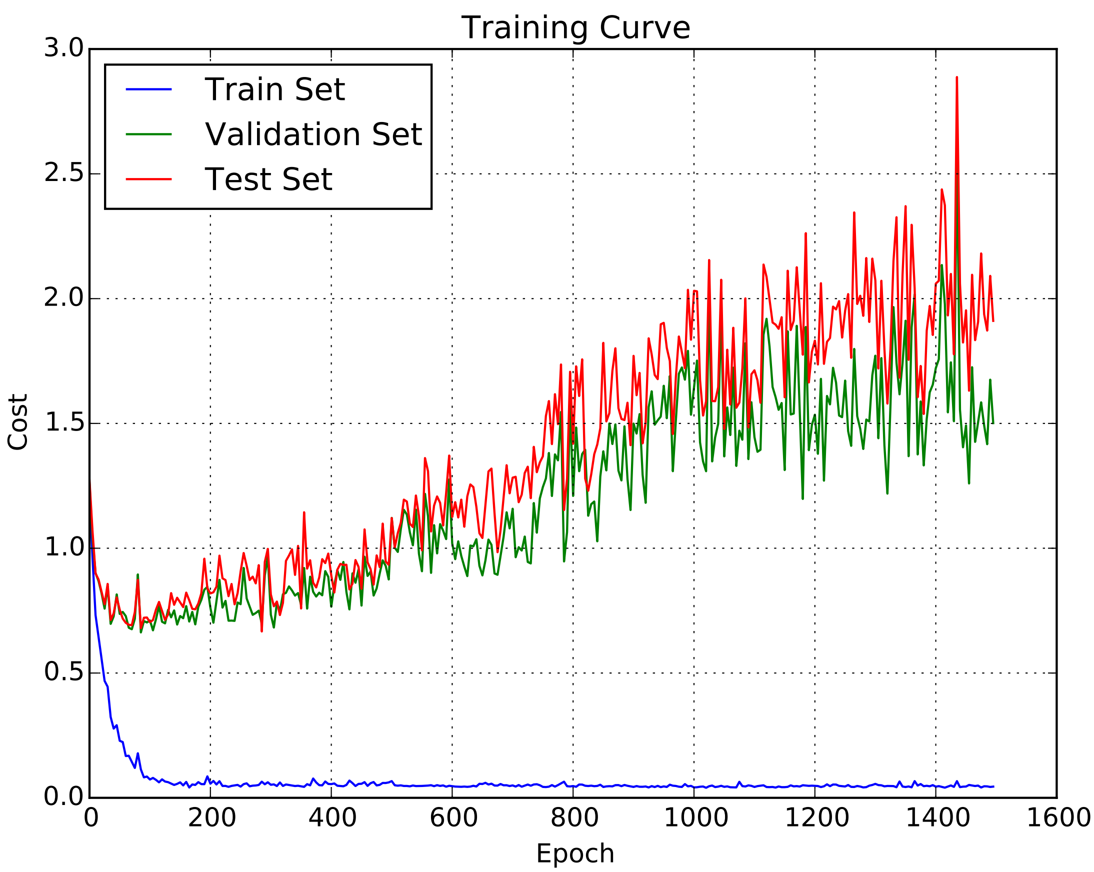

Figure 1: Cost curve for the model in Table 2.

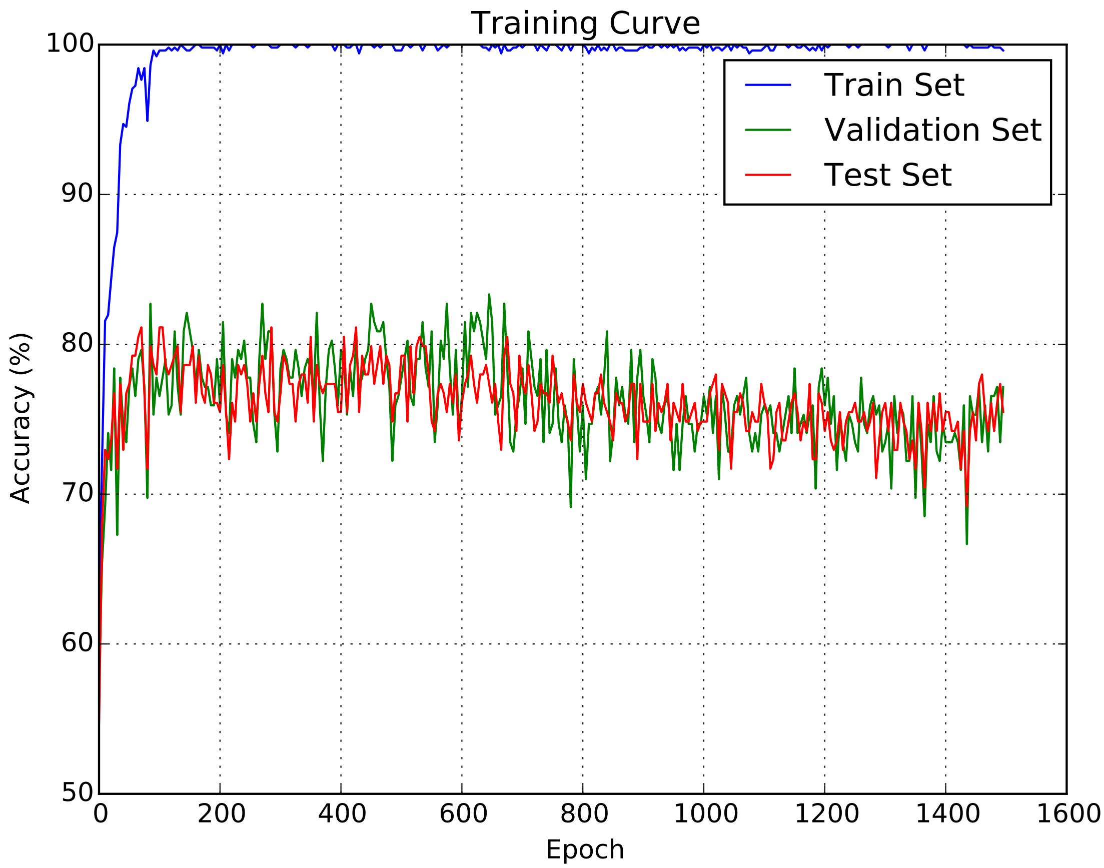

Figure 2: Accuracy curve for the model in Table 2.

Part 2

We follow a procedure similar to that discussed in Part 1, however we now have an extra hyperparameter: The AlexNet layer to be used as input, which we randomize from 1 to 5. We also change the normalization of the inputs in order to match the one used in AlexNet, so the input is resized to (227,227) then subtracted of its own mean. In order to speed the process up, we also pre-compute the AlexNet activations and produce the training, validation and set from them. The remaining aspects are kept the same and we obtain the results listed in Table 3.

	Hyperparameters				**Cost**			**Accuracy (%)**		
ID	**Units**	**Function**	**Regularization**	**Layer**	**Train**	**Validation**	**Test**	**Train**	**Validation**	**Test**
1	[683]	['relu']	0.0001	5	0.0127	0.0862	0.1658	100	97.5300	94.9700
2	[491]	['relu']	0.0050	4	0.4938	0.5952	0.6913	100	98.1500	93.7100
3	[437]	['tanh']	0.0006	4	1.5161	1.5455	1.5420	45.4900	45.6800	38.3600
4	[224]	['tanh']	0.0003	5	0.0297	0.1176	0.1964	100	99.3800	93.7100
5	[320]	['relu']	0.0007	4	0.2632	0.3711	0.4657	100	97.5300	93.7100
6	[555]	['relu']	0.0020	4	0.5116	0.6645	0.8136	100	96.3000	93.0800
7	[716]	['relu']	0.0060	5	0.1614	0.1760	0.2961	100	100	96.8600
8	[265]	['tanh']	0.0007	5	0.0448	0.1088	0.1622	99.8000	98.7700	95.6000
9	[480]	['tanh']	0.0050	3	2.6280	2.6325	2.6245	30.3900	31.4800	30.8200
10	[582]	['relu']	0.0050	5	0.1414	0.1549	0.3061	100	100	96.8600
11	[416]	['relu']	0.0080	1	7.5006	7.9082	7.8604	94.5100	83.9500	84.2800
12	[381]	['tanh']	0.0007	5	0.0853	0.1740	0.2115	99.8000	97.5300	95.6000
13	[515]	['relu']	0.0040	5	0.1430	0.2362	0.3494	100	98.7700	96.8600
14	[709]	['tanh']	0.0001	5	0.0493	0.1353	0.1970	99.8000	98.1500	94.3400
15	[768]	['relu']	0.0000	5	0.0107	0.1210	0.1880	100	96.3000	92.4500
16	[465]	['tanh']	0.0030	1	2.2840	2.2906	2.2918	23.1400	26.5400	22.0100
17	[382]	['relu']	0.0050	3	0.4350	0.5010	0.6517	100	98.1500	94.3400
18	[587]	['relu']	0.0001	2	0.0870	0.2639	0.4183	100	96.9100	91.1900
19	[839]	['relu']	0.0001	5	0.0155	0.0930	0.2161	100	97.5300	93.7100
20	[648]	['tanh']	0.0040	2	2.1450	2.1650	2.1770	18.3900	16.1500	11.6000

Table 3: Hyperparameter search.

From Table 3, we find the best model according to validation accuracy, which gives us Table 4.

Hyperparameters				**Cost**			**Accuracy (%)**		
Units	**Function**	**Regularization**	**Layer**	**Train**	**Validation**	**Test**	**Train**	**Validation**	**Test**
[716]	['relu']	0.0060	5	0.1614	0.1760	0.2961	100.0000	100.0000	96.8600

Table 4: Best model from Table 3.

For this model, we have the training curves shown in Figure 1 and Figure 2.

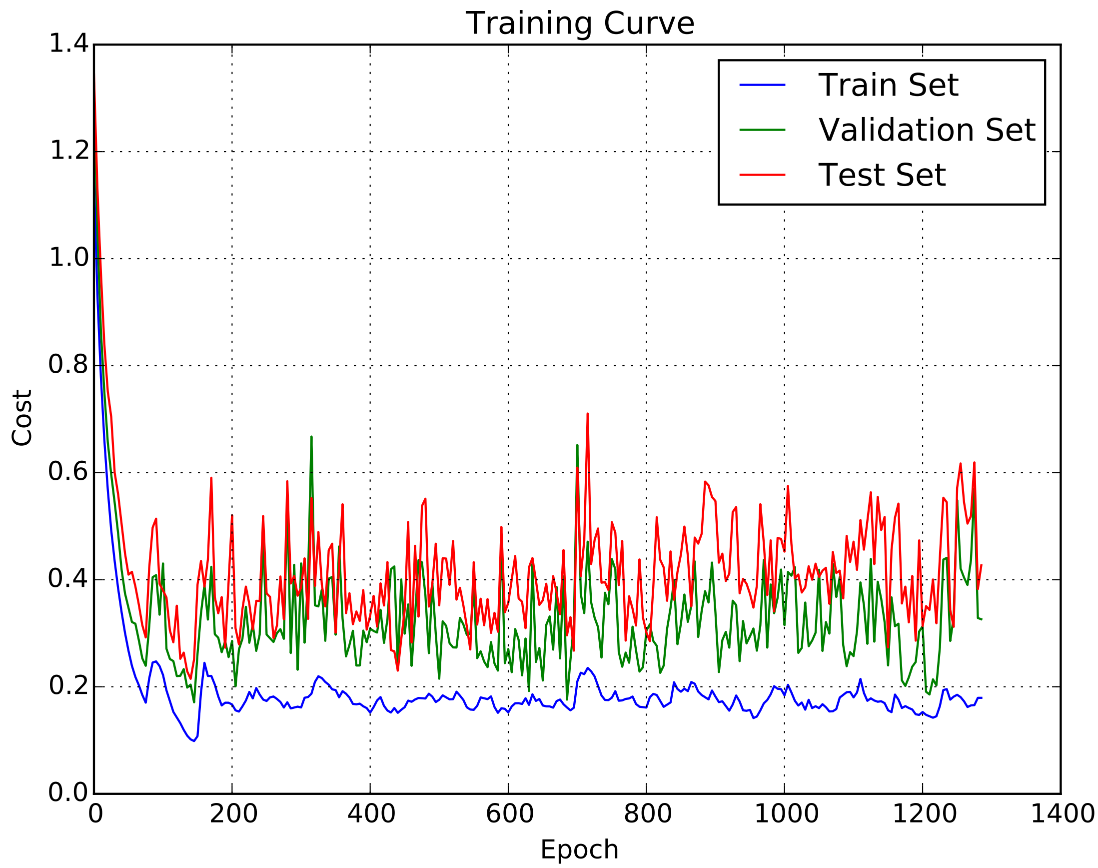

Figure 3: Cost curve for the model in Table 4.

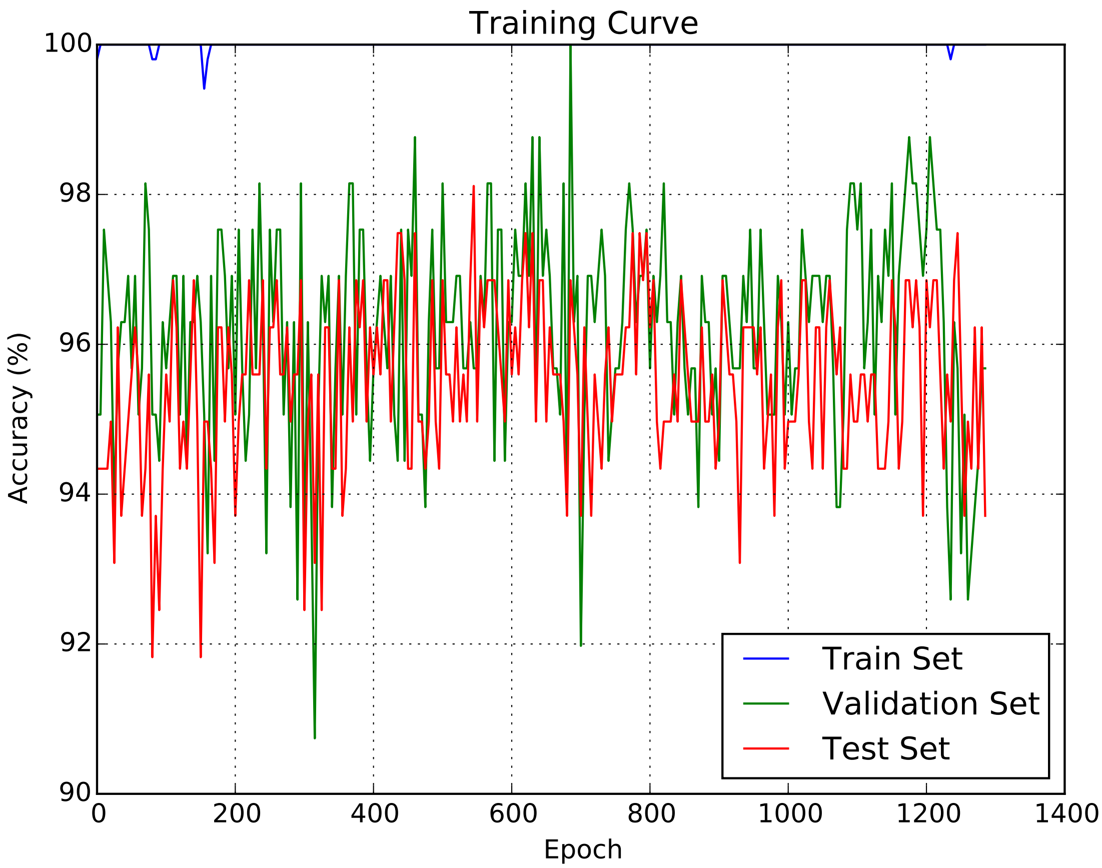

Figure 4: Accuracy curve for the model in Table 4.

Part 3

We first train two networks: One with 300 hidden units and another with 800, both using ReLU activation, dropout and a L2 regularization factor of 0.001. With these parameters we obtain models with an average accuracy of 82% and with the weights shown in Figure 5 and Figure 6.

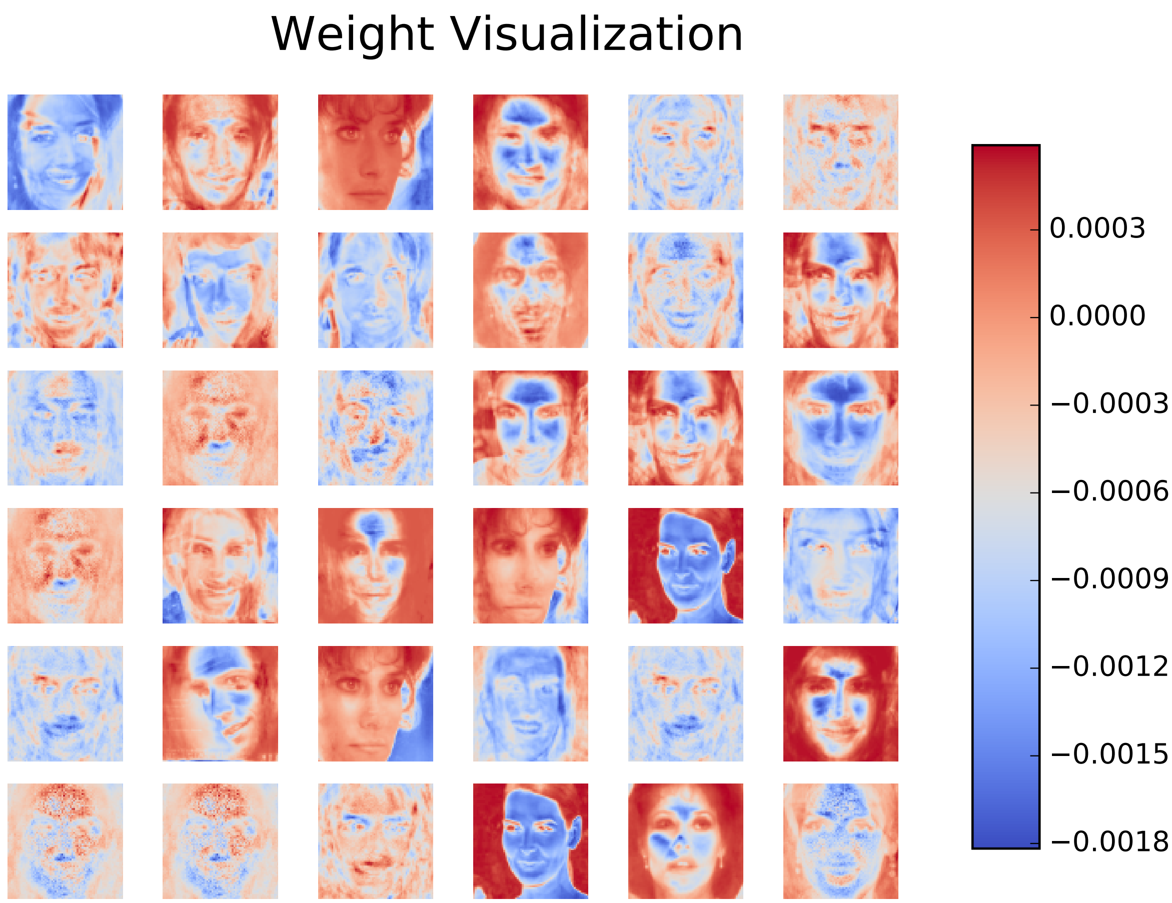

Figure 5: Sample weights for model with 300 hidden units.

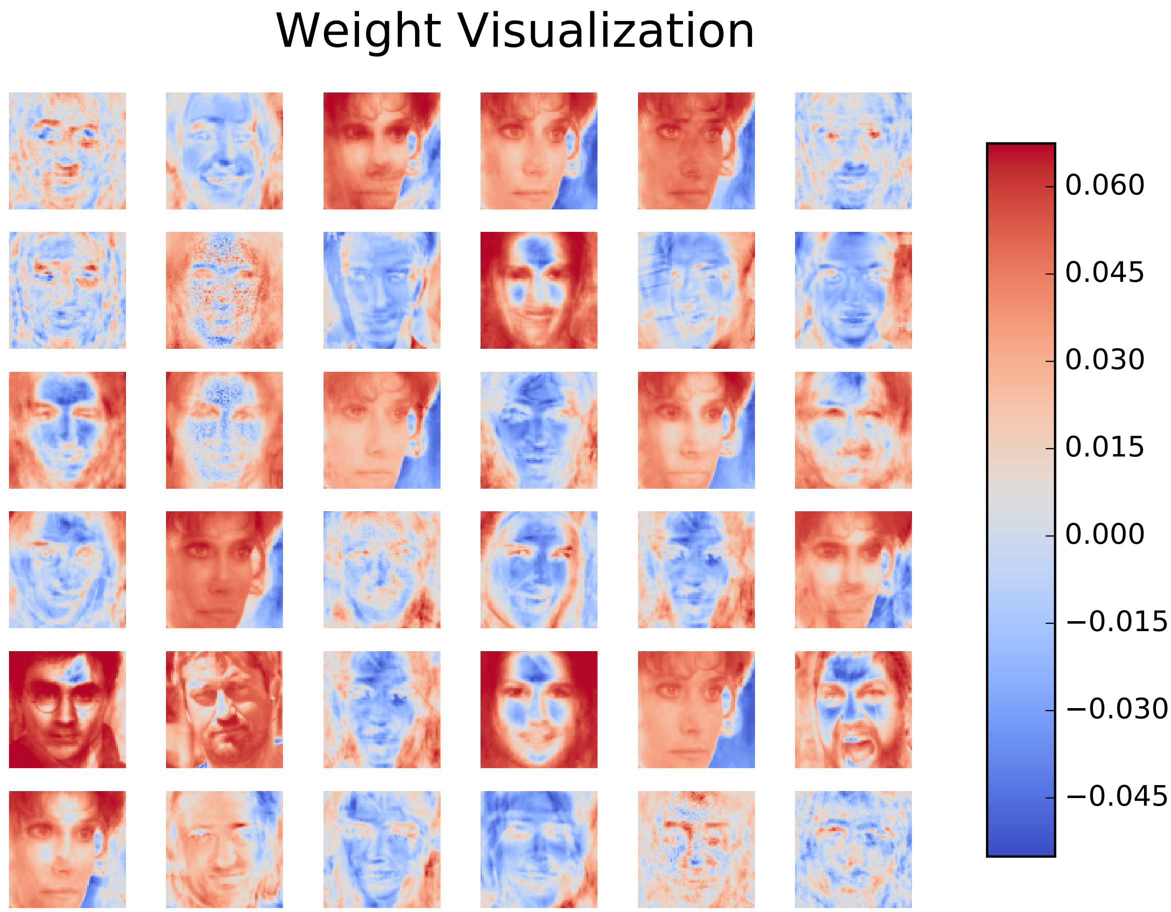

Figure 6: Sample weights for model with 800 hidden units.

From inspecting the weights we can notice that some of them contain overlays of many faces as an attempt to extract useful features. However, its also noticeable how the model still overfits to some training cases even with dropout as we can see that some units are just memorizing the inputs, and in fact even creating duplicate memories as an attempt to become resilient to dropout. This happens even more often in the bigger model, with 800 units, since it is effectively able to memorize more inputs, it could, in fact, memorize the entire training set and achieve 100% training accuracy.

Part 4

In order to make a unified system, we copy over the AlexNet architecture and plug in one of our previously trained densely connected networks (from Part 2) in the correct AlexNet layer. We can then feed the set of face images directly to it and get the relevant outputs (probabilities and classes)

Code 1: Using the system.

```
'''faces is a list of numpy images ans ff_params are the parameters of the densely
 connected layers, in the following order [w_in, b_in, w_out, b_out, alex_net_layer].

It returns two lists, "outputs" containing the softmax outputs and "classes"
 with the most likely class for each image'''

outputs, classes = eval_alex_net(faces, ff_params)
```

With the outputs of the system we produce Figure 7. For each image it displays the two classes with higher softmax probabilities (in parenthesis), texts in green means it got the correct class, while red means it misclassified.

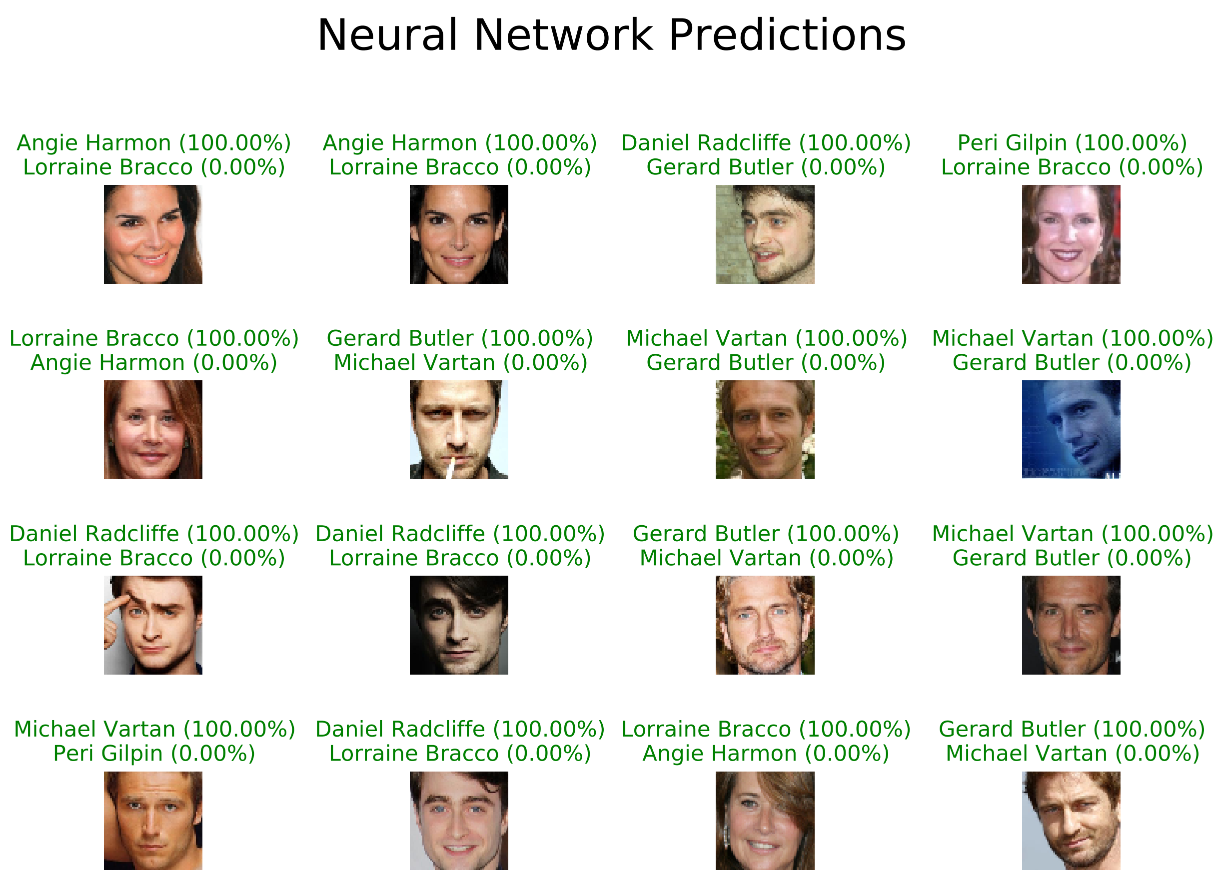

Figure 7: Predictions from the neural network.

Part 5

To produce the gradient for a specific input image we use the system developed in Part 4 and TensorFlow's built in function *tf.gradients()* to calculate the gradient of the highest softmax output with respect to the input. From this we get the results shown in Figure 8.

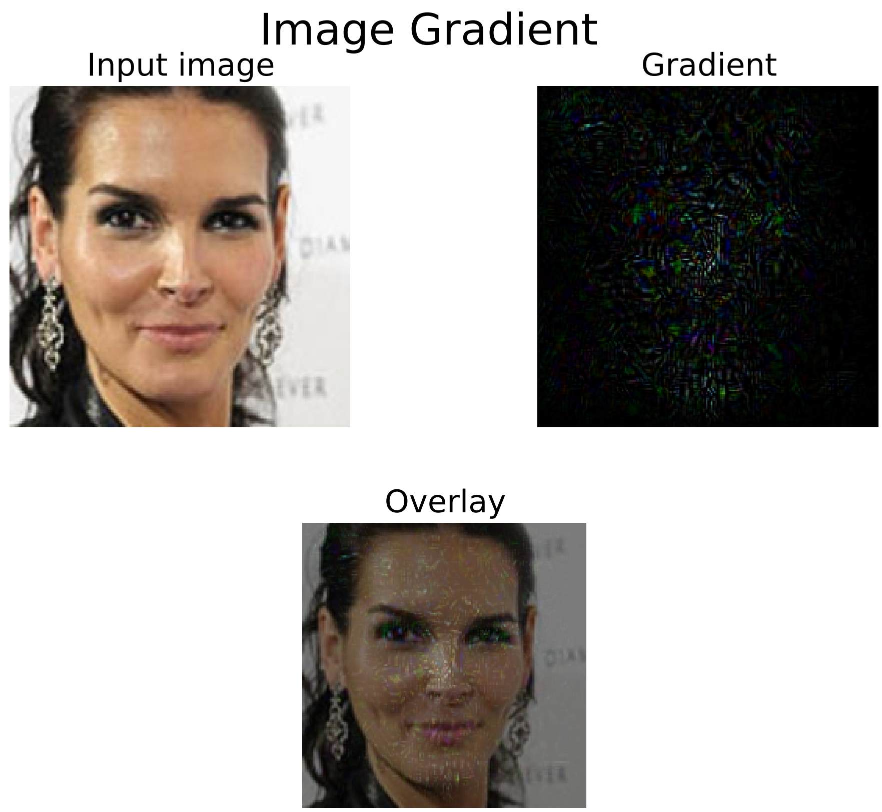

Figure 8: Gradient of the highest output probability with respect to the input.

For this picture of Angie Harmon we notice that the shape of the eye and eyebrows, the pointy nose and pronounced cheeks are strong indications for the classification. In Part 7 we will be able to confirm our hypothesis using guided backpropagation, which gives a clearer visualization.

Part 6

In order to train AlexNet to recognize our set of artists, we now train the entire network and, differently from Part 2, don't pre-compute the activations and now back propagate the error gradient throughout the entire network. To speed the process up, we initialize AlexNet with the provided weights and the densely connected layers with the best topology according to the findings of Part 2, however we reinitialize their values with random values sampled from $\mathcal{N}(0.0001, 0.0001)$, since otherwise we would already start the training at a local minimum and would be unlikely to find a better model. We also add regularization to all the network weights and dropout after the pooling layers and hidden densely connected layer in order to avoid overfitting.

With these parameters we obtain the performance listed in Table 5, with the training curves depicted in Figure 9 and Figure 10. We notice that the training accuracy rapidly converges to 100% and from this point forward the model just works to generalize itself in order to compensate for dropout and regularization. The final model is not better than the absolute best found in Part 2, however its performance is comparable to that of the best ranked models.

Cost			**Accuracy (%)**		
Train	**Validation**	**Test**	**Train**	**Validation**	**Test**
0.0480	0.1900	0.3226	100.0000	97.1429	94.5900

Table 5: Model performance.

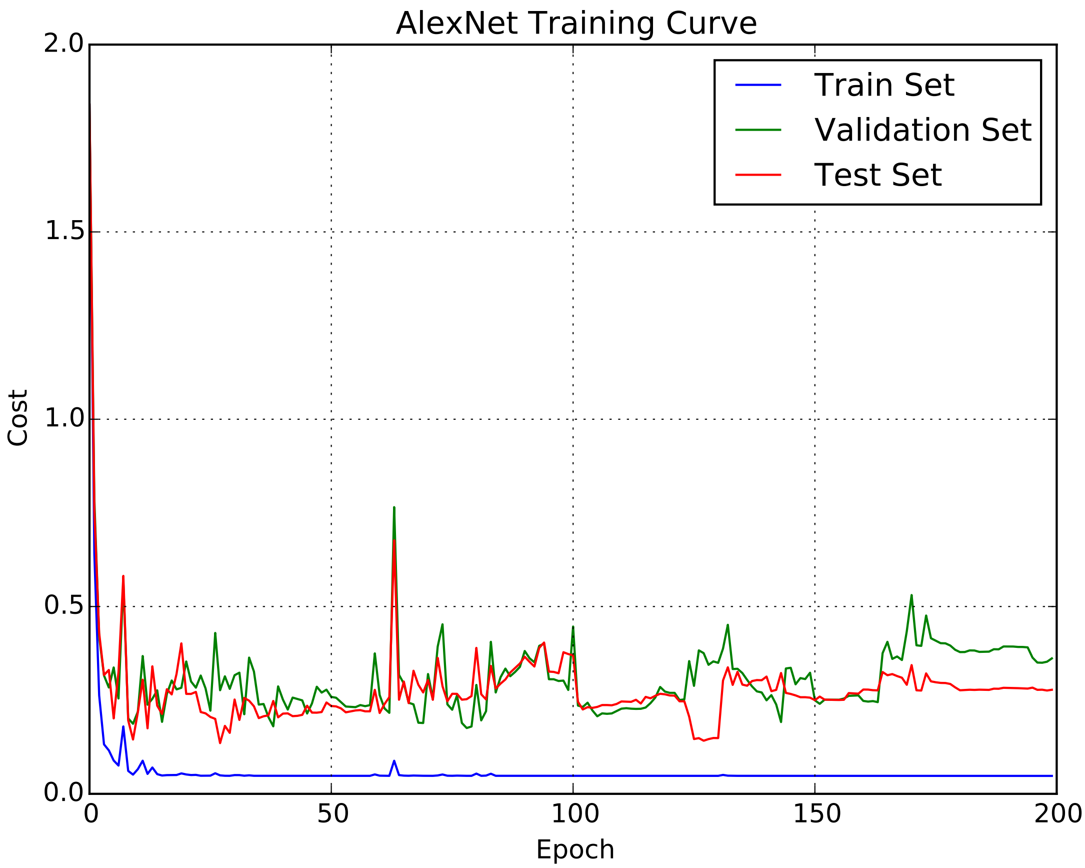

Figure 9: Cost curve for the model discussed.

Figure 10: Accuracy curve for the model discussed.

Part 7

For guided backpropagation we follow a procedure similar to the one discussed in Part 5. However, we now have to zero out negative gradients, to do so we take in the fact that our network only has ReLU units, therefore we can change it's gradient to a more convenient calculation using TensorFlow's *@tf.RegisterGradient()* function decorator and then map the ReLU units in the network graph to our custom ReLU.

ReLU's gradient is, by default $\frac{\partial}{\partial x}f(x) = [x > 0]$, so during backpropagation it effectively propagates the gradient if it's input is positive, or $\delta_i = \delta_{i-1}[x > 0]$. We add another indicator function to the gradient, so it only propagates previous positive gradients, or $\delta_i = \delta_{i-1}[x > 0][\delta_{i-1} > 0]$. The final TensorFlow implementation is listed in Code 2.

With this method we obtain the outputs depicted in Figure 11. We can see that the gradient is higher around the eyes, nose and cheekbones. Thus confirming our hypothesis from Part 5.

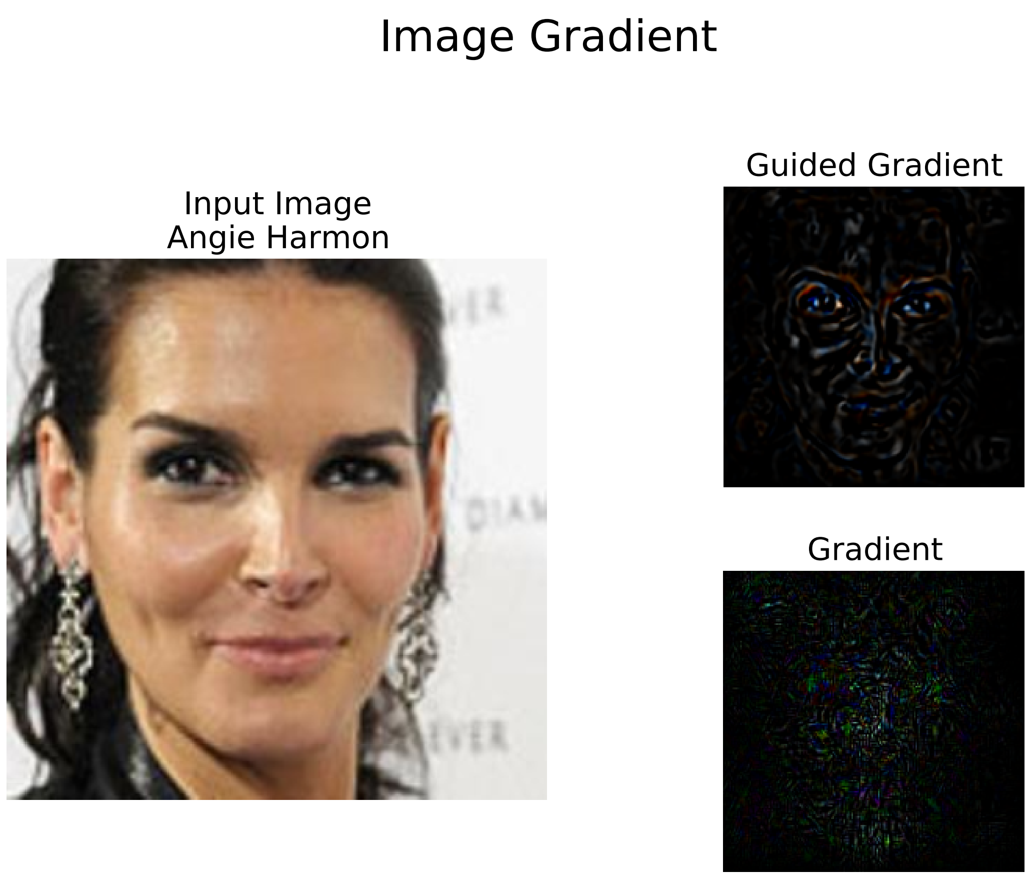

Figure 11: Gradients obtained with and without guided backpropagation.

Code 2: Custom ReLU Gradient.

```
@tf.RegisterGradient("CustomRelu")
def _custom_relu(op, grad):
  zero = tf.Variable(0.)
  relu = op.outputs[0]
  relub0 = tf.cast(tf.greater(relu, zero), tf.float32)
  gradb0 = tf.cast(tf.greater(grad, zero), tf.float32)
  return relub0 * gradb0 * grad
```