Lilli Lewis
Lech Syzmanski - COSC420 Deep Learning
April 15

Assignment 1 - Oxford Flowers

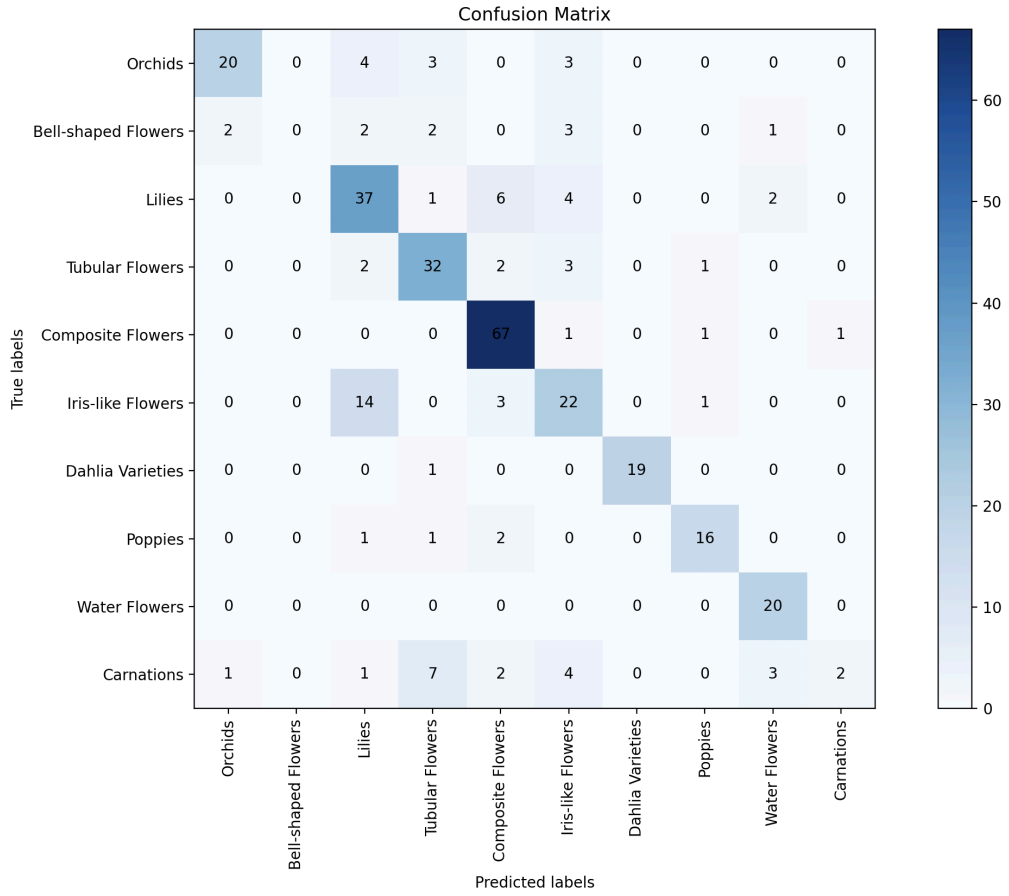
Part 1

Through heaps of trial and error, I came to the following architecture for my CNN: two 32 filter convolutional layers followed by a batch normalizing layer and a max pooling layer. I repeated this process twice, increasing the filters to 64 and 128 for the next two batches. In each convolutional layer, the kernel size is 3x3, the strides are 1x1, the activation is ReLU, the input shape is 96, 96, 3, and the padding is “same”. After these first three batches (12 layers total), I add a flattening layer, and then I add a dense ReLU layer with 512 neurons with a regularizer of .2. I add a dropout layer of .2, then add another dense ReLU layer with 512 neurons with a regularizer of .2. Then I have another dense layer with the same number of neurons as classes (softmax instead of ReLU) before my final dropout layer of .2. I tried a multitude of combinations for number of convolutional layers, number of filters and order in which to increase them, where to put the pooling layers, how high of a regularizing value to use, how high of a dropout value to use, etc., and found that this combination brought me the most success. I use an Adam optimiser with a learning rate of .0001, a batch size of 32, and 300 epochs. Before running the model, I explicitly normalize my data by dividing the images (as type float) by 255, making all of their values between 0 and 1. I tried addressing class imbalances by randomly replicating images from the classes with fewer data points in the training set so that by the time the model begins running, each class in the training dataset has 350 images. However, this decreased my accuracy.

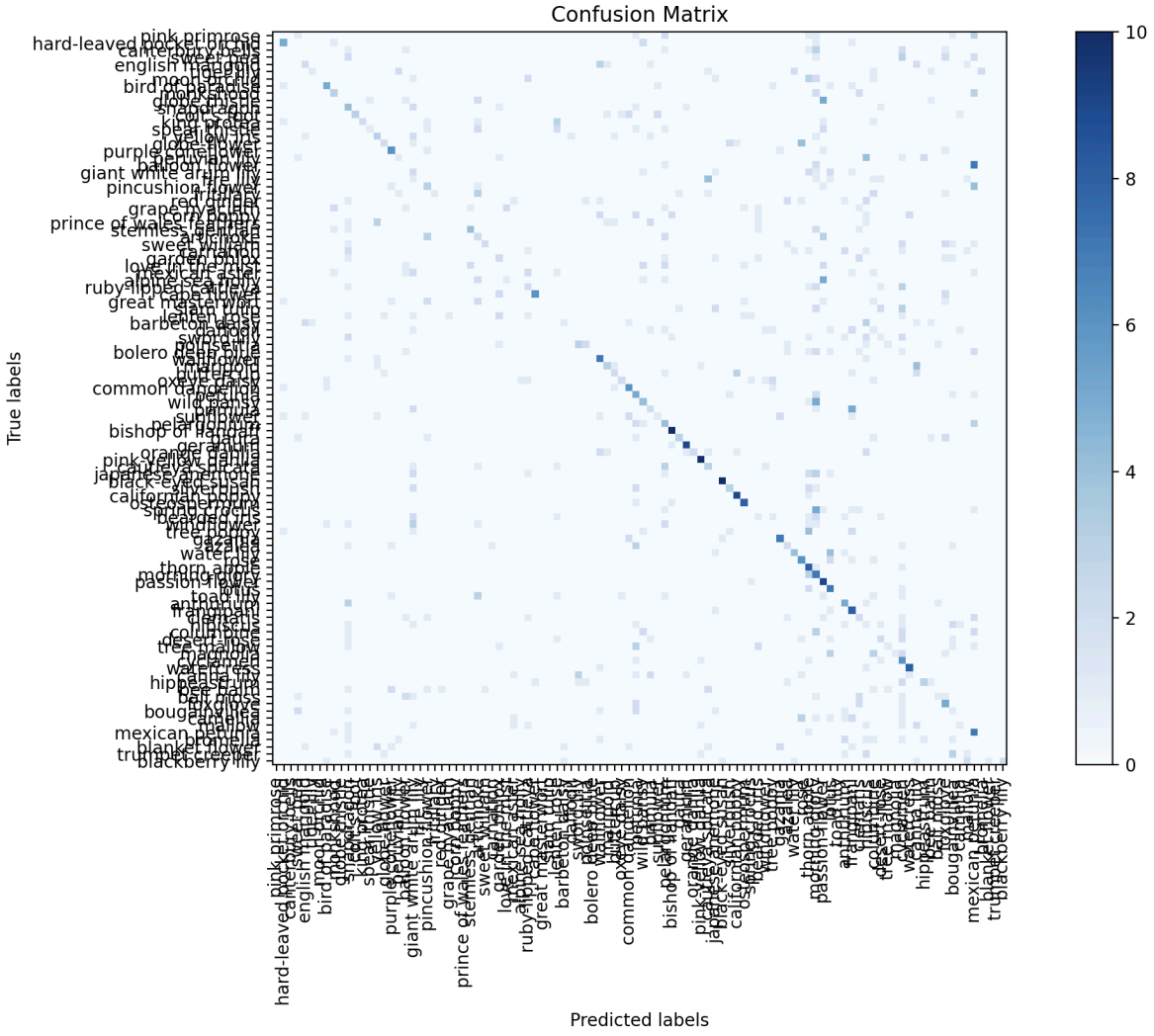
	With class imbalances	**With balanced data**
Train accuracy	.99	.85
Validation accuracy	.70	.59
Test accuracy	.73	.63

Since my computer is not powerful enough to run the epochs at 10,000 within the scope of this assignment, I don’t know which model is better. I’m inclined to say the model with balanced data is better since there is less overfitting, however, I’ve included and am doing the analysis with the model with the class imbalances since its accuracy is 10% higher. Here is the

confusion matrix for the model with 73% accuracy on the coarse data:



And here is a confusion matrix for the fine data:



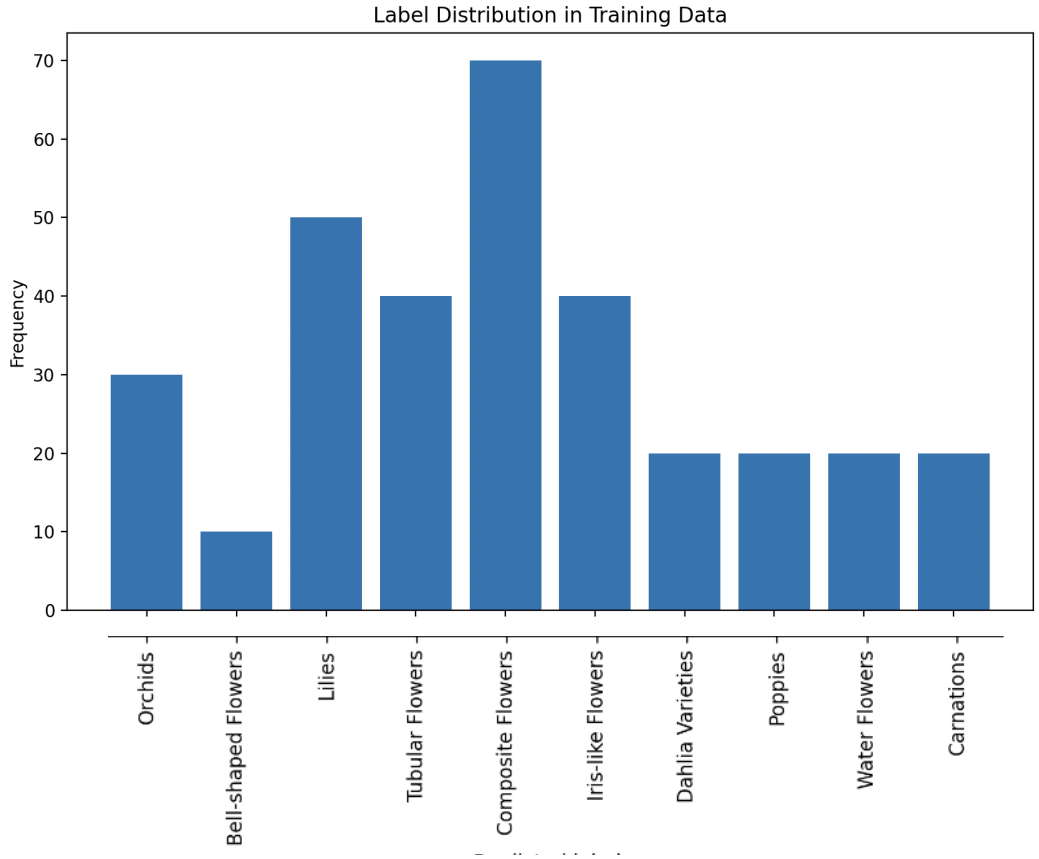
(I also provided a print out table in my fine file for this matrix, since I didn't figure you'd appreciate a confusion matrix table of 102 by 102 in this document).
Here are the accuracies between the two:

Accuracy	**Coarse**	**Fine**
Training	.99	.85
Validation	.70	.26
Testing	.73	.27

As you can tell, the model was much less successful on the fine data. I am relatively unsurprised: the model was designed for a much simpler task. If I had trained and designed the model on the finer data, I'm sure I could've gotten a better accuracy. I would guess that this other model would have probably had more convolutional layers, a much higher number of

epochs, and many more filters. With the resources I have available and the task at hand, though, I think I did a good job.

I found any batch size other than 32 to result in a lower accuracy. I tested on batch sizes 8, 16, 32, and 64. I found the same for any learning rate beside .0001, and I tested on rates .00001, .00005, .0001, .0005, .001, and .005. I tried all sorts of combinations of convoutional layers, switching around the number of filters, the number of convolutional layers, and the order in which I placed them. I tried adding dropout layers and tried dropouts of .1-.5, and I tried dense layers of varying filter sizes and varying regularizing weights. The model I ended up with was best suited to the task at hand, so finding out that it was not well suited to a much more complex task is unsurprising



This is a histogram of the training data distribution. It is unsurprising to see that the model predicted 0 of the bell-shaped correctly, given that it had fewer than 10 examples to train on. It makes sense that the model performed well on the composite flowers, given that it had the most examples of composite flowers to train on. As I mentioned, though, balancing the data did not help the accuracy within the number of epochs my computer (and I) had time to run.

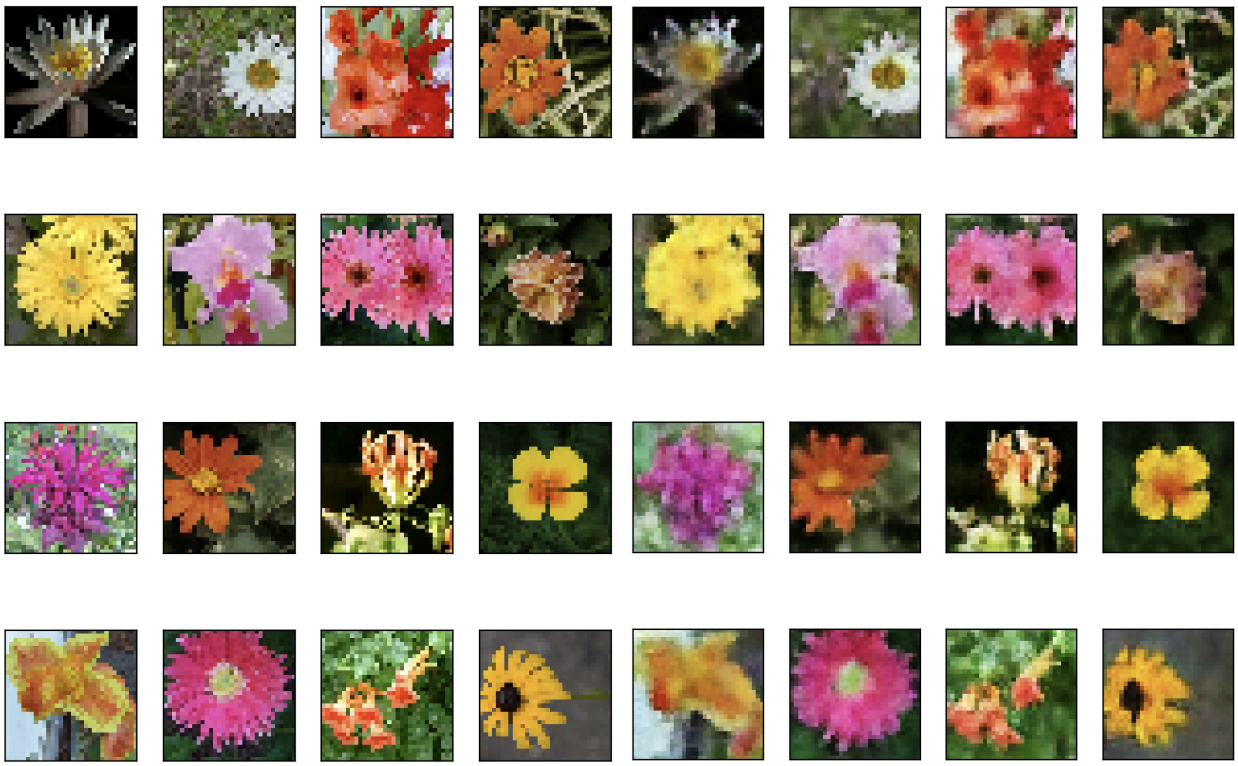
Part 2a

For this portion of the task, I began by recreating my model from part 1 in the encoder section of the autoencoder, then using a combination of convolutional and max pooling layers in my decoder. I found, however, that if I removed the dense, fully connected layers from the encoder and the max pooling layers in the decoder, I got much more accuracy. So, the encoder

is, similarly to my CNN, three sets of: two 2D convolutional layers, one batch normalizing layer, and one max pooling layer. Their filter numbers are 32, 64, and 128, just like in my model, and every single convolutional layer has a 3x3 window, same padding, and a ReLU activation function. In the decoder, there are three convolutional layers following the same criterion as above, with 32, 128, and 512 filters each (in that order). The last layer in the decoder is also a convolutional layer with 3 filters, all of the same specifications as usual, and a sigmoid activation function. I've found the most success with 100 epochs, a batch size of 32, a learning rate of .001, and an Adam optimiser. I used the same data normalizing code as I did before running my CNN, and the same program structure from example 3 in terms of if statements, saving files, etc. I recycled a lot of code between example 3 and both tasks, so don't be alarmed to see that. The main differences between my model from task 1 and my autoencoder is that my autoencoder does not have dropout layers or fully connected dense layers. I found that removing them helped improve my model's accuracy.

Here are 16 photographic examples of my input and output images from the autoencoder:

Input: Output:



Here are the mean error per pixel and standard deviation of error per pixel from my best run:

Mean error per pixel:	.07319
Standard deviation of error per pixel:	.08033

And from the last run that saved the file (same model):

Mean error per pixel:	.07708

Standard deviation of error per pixel:	.08424

And the 15 graphic side by sides from the saved run:

Input: Output:

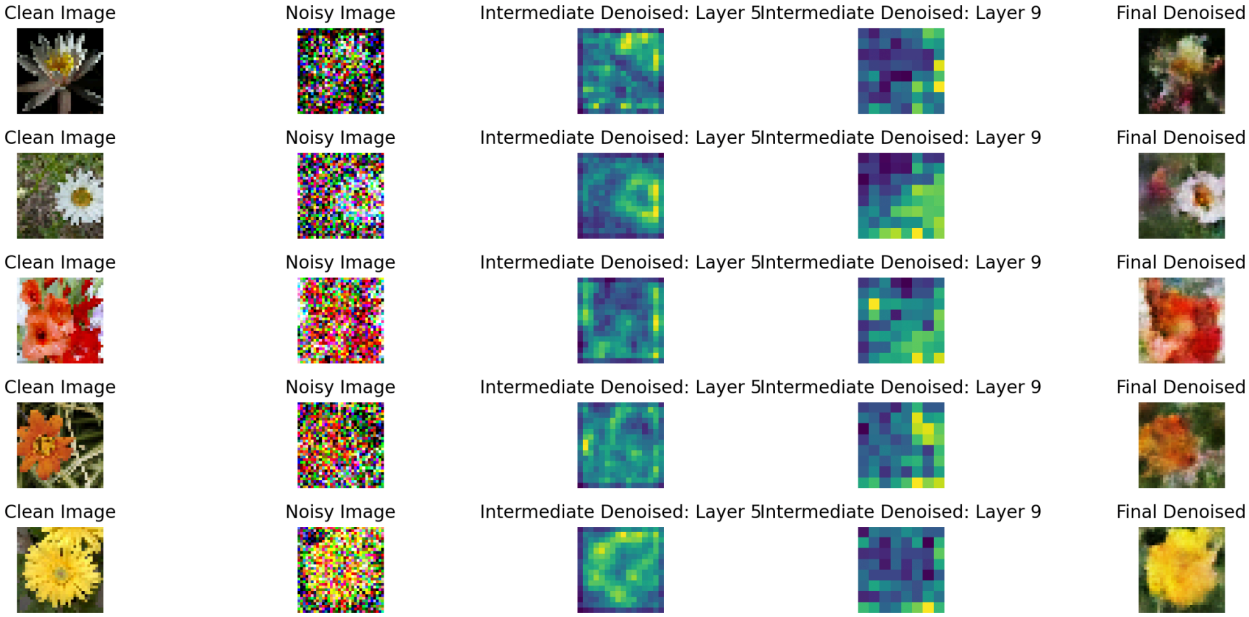
I include both runs because the best run was slightly better.

I think that to improve this autoencoder, I could run it with more epochs. I tried it with 300, which led to a lower accuracy than 100, but I wonder if doing a higher number such as 10,000 would improve the model's accuracy. I tried all sorts of combinations of filter numbers and orders and number of convlutional layers, in both the encoder and decoder. I tried learning rates of .0001, .0005 .001, and .005, batch sizes of 8, 16, 32, and 64, and epochs of 30-300. The model's values that I currently have were best suited to the task within the epochs my computer (and I) was willing to run. I also probably could have improved this autoencoder and my CNN by using the 500x500 images, but they made the models run so slowly I couldn't have feasibly used them on my computer. I was able to use 96x96 for the CNN but only 32x32 for the rest.

Part 2b

I made a separate file for this task, just for clarity for myself. I replicated my autocoder's architecture exactly, put in the appropriate noise-adding code (using a Gaussian noise matrix), and trained my diffusion model on pairs from the noisy and clean test and validation data. The intermediate photos included below are so colored because they are averages across the channels of the images during layers 5 and 9, before the images are the correct shape for presentation. I thus had to use a colormap to represent them, and I chose viridian as the color scheme because it was dissimilar to the colors in the images.

Results of 5 images being passed through the diffusion model:

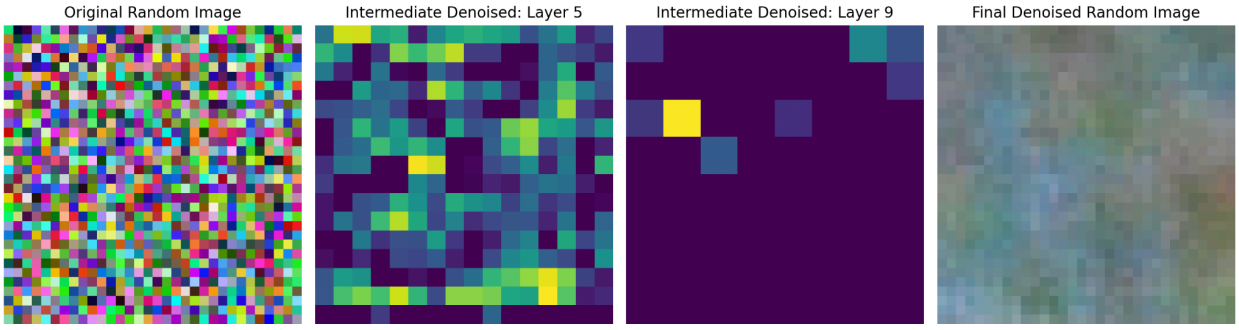


Mean error per pixel and standard deviation of error per pixel of final denoised images:

Mean error per pixel:	0.12371
Standard deviation of error per pixel:	0.12119

I think the model did a reasonably good job, given that I did not adjust it at all from the autoencoder from part 2a.

Here is the model working on a random image I generated:



And the errors:

Mean error per pixel:	0.24891
Standard deviation of error per pixel:	0.14843

As expected, the model just produced a blurry blob. I chose layers 5 and 9 to demonstrate the intermediate denoising phases because my model has 16 layers, so I found those to be the thirds(ish) of the stage. The output suggests that the program can generate something, even if it's just a few blurs of color.

In this entire project, I recycled lots of structure from example 3. I also used chat GPT for tasks such as graphic production (ex: to see class distribution or to print the image above) or generating a Keras line so I could see the proper format, but the models' structures were entirely mine. I would've used it less if I'd known more Python- I'm still learning, I started in January, and in fact this project is the first time I've ever used chat GPT! This project was definitely a good way to learn/practice some Python, though. I learned/used some of the code present within the project from Stack Overflow, and some of it from The Keras Blog. I tried my best to denote in the comments which code was mine and which was borrowed, but I want you to know that I wasn't just copy-pasting, I was careful and tested everything I used thoroughly before deciding whether or not to implement any changes. Lastly, I want to thank you, Lech, for all of your patience and assistance during this project. You have been monumntally supportive and helpful, and I really appreciate it!

Model names key (it's so self explanatory but I don't know if you require this or not):

Pt 1 (coarse)	Model	oxford_flowers102_pt1_coarse_cnn_net.h5
	History	oxford_flowers102_pt1_coarse_cnn_net.hist
Pt 1 (fine)	Model	oxford_flowers102_pt1_fine_cnn_net.h5
	History	oxford_flowers102_pt1_fine_cnn_net.hist
Pt 2 A	Model	oxford_flowers102_2a_cnn_net.h5
	History	oxford_flowers102_2a_cnn_net.hist
Pt 2 B	Model	oxford_flowers102_2b_cnn_net.h5
	History	oxford_flowers102_2b_cnn_net.hist