Neural Network Classification of Double-Digit Images

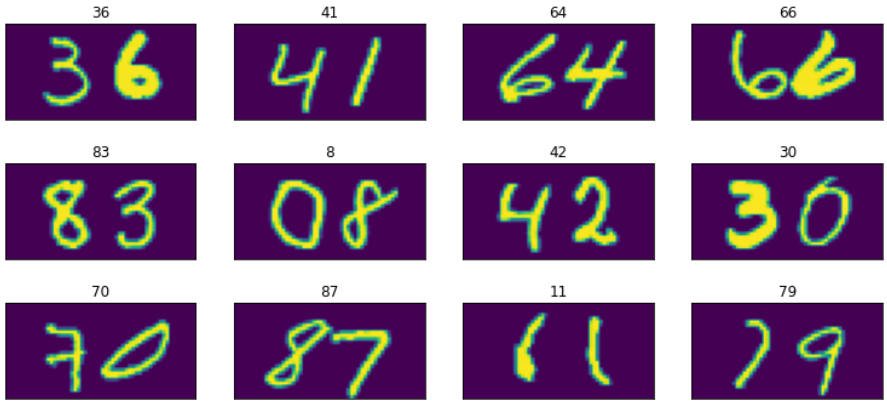
Udacity Machine Learning Engineer Nanodegree: Final Project

I. Introduction

Purpose

The purpose of this project is to compare the performance of different neural network models on an image recognition task, under several levels of difficulty. The task will be to categorize images of handwritten double-digit numbers (00, 99) constructed from the classic single-digit MNIST database[1] (examples shown).

Four models each of five neural network designs will be implemented for training on each of the four categories of "noiseiness", for a total of 20 trained models to evaluate. Each will be evaluated on sets of test images; one set for each noise condition. The final results will then be a total of 80 accuracy measurements.



The overall goal of the project is to determine how well each model design performs for each combination of training and testing noise-level conditions. Specifically, it is to identify under which of these combinations model-types can meet or exceed the benchmark for accuracy at 95%.

Questions and Expectations

The questions to be directly answered by this project are:

- Can the "minimum" model design (with just three layers: Input, Flatten, Output) do any better than by chance alone (1%)? Even on the "no-noise" datasets? (Expectation: no; doubtfully.)
- How did the DNN algorithms compare to CNNs, across conditions? (Expectation: worse.)
- Does the comboNN design do any better than the CNN? (Expectation: maybe.)
- How much does training under a given condition affect accuracy on the other conditions? (Expectation: a lot.)
- How well will the "over-complicated" ocNN models perform, compared to the other four algorithm design structures? (Expectation: better than miniNN, worse than the others; and more buggy, or at least more resource-greedy.)

[1] Official MNIST repository

II. Dataset Construction

Double-Digit Images

The data used for this evaluation will be derived from the canonical MNIST single-digit data.[2] Tensorflow[3] provides this data with a convenient-to-access built-in function, consisting of two sets of pixel arrays, 60,000 for training and 10,000 for testing. Both sets come from the same statistical distribution, according to the dataset designers. Each element is a 28x28 array of numbers ranging (0, 255) to be interpreted as values on a single greyscale color channel. These values represent handwritten images of a single digit, (0, 9). Paired with each image array is its corresponding label array with values (0,9) to indicate which digit the images are meant to represent.

From this raw dataset of examples-and-labels, a new dataset of double-digit images, along with their labels, (0,99).

To do so, I randomly selected a single-digit example to be the left-hand tens digit, and separately selected a random digit to be the right-hand ones digit. I then simply added these two arrays to each side of an empty array of 28x56 pixels.

First I made sure to normalize each single digit image array from (0,255) down to (0,1), and it was important to do this before adding them to the new array, because each one had a different distribution of values (different means and standard deviations). Once normalized, they all had comparable brightness levels and ranges of grayscale values, and looked natural when combined (examples shown in Figure 1).

There was some occasional overlap between the two digits, and since I was adding the two smaller arrays together into the large one, I also had to be careful to clip the resulting values at 1.0, and this may have altered the original distributions very slightly. Labels for these new examples were calculated as 10 times the tens digit plus the ones digit, so were in the range (0,99).

I would use 100,000 of these new arrays (now normalized to the range (0, 1)), as a training set; another 10,000 image arrays on which to validate while training (ie. to be measured between epochs); and then another set of 20,000 double-digit images to be used for the final evaluation after training.

Drawing from the original 60,000 MNIST training images, reducing the likelihood of an algorithm seeing the same double-digit image twice during training, at least sufficiently to help keep models from overfitting to the training dataset.

The validation and evaluation double-digit sets were separately drawn at random from the original test-set of 10,000 single-digit images provided by MNIST. (This was to ensure neither digit of an image in the test set had also been present in the training set.)

---

[2] Tensorflow Dataset Catalogue

[3] MNIST Dataset Customization (specifically: `mnist.load_data()`)

Noise Conditions

Next, all of these images will have three different noise conditions applied to them: low noise, high noise, and variable noise. The "noise" was given random pixel values drawn from a normal distribution centered on the mean of the double-digit pixel values, with variation expressed in relation to the standard deviation of those original values. The low noise category could then be defined as 0.5 times the image SD. High noise was 1.5 times the SD. The variable category ranged in values from 0 to 1.5 times the SD, with no noise being zero noise added. (Note: at about twice the SD, images were noisy enough to be consistently indiscernible to a typical human test subject).

The remaining difficulty in implementing the dataset construction is that all the images get normalized *twice*. The first normalization has to occur before the single-digit images are combined, because the distribution of pixel values differs greatly from one image array to the next. Then, they have to be normalized again after being combined, because the addition of noise de-normalizes them again. There is surely a better way to do this, but it will have to wait to be implemented in future investigations.

The specific algorithms used in these constructions can be found in the project notebook, in the section entitled "Construct the Data".

Preprocessing

The MNIST dataset is much-used and well-tested. Most of the heavy statistical preprocessing and normalization has already been done, and it is just a matter of maintaining the statistical distributions of that raw data for the purpose of this project.

Preprocessing:

Normalized pixel values, from (0, 255) to (0, 1), so that images with different distributions of pixel values can be meaningfully compared with each other. Values were scaled to the maximum value minus the minimum value for each image array, instead of to 0 and 255, for this reason, ie. to combine single-digit images with different levels of brightness so they do not seem obviously different when put together.

As usual with this dataset, a single grayscale channel dimension was added to the image array before presenting them to the algorithms for training.

No pre-reshaping was done for the minimal and dense model algorithms. The image input layer shared by all models was simply immediately flattened to a dense layer with 28*56 neurons.

Metrics

The metric for model performance will be *accuracy*, defined as the number of correct predictions made divided by the total number of predictions made by the model during testing.

Baselines

The *qualifying* or *threshold* baseline will be the accuracy expected for a model by chance alone; ie. its expected score if it were guessing uniformly at random, which in this case would be 0.01,

or 1% (given the 100 possible answers, (0, 99)). Any functioning model with an accuracy above this bare minimum will be considered an acceptable algorithm, and worth comparing with the other models. If it can not achieve at least this level of accuracy, a redesign of that model type will have to be considered before it can be meaningfully compared to the other models and conditions.

The *successful* baseline of a model's performance will be set at 95% accuracy for this double-digit task. Any model performing at better than 95% will be considered a successful model. This is an arbitrary number and could be set differently. This number seems a good rough baseline for evaluating performance on this task, given that accuracies above 98% are easily achievable on the original ("no-noise"), single-digit dataset. (See *Benchmarks* section).

Another baseline for comparison will be the set of five algorithms trained on images without any added noisiness. The performance of these trained models on the no-noise validation set will produce five accuracy measurements, which will then serve as a baseline for comparison with the other combinations of training/testing conditions, and with each other. This key baseline criterium might also reflect on the question of how well each variety of model performs on discrimination of no-noise double-digit images, compared to single-digits. (It would be interesting to run some comparisons between models' performances on single- and double-digit images, but that will be outside the scope of the current project).

Other metrics and baselines for comparison between models and conditions could be considered; for example, the training epochs required by each model-type to reach its asymptotic limit, or how the number of a model's trainable parameters affects its performance. However, these other metrics lie mostly outside the scope of this current project.

Benchmarks

There are five different model types, so each type will require a benchmark model for comparison of type performances between noise conditions. These benchmarks will be a model of each type trained on the original no-noise images. Their performance on the no-noise test data will then be compared to that of models of the same type but trained on the other noise conditions.

The baseline for noise-levels will be the no-noise condition.

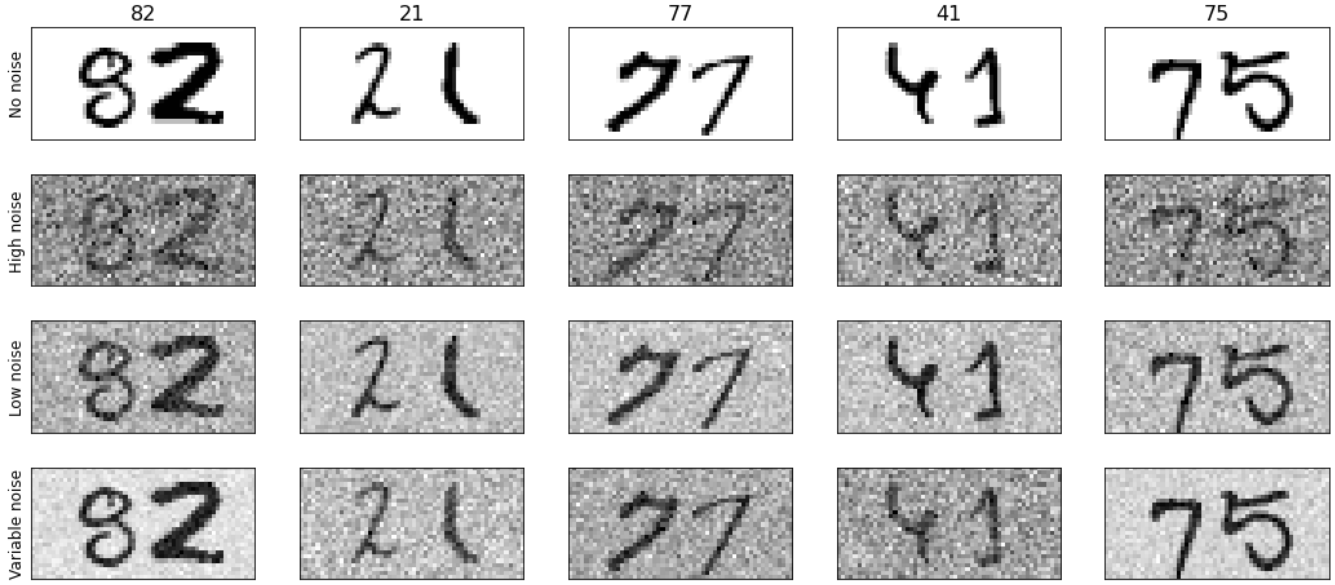
The benchmark for accuracy across all model types and noise conditions will be 95%. That seems the most reasonable benchmark to use after consideration of typical accuracy scores for single-digit images, which is routinely above 99%.

Dataset Analysis

Sample Set

The first row is a random sample of the original double-digit images, before adding any "noise" to them. The next three rows show the application to these images of the low, high, and variable noise levels (in that order). In the variable-noise dataset, each image had a different degree of noise added, ranging anywhere from *no* noise to *high* noise levels.

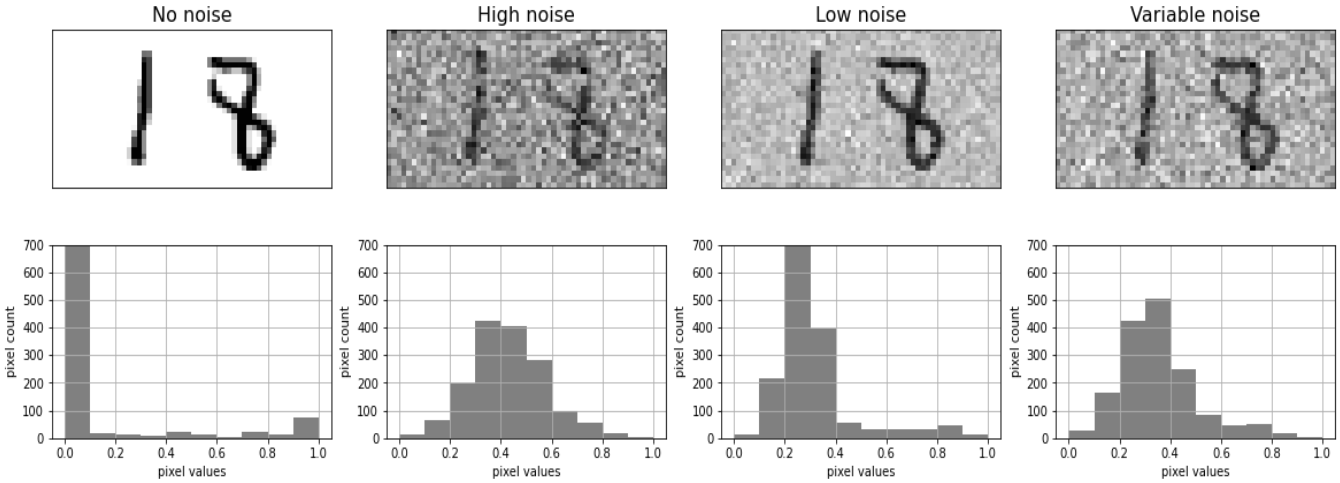
Figure 2. Samples of images with *no*, *low*, *high*, and *variable* noise added
(In that order, by row. The first row is from the baseline dataset: no added noise.)



Distribution and Normalization

As is discussed further in the project notebook, some scaling and normalization was required for all but the no-noise image sets. Care was taken to maintain suitable noise distributions, as visualized in Figure 3.

Figure 3. Distribution of pixel values under each noise condition



k-Fold Cross Validation

This data would be well-suited to a *k*-fold cross validation[4] analysis. However it was not possible to implement that kind of analysis within the constraints of this project.
For this project, formal statistical analysis will be minimal, and comparisons should be considered only approximately descriptive.

III. Methodology

Variables and Hyperparameters

To improve comparison between models, they all shared some common features. One input layer was defined for use by all the models, for example. Every model used standard ReLU activation for the layers above the logits layer. The logits layers all used softmax activation.

All the hyperparameters and global variables were set the same for every training session and every model. These were:

- Batch size/Val-batch size: 500/100
- Optimizer: RMSprop
- Learning rate: 0.001
- Size of training and test sets: 100000 to train, 30000 to test
- Epochs: 10
- Image size: (28,56)

[4] "*How to Develop a CNN for MNIST Handwritten Digit Classification*"; Jason Brownlee, **Deep Learning for Computer Vision**, May 8, 2019

Table 1. Trainable Parameters and Number of Layers for each model type

Model Type	Trainable parameters	Number of Layers
miniNN	156,900	3
CNN	236,730	7
DNN	1,016,527	7
comboNN	5,127,420	7
ocNN	16,941,577	31

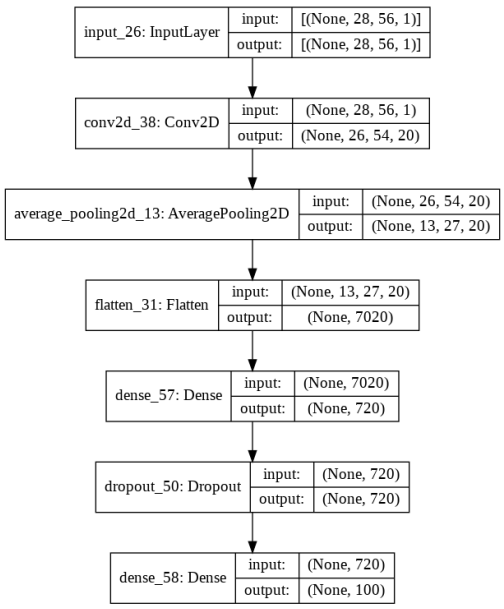
Implementation

Algorithms and Model Designs

The models were implemented in python with the Keras *Functional API*[5] on the TensorFlow[6] platform. The code developed for this implementation can be found in the Colab notebook included with this project, in the section "Build and Train the Models".

Five kinds of neural network models will be designed and trained on the data, and then compared on the basis of their post-training performance on new, unseen data. The difficulty levels will be defined by four degrees of random "noise" added to the images: no noise, low noise, high noise, and variable noise. Each neural network will be trained on each noise condition, and then subsequently evaluated on each noise level so that the differences between models can be compared.

Diagram 1. Layer structure for comboNN model design



Five different model designs will be implemented: a minimal model ("miniNN"), essentially just a layer for input and a layer for output; a convolutional neural network ("CNN"), with only convolutional and pooling layers; a densely connected network ("DNN"), with only dense and dropout layers; a hybrid model made of both convolutional and dense layers ("comboNN")[7]; and finally an "overcomplicated" model, with unnecessarily many dense and convolutional layers, dividing apart and concatenating together again along a superfluously complicated scheme of information-flow ("ocNN").

Four models each of these five design-types will be trained on each of the four categories of noiseiness, for a total of 20 trained models. Each of these trained models will then be tested on four sets of validation images, one for each noise condition, to yield a total of 80 accuracy measurement results to record and analyze.

[5] the *Functional API* at Keras
[6] the TensorFlow platform
[7] Example shown; see Appendix A for the full set of model summaries and diagrams.

The python implementation of the neural network algorithms can be found in the project notebook under the heading, “Build and Train the Models”.

Obstacles and Adjustments

Only the DNN model had difficulty rising above the 1% accuracy to be expected by chance alone, under the training conditions. It seemed unable, through many runs, to learn any meaningful patterns under the high and the variable noise conditions. (The results reported here are from a typical run.) It is not yet clear how to explain these spurious results. The fact that this did not *always* occur suggests that it is not a fundamental problem with model design, or with the data itself. Trying different hyperparameters might solve this problem, or perhaps a different model design. Implementing a *k*-fold cross validation analysis might help to clarify why and how often these training failures occur. Meanwhile, give it a slower learning rate and more epochs to train on.

Did not fix the problem with constructing the no-noise images: don’t want to send no-noise to get_noise() to get scaled unnecessarily for a second time. No easy workaround yet found.

Should have started out with a *k*-fold cross validation analysis. This choice of benchmark for model performance made it hard to tell any difference between them. It will still be considered worthwhile for approximate comparison with the same model’s noise conditions.

Adjustments made to hyperparameters, learning rate, epochs.

Increased the noisiness factors from (0.3, 1.3) to (0.5, 1.5).

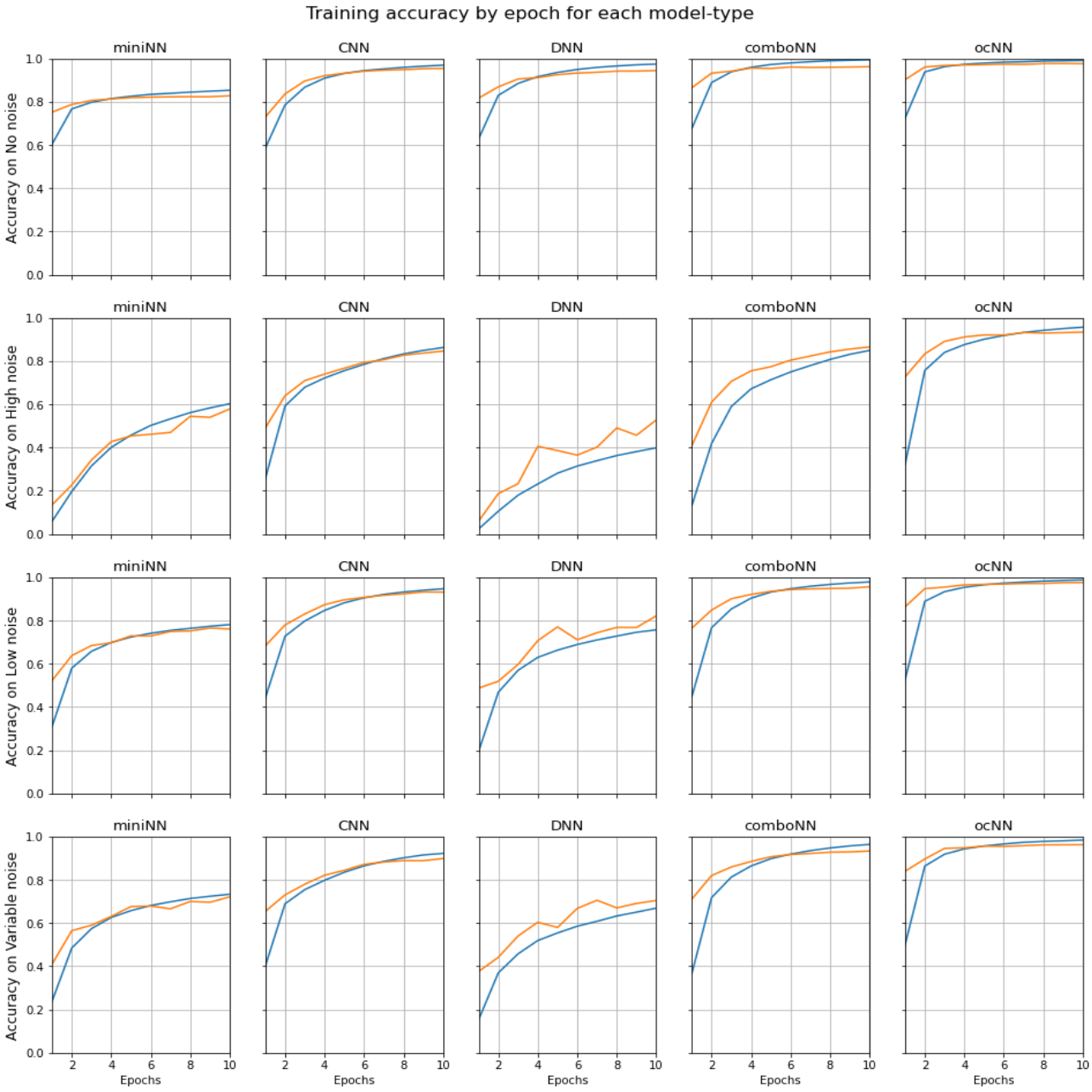
No computation time recorded for comparison.

IV. Results

Training Stats

Each plot in Figure 4 displays the learning curve for one of the twenty models training on each of the different noise levels. The first row are the benchmark-noise models: one of each type trained on unretouched images with no added noise distortion. These results show very little difference between model types under this condition.

Figure 4. Learning curves for each model type, training under each noise condition.

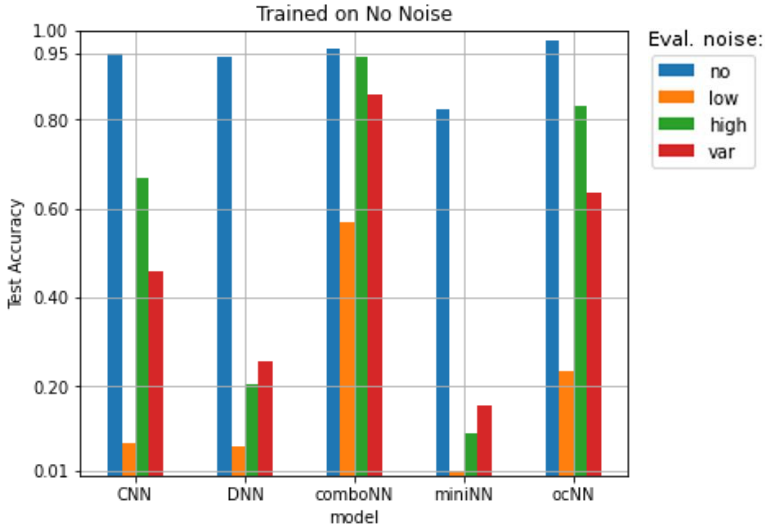


Blue line: training accuracy at each epoch;
Orange line: validation accuracy at each epoch.

There is a slight hint of overlearning visible in some of these curves. This may be what is happening when the blue training curve overtakes the orange validation curve. It is a very small difference and can only be seen near the asymptotic limit here, but it suggests a larger dataset may improve training for at least some of these model designs (especially the “over-complicated” algorithm).

The jumpiness in the orange validation lines could be remedied with some hyperparameter adjustment, most likely a decrease in learning rate (RMSProp optimizer defaults to 0.001).

Figure 5. Evaluation results for each of the baseline models.



The baseline accuracy of models trained on no-noise images and evaluated on no-noise images is shown in Figure 5 (blue bars). Only the minimal model type "miniNN" scored significantly lower than the 95% benchmark under these baseline no-noise training and test conditions. The other model styles just barely crossed above it under these baseline conditions, with ocNN reaching as high as 98% on some runs.

Evaluation

The final results are tabulated below.

Table 2. Accuracy for trained models under evaluation on each noise level

Model	Train cond:	Test condition: no	high	low	var
miniNN	no	0.8263	0.0300	0.1646	0.1941
	high	0.6503	0.5803	0.6728	0.6462
	low	0.7186	0.4253	0.7668	0.6609
	var	0.7806	0.6281	0.7636	0.7241
CNN	no	0.9534	0.1095	0.8628	0.6041
	high	0.8960	0.8436	0.8964	0.8827
	low	0.9309	0.7739	0.9265	0.8940
	var	0.9215	0.8269	0.9178	0.8970
DNN	no	0.9438	0.0244	0.2673	0.2530
	high	0.6754	0.5282	0.6601	0.6263
	low	0.8225	0.5767	0.8200	0.7558
	var	0.8537	0.5030	0.7835	0.7035
comboNN	no	0.9582	0.4259	0.9300	0.7998
	high	0.9044	0.8599	0.9006	0.8940
	low	0.9552	0.6577	0.9510	0.8944
	var	0.9531	0.8631	0.9481	0.9279
ocNN	no	0.9771	0.1223	0.7613	0.5430
	high	0.9632	0.9332	0.9625	0.9567
	low	0.9761	0.8776	0.9730	0.9545
	var	0.9719	0.9283	0.9690	0.9604

Blue values: above the benchmark accuracy of 95%
Red values: close to the "by-chance" baseline accuracy of 1%

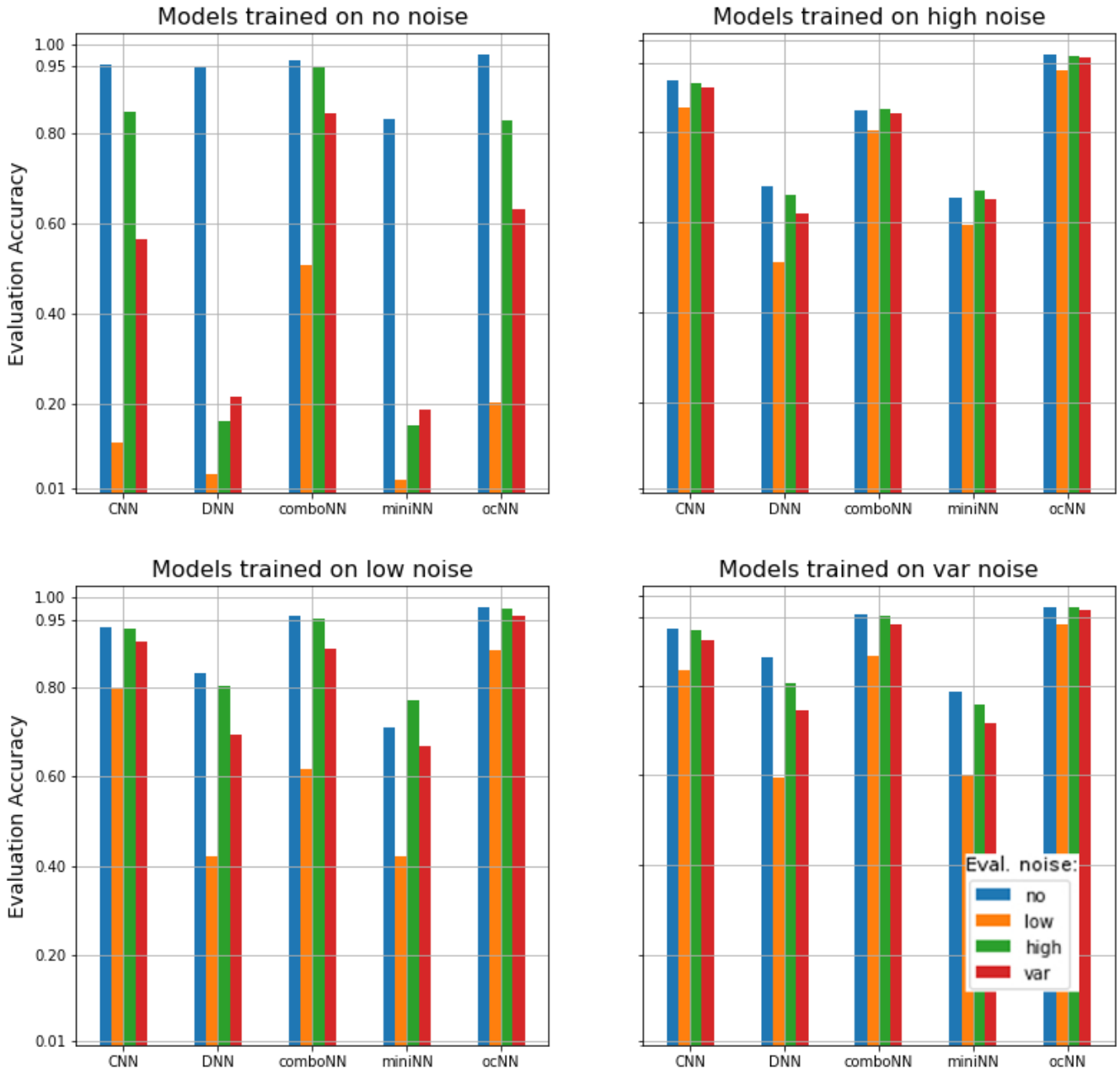
Except for the "minimal" baseline model, every model-type performed close to or above the 95% accuracy threshold for the baseline set of conditions.

Between-Models Comparisons

Across noise conditions, a pattern of performance began to appear. Roughly speaking, the CNN, comboNN, and ocNN models had very similar patterns of performance. These three designs

also consistently outperformed the other two, miniNN and DNN, which in turn performed nearly identically to each other across all categories.

Figure 6. Evaluation results for each training condition, by model type

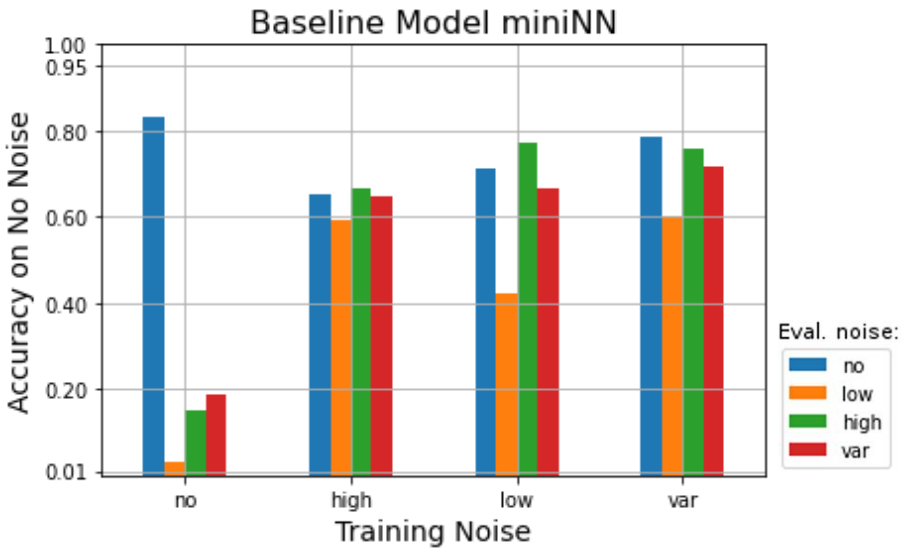


The charts in this figure show the evaluation performance results for each model after training on each condition.

From the upper left plot in Figure 6 it is clear that none of the models trained on the no-noise baseline condition performed better on testing under the other noise conditions. Models DNN, ocNN, and miniNN almost failed to pass the minimum benchmark of 1%, with DNN and miniNN also showing difficulty with the “low” and “var” conditions, too. However the others did

nearly as well with “low” noise as with none. The best all-around performer for this baseline training condition was the comboNN model. CNN and ocNN performance were almost identical across test conditions after training on the clear, original double-digit dataset.
Bottom row of Figure 6 are interesting: every model trained on “high” or “variable” noise performed about as well as on all the other evaluation conditions. This indicates a more generalized and robust representation of NNs trained on these conditions, as compared to those trained only on the “no” or “low” noise levels.

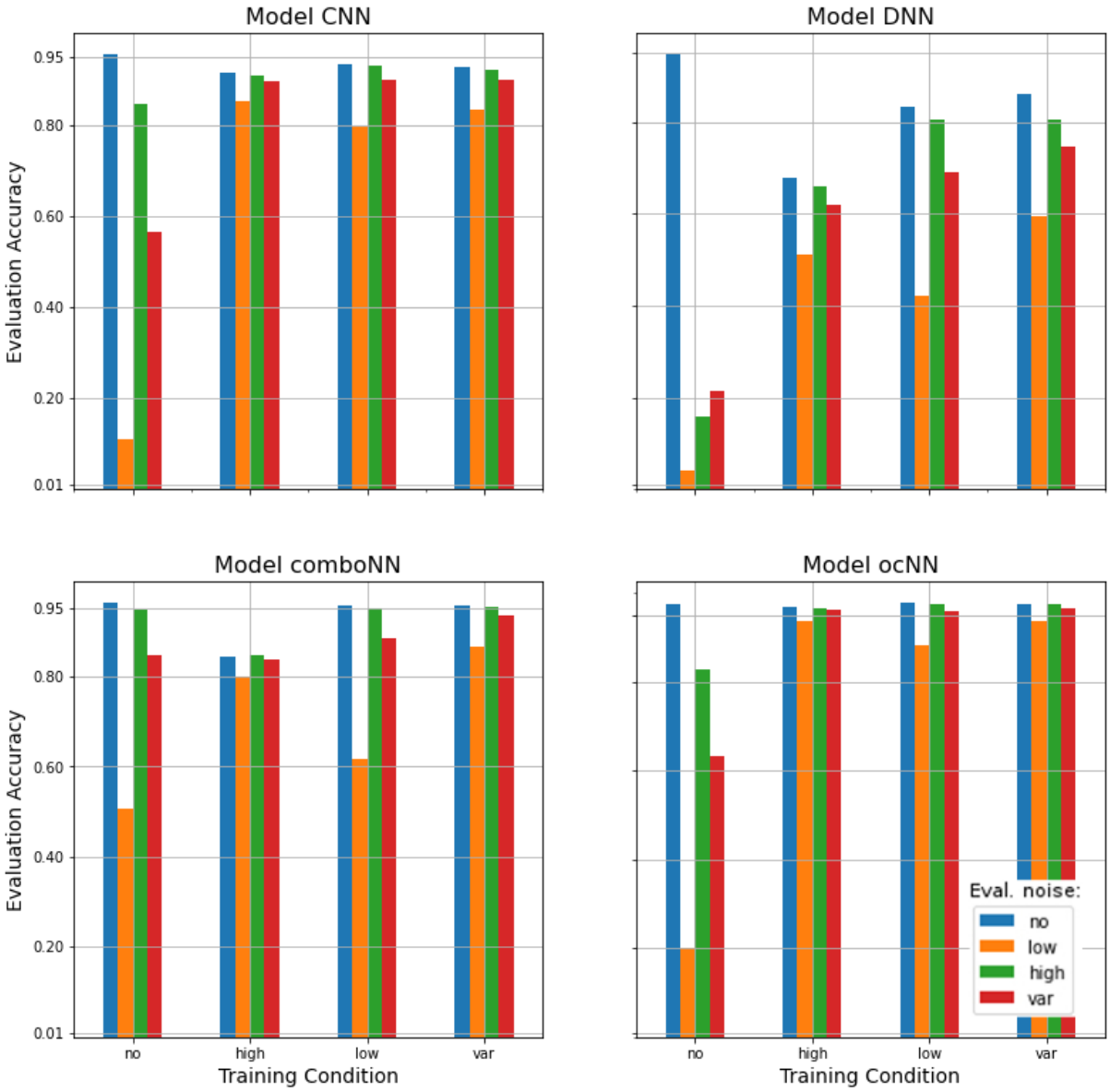
Figure 7a. Bar chart for model miniNN, tested on evaluation noise levels.



The blue bars in Figure 7a represent the test accuracy for the miniNN design-type when evaluated on the unaltered, no-noise digit images. Its best performance, of just over 80% accuracy, was on evaluation with the no-noise test set. This design never even approached the benchmark 95% line, but it did at least consistently pass the 1% by-chance baseline (though not quite significantly on the high-noise test condition). Clearly the no-noise baseline training condition was not sufficient for this model to translate its learning to the other test noise conditions. However it did perform surprisingly well after training on the noisier datasets! Some extrapolation of its learning on these images was applicable to the other conditions, giving it an average accuracy in the 60% to 70% range across the board. These results were much better than expected for this design type.

Noise-Condition Comparisons

Figure 7(b-e). Evaluation results for each model type, by training condition.



It was unexpected for the DNN design to perform no better than the bare minimal 3-layer design (bar plots 7a and 7c). Clearly the flat, linear approach does not do well on image data, but it would be expected to at least do better with more layers and parameters to world with.
Also of interest here was model-type comboNN's significantly better performance than the others when validating on high noise after training on low noise. The others all scored at or below 10% accuracy on this measure. Though the comboNN hovered just below the 95% benchmark across conditions, it also seemed to have acquired a more robust generalization of the data than the other models, even if only marginally.

V. Conclusion

Discussion

The initial benchmark of 95% accuracy was met or surpassed by all but the "minimal" algorithm. As a challenge to improved model designs. a new benchmark of 98% would be reasonable given this performance. And, that is also the accuracy reached by the highest scoring model type under two condition sets, which was the ocNN type trained on no noise and then evaluated on the no- and low- noise test sets.

Without noise distortion, performance above 99.999% should be achievable with an optimized design, as benchmarks that high have long been achievable for single-digit images without distortion.

While working through the full project end-to-end, some improvements in protocol could be discerned. It might have been more straightforward to have examined the effects of *either* doubling the digits *or* adding noise, but not both these variables at the same time. It would also have been possible to more meaningfully compare results, if the appropriate statistical analysis could have been applied from the start. In this case, the useful statistics that weren't used would have probably been a k-fold cross-variation analysis.

One of the most informative findings was that additional sequential layers do not really improve the dense-layer model design's accuracy on this task. And similarly it seems much clearer now how effective the convolutional strategy really is when working with image data.

Questions Revisited

The specific questions posed at the beginning of this project can now be considered in light of the results.

Did the "minimum" model design do any better than chance?

The expectation was that no, it would not. The results however were better than expected. Overall average accuracy was around 65%, and it only remained near the 1% baseline under the most difficult "no train noise, high test noise" set of conditions.

How did the DNN algorithm compare to CNN designs across conditions?

It was expected to be less accurate, and it turned out to be less accurate. A dense design alone is not well suited for visual discrimination tasks; at least, not a linear stack of dense layers.

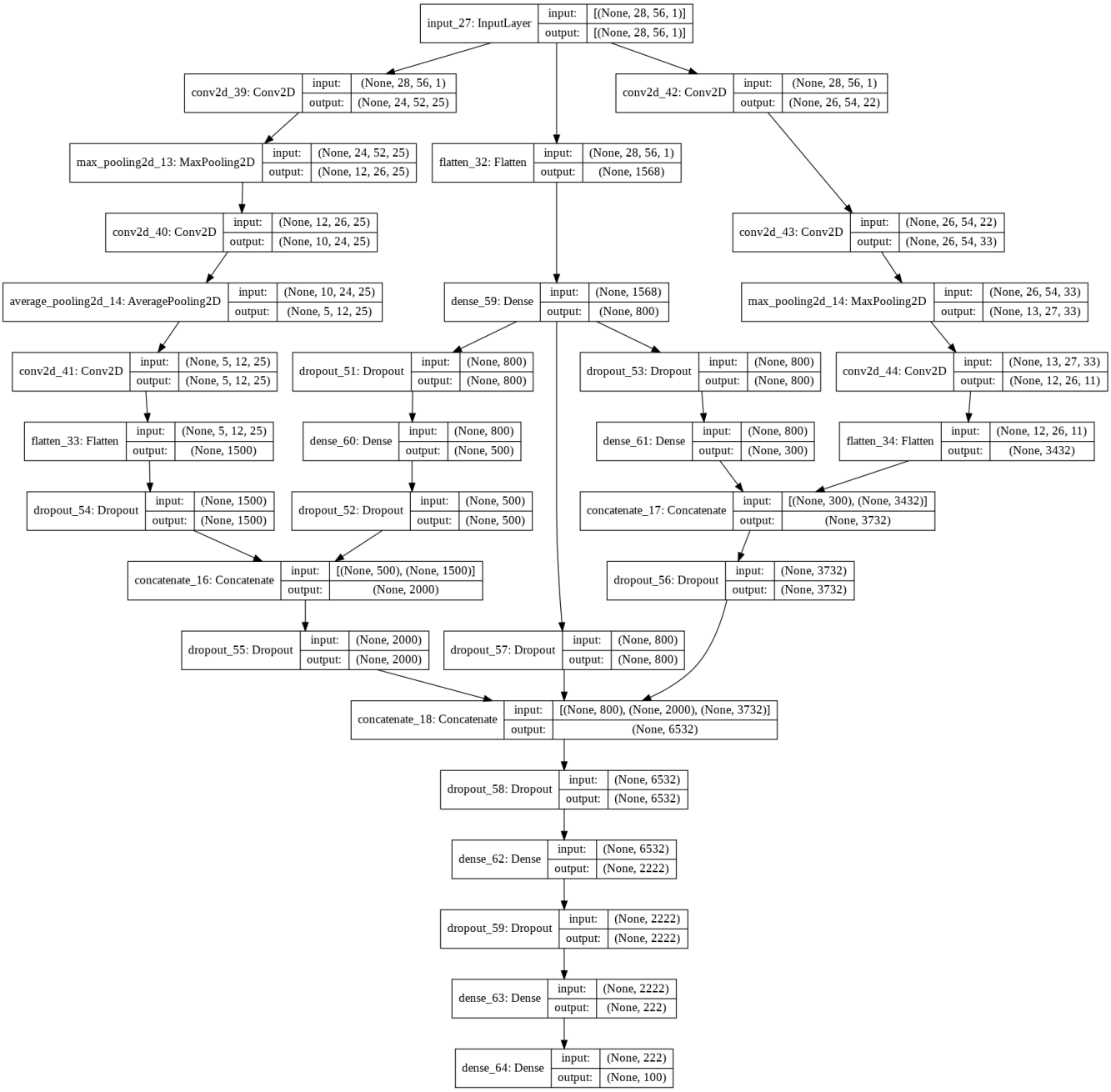
Did the comboNN design do any better than the CNN?

It was originally expected that it might. As it turned out, the three model types with any convolutional layers in them performed about equally well across all conditions: CNN, comboNN, and ocNN. The overcomplicated design did slightly better overall, but the comboNN performed more robustly to different train/test combinations than the CNN.

How much did training a model under one condition affect its accuracy on other conditions?
The initial expectation was that which noise condition a model trained on would have a significant impact on its capacity to classify images from different noise-level datasets. The variable-noise condition was conceived in the hope that training on an image-set with the entire variety of noise distributions, would improve any algorithm's test accuracy on image-sets with a single static degree of noise applied to all its images.

How well did the "over-complicated" ocNN design perform as compared to that of the other model types?
It was not entirely clear beforehand if such an over-complicated, randomly connected design would work at all. The expectation was that it would at least keep pace with the minimal, baseline design, miniNN, but not likely that there would be any real advantage to all that complication. On the other hand, its many pathways and abundant parameters might make it more robust across conditions. Sixteen million parameters to train! It seemed likely to take longer to train, just given all those neurons, and would surely use up more than its fair share of memory resources, too. It also seemed likely to throw a lot of errors during implementation, with all those branching pathways and interconnected layers at play.
The results were surprisingly good, which in hindsight does make sense. Also it did not turn out to be any buggier than the other algorithms; nor were its training times exceptionally slower, compared to those of the others.
Working out the ideal arrangement for a design with so many variables would be a good next-step to pursue.

Diagram 2. Design scheme for model type "over-complicated". Best performing overall model type.



Next Questions

It would be nice to experiment with different hyperparameters, including learning rate and number of training epochs. Also, which size dataset is sufficient for all the models to train successfully, if any? Smaller, larger?

How does design performance compare between single-digit and double-digit images? What about triple-digits? Two digits required just 100 classification categories; five-digit images would already be 100,000 possible classifications for the algorithm to learn! How many digits can an algorithm take? And what about the alphabet? There must be a limit somewhere on the number of categories for an algorithm to choose from.

How much noise is too much? Can algorithms classify images better than humans, above a certain noise threshold? << insert extremely noisy examples here >>

Some of the lessons learned here might apply well to other public datasets that are already noisy as they are, without any more added noise to them, such as the SAT-6 land-cover imagery dataset[8], for example. That data could be divided into subcategories of noisiness levels, and some pretraining on these might improve performance on the full set.

The densely connected algorithms could be redesigned to not fail the minimum accuracy baseline of 0.01 so often. Are there any parameters or settings that would help this model learn under the high noise condition? Give it more epochs? A looser learning rate? A larger training set to train on? What about implementing parallel sequences of single dense layers, instead of stacking them vertically? Would more layers and neurons help any, or is this design just not properly suited for this task?

[8] SAT-4 and SAT-6 airborne datasets