Neural Network Classification

Aaron Gonzales, Allison Ko, and Ruize Zhang

Abstract

The gap between the intelligence of a human and machine draws smaller as various fields continue to pursue the development of machine learning algorithms that better emulate the various ways that humans obtain and analyze information. For this specific task we created a neural network that was able to classify images from the Caltech silhouettes dataset that includes 101 classes of over 10,000 images. We decided to break up the task of classification into several parts based on the neural network algorithm. Although the breakdown provides good insight into the understanding of neural networks, they provided little in terms of improvement of accuracy. At the end we will also introduce the effects of PCA on the data and how this modification affected the run-time and accuracy of the model.

Introduction

Neural networks in general work in several steps that can be summarized in 5 major steps. Initially we have an input from the training set that is read into the input layer of the network. This data is then fed forward through the network and manipulated by the weights and biases, followed by an activation function at each subsequent node in layers beyond the input layer.

Once reaching the final layer, i.e. output layer, we then calculate the gradient of the cost function that we are utilizing and begin to perform backward propagation. Simply put, backward propagation aims to find the error associated with each node in the previous layers and decrease each one through some iterative optimization method. Once the propagation is complete we are able to repeat the process until we hit a number of iterations of optimization performed on the training set. It is important to take note of a few equations that impact the performance of the neural network. Firstly is the activation function, which takes the linear combination of the input and the weights plus the bias generally represented as $\sigma(w^T x + b)$. Sigma can vary in terms of the function it represents and for the purpose of the paper we will refer to its input as z for simplicity. The second is the calculation of the error at the end of forward propagation represented as $\delta^{x,L}$ where L refers to the layer and x the node for which the error pertains to. The output layer's error is special since it depends directly on the choice of cost function such that

$$\delta^{x,L} = \nabla C_x * \sigma'(z^{x,L})$$

where * refers to an element-wise product. Essentially we calculate the rate of change of the cost function at this node with respect to its activation, the σ' is just a result of the chain rule. The rest of the functions shall be introduced later, but for now this is sufficient to continue.

The algorithm involves several factors that have room for customization making the neural network extremely versatile in many situations. We attempt to improve our method in several ways by inspecting the effects of different initializations of weights, activation functions, hyper-parameters, and optimization methods. Although it is difficult to interpret how exactly the network is classifying the images, we can get a better intuition for which implementation works the best for this particular task.

Activation Functions

Activation functions play a major role in the neural network algorithm so understanding which functions characteristics are good in what cases can come in handy, especially since some specialize in certain tasks. In this project we experimented with several activation functions.

Sigmoid

The standard function to use is the sigmoid function given by

$$\sigma(z) = \frac{1}{1 + exp(-z)} \tag{1}$$

The function provides a way to squash values to between 0 and 1, analogous to the firing of an actual neuron. The differentiability of the function makes it easy to implement with in feed forward and backward propagation. The problems associated with the function include the fact that it can saturate neurons particularly fast, meaning that the derivative of the sigmoid with respect to the linear combination of the weights and input, z in the function above, go to 1 or 0 quickly, ”saturating” the neuron and making it incapable of learning any more. This is a problem that is also closely associated with the initialization of the weights which will be discussed later.

Tanh

A function closely associated to the sigmoid is the tanh function. Being so similar it is trivially susceptible to the same problems that we discussed with the sigmoid function. The only difference is that the function is actually zero centered meaning that it exist between [-1, -1], so it is typically preferred over sigmoid since it can deal with negative values. The tanh follows the structure,

$$\tanh(z) = 2\sigma(z) - 1 \tag{2}$$

We have written the function in terms of the sigmoid to allow for reusability of the sigmoid function and show its similarity.

ReLU

Lastly, we tried to use the rectified linear function also know as, ReLU or rectified linear function. This activation function computes,

$$ReLU(z) = max(0, z) \tag{3}$$

The function is simple yet effective because it generally allows for acceleration in convergence of stochastic gradient descent due to its linear, non-saturating form. The function is also differentiable and has low computational cost since it just uses a max function. This has the drawback of many neurons being set to zero or dying, disabling the ability for the network to learn since all the activations will be set to zero. There are ways to avoid this, but that is out of the scope of this paper. We also will not look much at this function due to its poor performance in our case as seen in Fig. 1.

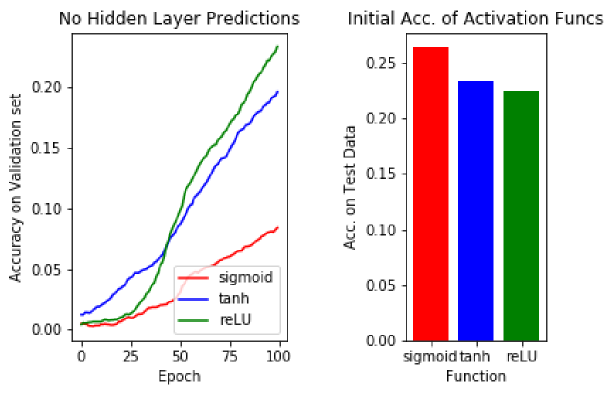


Fig. 1 Init. Predicted Values w/ functions

In the figure above we have used a bare linear perceptron, meaning our neural network is devoid of hidden layers. Not having any hidden layers means that our network simply calculates weights as linear combination of just the input without trying to capture any of the special features of the samples as many networks do with multiple hidden layers. These leads to a very simple model, and in our case, to poor values when using 100 epochs. As a note we used stochastic gradient descent (SGD) with 100 batches with a learning rate of .5 and a L2 cost function which will be the standard method we use unless otherwise stated. From the graph we can see characteristics of the functions already, with reLU doing far better initially for the validation set. Also, the sigmoid function clearly learns slowly meaning the neurons may have been saturated to begin with and had poor initialization and whereas tanh and reLU are still increasing. As we see in the bar graph though the sigmoid does the best when it comes to the test though, with tanh and reLU not far behind. This is a naive run since we optimized no parameters and have no hidden layers meaning we can most likely achieve a higher accuracy on the data than shown. The first optimization will be tuning the learning parameter of SGD that affects how the model learns.

Stochastic Gradient Descent

Stochastic Gradient Descent allows us to use a sample mean of gradient descent to achieve a similar descent to a local minimum. The function takes the form

$$w^{new} = w^{old} - \frac{\omega}{batch\ size} \sum_x \delta^{x,l} (a^{x,l-1})^T \tag{4}$$

$$b^{new} = b^{old} - \frac{\omega}{batchsize} \sum_x \delta^{x,l} \tag{5}$$

For all the batches, we take, for each node, the sum of the gradient of the cost function with respect to this nodes weight and subtract a fraction of this from the current weight value. The same goes for the bias, but is much simpler since the gradient w.r.t the bias is just the error.

Cost Functions

The L2 norm is an easy choice since it is differentiable and is a common metric for error. The problem with this cost function in terms of our particular problem is that we are trying to classify items that have an exclusive class. Why does this matter? Well when observing the characteristics of the L2 norm we see that it is derived from assuming the target is continuous and normally distributed and when maximizing the likelihood results in the MSE or L2 norm. In our case the data is not normally distributed nor is it continuous so the L2 norm does not properly represent the error we want to calculate. Therefore we can use a multinomial cost function that takes explicit classes into account. That cost function is the cross-entropy function which has the form

$$C = \sum_x \sum_j [y_j ln(a_j^L) + (1 - y_j) ln(1 - a_j^L)] \tag{6}$$

where we sum over all the samples in the set (x), and the number of classes (j). This models the probability of being part of a certain class, which is more suitable to the scope of this problem. The cross-entropy function is also very convenient since it does not depend on the derivative of the activation function (no more slow learning), but only on the difference between the activation and the label, making it a linear function. Since the cost function is derived from the log-likelihood when the output layer is a softmax function, it is recommended we change the output layer activation to it, since sigmoid is more appropriate for a binary classification. The results of the change seen in Fig. 2 are indicative of improvements in the right direction.

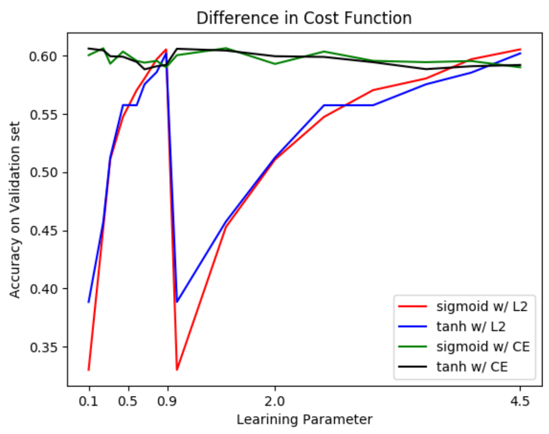
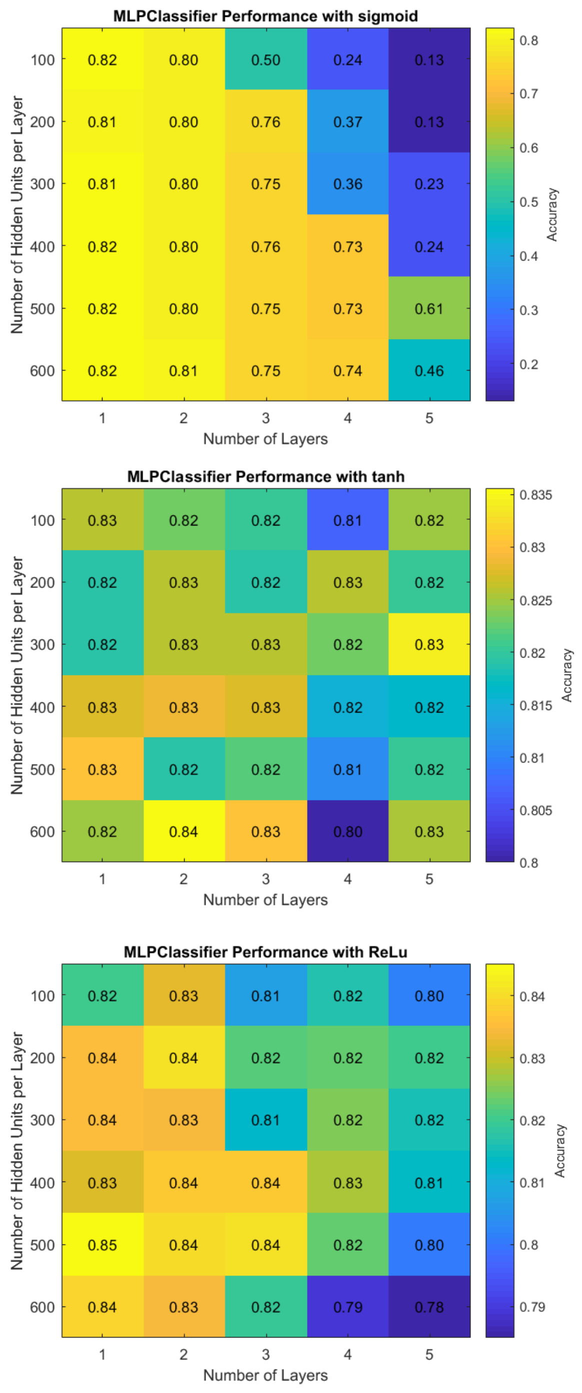


Fig. 2 Accuracy of Tests on Validation set

From the test above going through several values for λ we see that the cross-entropy function almost always performs better. Clearly the optimal choice is the cross-entropy with the softmax activation output layer following what we had originally assumed above.

Layers and Sizes

The number of hidden layers and units greatly determine the complexity of the model and henceforth sway the bias-variance trade off. When using more neurons we can see that we hit higher accuracy on the training data but a lower value on the validation set, a clear sign of over-fitting. A good way to deter this is first to determine a layer that gives good accuracy with minimal over-fitting. Later we will try to regularize the over-fitting if needed, but seeing our results as of now this is unnecessary. We ran a series of tests with the Python library scikit-learn, using the MLPClassifier model. The results are shown in Figure 3.



MLPClassifier Performance with sigmoid

Hidden units	1	2	3	4	5
100	0.82	0.80	0.50	0.24	0.13
200	0.81	0.80	0.76	0.37	0.13
300	0.81	0.80	0.75	0.36	0.23
400	0.82	0.80	0.76	0.73	0.24
500	0.82	0.80	0.75	0.73	0.61
600	0.82	0.81	0.75	0.74	0.46

MLPClassifier Performance with tanh

Hidden units	1	2	3	4	5
100	0.83	0.82	0.82	0.81	0.82
200	0.82	0.83	0.82	0.83	0.82
300	0.82	0.83	0.83	0.82	0.83
400	0.83	0.83	0.83	0.82	0.82
500	0.83	0.82	0.82	0.81	0.82
600	0.82	0.84	0.83	0.80	0.83

MLPClassifier Performance with ReLu

Hidden units	1	2	3	4	5
100	0.82	0.83	0.81	0.82	0.80
200	0.84	0.84	0.82	0.82	0.82
300	0.84	0.83	0.81	0.82	0.82
400	0.83	0.84	0.84	0.83	0.81
500	0.85	0.84	0.84	0.82	0.80
600	0.84	0.83	0.82	0.79	0.78

Fig. 3 Performance of Neural Networks with Different Architecture

These tests were run on smaller datsets of 10 classes, and the average accuracy of 5 trials was taken. For this task, the model using ReLU with 500 units and 1 hidden layer seemed to provide the best results. When using the ReLU and tanh activation functions, we found that the architecture did not have a huge impact on the results, but when we used sigmoid, more hidden layers led to significantly worse performance.

Weight Initialization

Weights of the data and how they are initialized is important to getting good results (which we are lacking) since our initialization of the weights and bias help determine the local minimum that we eventually converge to during gradient descent. Say for instance we were to initialize weight values as a high integer, this would lead to saturation of the weights of neurons resulting in slow convergence during back-propagation. Why is this? When taking for instance the sigmoid or tanh function its easy to observe that near 1 or 0 the values of their derivatives are very small. This means in back-propagation that a correctly predicted label is unlikely to move, but an incorrect one will change very slowly. There are many ways researched to initialize the weights, but one simple method is to divide each weight matrix by the square root of the size of the output layer. You can think of this intuitively since the initialization of weights is Gaussian distributed with mean 0 and SD 1 meaning that our input to the activation functions are also. The variance of these values increase with the size of the data and therefore a way to reduce this variance is to divide by the size of the layers such that our new weights are initialized by, $\frac{w}{\sqrt{n_{in}}}$.

We can see the results below and how they effect convergence of SGD saving us computation time since we are reaching a minimum much faster.

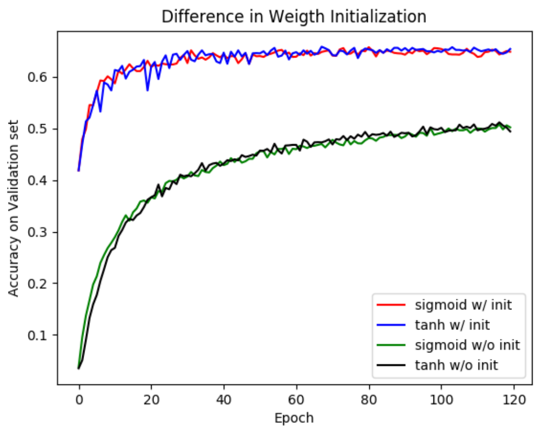


Fig. 4 Differences in Init. of weights

The accuracy shown as the starting accuracy for the runs with the new weight initializations is not the standard, but this does highlight the differences in the methods showing that one has more potential to converge to a better minimum than the other. One major thing to notice is that the new method converges within 40 epochs meaning we can run way less iterations saving us much more time.

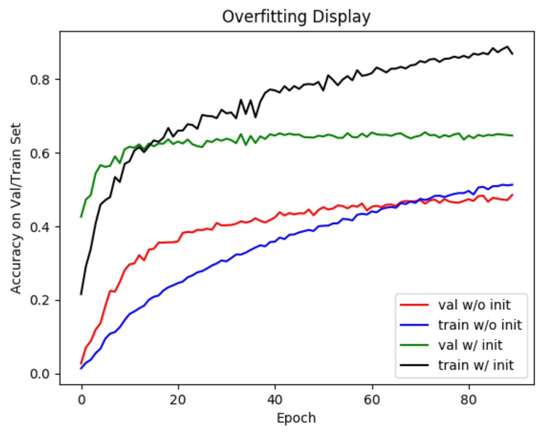


Fig. 5 Over-fitting from new weight init.

Regularization

Now that we have have testing scores, it is apparent that we now have over-fitting. Meaning that we must regularize the cost function by adding a parameter to penalize predictions and reduce how well we can map the training data. The cost function becomes,

$$C = \sum_x \sum_j [y_j ln(a_j^L) + (1-y_j) ln(1-a_j^L)] + \frac{\lambda}{2n} ||w||^2 \quad (7)$$

where λ is an adjustable parameter. The gradient w.r.t the weights will also change making the gradient descent iteration become,

$$w^{new} = (1 - \frac{\lambda\eta}{n}) w^{old} - \frac{\omega}{batch\ size} \sum_x \delta^{x,l} (a^{x,l-1})^T \quad (8)$$

showing that we take only a percentage of the weight compared to last equation. The variable n refers to the size of the training size. Finding the optimal parameter will help deter the affects to over-fitting and also give us a boost in prediction accuracy since we will have more room to optimize parameters!

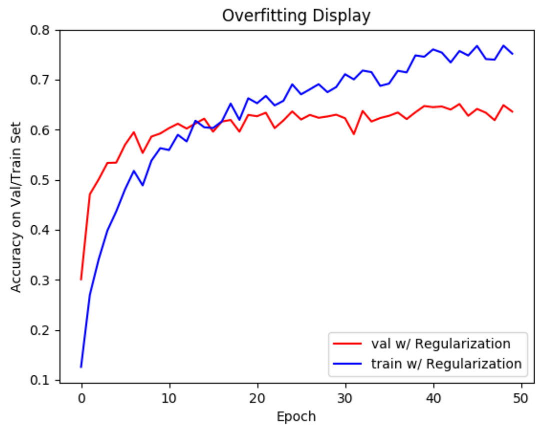


Fig. 6 Scores w/ Regularization

With regularization now, we are not achieving much higher scores, but we are no longer over-fitting the data which is great since it means our model is actually making predictions here rather than memorizing the labels. With PCA regularization had a much greater effect increasing the score from a .55 to a .64 which means that it allowed the model to learn more efficiently with preprocessed data.

PCA Comparison

Principal Component Analysis is a method used to represent data in a lower dimension. For our case we are given data that when used as input for the network results in an input layer of 784 which is the largest layer we have. Reducing this could reduce the time spent on back-propagation, but also result in a decrease in prediction accuracy. PCA runs into the same problems in terms of optimization of the parameters, but using the same methodology above we can once again optimize our results.

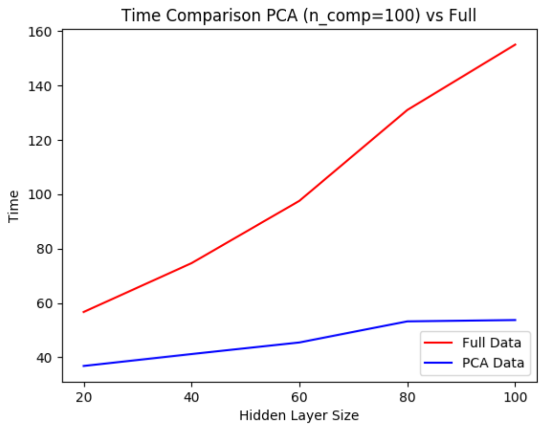


Fig. 7 Runtime Comparison with PCA

We found that using PCA, 100 components explained 82% of the data, making it a good choice for this task.

Model Comparison

Using the Python library scikit-learn, we tested the performance of various models on the dataset to compare them to our neural network. These models included a support vector machine, a logistic regression model, and a neural network. For the SVM, we used the scikit-learn class SVC, which implements one-versus-one classifiers for multiclass classification. We tested several available kernel functions, and found that the linear kernel had the best performance. The class MLPClassifier is a neural network that we ran with its default settings. It has one hidden layer with 100 units, uses the ADAM optimizer, with the ReLu activation function and cross entropy. The results of our comparisons are in Figure 8.

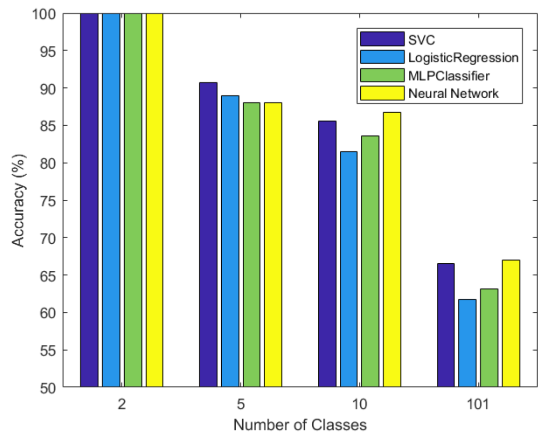
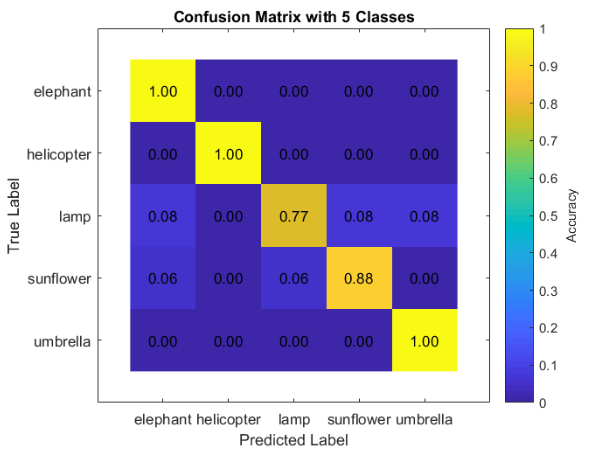
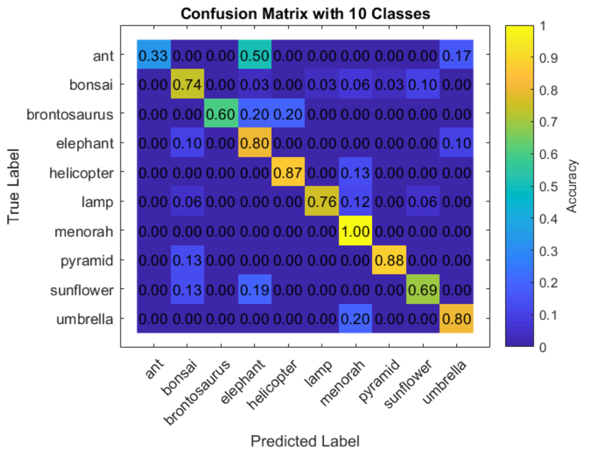


Fig. 8 Accuracy of Various Models

Conclusion





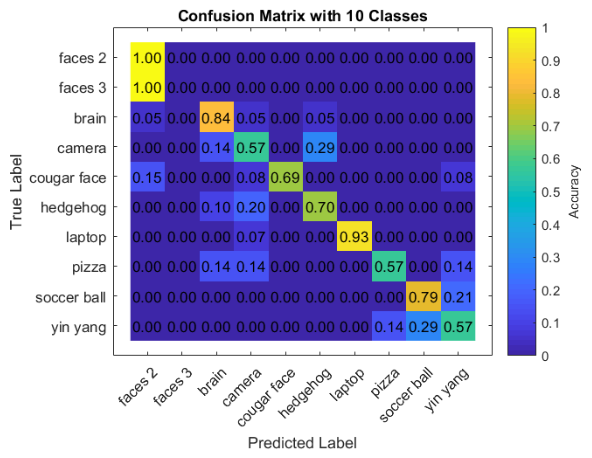


Fig. 9 Confusion Matrices for Various Classes

Examining the results class by class, we can see that in general use cases the model can decently classify the images, but when the images are extremely similar, e.g. pizza, soccer ball, and yin yang, the model performs much worse. Especially when classifying face 2 and face 3, as expected, the model does not distinguish between the two and lumps them into one category.

	Regular Data	PCA Data
Scores	.67	.65

Table 1: Best Scores table.

The results on the tables above are from tuning every parameter and experimenting with the model to the fullest extent. We were able to achieve a .65 with PCA without overfitting with a learning rate of .9 and actually no initialization of the weights which is odd. The unprocessed data was able to reach a .67 with a learning parameter of .2 and weight initialization. We coudl only assume that PCA allowed for the model to find a better minimum with PCA when regularized since it changes the input size, reducing the amount of variation between the values and therefore no longer needing the other weight initialization. It is intersting to note that during several trials that PCA even did better than our model with the same parameters. This should not have occurred even if it was slightly. The process of understanding the network by tuning every possible parameter and design highlighted major drawbacks of the Neural Network which is the time constraint it has. Our code took sometimes very long to run especially to test out k-folds, but regardless they are powerful as seen in their classification of over 100 images.

References

Nielson, M. (2015). *Neural Networks and Deep Learning*. Determination Press.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.