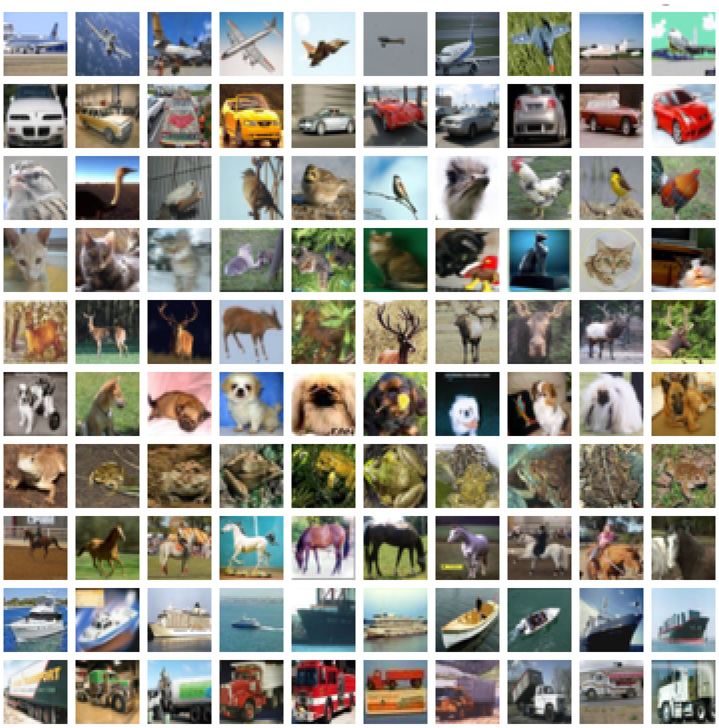
Image Classification through Convolutional Neural Networks on the CIFAR-10 Dataset using AWS

Capstone Project for Springboard by Ken Wallace, January 2018

Background

Objective

The goal is to create a Convolutional Neural Network to perform a computer vision image classification on the CIFAR-10 dataset that achieves better than 80% accuracy on the test set. The data (collection of images) is publicly available. This dataset contains 60000 images in 10 categories, with 50000 for training and 10000 for testing of the model.



Various tools and techniques will be used, primarily Keras as a wrapper for Tensorflow (for easier model generation), but also Numpy, Scikit-Learn, and Matplotlib.

Project Plan

The overall plan for this is to:

1. Learn how to create a baseline neural network in Keras, including the use of GridSearchCV (from the scikit-learn API) to get a sense of what impact can be made to the accuracy of a

model by adjusting the learning rate, batch size, and number of epochs. The purpose is to see how much better a model with convolution can be than one without it.

2. Add convolutional/hidden layers to improve the accuracy by using a deeper network. In this approach, the number of layers, the number of filters and their size, as well as the kernel size, will be adjusted in order to see what impacts can be made by tuning those hyperparameters. If it makes sense, additional tuning can be made to the optimizer and dropout rates (to help prevent overfitting).
3. At the start of this project, it wasn’t clear how it would be critical to use a GPU for model training. As a result, I started using Amazon Web Services (AWS), in particular a p2.xlarge instance of the Amazon Linux Deep Learning AMI, which was chosen because it has a GPU and also Tensorflow, Keras, and other libraries pre-installed.

Identify Use Case

There is no specific client, but this would be a tool that could be used by anyone looking to perform image classification, from applications ranging from self-driving cars to satellite image analysis to medical image (e.g. X-ray, MRI, CT) analysis to smart kitchen appliances. The tools learned through this process, such as Tensorflow and Keras, creating convolutional neural networks, and grid search, will be invaluable in future machine learning projects. In addition, learning to use AWS will be beneficial in future projects for creating deep neural networks without having to build dedicated computer systems.

Data import and pre-processing

The data didn’t require wrangling in the same way that other data does, rather the challenge comes in when attempting to build on a baseline model in order to improve classification accuracy. That being said, there were two different methods for importing the data, one was from the original source and the other using the data built-in to Keras.

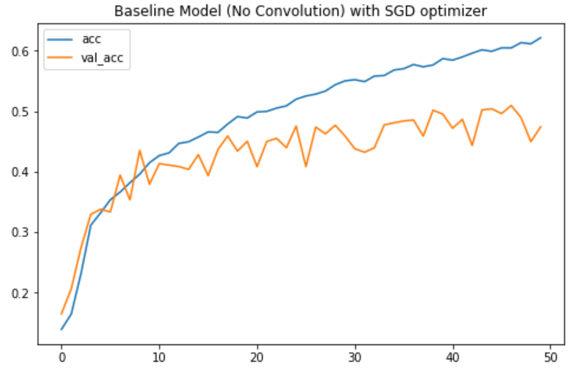
Initially I used the downloaded and extracted data, but after not getting the accuracy to the desired level, I ran a small experiment to see if there was a difference between the downloaded and imported data and found a nearly 7% difference (imported was higher). So all further models were trained using the imported data. Imported data came in as shape (50000,32,32,3) for the training data and (10000,32,32,3) for the test data. For the baseline models, this should be reshaped to (50000,3072) and (10000,3072), respectively, using numpy.reshape(x_train, (x_train/test.shape[0],-1)). The downloaded data came in as (50000,3072) and (10000,3072) which required reshaping for convolution. It’s unclear what caused the accuracy difference.

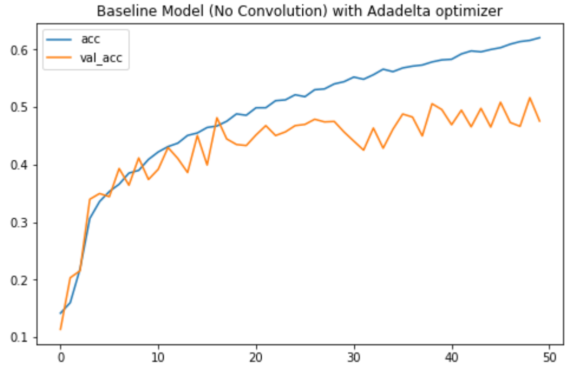
Potential data sets on which these techniques could be used

Other datasets that would utilize skills learned here would include the MNIST handwritten digit classification and likely many proprietary data sets containing satellite images, environmental images (e.g. weather patterns), street views for self-driving cars, medical images, and image sets of anything that might go into a refrigerator.

Baseline Modeling

1. The non-optimized baseline model was originally attempted with the Adam optimizer using the default learning rate (0.001) and all models only achieved a 10% accuracy. To see if the issue was that this optimizer was not suited to this data set, three baseline models were run with different optimizers. One of those was the Adam optimizer, but with a smaller learning rate than the previous attempt. The results with all three optimizers were similar, but Adadelta (47.59%) and SGD (47.41%) were the highest (while Adam was 47.39%).


Baseline Model (No Convolution) with SGD optimizer


Baseline Model (No Convolution) with Adadelta optimizer

2. The next step was to perform a GridSearchCV to see the effects of different batch sizes and numbers of epochs. I originally ran this on my laptop and it took 3.5 days to complete. Due to computational costs, I ran this again later using the AWS GPU, but I limited this to batch_size=[64, 128] and epochs=[100]. I used Adam as my optimizer even though this was the lowest (assumedly within a margin of error) in the baseline models because this was referenced as the most advanced optimizer in many papers I read.
3. Initially the models with convolution used batch size of 128 due to a misreading of the GridSearch results, and eventually batch_size=64 was used on a number of models. As it turned out, there was not a significant difference between the two, especially when other parameters were adjusted as well.

Models with Convolution

1. All convolutional models were run using the Keras Sequential model API with the Adam optimizer, as that seems to be the newest and most robust available. I suspected my initial issues with Adam may have been due to a too large learning rate. This was actually proven during a series of tests where only the learning rate was adjusted (from .0001 to .001 to .01, where lr=.01 resulted in 10% accuracy as seen much earlier).
2. Initially I tested the most basic of convolutional models just to prove that it works and to see what the “worst” CNN would do compared with the optimized baseline model. The “worst” of the CNN models I tested (72.23% accuracy) was nearly 25% better than the best baseline model.
3. Subsequent model iterations were created to measure the impact of adding various hidden layers, with varying numbers of filters and kernel sizes, and varying amounts of dropout at different points within the model graph.

Initial Results

Initial results found that the baseline model, using the Adadelta optimizer, batch size of 500 and number of epochs = 500 yield a classification accuracy of 52%. This is quite low, though better than the non-optimized baseline model. Subsequent trials were done due to computing challenges and found a slightly lower baseline accuracy, though similar, for the three optimizers tested.

After running the initial baseline models, the next step was to create a convolutional neural network with a minimum number of hidden layers to see how a basic CNN compared with the baseline models (no convolution).

Secondary Results

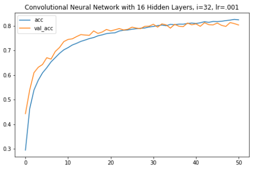
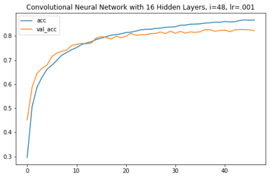
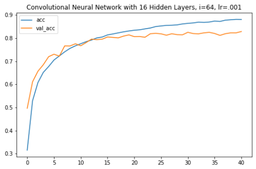
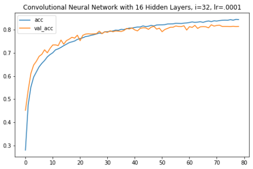
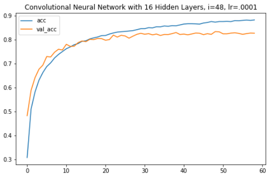
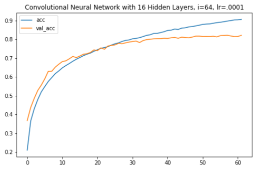
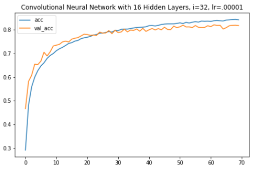
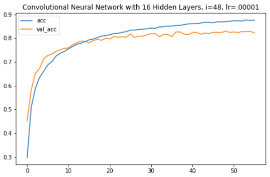
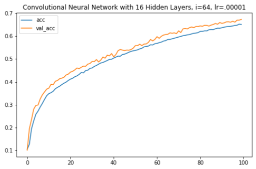
After running the basic CNN models, the next step was to create a deeper convolutional neural network with as many hidden layers as necessary (and in some part as many as possible that can be run on the selected GPU instance using AWS - in particular a p2.xlarge instance) to improve the test set accuracy to at least 80%.

Through many iterations to test the effects of convolution, the following hyper parameters were adjusted (somewhat randomly):

- Batch size
- Learning rate
- Numbers of hidden layers, including:
    - Number of filters
    - Kernel size
    - Number of MaxPooling layers
    - How many dropout layers and the amount of dropout
    - How many dense layers and their size

Batch size during convolution was kept at 128 despite the earlier GridSearchCV based on seeing some commonality among some example models. At the end a couple of the best configurations with batch_size=128 were tested with batch_size=64 and run for a full 200 epochs. This could definitely be explored further with additional resources. Learning rate seems most correlated with how many epochs are required to get a stable result. For example, in two different “sets” of runs where only learning rate was adjusted (i.e. number of filters was consistent for each set, but not from one set to the next), a learning rate of .001 achieved 81.8% accuracy in 25 epochs in set 1 and 79% accuracy in set 2.

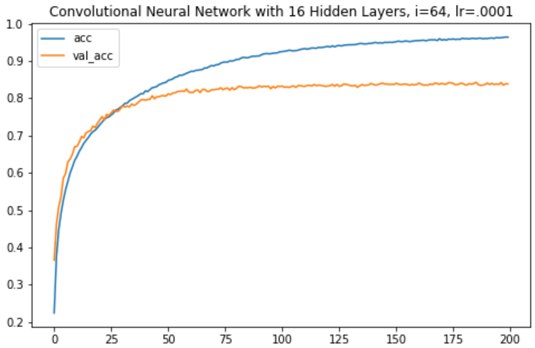
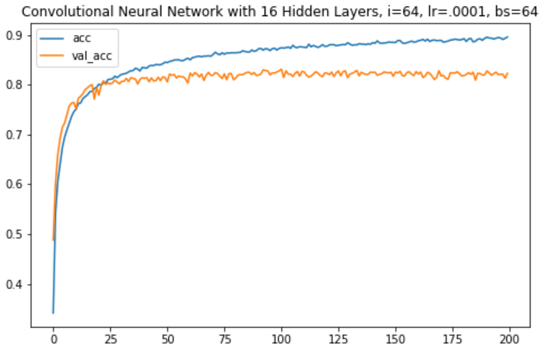
Tests not shown here with a learning rate of .01 only achieved a 10% accuracy. This rules out using a learning rate of .01 from this point forward.

Learning rate	Set 1 Test Accuracy	Set 2 Test Accuracy	Set 3 Test Accuracy
.001	80.36% (51 epochs)  	82.16% (47 epochs)  	82.82% (41 epochs) 
.0001	81.44% (79 epochs)  	82.69% (59 epochs)  	82.13% (62 epochs) 
.00001	81.82% (70 epochs)  	82.24% (56 epochs)  	67.35% (100 epochs) 

Set 1 contained convolutional layers with 32, 32, 64, 64, 128, and 128 filters.

Set 2 contained convolutional layers with 48, 48, 96, 96, 192, and 192 filters.

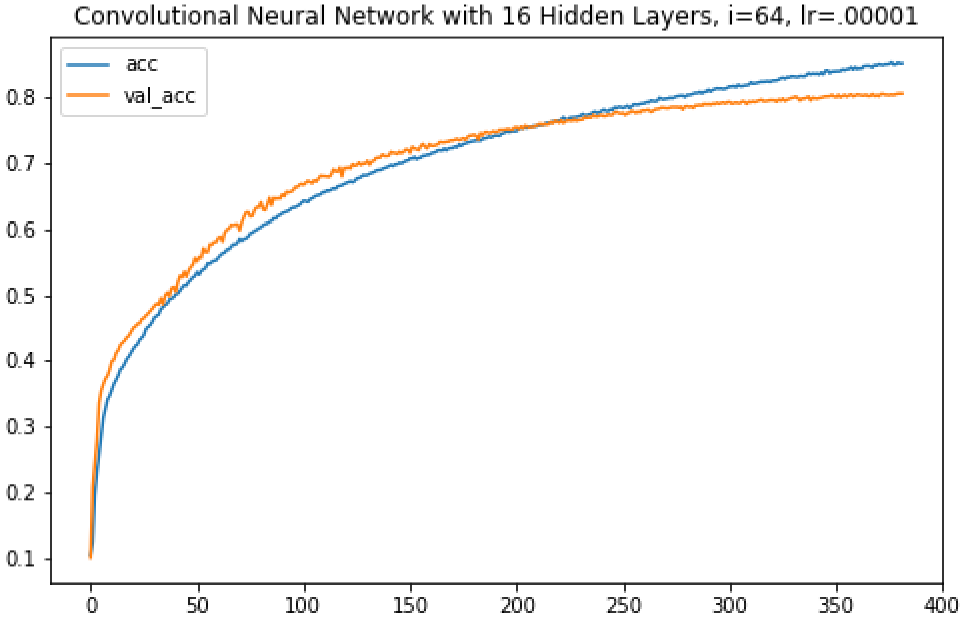
Set 3 contained convolutional layers with 64, 64, 128, 128, 256, and 256 filters.

Set 4 Test Accuracy (lr=.0001, batch_size=64, 200 epochs)	Set 5 Test Accuracy (lr=.001, batch_size=64, 200 epochs)
83.87%  Convolutional Neural Network with 16 Hidden Layers, i=64, lr=.0001 	82.27%  Convolutional Neural Network with 16 Hidden Layers, i=64, lr=.0001, bs=64

Set 4a contained convolutional layers matching Set 2, but with batch_size=64 instead of 128. This resulted in a test accuracy of 83.87% after 200 epochs.

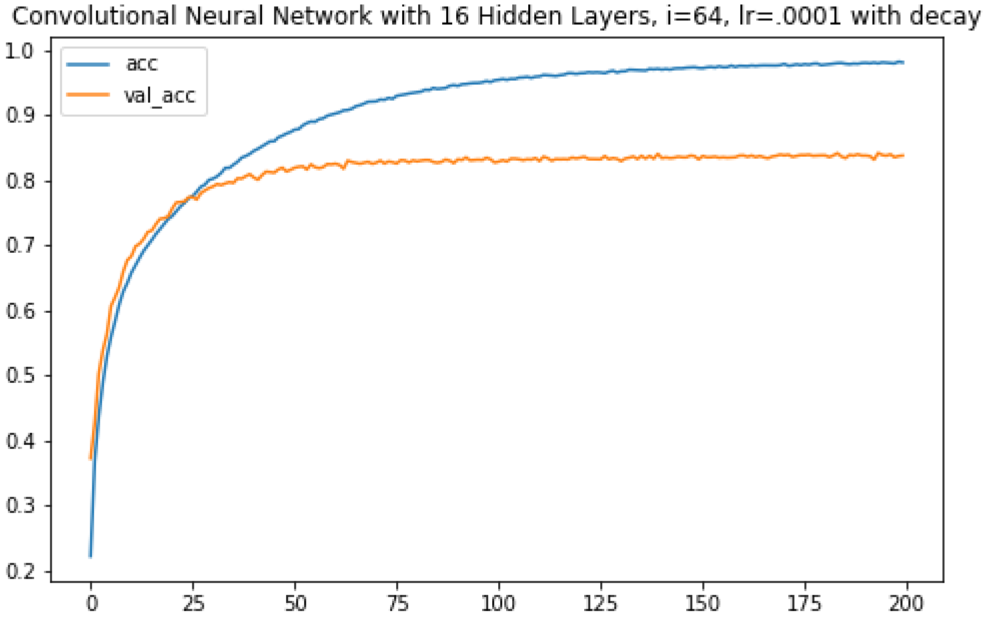
Set 5 contained convolutional layers matching Set 3, but with batch_size=64 instead of 128. This resulted in a test accuracy of 82.27% after 200 epochs.

Set 6 replicated Set 3, but left to run for longer (382 epochs), and still only achieved 80.60% accuracy, but was much less overfitted to the training data. Training accuracy was only less than 5% higher, compared with the training accuracy of Set 7, for example, which had a training accuracy that was almost 15% higher than the test accuracy.

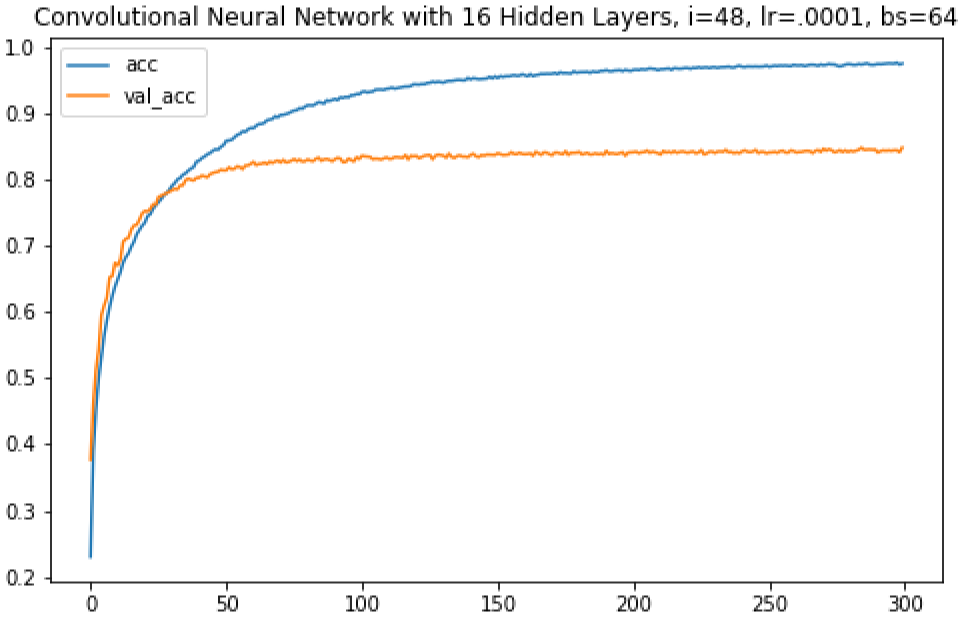


Set 7 replicated Set 3, but left to run for 500 epochs. Test accuracy was 84.73%, the highest so far.

Set 8, used the same parameters as Set 3, but included a decay of .000001 per epoch on the learning rate. The test accuracy was 83.77%. This only ran for 200 epochs, so this is another one that might have been slightly higher with a longer run, but considering how flat the accuracy curve gets after 75 epochs, it may not be a significant difference.



The last configuration, Set 4b, replicated the settings for Set 4a but was left to run a full 300 epochs. This had the best results of all, achieving 84.84% accuracy on the test set.



All sets used MaxPooling and dropout layers after each pair of convolutional layers, and 2 Dense layers of different sizes.

Of the above configurations, the best results were 84.73% accuracy on Set 7. Sets 1-3 indicated that increasing the number of filters had a positive effect on overall accuracy, but this stopped being true at the next set (not shown in final notebook).

Another test that was performed was to see if changing only the kernel size for set 3 from 2 to 3 had a positive or negative impact on accuracy. Note: these accuracy results were from interim configurations (not shown in final notebook), so will not correlate to values above.

Kernel size	Set 3 Test Accuracy	Set 3 Test Accuracy
2	84.42% (200 epochs)	81.70% (45 epochs) (86.21% on training at 45 epochs)
3	NA	82.95% (45 epochs) (92.69% on training at 45 epochs)

So while the accuracy for kernel size 2 was higher after 200 epochs, it was lower than for kernel size 3 after 45 epochs. However, the model with kernel size = 3 was much more overfitted to the training set than the one with kernel size of 2. So even though the accuracy was slightly lower, kernel size = 2 still seems like a better model.

Conclusions

Overall, the final model resulted in 84.84% accuracy to the test set, which I feel is pretty strong for what are ultimately a low number of low-resolution images. Given more resources, I’d like to try image augmentation to at least increase the number of images available for training, with the hope that this would have a positive impact to the accuracy of the model. And deeper models could also be attempted. One example I saw achieved 92% accuracy using a network an order of magnitude larger than anything attempted here, mainly due to computational limitations.

There are nearly infinite combinations of hidden layers (including number of layers, filter and kernel sizes for each layer, number of Dense layers and their sizes, learning rates, optimizers, loss functions, and activation types that can be tried, so it is effectively impossible to try them all. Therefore the best that can be hoped for is a model that is reasonably accurate, has as little over- or under-fitting as possible, and is as computationally inexpensive as possible. The intent of the various hyperparameter tunings in this project was to attempt to understand the potential effects that these adjustments can have, with the understanding that this may be completely different for a different data set. That being said, it is helpful to have a process for determining which of those hyperparameters should be tuned for a given data set.

There are examples of models getting as high as 96.53%[1] using more complex techniques such as MXNet Gluon[2] and DropConnect[3], and using image augmentation[4]. There is no DropConnect implementation for Keras at this time, so it would not be feasible for me to implement that. Another option that could get a small bump in accuracy would be to use a Support Vector Machine (SVM) as the final layer activation rather than Softmax. But again, this doesn't appear to have a built-in implementation in Keras.

[1] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#43494641522d3130
[2] https://www.kaggle.com/c/cifar-10/discussion/47798
[3] Regularization of Neural Networks using DropConnect https://cs.nyu.edu/~wanli/dropc/
[4] https://www.kaggle.com/c/cifar-10/discussion/40237

Appendix A:

List of files (model.h5 and model.log)

	Config	**model.h5**	**model.log**	**Accuracy %**	**# epochs**	
Baseline	SGD, lr=.0001	SGD_res.h5	model_SGD.log	47.41	50	
	Adadelta, lr=1.0	AD_res.h5	model_AD.log	47.59	50	
	Adam, lr=.0001	Adam_res.h5	model_Adam.log	47.39	50	
Basic CNN	Adam, lr=.001	model_conv_001.h5	conv_001.log	73.78	100	
	Adam, lr=.0001	model_conv_0001.h5	conv_0001.log	72.23	100	
	Adam, lr=.00001	model_conv_00001.h5	conv_00001.log	73.94	100	
Deep CNN (all use Adam opt.) -- number of epochs vary due to earlystopping callback, batch_size= 128.	filter1=32, lr=.001	model32_001.h5	cnn32_001.log	80.36	51	Set 1
	filter1=32, lr=.0001	model32_0001.h5	cnn32_0001.log	81.44	79	
	filter1=32, lr=.00001	model32_00001.h5	cnn32_00001.log	81.82	70	
	filter1=48, lr=.001	model48_001.h5	cnn48_001.log	82.16	47	Set 2
	filter1=48, lr=.0001	model48_0001.h5	cnn48_0001.log	82.69	59	
	filter1=48, lr=.00001	model48_00001.h5	cnn48_00001.log	82.24	56	
	filter1=64, lr=.001	model64_001.h5	cnn64_001.log	82.82	41	Set 3
	filter1=64, lr=.0001	model64_0001.h5	cnn64_0001.log	82.13	62	
	filter1=64, lr=.00001	model64_00001.h5	cnn64_00001.log	67.35	100	
Deep CNN (all use Adam opt.) -- 200 epochs, batch_size=64	filter1=48, lr=.0001	model48_0001_64.h5	cnn48_0001_64.log	83.87	200	Set 4a
	filter1=64, lr=.001	model64_001_64.h5	cnn64_001_64.log	82.27	200	Set 5
Deep CNN (all use Adam opt.) -- >200 epochs, batch_size=128	filter1=64, lr=.00001	model64_00001_500.h5	cnn64_00001_500.log	80.60	382	Set 6
	filter1=64, lr=.0001	model64_0001_500.h5	cnn64_0001_500.log	84.73	500	Set 7
	filter1=64, lr=.0001 with decay (.000001 per epoch)	model64_0001_500_d.h5	cnn64_0001_500_d.log	83.77	200	Set 8
	filter1=48, lr=.0001	model48_0001_64.h5	cnn48_0001_64.log	84.84	300	Set 4b