Machine Learning and Data Mining
COMP9417: Project Report

Due on Sunday, June 3, 2018

Christopher Pollock

Introduction

The goal of this project was to investigate the use of neural networks for image classification. I wanted to learn about the low-level details of neural network implementations, so rather than use a neural network package such as TensorFlow, I decided to implement my own neural network system from scratch. I give my system the rather whimsical name of *FishNet*. ("Net" is an obvious suffix for a neural network system, and a pollock is a type of fish.)

FishNet is written entirely in C++, using only the standard libraries. I reused a small amount of generic code that I had written for other projects, such as a Log class. The Log class is based on a design that I read about in Dr. Dobbs magazine, but apart from that the code is entirely my own work.

I will not define or explain neural network concepts and terminology in this report. Please refer to the book *Deep Learning* from MIT Press for a good explanation of neural networks. The book is available online at http://www.deeplearningbook.org.

FishNet implements feed forward networks, in which the image data forms the inputs to the first layer of the network, and the outputs of each layer form the inputs to the following layer. The outputs of the final layer form a one-hot representation of the network's prediction of the image class. Advanced architectures such as skip connections and recurrent networks are not supported.

FishNet supports three types of layers - fully connected, convolutional, and max pooling - which can be combined to create arbitrary network designs. The size and number of filters in a convolutional layer are configurable, although filter width and height must be the same. The filter stride and amount of zero-padding are also configurable. Dropout regularization can be used on fully connected layers.

FishNet uses a simple gradient descent algorithm, and provides basic implementations of learning rate decay and weight decay.

Most of the neural network code, for both training and classification, is multi-threaded, so FishNet makes good use of multiple CPU cores. Figure 2 shows how the number of threads affects the time taken for one epoch of training for two different networks. The numbers are from a FreeBSD 11.1 machine with 16GB of RAM and an Intel Xeon E3-1225 V2 running at 3.20GHz. The performance improvement is not linear, because the CPU's four cores have to compete for access to RAM, and small parts of the training algorithm are not multithreaded.

Unfortunately time constraints did not allow me to write code to take advantage of GPUs, so FishNet's calculations are entirely CPU based. I consider this to be the system's most serious shortcoming.

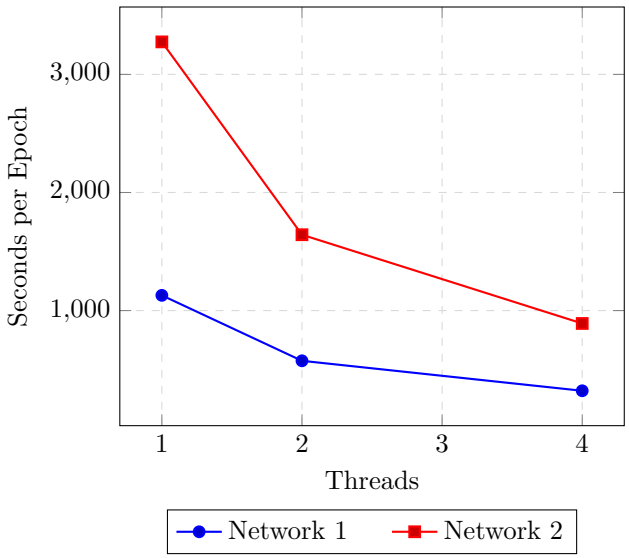

Figure 1: Multithreading Performance

FishNet includes code to load the following image datasets, which I used for testing and experimentation:

- MNIST (http://yann.lecun.com/exdb/mnist/)
- CIFAR-10 (https://www.cs.toronto.edu/ kriz/cifar.html)
- A faces dataset (http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html)

The network code does not make any assumptions about the format of the data, so it should be able to work with any image dataset. The only necessary work would be to write code to load the data and convert it into FishNet's simple internal format. It should even be possible to use FishNet with non-image datasets, although I have not tried this.

Using FishNet

Building FishNet

FishNet should be able to run on any recent version of Microsoft Windows, or most modern Linux or UNIX-like systems. I have successfully run it on Windows 7, Windows 10, Windows Server 2012, Ubuntu Linux 16.04, Ubuntu Linux 18.04, and FreeBSD 11.1. It includes a Solution file for Visual Studio 2017, and Makefiles for Linux or UNIX-like operating systems with version 4.0 or newer of the Clang C++ compiler. It should be possible to build with other modern C++ compilers such as gcc simply by changing the value of *CXX* in the the FishNet/build/common.mk file.

To build on Windows, extract the FishNet directory from the compressed archive, open the FishNet.sln file in Visual Studio, and do a full Release build. The solution includes a suite of tests for the Visual Studio unit testing framework; you can run these to verify that it is working correctly.

To build on a UNIX-like system, simply run the 'make' command inside the FishNet directory.

Obtaining the Data Sets

Before you can use FishNet, you need to obtain the datasets that it will work with. By default, FishNet expects to find the datasets inside the FishNet/datasets directory. This directory contains a script called download_datasets.sh to automatically download and extract the datasets. On most UNIX-like systems, you only need to run the following commands and let the script do the work:

```
$ cd FishNet/datasets
$ ./download_datasets.sh
```

The datasets directory also contains two files called test_faces.txt and training_faces.txt, which are used to divide the faces dataset into training and test sets. If you wish to use a dataset directory in a non-standard location, please be sure to include these files.

Running the Classifier

FishNet uses a command-line program called *Classifier* to train networks, or to test networks that have been previously trained. In training mode, Classifier reads training parameters and network definitions from one or more job definition files, trains the networks, saves the trained networks to files, and saves statistics about the networks to CSV files. In testing mode, Classifier reads one or more previously trained network files, tests them against a specified data set, and outputs statistics on each network. It saves a log file in both modes, so that the user can monitor its progress.

On Linux, the easiest way to run Classifer is to run the FishNet/Classifier.sh script. This sets some environment variables and runs the Classifier binary, which is put in the FishNet/Release/bin directory by the standard build. If you place the FishNet directory directly under $HOME and use the default directories for datasets, logs, and saved networks, this script should work as is. If you wish to use a different configuration, you will need to edit the script accordingly. The environment variables and their meanings are as follows:

FISHNET_LOG_DIR The directory for Classifier to save its log files.

FISHNET_DATA_DIR The directory in which the datasets are stored.

FISHNET_SAVE_DIR The directory for Classifier to save its trained network files and statistics files.

FISHNET_FACE_IMAGE_SIZE The faces dataset contains copies of each image in 3 different sizes - 128x120, 64x60, and 32x30. This selects which size to use - 1 means full-size, 2 means half-size, and 4 means quarter size.

The following command line options are used to control Classifier:

-dataset Specifies the dataset to be used in testing mode. Not used in training mode, because the dataset(s) are specified in the job files.

-dry Does a dry run in training mode. Classifier loads the parameters and network definitions from the job files, constructs the networks, and logs the details, but does not start training. This can be used to verify that the jobs are correct before running them.

-output <file> In testing mode, saves the output to the specified file. If this option is not used, Classifier writes to the standard output.

-test Runs Classifier in test mode. By default it runs in training mode.

-threads <count> Specifies the number of threads to use. By default, Classifier creates one thread per logical processor core, which should give the maximum performance. You may wish to use fewer threads to decrease the load on the machine. Increasing the number of threads is allowed but not recommended, since it should reduce performance.

Any command line argument not beginning with a dash is treated as the name of a network file in testing mode, or a job file in training mode. For example, the following command line would tell Classifier to run the training jobs in the files alpha.csv and beta.csv, using 3 threads:

```
Classifier.sh -threads 3 alpha.csv beta.csv
```

The following command line would tell Classifier to load a network from the file alpha.fish, test it on the CIFAR-10 data set using 4 threads, and save the output to a file called results.csv:

```
Classifier.sh -test -dataset cifar-10 -threads 4 -output results.csv alpha.fish
```

Training Networks

When run in training mode, Classifier first loads the network architectures and training parameters from the specified job file(s). If it detects any errors in the files, it outputs an error message and terminates. Before it starts training, it logs the details of each job, so that the user can verify that they are correct. The jobs are then run sequentially. Each job is divided into a number of training *epochs*, each of which proceeds as follows:

- Randomly shuffle the training images so that they are processed in a different order during each epoch.
- Divide the data into minibatches. Perform the gradient descent algorithm on each minibatch to update the network's weights and biases.
- Test the network's accuracy against the test set.
- Log learning progress.
- Save learning statistics.
- If the network's accuracy has improved, save the network to a file.
- Possibly reduce the learning rate if learning rate decay is in use.

Training can stop either after a set number of epochs, or when the network's accuracy has not improved for a specified number of epochs.

As training proceeds, Classifier automatically saves the network to a series of binary files, which have the extension .fish. To reduce the number of files that need to be saved, it only saves the network when it surpasses its previous highest accuracy. Network accuracy should tend to improve during training, but often fluctuates.

Classifier also saves the training parameters, network architecture, learning statistics, and final test classifications for each job to a CSV file.

Job Files

FishNet uses job files to specify both the architectures of the networks to be trained, and the parameters that should be used to train them. Job files are comma-separated text files, so they can be edited using a spreadsheet program or text editor. A single job file can contain multiple training jobs. Table 1 shows an example of a typical job file as it would appear in a spreadsheet program. This job file sets 8 training parameters, then defines two networks.

You can find many example job files in the FishNet/Jobs directory.

Dataset	CIFAR-10					
Learning Rate	0.1					
Learning Rate Decay	0.05					
Learning Rate Decay Point	0.99					
Weight Decay	0.0008					
Minibatch	16					
Epochs	100					
Give up after	10					
Network	Example1					
Layer	Layer Size	Filter Count	Filter Size	Padding	Dropout	Activation
Convolutional		32	3	1		ReLU
Max Pooling						
Convolutional		64	3	1		ReLU
Max Pooling						
Convolutional		128	3	1		ReLU
Fully Connected	400				0.5	ReLU
Fully Connected	10				0	Sigmoid
Network	Example2					
Layer	Layer Size	Filter Count	Filter Size	Padding	Dropout	Activation
Convolutional		32	3	1		ReLU
Max Pooling						
Convolutional		64	5	2		ReLU
Max Pooling						
Convolutional		128	5	2		ReLU
Fully Connected	400				0.5	ReLU
Fully Connected	10				0	Sigmoid

Table 1: Example Job File

Training Parameters

The syntax for setting training parameters is very simple - the parameter name goes in the first column, and the value in the second. Only a single training parameter can be set in each row. Once a training parameter is defined, it is applied to all the networks that follow it in the file. If you wish to train a number of networks with the same parameters, you therefore only need to set the training parameters once, at the start of the file. You can also set training parameters between network definitions, if you want to use different parameters for different networks.

The following training parameters are available. Parameter names are not case-sensitive. Dataset, Minibatch Size, Epochs, and Learning Rate are mandatory. All other parameters are optional.

Dataset
The name of the data set to train on. The name is not case-sensitive. The currently supported values are "cifar-10", "mnist", "emotions", "face-directions", "people", "sunglasses", and "directions-sunglasses". The first two values correspond to actual data set names; the others load the Faces dataset with the images classified in various ways.

Minibatch Size
This controls how many training examples are processed between each update to the network's weights and biases. For optimal performance, this should be a multiple of the number of threads, so that the work can be divided equally between threads.

Epochs
The maximum number of training epochs for each network.

Learning Rate
Controls how much the network updates its weights and biases in response to each training example.

Learning Rate Decay
Learning rate decay can be used to decrease the learning rate as training proceeds. For example a learning rate decay of 0.05 means to decrease the learning rate by 5% (i.e. multiply it by 0.95). By default the decay is applied at the end of every epoch, but this behaviour can be changed using the Learning Rate Decay Point parameter described below.

Learning Rate Decay Point
Rather than decreasing the learning rate after every epoch, it may be better to only decrease the learning rate when the training error stops decreasing, or is decreasing too slowly. The learning rate decay point specifies a minimum ratio of the current epoch's training error to the previous epoch's training error. After each epoch, the network calculates the error rate ratio, and decreases the learning rate if it is below this minimum. This is best explained by showing some examples:

Learning Rate Decay Point	Condition to decrease learning rate
0.95	Error has decreased by less than 5% since the previous epoch
1.00	Error has not decreased since the previous epoch
1.05	Error has increased by more than 5% since the previous epoch

Values less than 1 should be used with caution. If the error is decreasing too slowly because the learning rate is too low, this will make the problem worse.

Weight Decay
This parameter is used to apply a simple type of weight regularization. Weight decay decreases each weight in the network slightly at the end of each minibatch, before applying the weight updates from gradient descent. Each weight w in the network is reduced according to the formula $w \leftarrow w(1 - \eta d)$, where η is the learning rate and d is the weight decay parameter. The weight decay is scaled by the learning rate so that its effect will remain roughly constant as the learning rate is varied. This is necessary to prevent weight decay from overpowering learning when learning rate decay is in use.

Give Up After
Rather than training for a set number of epochs, it is usually more efficient to stop training when the accuracy of the network stops improving. After every training epoch, FishNet tests the network's classification accuracy against the test set. The *Give Up After* parameter causes FishNet to stop training if the accuracy has not improved for the specified number of epochs. For example, if *Give Up After* is set to 10 and the network reaches a peak accuracy of 80%, it will stop training if the accuracy does not exceed 80% within 10 more epochs.

Network Definitions

The start of a network definition is marked by a row with the word "Network" in the first column. The second column of this row may optionally define a name for the network. The networks in Table 1 are called *Example1* and *Example2*.

The network definition takes the form of a table, with each row representing a layer of the network, and each column representing a layer parameter. The first row of the table consists of column headings, which specify the parameter for each column. The columns can be in any order, and the columns for parameters that are not used can be omitted.

A network consists of a series of layers, with the outputs from each layer forming the inputs to the next. The first layer of the network is at the top of the table, and the output layer is at the bottom. The output layer must have one neuron for every classification in the dataset, and use sigmoid activation. There must be an empty row after the output layer to mark the end of the network definition.

FishNet supports three types of layers - fully connected, convolutional, and max pooling. Most parameters only apply to some layer types, and are ignored for others. FishNet will not allow nonsensical network designs, such as having a convolutional layer after a fully connected layer or having zero padding larger than the filter dimensions.

The following network parameters are available. Parameter names and values are not case-sensitive.

Layer
Specifies the type of layer. It should be "Fully connected", "Convolutional", or "Max Pooling".

Layer Size
Sets the number of neurons in fully connected layers. Not applicable to other layer types.

Filter Count
Sets the number of filters for convolutional layers.

Filter Size
Sets the size (width and height) of the filters in a convolutional layer. Note that FishNet's convolutional layers always use square filters, so the width and height cannot be set separately.

Stride
Sets the stride to use with convolutional layers. By default the filters are applied at every pixel offset within the image, but the stride can be set to only apply it at certain offsets. For example, a stride of 3 only positions the filter at every third pixel.

Padding
The amount of zero padding to use with convolutional layers. This allows filters to be applied while partially outside the image, with the missing pixels replaced by zeros. Padding must be less than the filter size. By default no zero padding is used.

Dropout
Used to apply dropout regularization to fully connected layers. The value should be the probability of a neuron being "dropped", e.g. 0.4 means a 40% chance. Dropout is not currently supported for convolutional layers.

Activation
Sets the activation function to be applied to the layer's outputs. Supported activation functions are "Sigmoid", "TanH", "ReLU", and "Leaky ReLU". Not applicable to max pooling layers.

Leakiness
Sets the leakiness of Leaky ReLU activation, which is the value by which negative inputs are multiplied. It defaults to 0.01. Not applicable to other activation types.

Loading an Existing Network Definition

FishNet allows you to load a network definition file that was saved during a previous training run, and do additional training. To do this, replace the word "Network" in the job file with "Network File", and put the path to the file in the cell that normally contains the network name. The network definition must be omitted, because it will be loaded from the file.

FishNet Performance on Multiple Data Sets

The MNIST Data Set

The MNIST dataset is a collection of 28×28 pixel greyscale images of handwritten digits. It has separate training and test sets of 60,000 and 10,000 images respectively. Classifying the MNIST digits is relatively easy, so it was the first task that I tried while developing FishNet.

Figure 2 shows the accuracy of 6 increasingly complex networks on the MNIST test set. All networks used ReLU activation for all layers except the output layer. They were trained with a small amount of learning decay and no weight decay. The networks had the following layer architectures:

1. Single Layer
    - Fully connected with 10 neurons
2. Two Layer
    - Fully connected with 200 neurons, dropout with 50% probability
    - Fully connected with 10 neurons
3. One Convolutional Layer
    - Convolutional with 16 filters of size 3×3, zero-padding of 1
    - Max pooling
    - Fully connected with 200 neurons, dropout with 50% probability
    - Fully connected with 10 neurons
4. Two Convolutional Layers
    - Convolutional with 16 filters of size 3×3, zero-padding of 1
    - Max pooling
    - Convolutional with 32 filters of size 5×5, zero-padding of 2
    - Fully connected with 200 neurons, dropout with 50% probability
    - Fully connected with 10 neurons
5. Three Convolutional Layers
    - Convolutional with 16 filters of size 3×3, zero-padding of 1
    - Max pooling
    - Convolutional with 32 filters of size 5×5, zero-padding of 2
    - Max pooling
    - Convolutional with 64 filters of size 5×5, zero-padding of 2
    - Fully connected with 200 neurons, dropout with 50% probability
    - Fully connected with 10 neurons

It is immediately obvious that achieving a moderately high accuracy on the MNIST dataset is not difficult. Even the single layer network, which is a linear classifier, reached nearly 92% accuracy. As expected, the deeper networks did better. The network with 3 convolutional layers reached 97.6% after a single epoch of training, and 99.4% after 50 epochs.

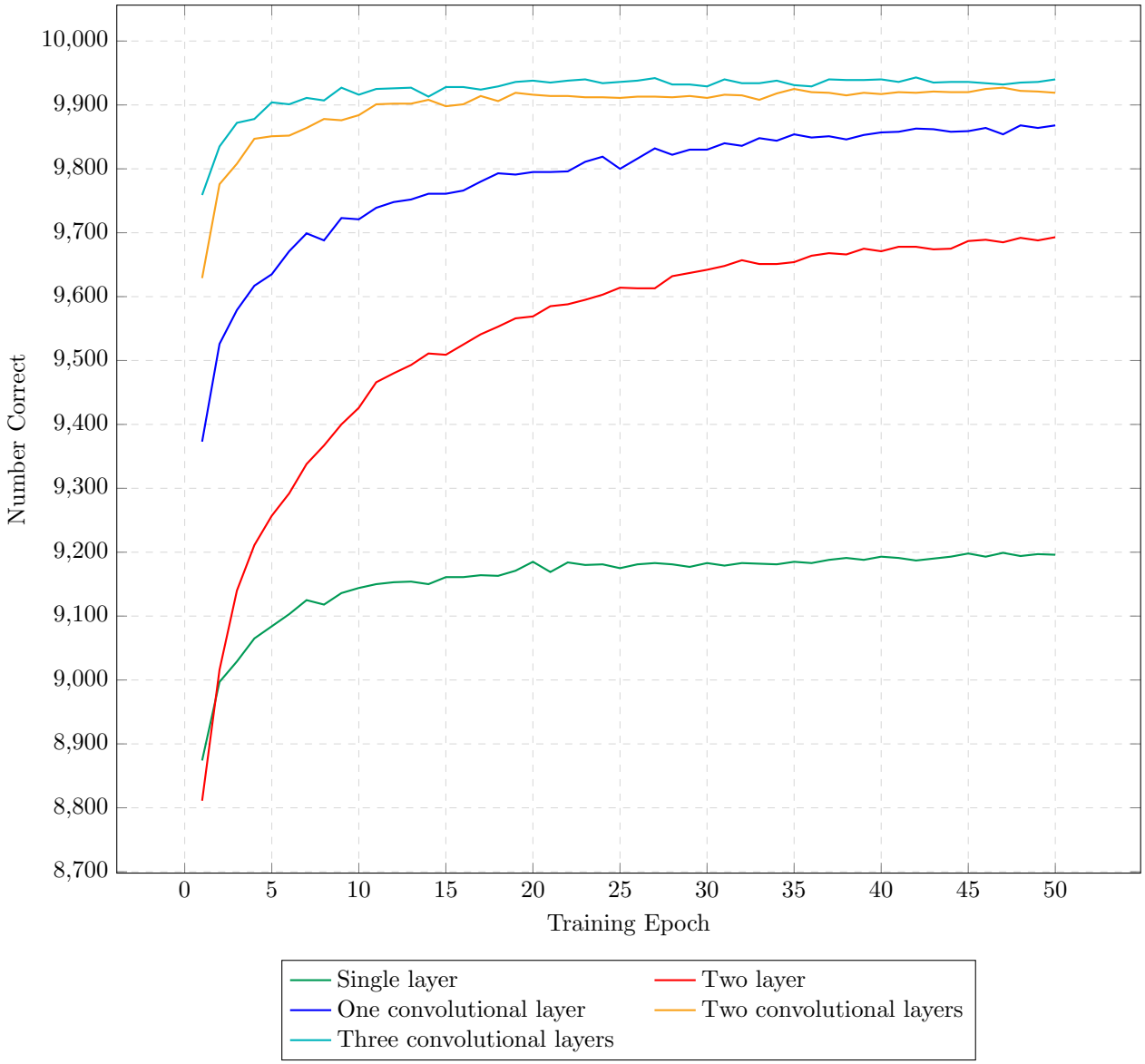

Figure 2: Accuracy on the MNIST dataset

I experimented with four variations of the three layer network, with differences in their first convolutional layers and fully connected layers. The configurations were as follows:

- 16 filters of size 3×3 in the first layer, 200 neurons in the fully connected layer.
- 16 filters of size 3×3 in the first layer, 400 neurons in the fully connected layer.
- 32 filters of size 5×5 in the first layer, 200 neurons in the fully connected layer.
- 32 filters of size 5×5 in the first layer, 400 neurons in the fully connected layer.

Figure 3 shows their accuracies. The differences in accuracy were too small to draw any firm conclusions, but the larger filters and a smaller fully connected seem to slightly more effective. I can only speculate the the larger fully connected layer is more prone to overfitting. The best configuration correctly classified approximately 99.5% of the digits.

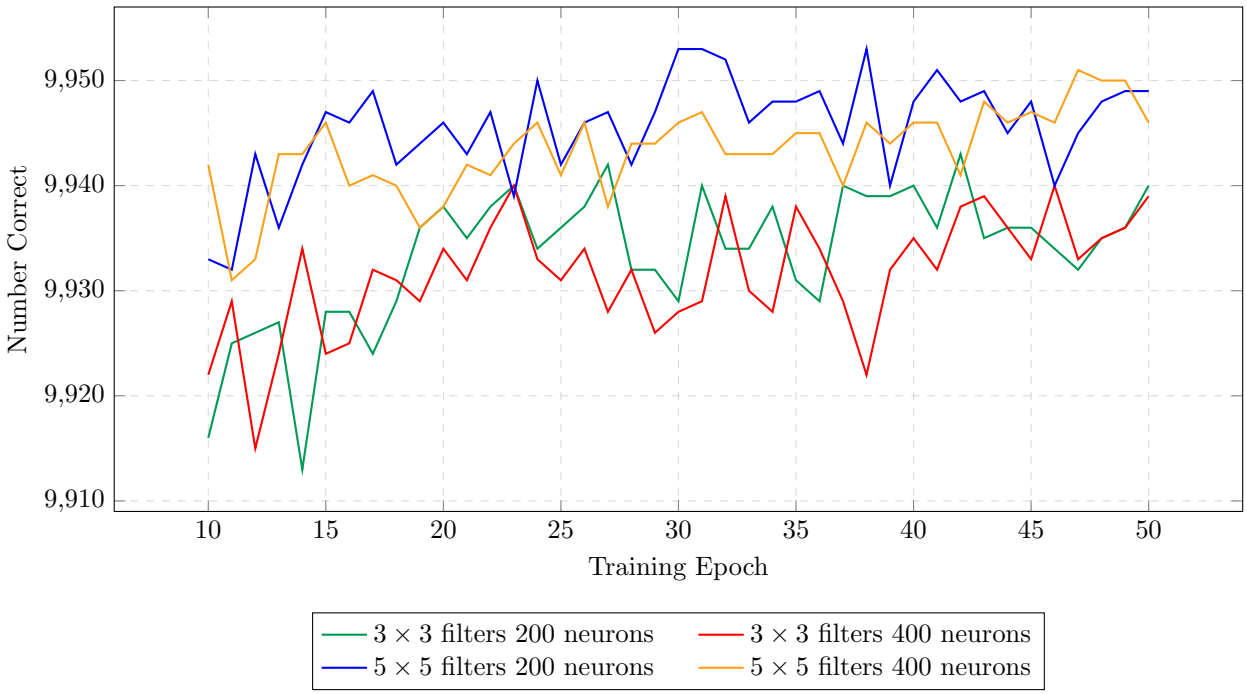

Figure 3: Different configurations of three-layer networks

Dropout and Weight Decay Regularization

Overfitting is a common problem in machine learning, and complex neural networks can be particularly susceptible. In an attempt to alleviate this problem, I implemented dropout regularization on FishNet's fully connected layers. For most of my tests I used dropout with 50% probability on the fully connected layer immediately before the output layer.

I later implemented a simple type of weight decay, but unfortunately I didn't have time to experiment with it as much as I would have liked.

Figures 4 and 5 show the effects of 50% dropout and weight decay of 0.0002 on two three-layer networks. The network in Figure 4 had the following architecture:

- Convolutional with 16 filters of size 3×3, zero-padding of 1
- Max pooling
- Convolutional with 32 filters of size 5×5, zero-padding of 2
- Max pooling
- Convolutional with 64 filters of size 5×5, zero-padding of 2
- Fully connected with 200 neurons
- Fully connected with 10 neurons

The network in Figure 5 was the same except that it had 400 neurons in its first fully connected layer.

The differences are not dramatic, but dropout is definitely helping, especially with the smaller fully connected layer. Weight decay on the other hand seems to make the accuracy slightly worse. Perhaps I could have achieved better results by tuning the weight decay, but time constraints prevented this.

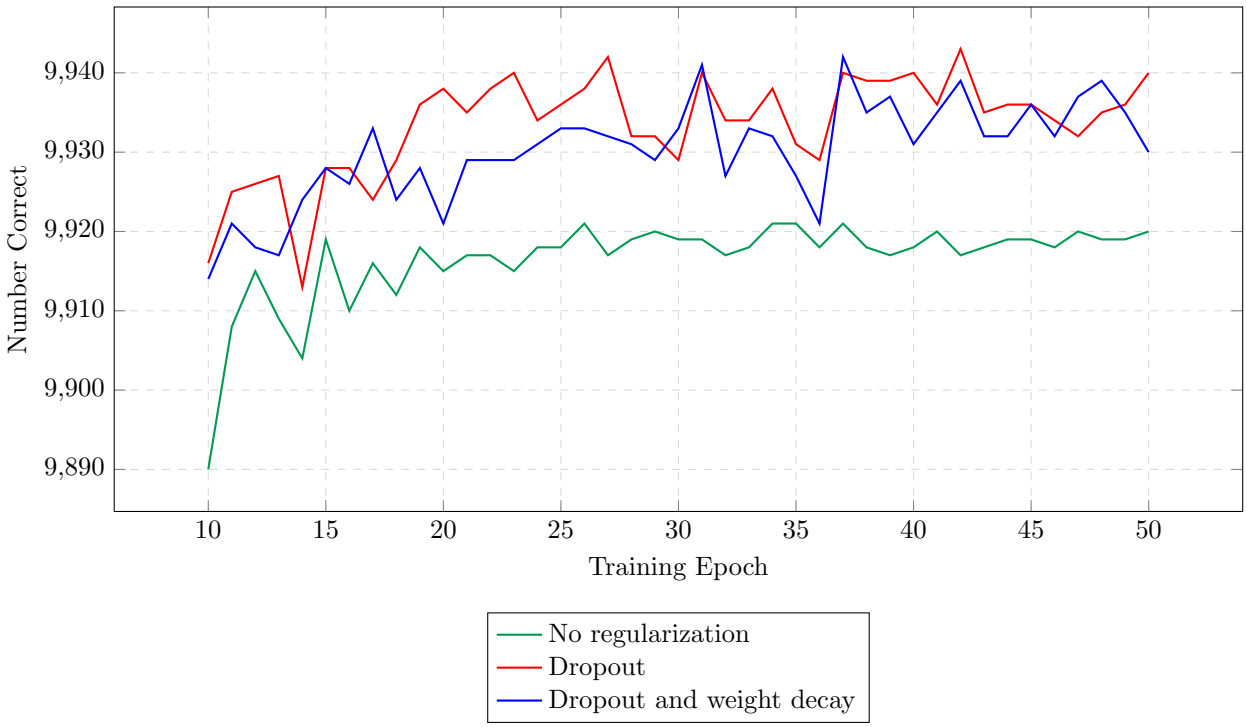

Figure 4: Dropout and weight decay with a 200 neuron fully connected layer

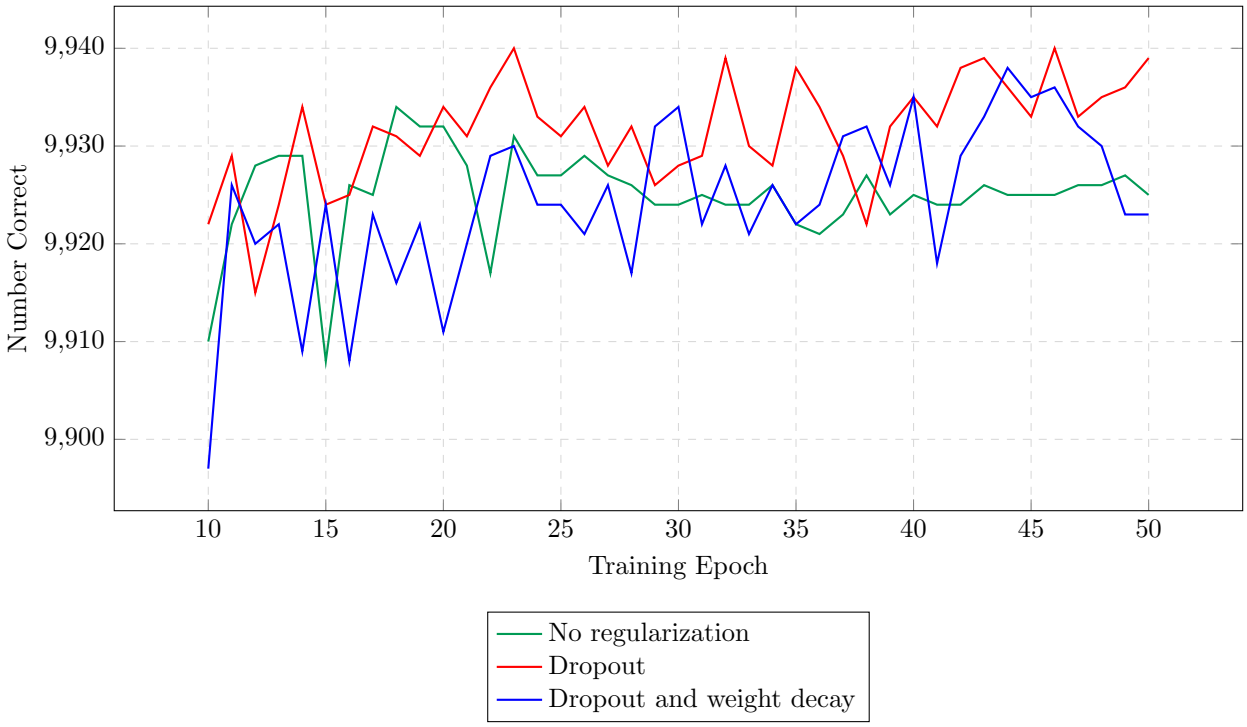

Figure 5: Dropout and weight decay with a 400 neuron fully connected layer

Networks with Four Convolutional Layers

In a final attempt to improve accuracy, I trained two networks with four convolutional layers. The first network had the following architecture:

- Convolutional with 32 filters of size 5×5, zero-padding of 2
- Max pooling
- Convolutional with 32 filters of size 5×5, zero-padding of 2
- Max pooling
- Convolutional with 64 filters of size 5×5, zero-padding of 2
- Convolutional with 64 filters of size 5×5, zero-padding of 2
- Fully connected with 200 neurons, 50% dropout
- Fully connected with 10 neurons

The second network differed only in having 128 filters in the last convolutional layer. I did not use weight decay, since it had not previously helped on MNIST.

Unfortunately these deeper networks failed to match the accuracy of my best network with three convolutional layers, even when given additional training.

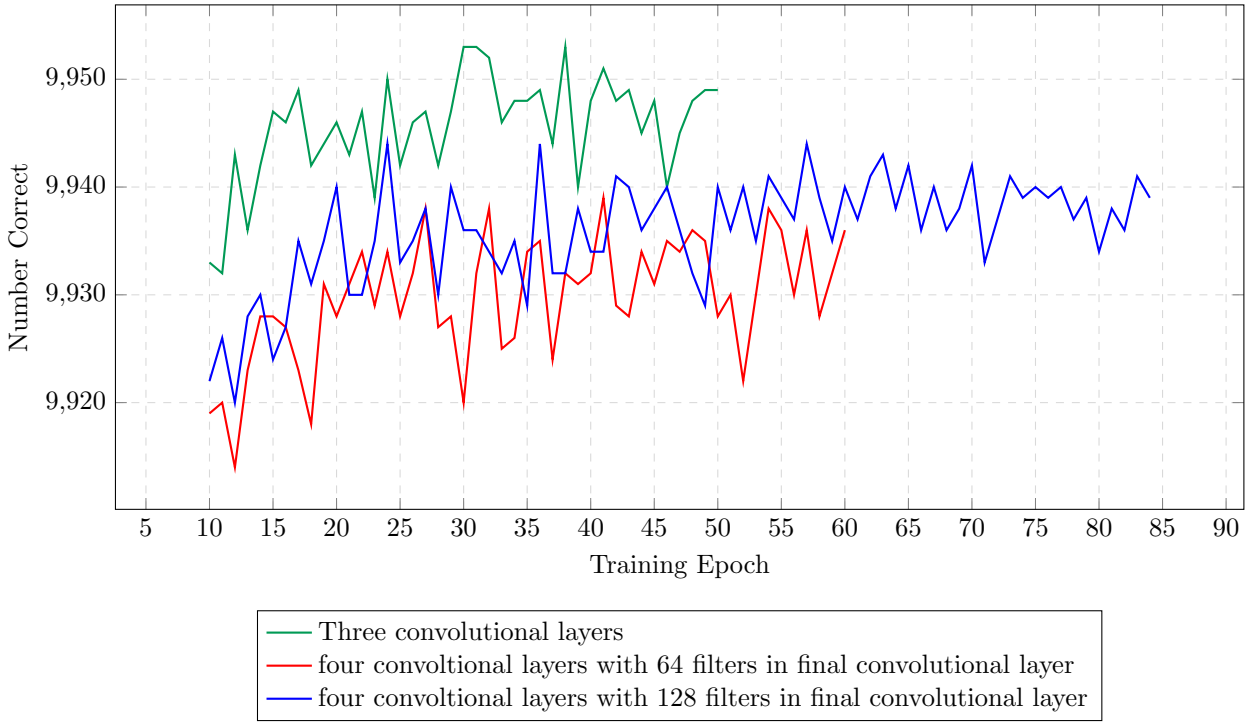

Figure 6: Different configurations of four-layer networks

The CIFAR-10 Data Set

CIFAR-10 consists of 32×32 pixel colour photographs of real objects belonging to 10 classes. It has a training set of 50,000 images and a test set of 10,000 images. CIFAR-10 is a far more challenging classification task than MNIST, but the images are still small enough to allow networks to be trained on modest hardware. I therefore considered it to be an ideal dataset on which to test FishNet.

I tested a wide variety of network architectures against CIFAR-10. I will present my results in order of increasing network complexity, starting with fully connected networks. Figure 7 shows the performance of the following network architectures:

- Single Layer
    1. Fully connected with 10 neurons
- Two Layer
    1. Fully connected with 400 neurons, dropout with 50% probability
    2. Fully connected with 10 neurons
- Three Layer
    1. Fully connected with 400 neurons
    2. Fully connected with 400 neurons, dropout with 50% probability
    3. Fully Connected with 10 neurons
- Four Layer
    1. Fully connected with 400 neurons
    2. Fully connected with 400 neurons
    3. Fully connected with 400 neurons, dropout with 50% probability
    4. Fully Connected with 10 neurons

Accuracy ranged from about 40% for the single layer network to around 55% for the three and four layer networks. Interestingly, four layers did not show any obvious advantage over three layers.

I also tried slight variations of the three and four layer networks which used dropout on all layers except for the output layer. However, this change reduced the accuracy, as can be seen in Figure 8. These networks may have improved with more training, but I knew that to do significantly better I would have to use convolutional networks.

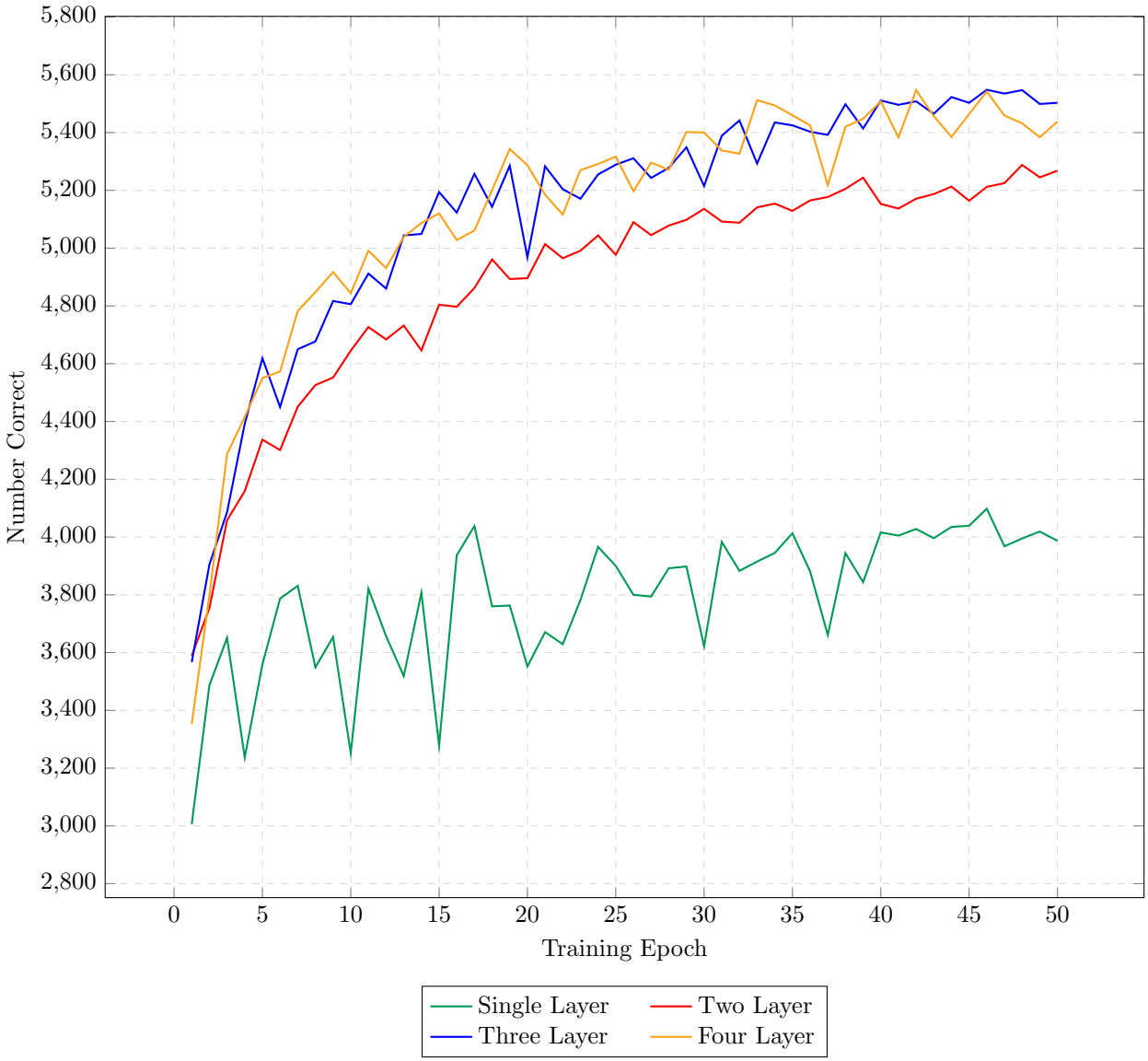

Figure 7: Accuracy of fully connected networks on the CIFAR-10 dataset.

Figure 8: The effect of applying dropout to all layers apart from the output layer

Convolutional Networks

I will now discuss the effectiveness of the following convolutional network architectures of increasing depth:

- Single Convolutional Layer
  1. Convolutional with 32 filters of size 3×3, zero-padding of 1
  2. Max pooling
  3. Fully connected with 400 neurons, 50% dropout
  4. Fully connected with 10 neurons
- Two Convolutional Layers
  1. Convolutional with 32 filters of size 3×3, zero-padding of 1
  2. Max pooling
  3. Convolutional with 64 filters of size 3×3, zero-padding of 1
  4. Max pooling
  5. Fully connected with 400 neurons, 50% dropout
  6. Fully connected with 10 neurons

- Three Convolutional Layers
  1. Convolutional with 32 filters of size 3×3, zero-padding of 1
  2. Max pooling
  3. Convolutional with 64 filters of size 3×3, zero-padding of 1
  4. Max pooling
  5. Convolutional with 128 filters of size 3×3, zero-padding of 1
  6. Max pooling
  7. Fully connected with 400 neurons, 50% dropout
  8. Fully connected with 10 neurons
- Four Convolutional Layers
  1. Convolutional with 32 filters of size 3×3, zero-padding of 1
  2. Max pooling
  3. Convolutional with 64 filters of size 3×3, zero-padding of 1
  4. Max pooling
  5. Convolutional with 128 filters of size 3×3, zero-padding of 1
  6. Convolutional with 128 filters of size 3×3, zero-padding of 1
  7. Max pooling
  8. Fully connected with 400 neurons, 50% dropout
  9. Fully connected with 10 neurons

Figure 9 shows the accuracy of the four networks. Unsurprisingly, the deeper networks achieved better accuracy, with the four layer network reaching approximately 77.5%.

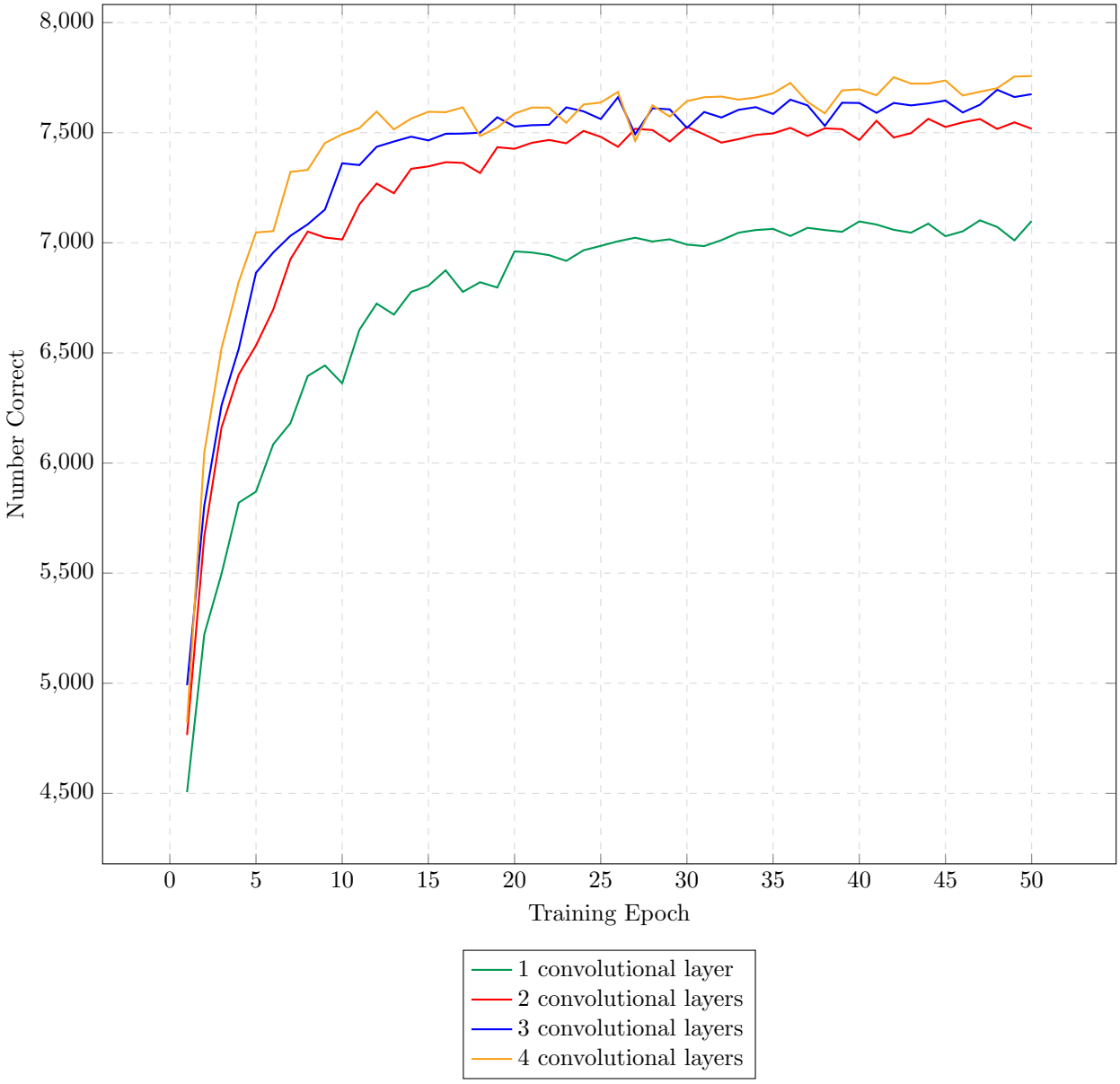

Figure 9: Accuracy of convolutional networks on the CIFAR-10 dataset.

Dropout and Weight Decay Regularization

Figure 10 shows a comparison of the three convolutional layer network trained without regularization, with dropout, and with both dropout and weight decay of 0.0004. Once again dropout improved the accuracy somewhat. Contrary to the MNIST results, weight decay also seems to have helped slightly.

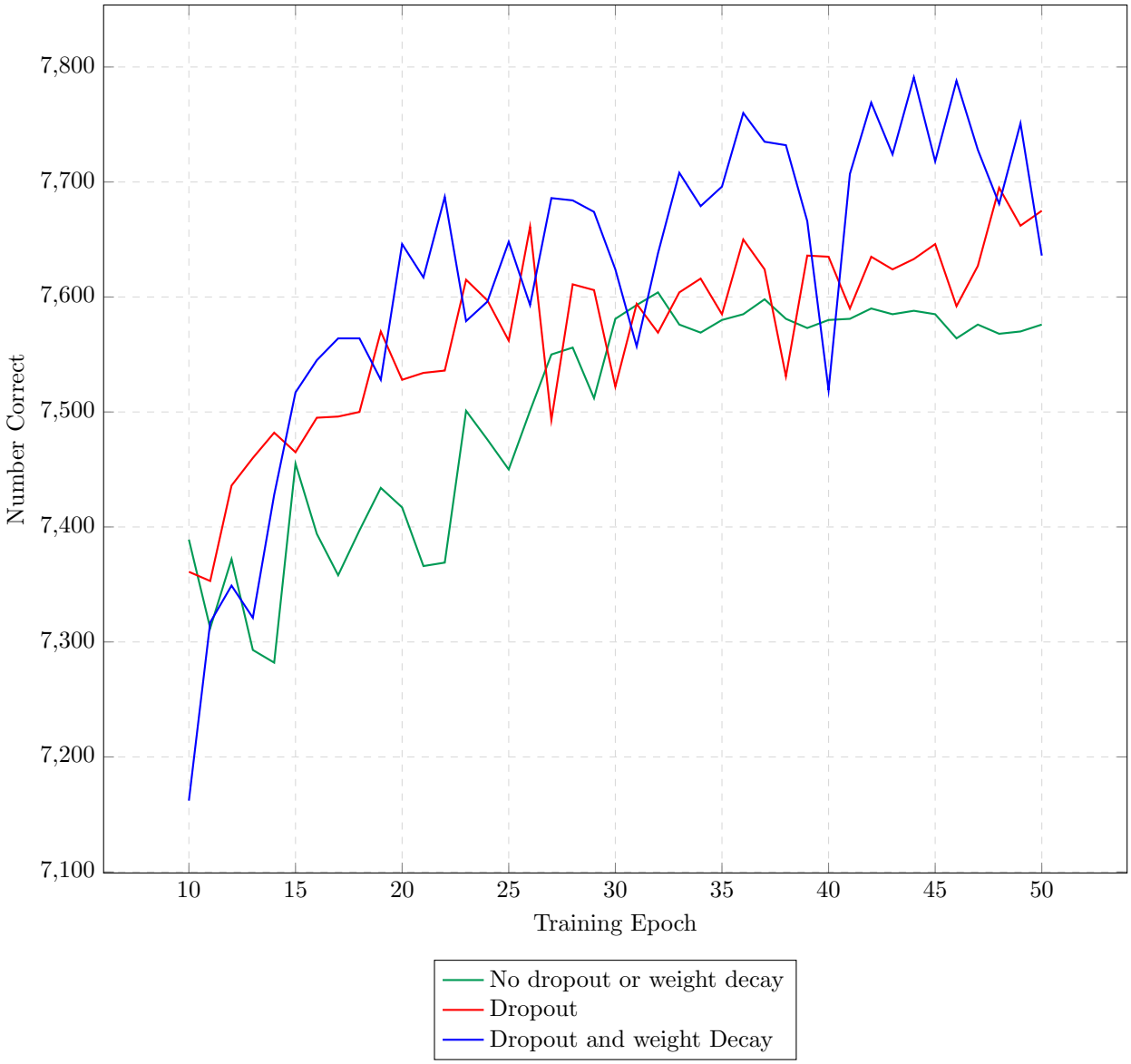

Figure 10: Effects of dropout and weight decay on network with 3 convolutional layers

Figure 11 shows a comparison of the four convolutional layer network trained with only dropout, and with both dropout and weight decay of 0.0004. Weight decay seems to have helped this network too, allowing it to touch 79% accuracy. Careful tuning of weight decay and more training iterations might have improved the accuracy even further, but unfortunately time did not permit this.

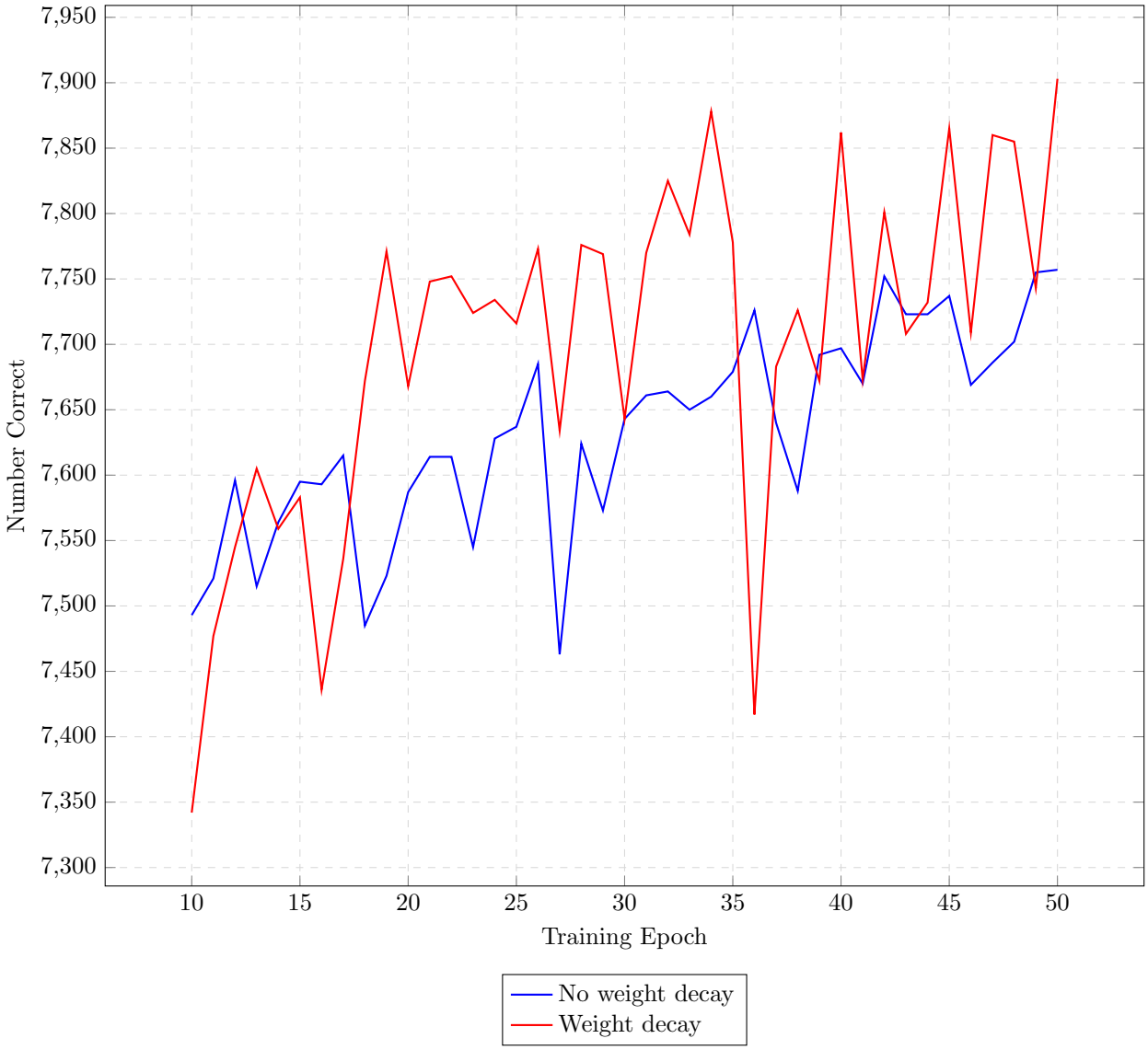

Figure 11: Effect of weight decay on network with 4 convolutional layers

Larger Filters on the Three Convolutional Layer Network

I tried using 5×5 filters in network with three convolutional layers instead of 3×3, but these gave slightly worse accuracy, as Figure 12 shows. I used weight decay of 0.0004 for this test.

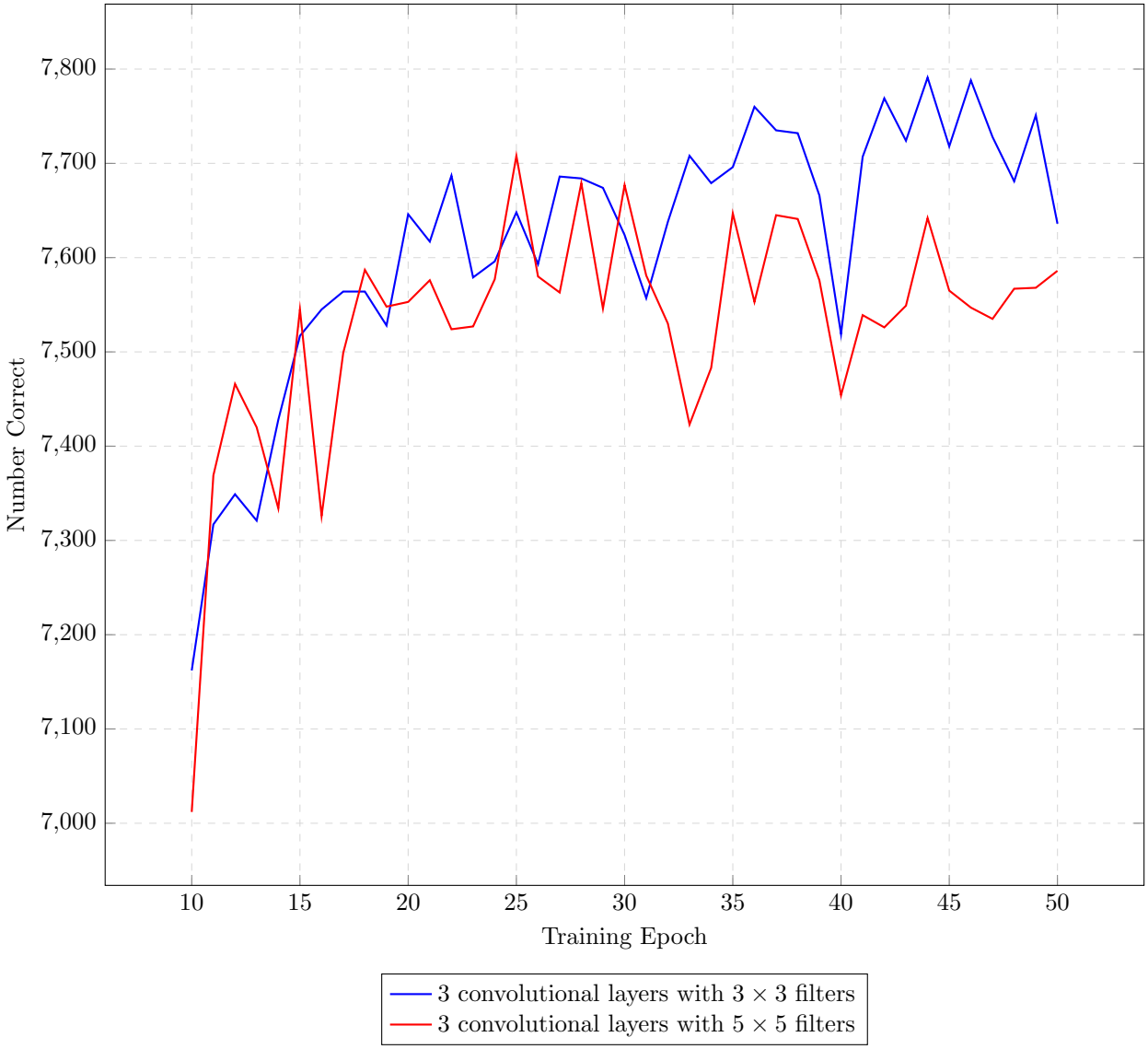

Figure 12: 3×3 filters vs. 5×5 filters on the three layer network

A Larger Four Convolutional Layer Network

I tested a variation of the four convolutional layer network with 5×5 filters on the second, third, and fourth convolutional layers, and 800 neurons in the fully-connected layer. Figure 13 shows its accuracy compared to the original network. The results are similar, but the original design is perhaps slightly more accurate, as well as being less computationally costly. Both networks were trained with weight decay.

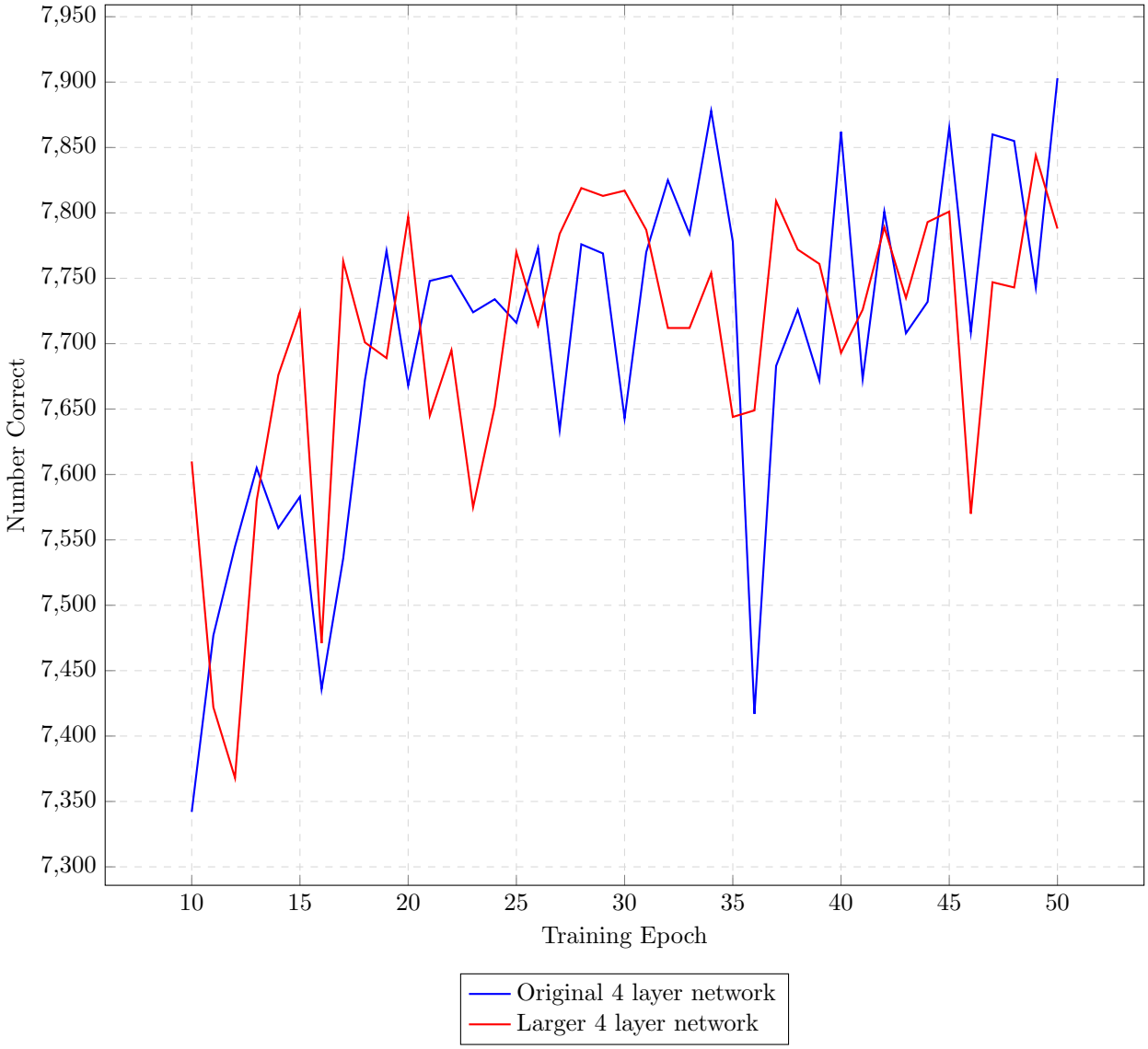

Figure 13: Variations of the four layer network.

Varying the Activation Function

I tested various activation functions on the three layer network - Sigmoid, ReLU, and Leaky ReLU with leakiness of 0.01 and 0.02. I did not use weight decay for this test. Figure 14 shows the results. I knew that sigmoid activation causes the gradients to become very small after a few layers, leading to slow training, and my test clearly confirmed this. The Sigmoid network was still learning and would presumably have improved if trained for longer, but it was clearly inferior to the ReLU networks.

The ReLU networks had very similar accuracy, but leaky ReLU seems to show a slight advantage. The level of leakiness did not make an obvious difference. This would be a promising area for future exploration.

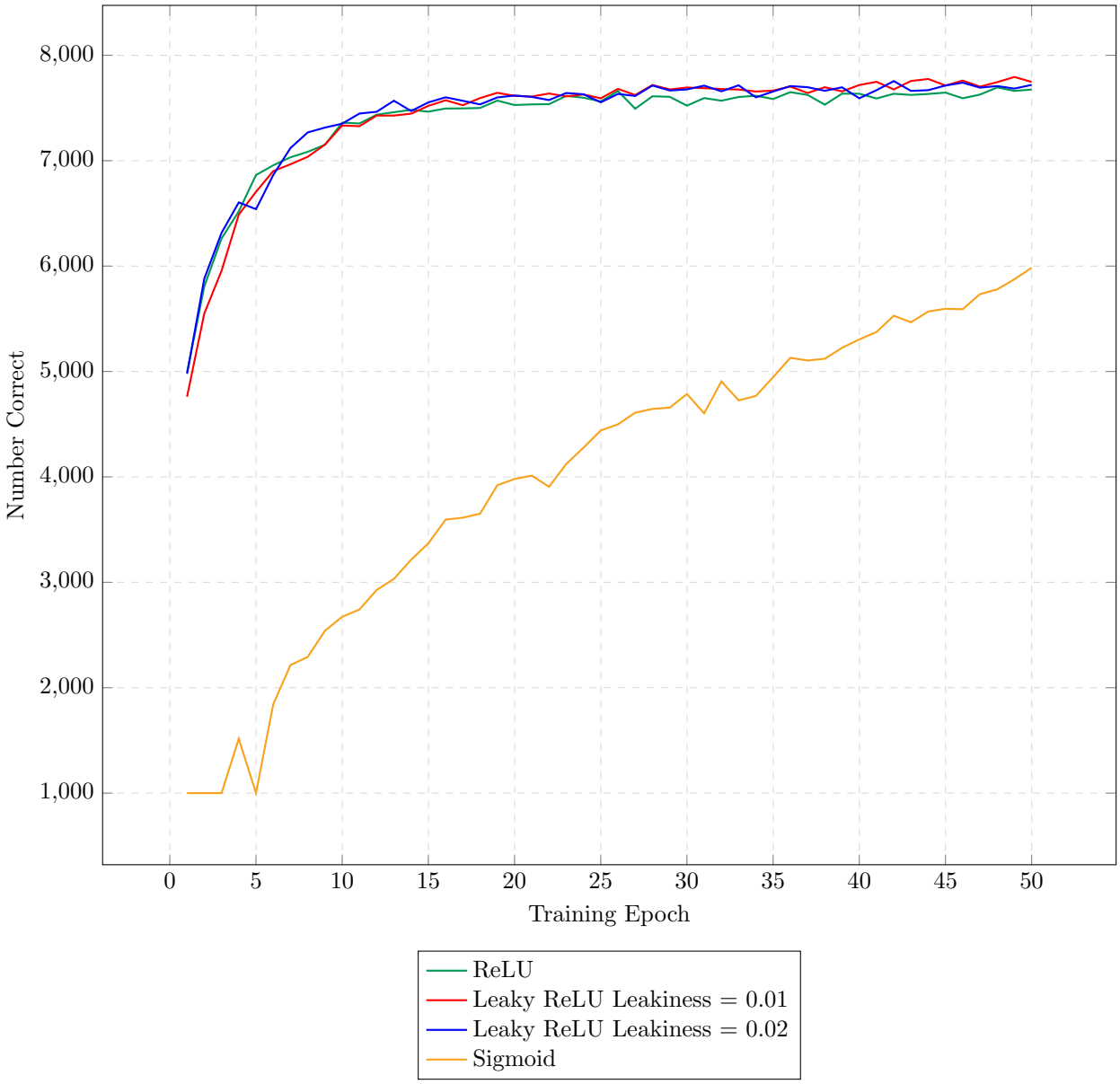

Figure 14: Different activation functions on a 3 convolutional layer network.

Increasing the Stride

I tried using convolution with a stride of 2 rather than a stride of 1 followed by 2×2 max pooling. A larger stride has the same effect of reducing the size of the input to the following layer, but is less computationally expensive because it only applies the filters at every second pixel location. It also avoids the cost of the max pooling calculations. However, because because it applies the filter more coarsely, I expected it to give poorer results.

I trained two networks with three convolutional layers containing 32, 64, and 128 filters of size 3×3, followed by a fully connected layer of 400 neurons with 50% dropout. Both networks used a small amount of learning rate and weight decay. The only difference was that one network used a stride of 1 followed by max pooling, and the other used a stride of 2.

As I expected, a stride of 2 was decidedly inferior. A different configuration, such as larger filters, may yield better results for striding, but I did not have time to investigate this further.

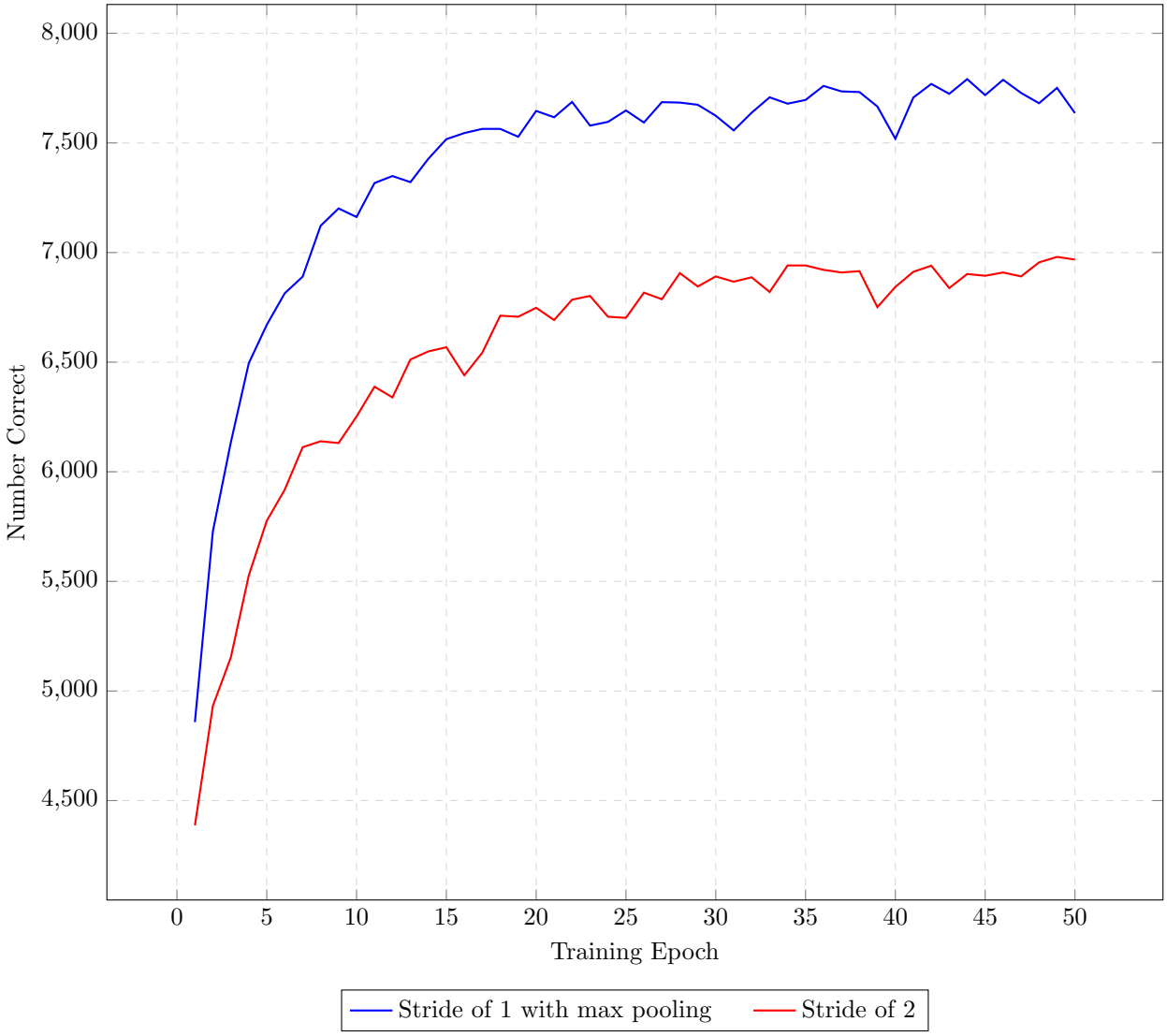

Figure 15: Stride of 2 vs. stride of 1 with max pooling

The Faces Data Set

The faces data set consists of 624 greyscale images of 20 people facing in various directions, with various facial expressions. The subjects are wearing sunglasses in about half the images. The data set contains copies of each image in three different sizes - 128×120, 64×60, and 32×30. FishNet can be configured to use any of these sizes, as previously explained. I chose to use the 64×60 images to reduce the computational costs.

I reserved 100 randomly selected images for a test set, leaving me with a training set of only 524 images. This is a very small number by modern standards - about 100 times smaller than MNIST and CIFAR-10. This provides an interesting opportunity to see how well the system learns on such a small training set.

The images can be classified in various ways. I attempted to train networks for the following five classifications:

- The direction that the person is facing - left, right, straight, or up
- Whether or not the person is wearing sunglasses
- The direction that the person is facing, and whether or not the person is wearing sunglasses
- The identity of the person
- The facial expression - angry, happy, neutral, or sad

I did not have time to design specialised networks for each classification task, so I used the same three convolutional networks of increasing depth for all five tasks. I will refer to them as the shallow, medium, and deep networks. (Even the "deep" network is shallow by modern standards, but the terms are relative.)

- Shallow Network
    1. Convolutional with 32 filters
    2. Max pooling
    3. Convolutional with 64 filters
    4. Max pooling
    5. Fully Connected with 400 neurons
    6. Fully Connected with one neuron for each category
- Medium Network
    1. Convolutional with 32 filters
    2. Max pooling
    3. Convolutional with 64 filters
    4. Max pooling
    5. Convolutional with 64 filters
    6. Fully Connected with 400 neurons
    7. Fully Connected with one neuron for each category

- Deep Network
    1. Convolutional with 32 filters
    2. Max pooling
    3. Convolutional with 64 filters
    4. Max pooling
    5. Convolutional with 64 filters
    6. Convolutional with 64 filters
    7. Fully Connected with 400 neurons
    8. Fully Connected with one neuron for each category

The network architectures and training parameters were the same apart from the number of convolutional layers. The convolutional layers used 5×5 filters, a stride of 1, and zero-padding of 2. The first fully connected layer used dropout with 50% probability. All layers used ReLU activation, except the output layer, which used sigmoid. The learning rate was 0.05, and the learning rate decay point was 0.995. I did separate training runs without weight decay, and with a weight decay of 0.0004.

Determining Face Direction

All three networks did quite well at determining face direction. The final accuracy was the same both with and without weight decay, but the networks with weight decay made somewhat smoother progress, so I have shown them in Figure 16.

The deep network correctly classified 97 out of 100 test images, and the other two networks managed 96. This difference is not statistically significant. I trained for 100 epochs, but the accuracy did not improve beyond this point.

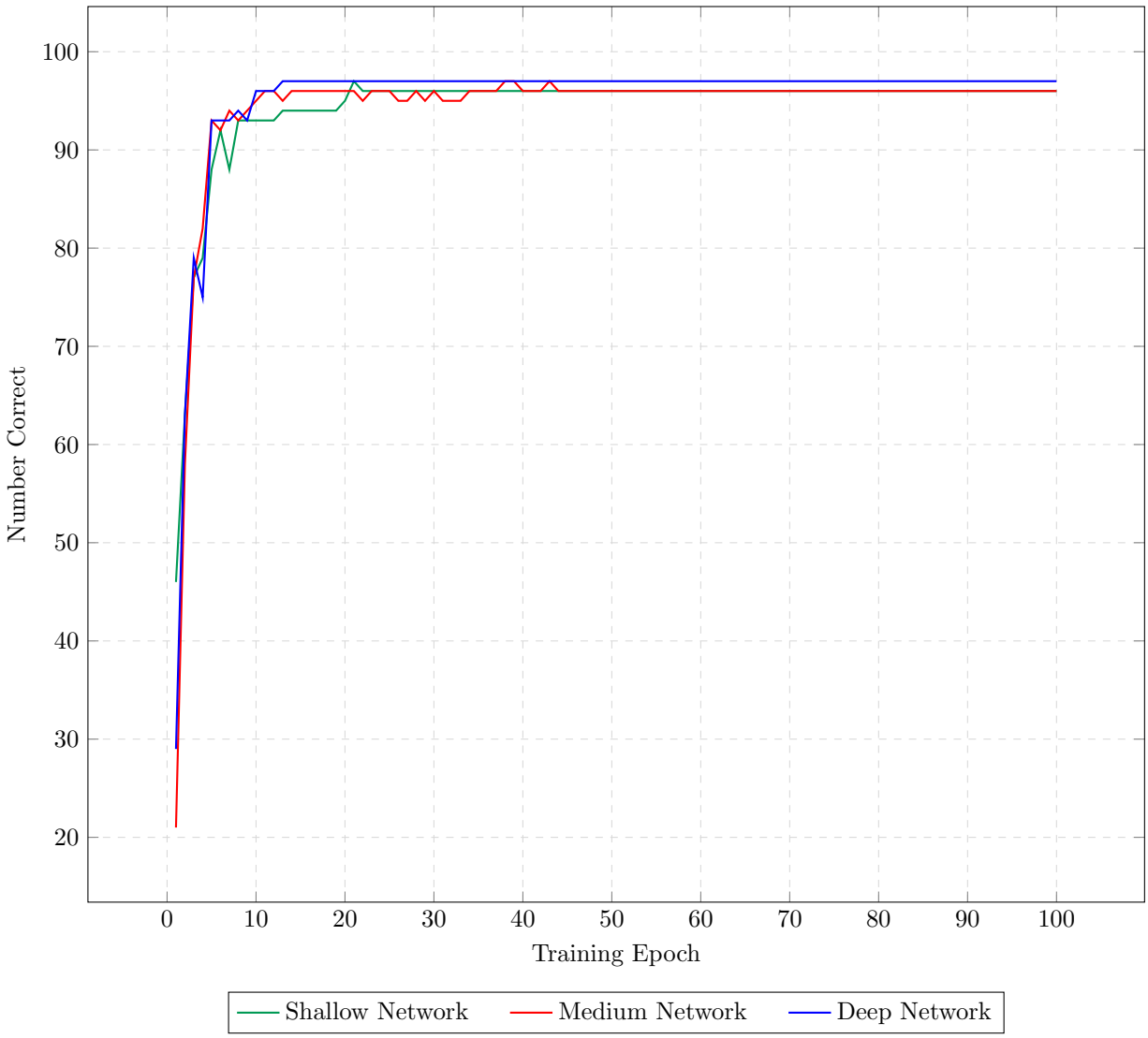

Figure 16: Face direction

Detecting Sunglasses

The networks also achieved moderately good accuracy at detecting sunglasses, as shown in Figures 17 and 18. The maximum accuracy was 92% both with and without weight decay. However, with weight decay the medium network beat the deep network, and without weight decay the deep network beat the medium network. Given the small number of training and test examples I don't think any conclusion can be drawn from this observation.

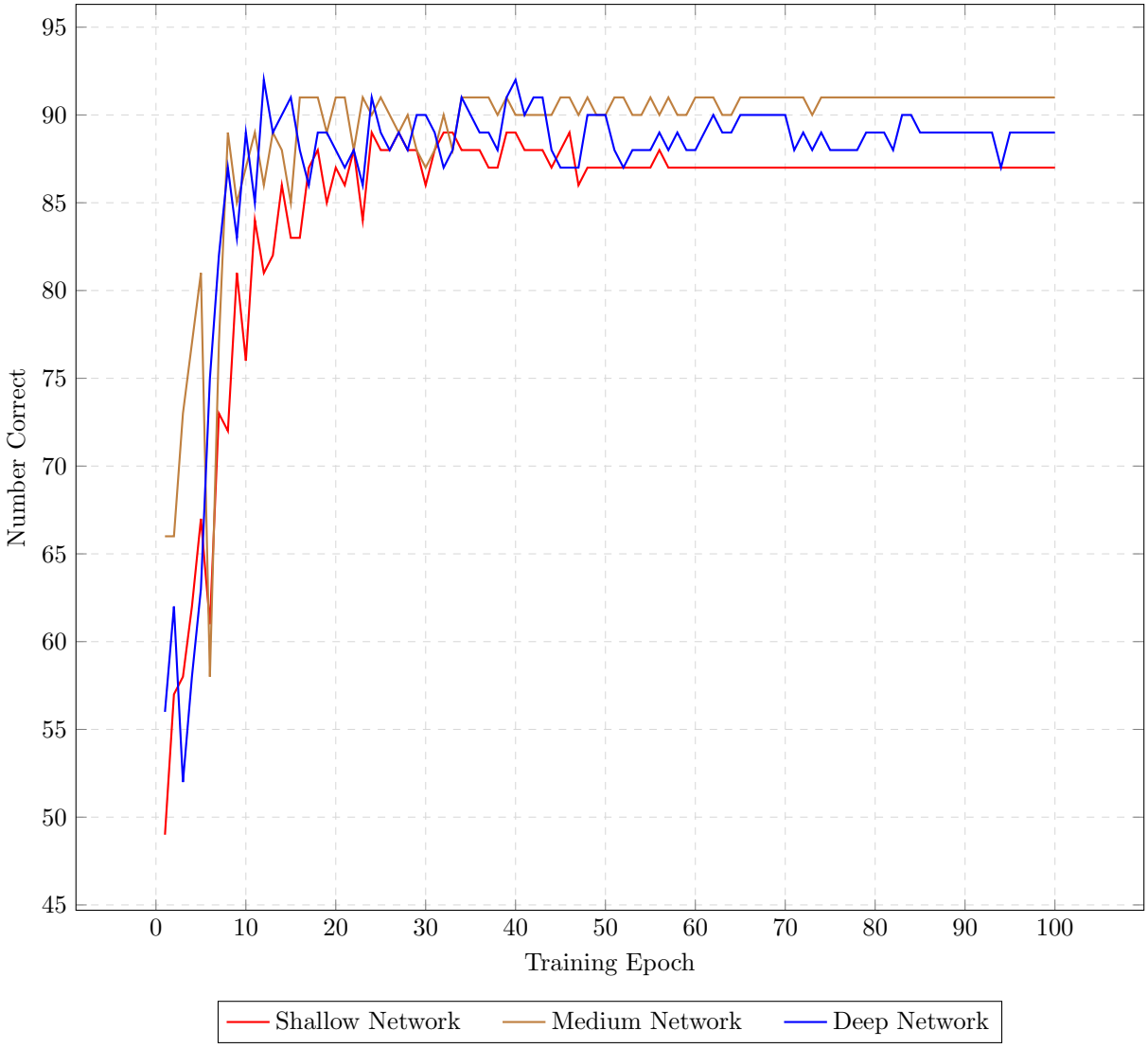

Figure 17: Sunglasses (with weight decay))

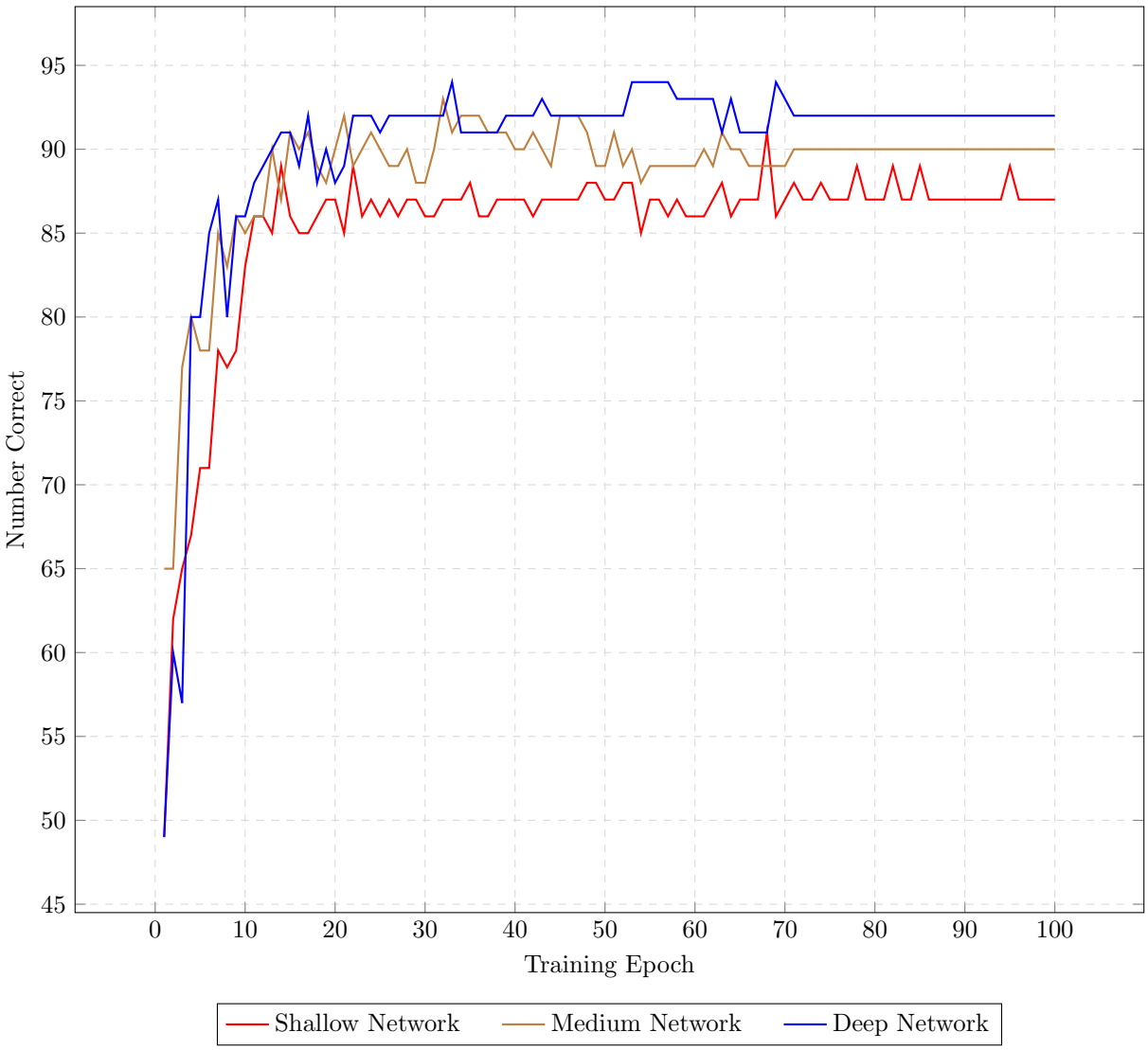

Figure 18: Sunglasses (without weight decay)

Detecting Face Direction and Sunglasses

This task divides the images into 8 classes (4 directions with and without sunglasses), so it is inherently more difficult than the previous tasks. Figures 19 to 21 show the results of three training runs, two with weight decay and one without.

As can be seen in Figure 20, the deep network failed to learn at all on the second run with weight decay. Learning seems to be quite unstable on this data set, which is not surprising given the small number of training examples. On this occasion the network somehow got into a bad state from which it was unable to recover. Note that the same network design worked quite well during the first training run, which used identical parameters apart from the number of epochs. Presumably the difference was due to different weight initializations, and the order in which it trained on each example, which are both randomized.

The best accuracy was achieved by the medium network with weight decay, which touched 90%, but the numbers are too small to draw any conclusions about relative performance.

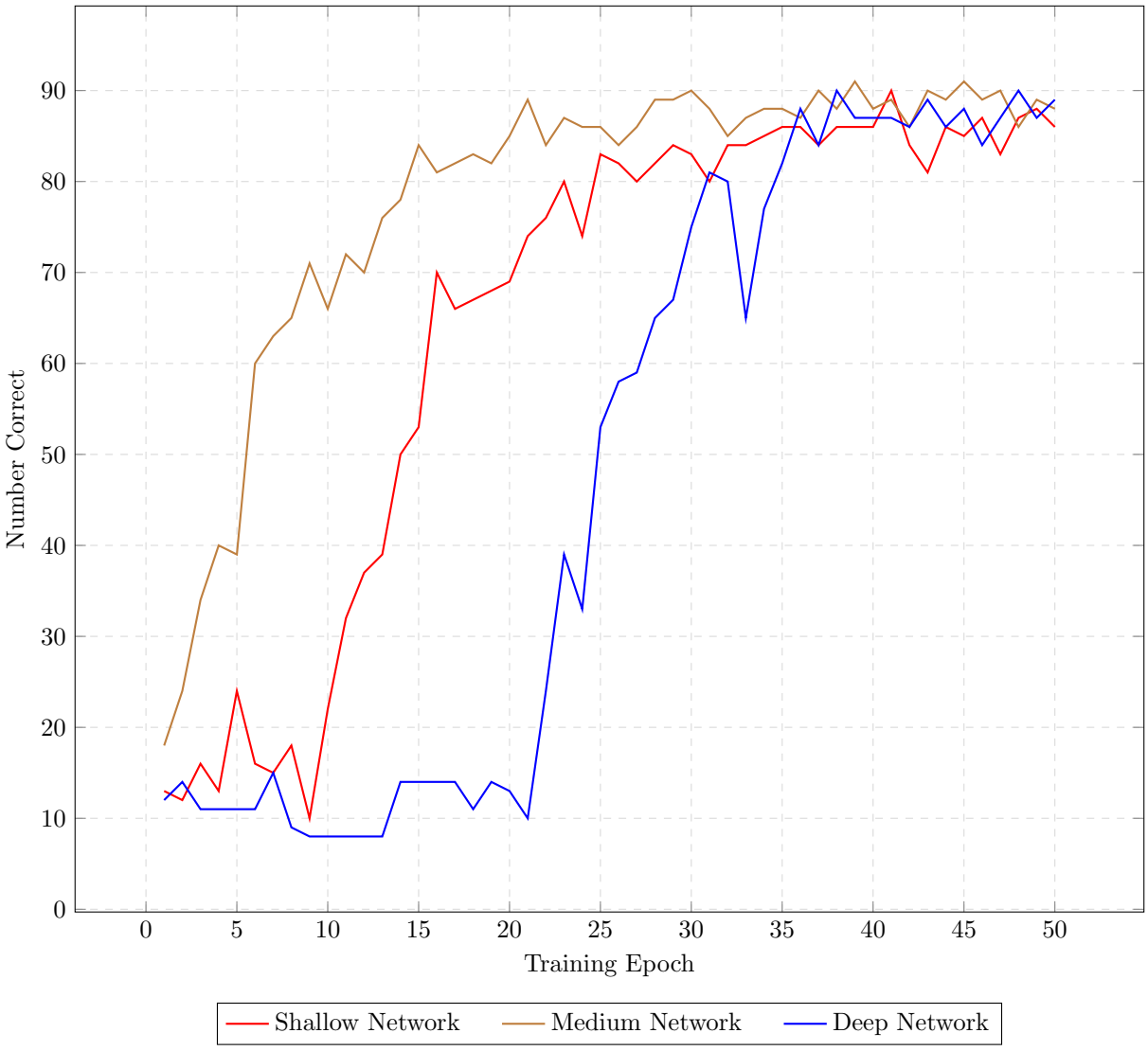

Figure 19: Face direction and sunglasses (50 epochs with weight decay)

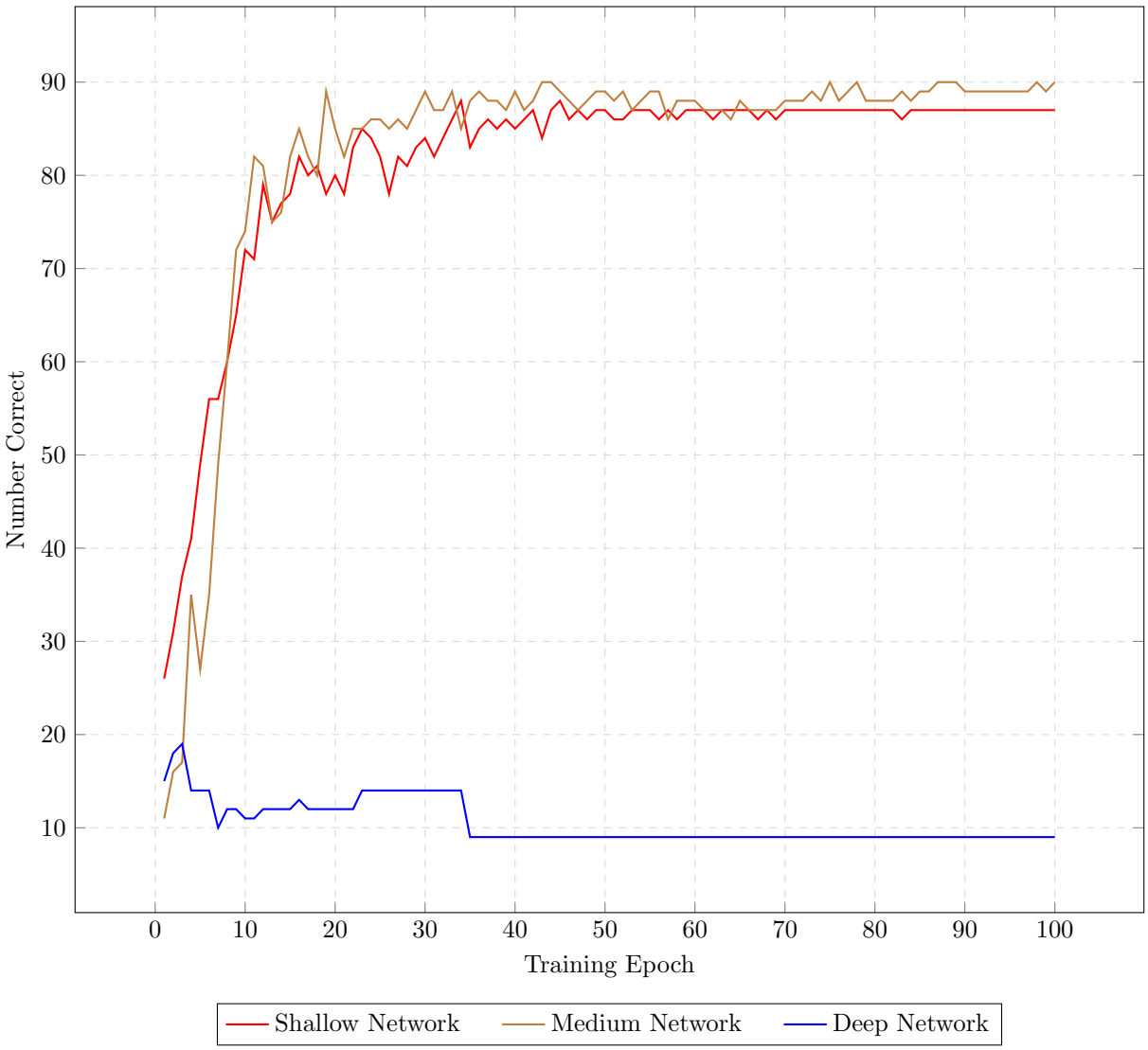

Figure 20: Face direction and sunglasses (100 epochs with weight decay)

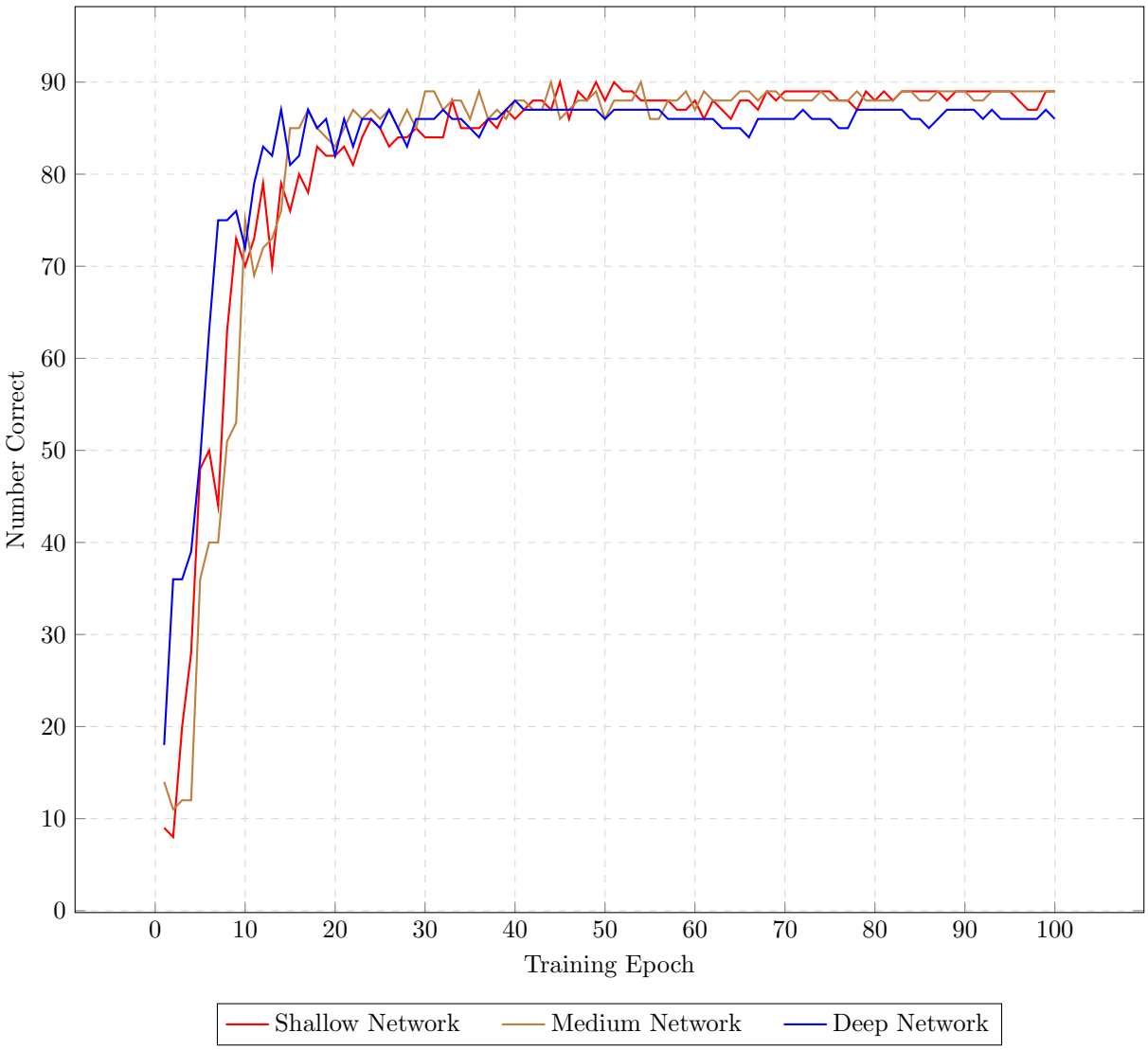

Figure 21: Face direction and sunglasses (100 epochs without weight decay)

Recognising Faces

Figures 22 to 24 show three training runs, two with weight decay and one without. They provide further evidence of unstable learning on this data set. The shallow network learned to classify the faces with 100% accuracy on all three runs, but the number of epochs that it required to do so varied greatly. The deep network eventually started to learn well on the second run, and might have reached 100% with a few more epochs of training. In all other cases the shallow and deep networks failed to learn significantly.

The impressive performance of the shallow network was surprising and rather puzzling. This is the only task at which a network ever learned to classify the data perfectly. I examined some of the images visually, and noticed that there is a lot of background detail. The backgrounds seem to be the same for all shots of the same person, but vary between people. I therefore suspect that the network learned to recognise the backgrounds rather than the faces.

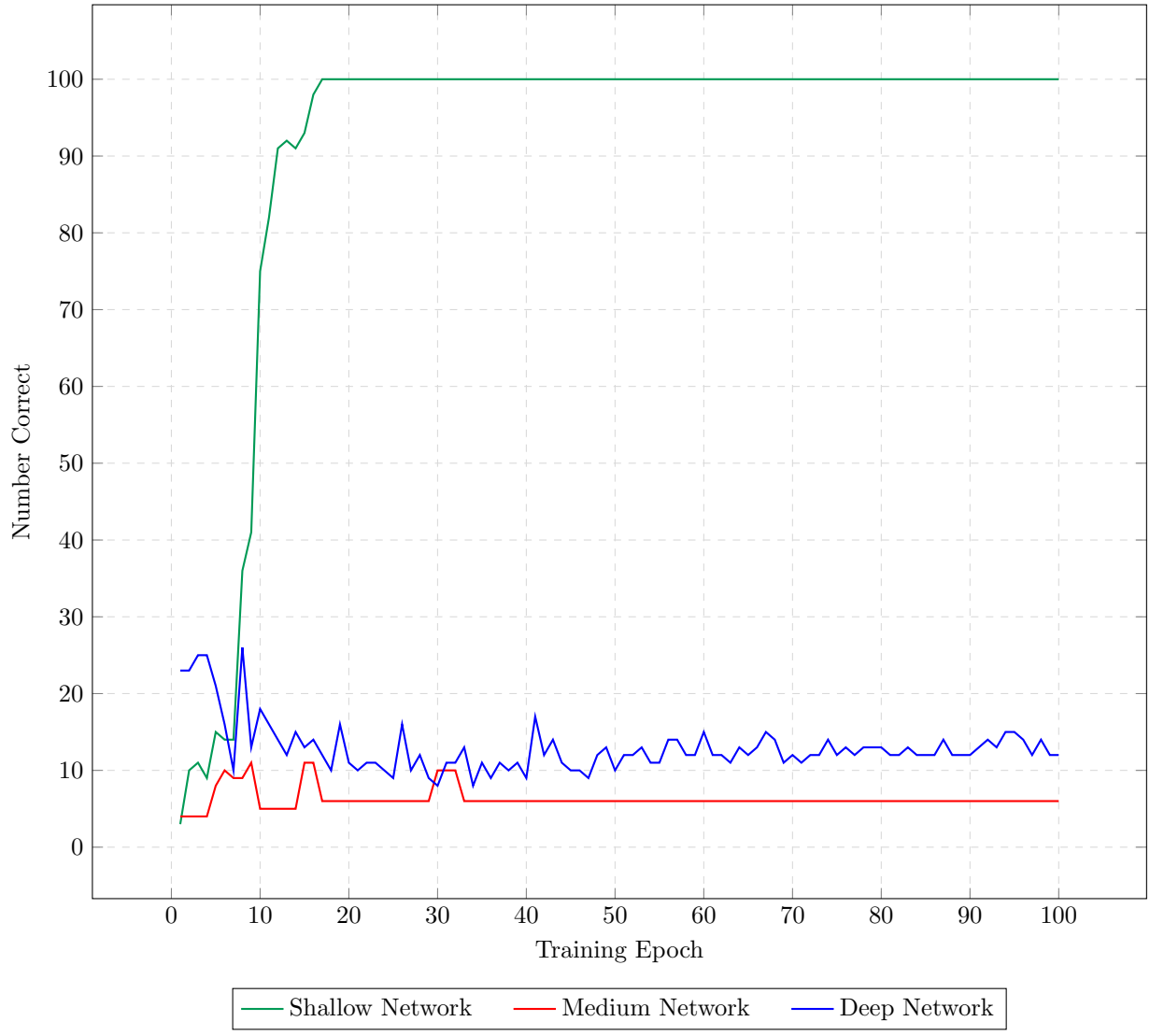

Figure 22: Recognising faces (first run with weight decay)

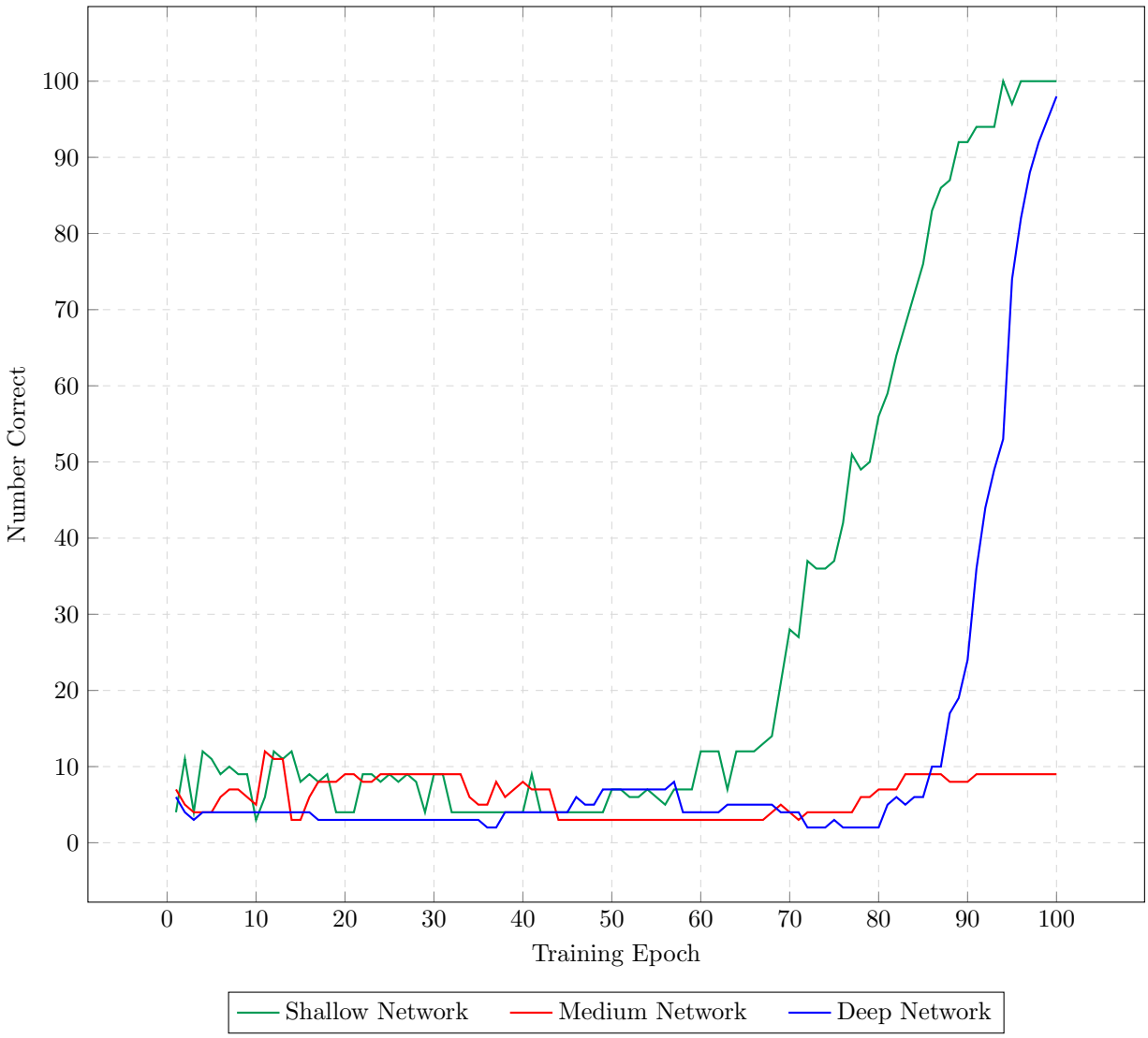

Figure 23: Recognising faces (second run with weight decay)

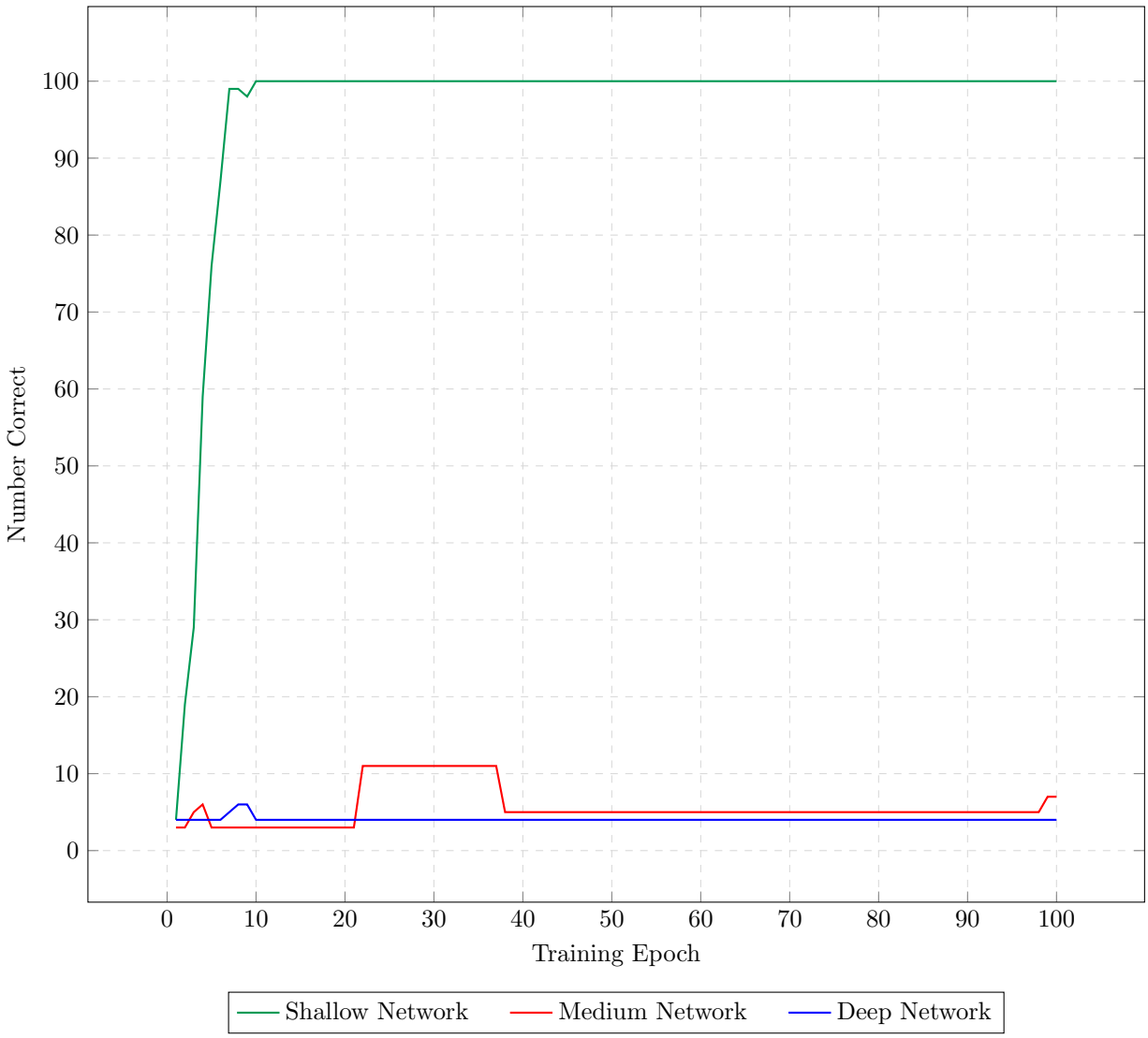

Figure 24: Recognising faces (without weight decay)

Classifying Facial Expressions

Given the subtleness of human facial expressions compared to features such as face direction or the presence of sunglasses, the small size of the training set, and the many distracting features, I did not expect my relatively simple networks to have any success on this task. Furthermore, the (presumably acted) expressions of the subjects did not look very realistic to me. I suspect that the subjects were not skilled actors.

As I expected, the networks failed miserably, as can be seen in Figure 25. The only surprise was that as training progressed, the accuracy became significantly *worse* than would be expected by chance. I saw the same effect on all three networks, both with and without weight decay. Figure 25 shows the results without weight decay.

Tables 2 to 4 show the predicted classes from the networks at the end of training. I am unable to think of an explanation for these strange results.

Actual Category	Predicted angry	Predicted happy	Predicted neutral	Predicted sad
angry	5	3	4	13
happy	9	3	10	6
neutral	4	12	2	5
sad	11	5	8	0

Table 2: Expressions predicted by the shallow network

Actual Category	Predicted angry	Predicted happy	Predicted neutral	Predicted sad
angry	3	3	7	12
happy	4	4	11	9
neutral	4	10	4	5
sad	10	6	7	1

Table 3: Expressions predicted by the medium network

Actual Category	Predicted angry	Predicted happy	Predicted neutral	Predicted sad
angry	2	6	7	10
happy	6	5	10	7
neutral	5	12	0	6
sad	9	8	6	1

Table 4: Expressions predicted by the deep network

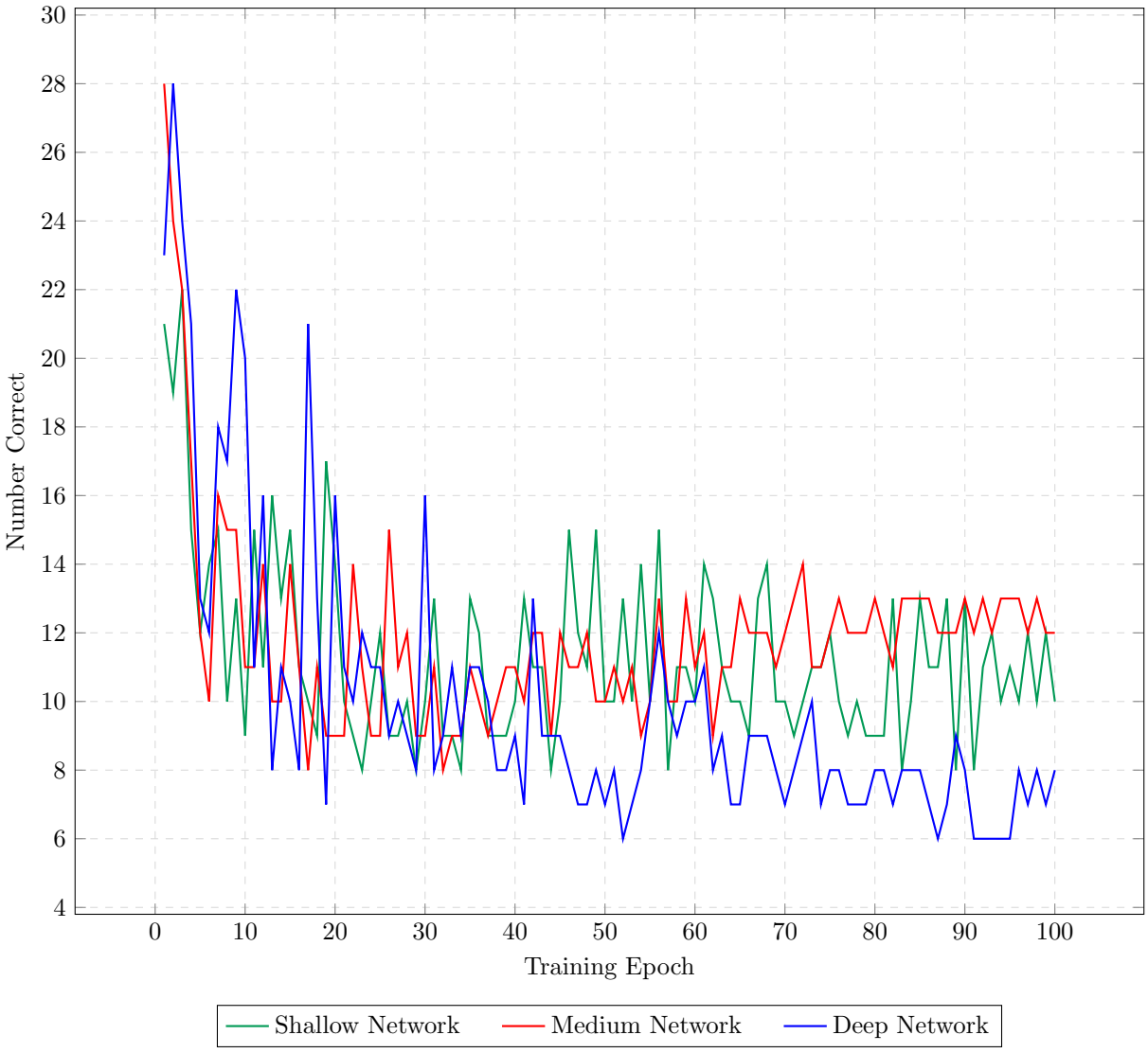

Figure 25: Failing to classify facial expressions