UNIVERSITY OF CAMBRIDGE
DEPARTMENT OF ENGINEERING

Isaac Brown
iwb21@cam.ac.uk
github.com/Isaac-W-Brown/Deep_Learning

Deep Learning: Character Recognition

Isaac Brown - Pembroke College Cambridge

May - July 2020

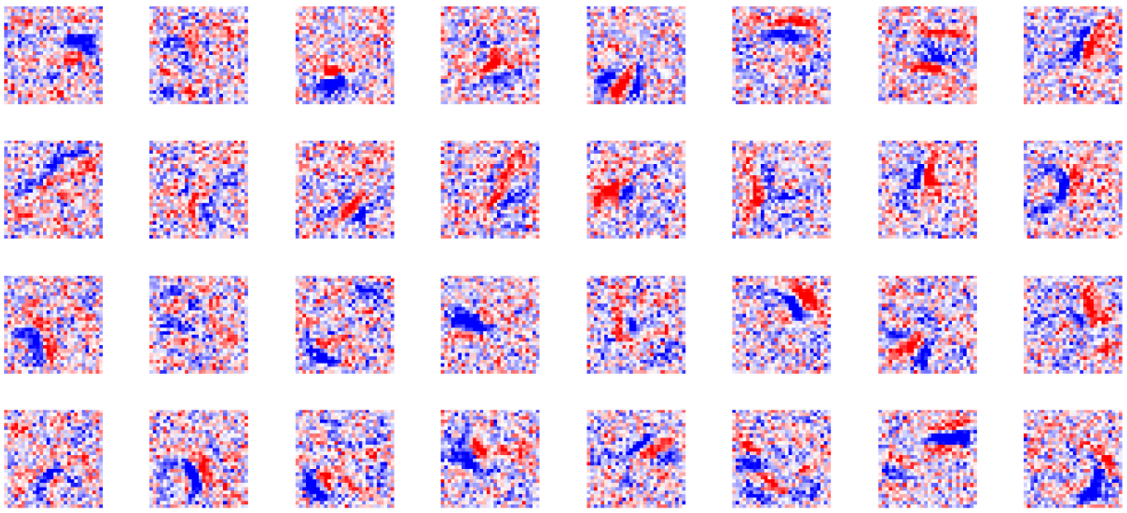


Figure 1: First Layer activations in a Neural Network

1 Introduction

Inspired by a second year lecture series on Deep Learning and Computer Vision, I plan to investigate neural networks further by creating one. The problem I have chosen to tackle is number recognition. This has been done many times before and as such guidance on how to proceed should be prevalent. The language I will use is Python, although I may convert it to C++ for speed improvements or to improve my understanding.

1.1 Aims

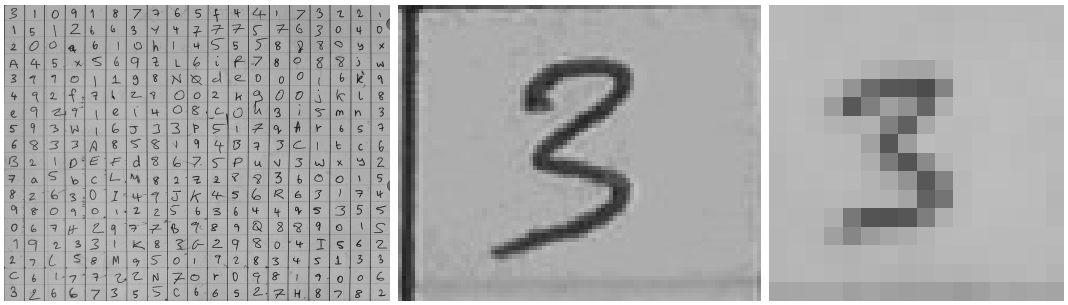
- Code derivatives and back-propagation involved in neural networks.
- Show successful optimisation on one image.
- Optimise a set of numbers.
- Create a Neural Network which reads numbers to a high degree of accuracy.
- Create a Convolutional Neural Network to solve the same problem and discuss any benefits.
- Find a way to visualise the decisions the Neural Networks are making.

2 Simple Neural Network

I began this project before the university lectures on deep learning had taken place. All my knowledge on the subject came from a YouTube series by 3Blue1Brown [1] which covered some of the maths involved in training a Neural Network. For this first network the cost function would be a sum of the squares of differences between the desired output and that predicted by the network. As I would soon discover, this has some large drawbacks.

2.1 Starting features of the Neural Network

- 18 training examples per number. I hand drew these myself, scanned them and compressed them to 16x16 images. Hence the input vector to the Neural Network is 256 dimensional.



(a) Number sheet (b) Full quality scan (c) Compressed to 16x16

Figure 2: The process of making my own numbers for training

- One dummy class. I created 18 images which were not numbers, to ensure the network can recognise when a character is not at all what it is looking for. This class was labeled '10' and meant there were 11 classes for the network to classify.
- Single layer network. The 256 inputs are simply linearly multiplied and added to form the 10 output neurons, before being passed through a Sigmoid function. The code had the ability to handle more layers but with so few training examples this was not necessary.

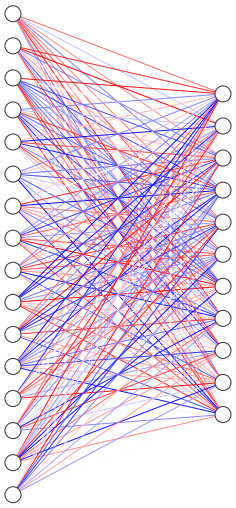


Figure 3: Single Layer Neural Network [2]

- Squared cost function. The true output node has a target value of 1 and the other nodes 0. The cost function is simply the sum of the squares of the differences between the network outputs and the target values. There is no Softmax layer yet.

- Non-stochastic gradient descent. With so few training examples it was possible to optimise the cost function using all the images every iterations.
- Steepest gradient (linear) optimisation. The change in the parameters was some constant learning rate multiplied by the negative derivative of the cost function with respect to that parameter.

I coded the network up from scratch, not yet using Numpy arrays and matrices to represent the weights, biases and values. Therefore this network was inefficient.

Inefficient forward propagation through network

```
# Initializes list of values of next layer
layer = [0] * layers[i+1]

# Iterates through weights to calculate next layer
for end in range(layers[i+1]):
    for start in range(layers[i]):
        layer[end] += weights[i][start][end] * vals[-1][start]

# Adds bias and does logistic function
adj_layer=[]
for n in range(len(layer)):
    adj_layer.append(1 / (1 + np.exp(-(layer[n] + biases[i][n]))))
```

Inefficient backward propagation through network

```
l = 1  # Iterates through layers
while l < len(layers):
    start = 0  # Iterates through start points
    while start < layers[-l-1]:
        end = 0  # Iterates through end points
        while end < layers[-l]:

            if image[0] == end:
                g = 1  # Goal is 1 if end point is true
            else:
                g = 0  # Goal is 0 if end point is true
            d = vals[-1][end]
            c = vals[-2][start]

            # Calculate weight and bias grads
            weight_grads[-1][start][end] = 2 * (d-g) * d * (1-d) * c
            weight_grad_sums[-1][start][end] += weight_grads[-1][start][end]
            if start == 0:
                bias_grads[-1][end] = 2 * (d-g) * d * (1-d)
                bias_grad_sums[-1][end] += bias_grads[-1][end]

            end += 1
        start += 1
    l += 1
```

While the network did eventually learn the training images, it would learn by optimising the majority of the images at the expense of other images which would predict completely the wrong number. This is down to the squared cost function; a log cost function is better as the sum of the logs of the predicted values produces a cost function which will optimise the product of the output probabilities, which is what our aim is.

2.2 Stochastic gradient descent (V1)

To make the code faster I gave it a stochastic gradient descent. Here I grouped the images in batches of 10, 1 image per number. This gave speed improvements of about 4 times, approximately the square root of the 18 times fewer images which were being used to train the network each iteration. The code was more or less identical, other than a line instructing the program to iterate through the batches of numbers.

Initial batch implementation

```
for batch in training_data:
```

Later on I implemented stochastic gradient descent by treating the multiple input images as a 3D tensor.

2.3 Log cost function and Softmax

After the lectures on Deep Learning [3], I decided to implement a log cost function. Here only the true output class would be used to calculate the cost function by taking the negative log of the probability of this output predicted by the network. This resulted in dramatically improved learning rates and all the image predictions improving in a balanced fashion.

$$G = -\ln(y_i)$$
$$\frac{\partial G}{\partial y_i} = -\frac{1}{y_i}$$

Where G is the cost function and y_I is the true output of the network.

As this is a classification problem, a Softmax final layer is appropriate. This ensures that the outputs add to 1, making them analogous to probabilities, which in a desired characteristic of a neural network.

$$y_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

A Softmax layer, where x is are the inputs and y are the outputs.

A further benefit of a Softmax is that is makes the back propagation maths neat. To find the rate of change of the true Softmax output y_i with respect to the true Softmax input x_i, we can write the Softmax formula (equation 1) as:

$$y_i = \frac{e^{x_i}}{C + e^{x_i}}$$

Find the derivative and rearrange:

$$\frac{\partial y_i}{\partial x_i} = \frac{e^{x_i}}{e^{x_i} + C} - \frac{e^{2x_i}}{(e^{x_i} + C)^2}$$
$$\frac{\partial y_i}{\partial x_i} = \frac{Ce^{x_i}}{(e^{x_i} + C)^2}$$
$$\frac{\partial y_i}{\partial x_i} = \frac{C}{e^{x_i} + C} \cdot \frac{e^{x_i}}{e^{x_i} + C}$$
$$\frac{\partial y_i}{\partial x_i} = (1 - y_i)y_i$$

Multiplying by $\frac{\partial G}{\partial y_i}$:

$$\frac{\partial G}{\partial x_i} = y_i - 1$$

Similarly, to find the rate of change of the true Softmax output y_i with respect to a false Softmax input x_j (where $j \neq i$), we can write the Softmax formula (equation 1) as:

$$y_i = \frac{C_1}{e^{x_j} + C_2}$$

Find the derivative and rearrange:

$$\frac{\partial y_i}{\partial x_j} = -\frac{C_1 e^{x_j}}{(e^{x_j} + C_2)^2}$$
$$\frac{\partial y_i}{\partial x_j} = -\frac{C_1}{e^{x_j} + C_2} \cdot \frac{e^{x_j}}{e^{x_j} + C_2}$$
$$\frac{\partial y_i}{\partial x_j} = -y_i y_j$$

Multiplying by $\frac{\partial G}{\partial y_i}$:

$$\frac{\partial G}{\partial x_j} = y_j$$

Therefore

$$\frac{\partial G}{\partial x_j} = \begin{cases} y_j & \text{for } j \neq i \\ y_j - 1 & \text{for } j = i \end{cases}$$

Hence the line of code "`probs[y] -= 1`" below, as in the code `y` represents the true output. Note that as the biases act directly on the vector $\mathbf{x}$, this is the gradient for the biases as well.

2.4 Linear Algebra

I used Numpy arrays to do the linear algebra associated with forward and back-propagation which sped the program up dramatically. This used the fact that moving forward through the network is essentially matrix multiplication followed by the addition of a bias vector, then a Sigmoid, ReLU or Softmax function.

$$\mathbf{x} = \sigma(\mathbf{W}\mathbf{a} + \mathbf{b}) \tag{2}$$

Where $\mathbf{x}$ is the output vector, $\mathbf{a}$ is the input vector, σ is the Sigmoid function, $\mathbf{W}$ is a matrix of weights and $\mathbf{b}$ is a bias vector.

To find the rate of change of the cost function with respect to the weights we note that equation 1 (without the Sigmoid) shows:

$$x_j = W_{jk} a_k + b_j$$
$$\frac{\partial x_j}{\partial W_{jk}} = a_k$$
$$\frac{\partial G}{\partial W_{jk}} = \frac{\partial G}{\partial x_j} \frac{\partial x_j}{\partial W_{jk}}$$
$$\frac{\partial G}{\partial \mathbf{W}} = \frac{\partial G}{\partial \mathbf{x}} \mathbf{a}^{\mathrm{T}}$$

Which is simply the outer product of the input vector to the weights and the derivative of the cost function with respect to its outputs.

Log cost function Softmax and linear algebra implementation

```
while it < 1000:
    it_cost = 0

    # Initialize derivative sums
    wt_derivs_tot = np.zeros((11, 256))
    bs_derivs_tot = np.zeros(11)

    # Iterate through one example of each number
    for n in range(11):

        # Choose training data
        x = np.random.randint(imagespnum)
        i = (n * imagespnum) + x
        y, image = training_data[i]

        # Calculate output
        output = np.matmul(weights, image) + biases
        exp_output = np.exp(output)
        softmax_sum = sum(exp_output)
        probs = exp_output / softmax_sum
        cost =- np.log(probs[y])
        it_cost += cost

        # Calculate derivatives
        probs[y] -= 1
        wt_derivs = np.outer(probs, image)
        wt_derivs_tot += wt_derivs
        bs_derivs = probs
        bs_derivs_tot += bs_derivs
```

2.5 Multi-Layer network

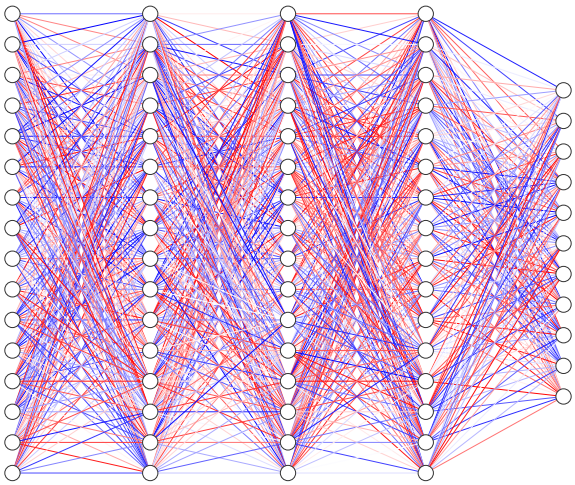


Figure 4: Multi Layer Neural Network [2]

To make the network *deep*, multiple layers with non-linearity are used to allow the network to act with some sort of intelligence. For the non-linearity I used Sigmoid functions on each of the nodes, other than the final layer which uses a Softmax. A Sigmoid has two useful properties; it maps all numbers to a range between 0 and 1, keeping the neurons within a controlled range, and it has easily calculable derivative, making the back propagation maths easier.

As before, the formula for moving through the layers is:

$$\mathbf{x} = \sigma(\mathbf{W}\mathbf{a} + \mathbf{b})$$

Which we can write as

$$\mathbf{x}' = \mathbf{W}\mathbf{a} + \mathbf{b}$$

Where $\mathbf{x}$ is $\sigma(\mathbf{x}')$ and $\mathbf{a}$ is $\sigma(\mathbf{a}')$ (the previous layer). We already know the derivative of the cost function with respect to $\mathbf{x}'$, so we can use the chain rule to propagate backwards:

$$\frac{\partial G}{\partial \mathbf{a}} = \frac{\partial \mathbf{x}'}{\partial \mathbf{a}} \frac{\partial G}{\partial \mathbf{x}'}$$
$$\frac{\partial G}{\partial \mathbf{a}} = \mathbf{W}^{\mathrm{T}} \frac{\partial G}{\partial \mathbf{x}'}$$

To find the derivative with respect to $\mathbf{a}'$, we find the derivative of the Sigmoid function:

$$\mathbf{a} = \sigma(\mathbf{a}')$$
$$\mathbf{a} = \frac{1}{1 + e^{-\mathbf{a}'}}$$
$$\frac{\partial \mathbf{a}}{\partial \mathbf{a}'} = \frac{e^{-\mathbf{a}'}}{(1 + e^{-\mathbf{a}'})^2}$$
$$\frac{\partial \mathbf{a}}{\partial \mathbf{a}'} = \frac{1}{1 + e^{-\mathbf{a}'}} \cdot \frac{e^{-\mathbf{a}'}}{1 + e^{-\mathbf{a}'}}$$
$$\frac{\partial \mathbf{a}}{\partial \mathbf{a}'} = \mathbf{a}(1 - \mathbf{a})$$

Therefore

$$\frac{\partial G}{\partial \mathbf{a}'} = \mathbf{a}(1 - \mathbf{a})\mathbf{W}^{\mathrm{T}} \frac{\partial G}{\partial \mathbf{x}'}$$

With this we can find the derivative of the cost function with respect to a layer's pre-Sigmoid inputs, only knowing the post-Sigmoid values of that layer ($\mathbf{a}$), the weights between the layers we are looking at ($\mathbf{W}$) and the derivative of the cost function with respect to the pre-Sigmoid layer we have already calculated ($\mathbf{x}'$). We can use this formula to work backwards through the layers. The derivatives with respect to the weights are an outer product of the input vector to the weights and the derivatives of its outputs, as before in the single layer case.

Multi layer back-propagation implementation

```
# Iterates backwards through layers (back-propagation)
for i in range(l-1):

    # Bias derivative calc, different if final layer (due to Softmax)
    if i == 0:
        b = vals[-1]
        b[y] -= 1
    else:
        b = vals[-i-1] * (1 - vals[-i-1]) * (weights[l-i-1].T @ b)

    # Weight derivatives are outer product of bias derivatives and previous layer
    w = np.outer(b, vals[-i-2])
```

3 Optimisation techniques

3.1 Regularisation

Another concept introduced in lectures was the principal of "regularising" a Neural Network. If left to its own devices, a network may over train - its weights and biases will become large as the network extrapolates the patterns in the data it has seen to other parts of the sample space with too much confidence. Regularisation aims to prevent this by reducing the magnitude of each weight after every iteration. The amount reduced is proportional to the square of the weight. This results in the weights being approximately normally distributed, hence the term regularisation.

Regularisation implementation

```
# Regularises weights and biases
reg_weights = [a * (1 - (reg * abs(a))) for a in weights]
reg_biases = [a * (1 - (reg * abs(a))) for a in biases]

rate = k  # Learning rate

# Updates weights and biases using batch derivatives
weights = [a - (rate * b) for a, b in zip(reg_weights, wt_derivs_tot)]
biases = [a - (rate * b) for a, b in zip(reg_biases, bs_derivs_tot)]
```

After experimenting with different values of k (learning rate) and reg (regularisation coefficient), it became clear that to have a stable learning process $k \cdot reg$ had to be roughly constant. If $k \cdot reg$ is too large learning is erratic, if it is too low learning is slow. The regularisation can then be tuned by altering k/reg. After learning the images, the below images show how regularisation decreases the magnitude of the weights and gives them a normal-like distribution.

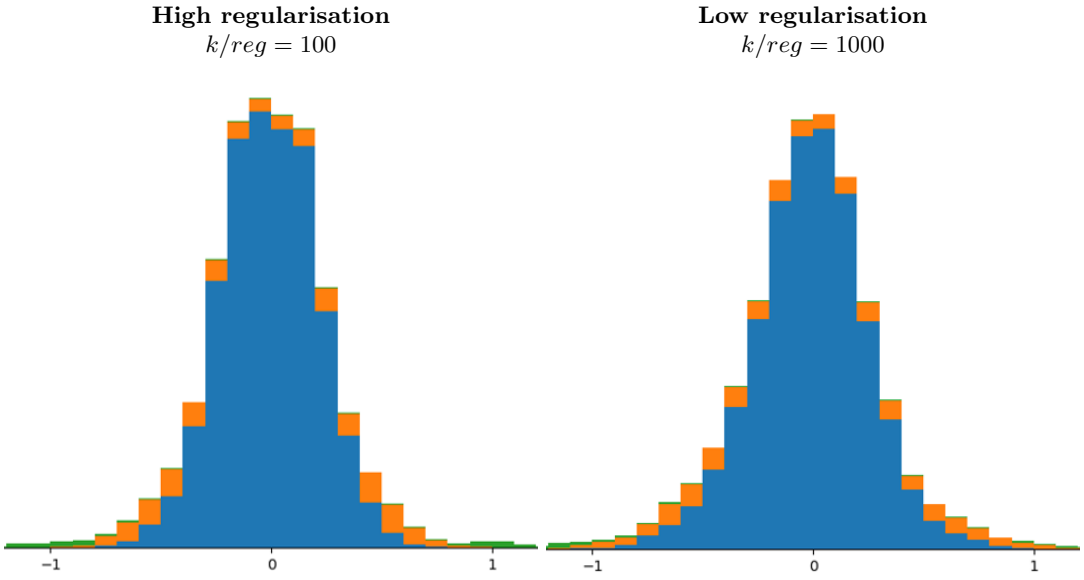


Figure 5: Histogram of weights. The left histogram represents a stronger regularisation and therefore its weights are smaller.

Interestingly the layers closer to the output have weights of greater magnitude. This means they must have a greater bearing on the output of the network, which makes sense. The network was able to make the cost function arbitrarily small, but increasing the regularisation significantly reduces the ability of the network to minimise the cost function. Furthermore it took longer to reach its final value, which I did not expect.

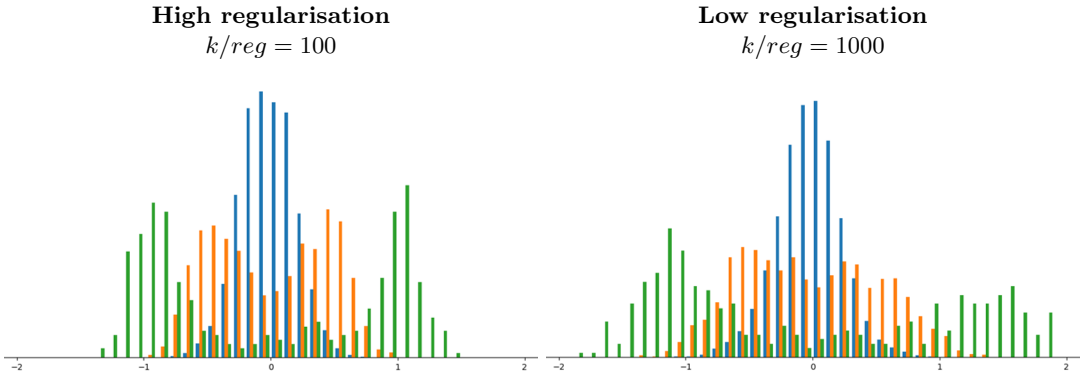


Figure 6: Histogram of weights, separated by layer. This better shows the significance of k/reg. Weights closer to the output (blue is 1st layer, orange 2nd and green 3rd) are larger.

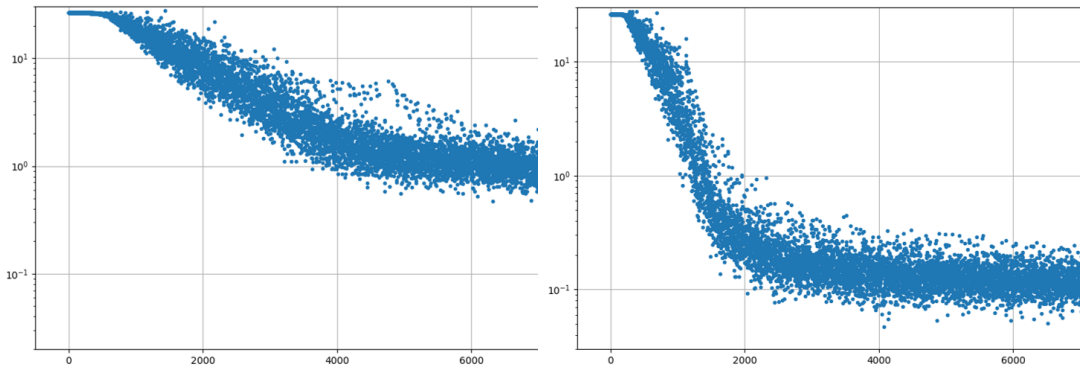
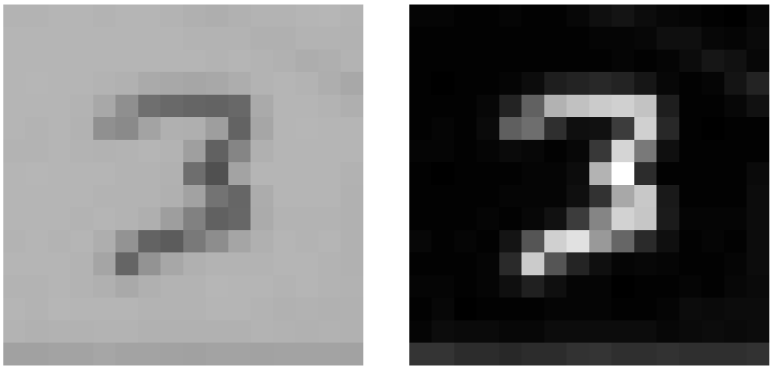


Figure 7: Learning rates, showing cost function against iteration showing how increasing the regularisation reduces the ability of the network to minimise the cost function and slows learning.

3.2 Tuning and Accuracy Testing

I produced a further 600 examples of each number - 500 to train the network and 100 to test it. The numbers I drew were deliberately noisy - I wanted the network to find the underlying pattern which makes a number. Before passing these numbers into the network I increased the contrast and inverted the images such that the lowest pixel value is mapped to 255 and the highest to 0. This is more representative of how a human brain adjusts for lighting conditions and how the black writing is 'on' and therefore should have a high value rather than the 0 a computer stores it as.

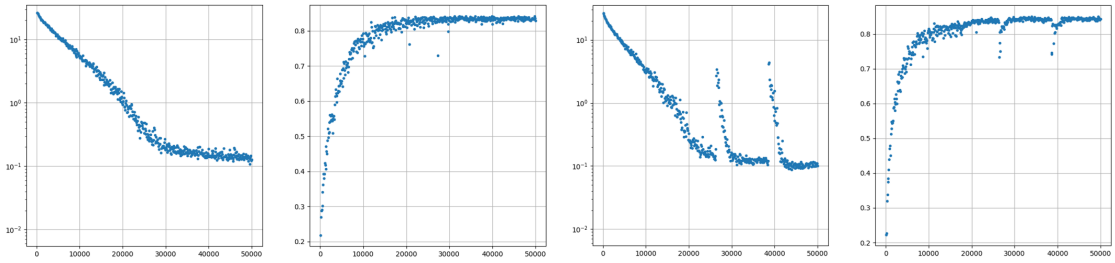


Before pre-processing Increased contrast and inversion

Figure 8: Pre-processing of images for lighting variations. Note the pixel imperfections.

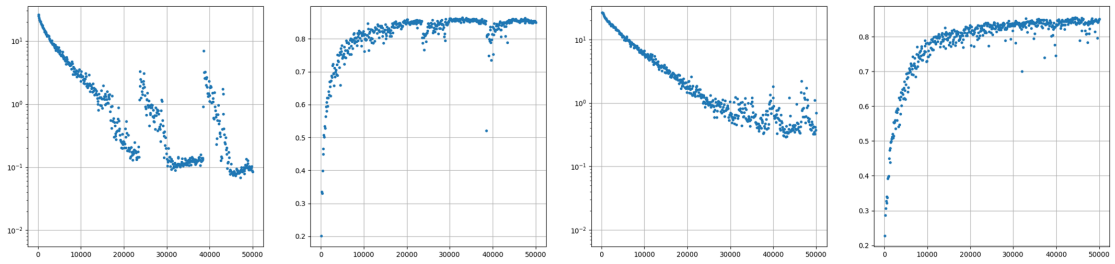
I could then begin to tell how accurate the network was at predicting numbers. I implemented a 256/128/64/32/11 architecture for the layers and normally distributed random starting weights ($\sigma = 0.25$), which were chosen after fine-tuning the network. I tested it with different values of k and reg and found some interesting behaviors:

- After a few thousand iterations the networks learning rate increased dramatically , before levelling off due to regularisation. This is a characteristic of the point a neural network learns some actual features of the images, rather than simply memorising the data [1].
- As I increased k for a constant k/reg the learning became very unstable to the extent that every few thousand iteration the cost function would explode. It would come down quickly. This is likely to be the network jumping over a ridge to reach another local minimum.
- Optimising the cost function was not necessarily best for accuracy. Namely, the unstable learning processes often resulted in higher cost functions but higher accuracy. This could be down to the fact that the explosions in cost function allowed the network to reset and 'breathe', finding a better local minimum.
- I obtained a maximum sustainable accuracy of 85.5%, not bad for the limited noisy and low resolution data.



$k = 10^{-1.5}, reg = 10^{-5}$, 83.6% accuracy

$k = 10^{-1.25}, reg = 10^{-4.75}$, 84.4% accuracy



$k = 10^{-1}, reg = 10^{-4.5}$, 85.5% accuracy

$k = 10^{0.75}, reg = 10^{-4.25}$, 84.6% accuracy

Figure 10: Accuracy testing the neural network. The left diagram in each case is cost function over iteration and the right diagram is accuracy over iteration. The graphs show how increasing k made the network more unstable, and how after approximately 20000 iterations the cost function decreases suddenly.

3.3 Elastic Distortion

Elastic distortion is a method of generating new training examples from existing data. Two random, uniformly distributed numbers are assigned to each pixel, representing a shift in the x-direction and y-direction. A Gaussian blur of some sigma is then used to smooth these numbers. They then serve as displacement co-ordinates from each pixel from which the new value will be linearly interpolated from the original pixel values. The effect is that the image is distorted but in a way which makes the image recognisably human. With large enough displacements, the new images have can have very little in common with the original on a pixel-for-pixel level.

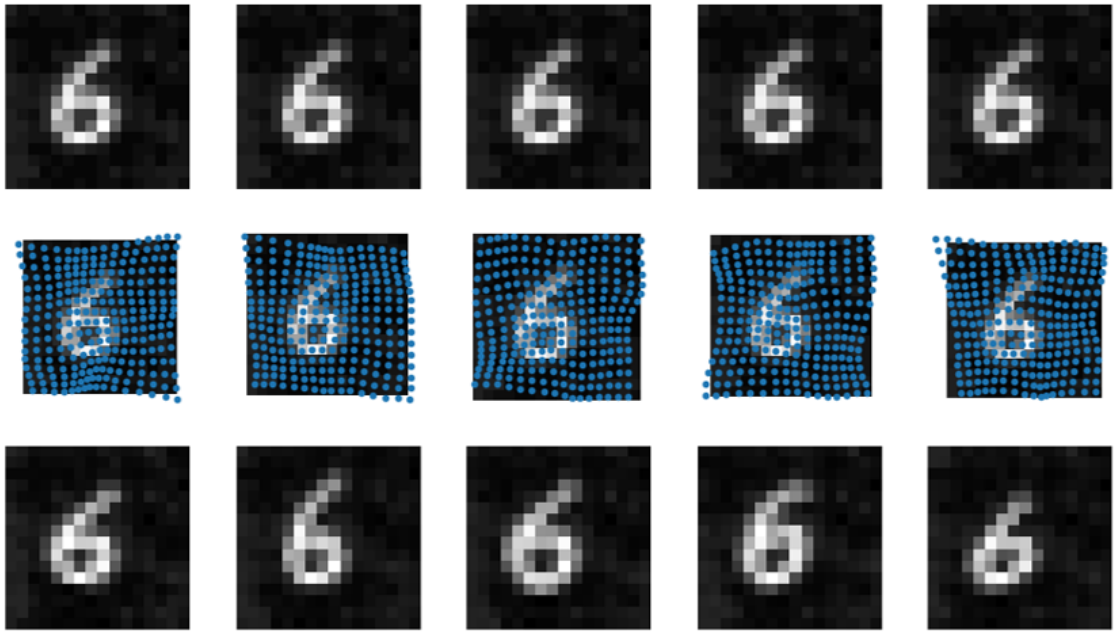


Figure 11: Elastic distortion of a number. The top row shows the original image. The blue dots on the second row show the location of the interpolations on the original image. The final row shows the new generated images, which are now distinct but still clear numbers.

Elastic distortion implementation

```
# Distorts training data creating new, still recognisable, training data
def elastic(image, u_rng, sigma):
    size = int(np.sqrt(len(image)))

    # Creates random filtered arrays and adds to square grid
    x_random_array = u_rng * (np.random.rand(size, size) - 0.5)
    x_smoothed_array = gaussian_filter(x_random_array, sigma=sigma)
    xs = np.array([np.arange(size),] * size) + x_smoothed_array
    y_random_array = u_rng * (np.random.rand(size, size) - 0.5)
    y_smoothed_array = gaussian_filter(y_random_array, sigma=sigma)
    ys = np.array([np.arange(size),] * size).transpose() + y_smoothed_array

    # Interpolate known image with random points to produce new image
    x = y = np.arange(size)
    z = np.reshape(image, (-1, size))
    f = interpolate.RectBivariateSpline(x, y, z)
    new_image = f.ev(ys.flatten(), xs.flatten())

    # Normalise new image
    new_image[new_image < 0] = 0
    new_image -= min(new_image)
    new_image *= 1 / max(new_image)

    return new_image, xs, ys
```

The overall effect is that the network now has infinitely many training examples to learn from, however it may still lack a true diversity of numbers. By employing this method and by tuning the network further, I was able to acquire large improvements in accuracy, up to 92%.

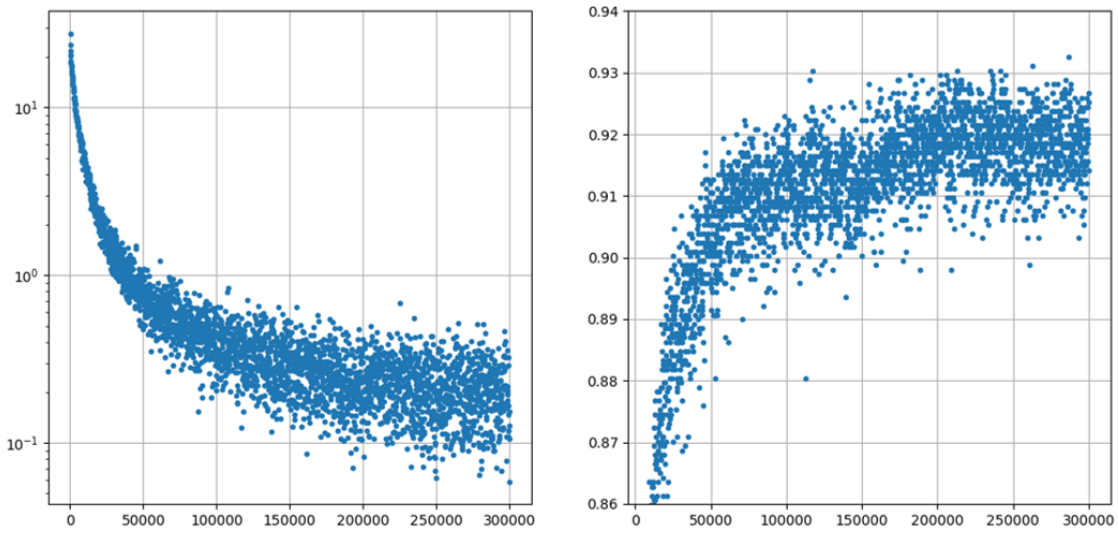


Figure 12: Learning with elastic distortions. The cost function stays higher as there are more images which is harder to optimise. 92% final accuracy with $k = 10^{-1.25}$, $reg = 10^{-5.5}$, elastic $u = 8$ and $\sigma = 2$, normally distributed start weights $\sigma = 0.25$ and a 256/256/256/256/11 architecture.

Figure 13: Some examples of the images the network got wrong, with the number it though it was above the image. The number 10 means the network thought it was not a number. The numbers it got wrong were by no means crystal clear failures, which is encouraging.

4 EMNIST dataset

The EMNIST database [4] (Extended Modified National Institute of Standards and Technology database) is a set of hundreds of thousands of hand drawn characters. They have already been adjusted for contrast and are much less noisy than the numbers I drew. Further, they are 28x28 rather than 16x16. The subset of the database I used has upper and lowercase letters as well as numbers, with each class having an equal quantity of examples. In total there are 47 classes, 112800 training examples and 18800 testing examples.

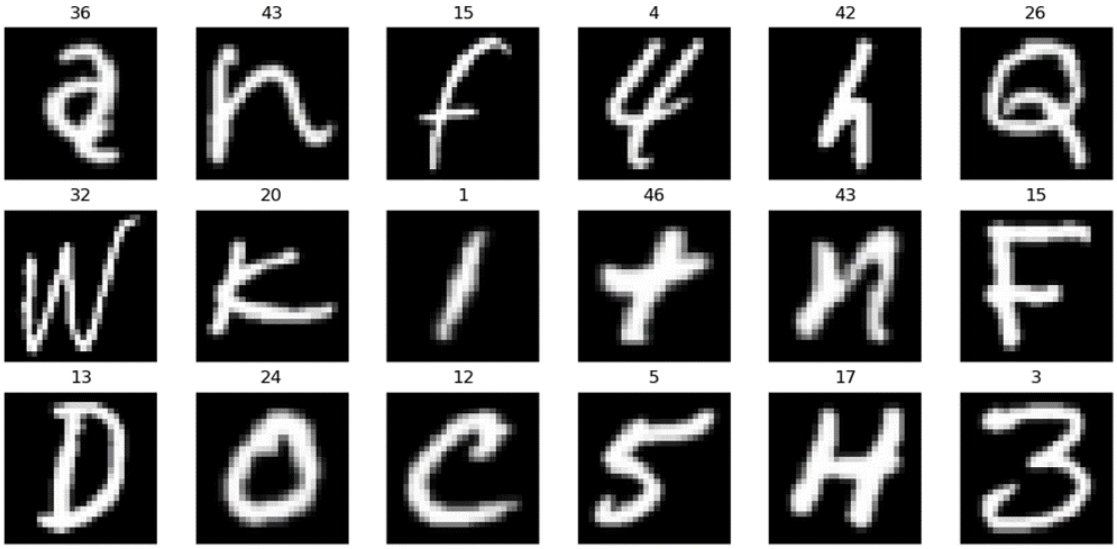


Figure 14: EMNIST training images

After experimenting with parameters, I used a 784/256/256/47 architecture and varied the hyper-parameters. Despite the large number of training images, elastic distortion helped improve accuracy significantly. Interestingly, when elastic distortion was used the network only lowered the cost function to 0.35 per image, meaning it thought each output was 70% likely on average. This suggests a bigger network would be able to improve accuracy further as the cost function has a lot of room for improvement. I implemented a 784/784/784/784/47 network to test this but it used to much RAM for my computer to be fast.

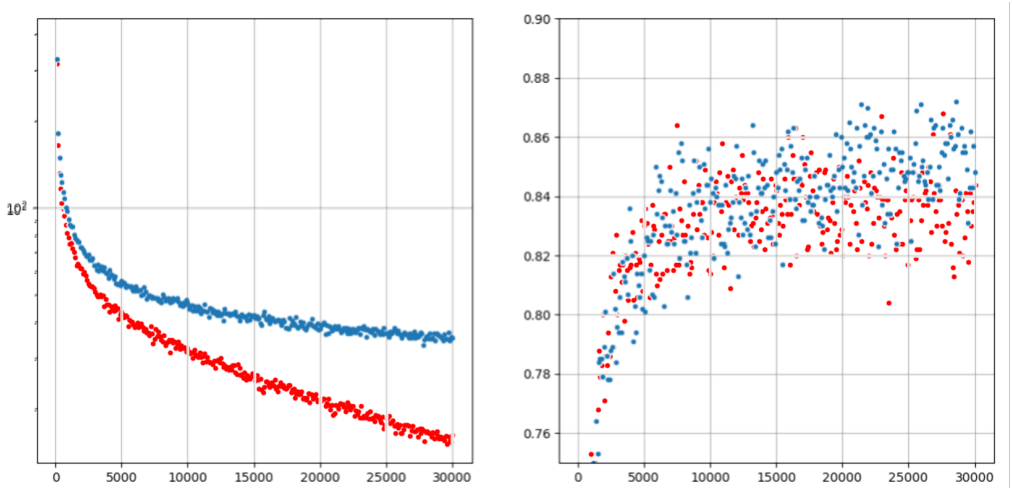


Figure 15: Network training. Red is without and Blue is with elastic distortion. This shows how elastic distortion inhibits the optimisation of the cost function (L), but improves accuracy (R).

During testing I noticed some of the images' labels were tenuous and the network was understandably confused some. Nevertheless it hit 86% accuracy, which for 47 classes and some questionable training data is good. Furthermore, some of the failures were understandable as the images were ambiguous. About 55% were acceptable, bringing accuracy to over 90%.

Label	Char	Label	Char	Label	Char	Label	Char	Label	Char	Label	Char
0	0	8	8	16	G	24	O	32	W	40	f
1	1	9	9	17	H	25	P	33	X	41	g
2	2	10	A	18	I	26	Q	34	Y	42	h
3	3	11	B	19	J	27	R	35	Z	43	n
4	4	12	C	20	K	28	S	36	a	44	q
5	5	13	D	21	L	29	T	37	b	45	r
6	6	14	E	22	M	30	U	38	d	46	t
7	7	15	F	23	N	31	V	39	e		

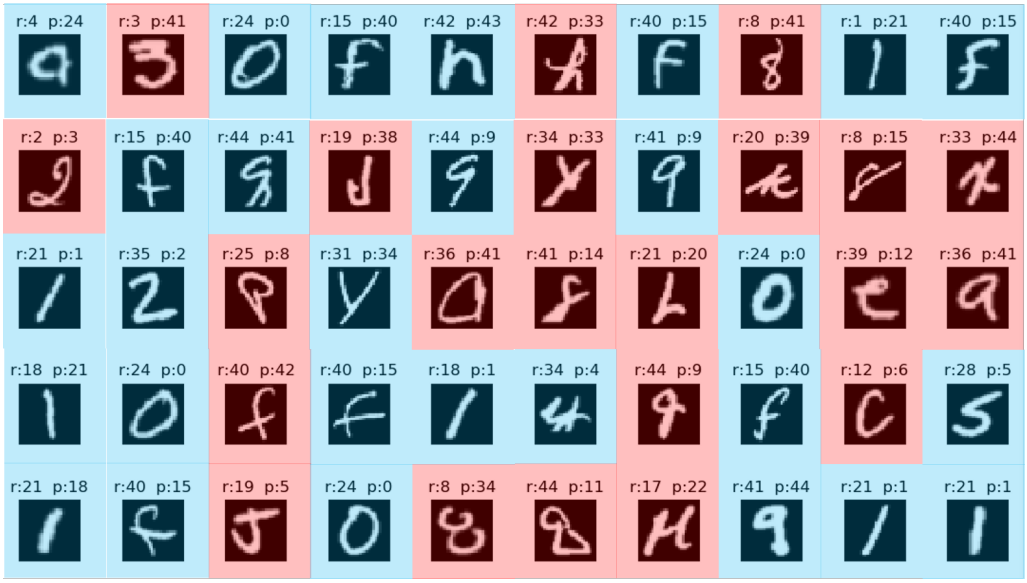


Figure 16: Neural Network failures. Each image shows the real and predicted label above it. The characters in blue are understandable failures, red are genuine errors.

I plotted the first layer activations and coloured their values on a scale from blue (-ve) to red (+ve). In these images you can clearly see the features the network is looking for, usually edges of some curvature and position.

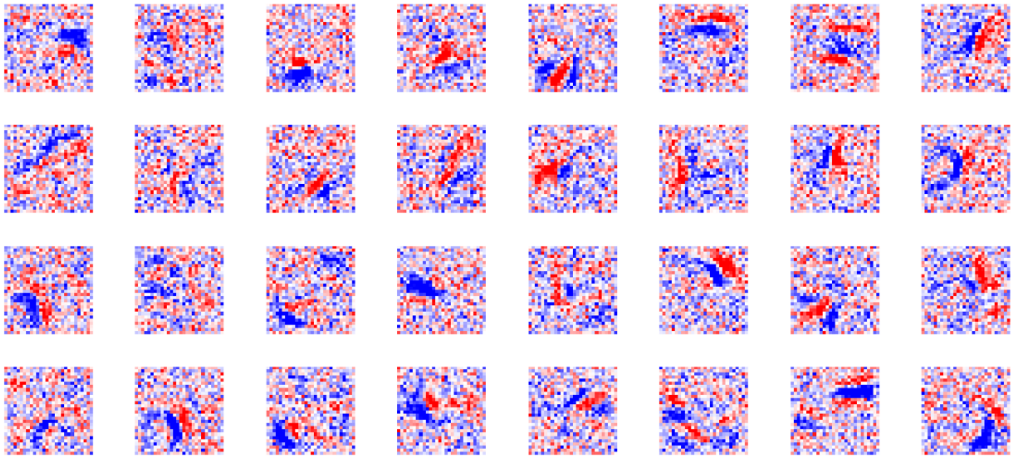


Figure 17: First Layer activations in the EMNIST Neural Network, showing clear key features.

4.1 Stochastic gradient descent (V2)

A better way of implementing stochastic gradient descent is by combining the input vectors of various images into a matrix, and then performing the forward and backward propagation with an extra "batch" dimension present. The formula for forward propagation now becomes:

$$\mathbf{X}' = \mathbf{W}\mathbf{A} + \mathbf{B}$$

Where $\mathbf{X}'$ is the pre-Sigmoid output matrix and $\mathbf{A}$ is the post-Sigmoid input. The new dimension in both is batch-wise. $\mathbf{W}$ is the matrix of weights as before and $\mathbf{B}$ is a matrix of biases formed by repeating the bias vector by however many images there are in the batch. Finding the derivative of the cost function with respect to the pre-Sigmoid inputs ($\mathbf{A}'$):

$$\begin{aligned}
\frac{\partial \mathbf{X}'}{\partial \mathbf{A}} &= \mathbf{W}^{\mathrm{T}} \\
\frac{\partial G}{\partial \mathbf{A}} &= \mathbf{W}^{\mathrm{T}} \frac{\partial G}{\partial \mathbf{X}'} \\
\frac{\partial \mathbf{A}}{\partial \mathbf{A}'} &= \mathbf{A} \circ (1 - \mathbf{A}) \\
\frac{\partial G}{\partial \mathbf{A}'} &= \mathbf{A} \circ (1 - \mathbf{A}) \circ \mathbf{W}^{\mathrm{T}} \frac{\partial G}{\partial \mathbf{X}'}
\end{aligned}$$

Which is the Hadamard product of the post-Sigmoid inputs, one minus the post-Sigmoid inputs and the matrix multiplication of the transpose of the weights and the known gradients of the weights' output matrix. The Hadamard product is an element-wise multiplication of two matrices, so they must have the same size and output a matrix of the same size. Finding the derivative of the cost function with respect to the weights:

$$\begin{aligned}
\frac{\partial \mathbf{X}'}{\partial \mathbf{W}} &= \mathbf{A}^{\mathrm{T}} \\
\frac{\partial G}{\partial \mathbf{W}} &= \frac{\partial G}{\partial \mathbf{X}'} \mathbf{A}^{\mathrm{T}}
\end{aligned}$$

Which is the matrix multiplication of the known gradients of the weights' output matrix and the transpose of the weights' input matrix. With these two formulae we can back propagate through the network in batches all in one go. This makes the code much cleaner and potentially faster.

Batch back-propagation implementation

```
# Iterates backwards through layers (back-propagation)
for i in range(l-1):

    # Bias (outputs) derivative calc, different if final layer (due to Softmax)
    if i == 0:
        b = vals[-1] + y
    else:
        b = vals[-i-1] * (1 - vals[-i-1]) * weights[l-i-1].T @ b

    # Weight derivs are matrix product of bias (output) derivs and input matrix transposed
    w = b @ vals[-i-2].T

    # Append derivatives to lists
    wt_derivs.append(w)
    bs_derivs.append(np.sum(b, axis=1))
```

4.2 Starting conditions

Until this point I initialised the starting conditions of the weights by randomly generating normally distributed numbers with a mean of 0 and a standard deviation of 0.25, except for the final layer whose weights were 0 (0.25/0.25/0.25/0). My reasoning behind this was that I wanted the inputs to each of the layers to be unique so as to ensure that the weights change by different amounts, giving more defined gradients.

However, after experimenting with different starting conditions, I found that doing the opposite (0/0.25/0.25/0.25) was more effective. As back-propagation starts from the end of the neural network, it is important to ensure the gradients are meaty in the final layer. When the weights were 0 in the final layer, the gradients were then 0 for the preceding layers. This slowed down learning in the first 3 layers considerably, which can be shown by plotting the standard deviation of the weights' derivatives against iteration.

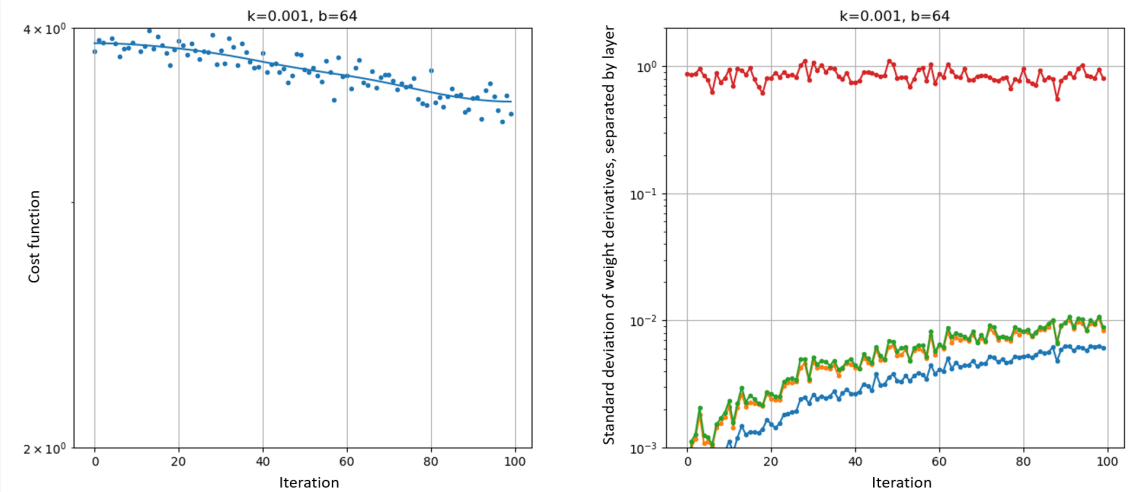


Figure 18: Cost function and standard deviation of weights' derivatives against iteration for $\sigma = 0.25/0.25/0.25/0$ starting weights.

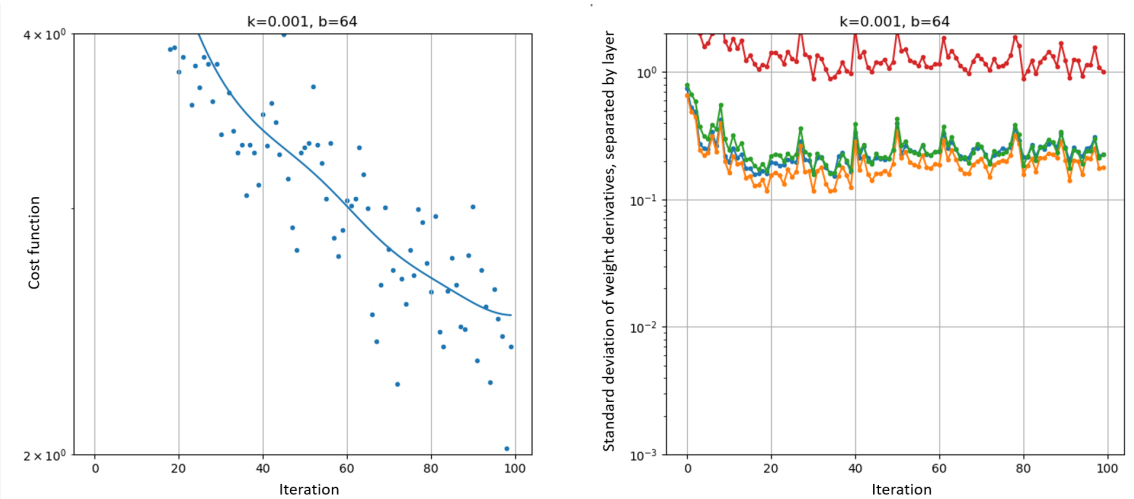


Figure 19: Cost function and standard deviation of weights' derivatives against iteration for $\sigma = 0/0.25/0.25/0.25$ starting weights.

The above graphs show how starting with the final weights as 0 decreased the magnitude of the derivatives in the first three layers (green, blue, orange) by over an order of magnitude. As a result the initial rate of learning in the second case is much faster. The reason behind starting the

first layer weights at 0 is that the first layer activation images are much less random, as all their value the has been learned without a random seed initialisation.

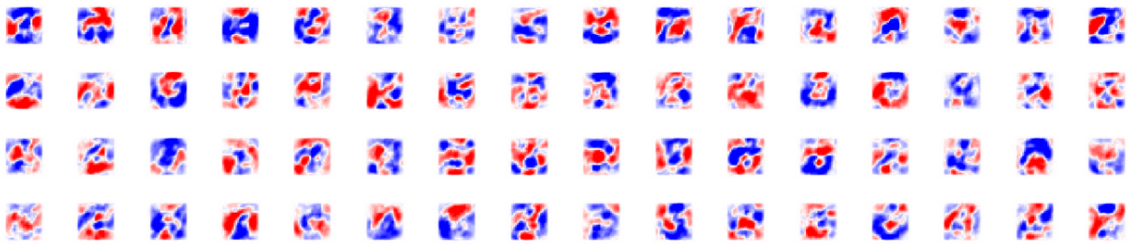


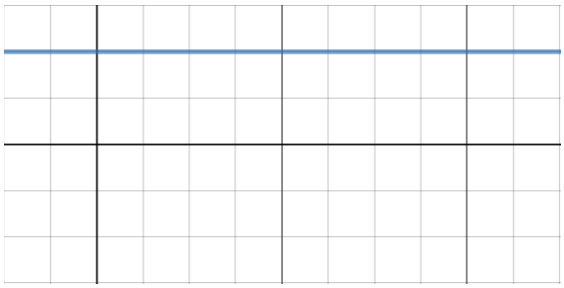
Figure 20: A selection of first layer activations after starting the first layer weights at 0, with much less random noise.

4.3 Adam optimiser

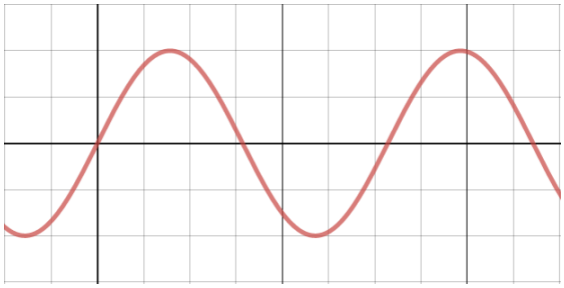
Adam optimisation [5] (Adaptive Moment optimisation) is a method of improving learning rates by varying the rate at which parameters of the network are changed with respect to their gradients. Before, the change of a parameter was proportional (k) to the instantaneous gradient of the cost function in the direction of that parameter. With Adam, the behavior of that gradient over time influences the learning rate.

Adam uses two exponentially decaying properties, momentum and energy:

- Momentum: An exponentially decaying weighted average of the previous gradients of a parameter. If a gradient is high for a long period of time, momentum will be high and the rate of learning will increase for that parameter.
- Energy: An exponentially decaying weighted average of the squares of previous gradients of a parameter. If the magnitude of the gradient is high for a long period of time, energy will be high and the rate of learning will decrease for that parameter.



A high momentum gradient



A high energy gradient

Together, this means that if a gradient is high and stable, the learning rate will be high. Conversely, if the momentum is high but the gradient oscillates, it will have a high energy and the learning rate will decrease. This ensures the network speeds up in potentially shallow but stable directions and slows in steep but oscillatory directions. The formula for the change in a given parameter is:

$$\theta_{n+1} = \theta_n - k \cdot \frac{m'_n}{\sqrt{v'_n + \epsilon}}$$

Where m' and v' is the iteration adjusted momentum and energy respectively and ϵ is a small offset to prevent division by 0. The formulas for momentum and energy are:

$$m_n = B_1 \cdot m_{n-1} + (1 - B_1) \cdot \frac{\partial G}{\partial \theta}$$

$$v_n = B_2 \cdot v_{n-1} + (1 - B_2) \cdot \left(\frac{\partial G}{\partial \theta}\right)^2$$

As both momentum and energy start at 0, they will be low when learning starts, so we adjust them:

$$m' = \frac{m}{1 - B_1^n}$$
$$v' = \frac{v}{1 - B_2^n}$$

$B_1 = 0.9$, $B_2 = 0.999$ and $\epsilon = 10^{-8}$ are all tunable hyper-parameters, but I kept the values recommended in the papers and articles I had read.

Adam optimiser implementation

```
def adam(params, derivs, ms, vs, it):
    i = 0
    for a, b in zip(params, derivs):
        # Updates momentum and energy values
        ms[i] = B1 * ms[i] + (1 - B1) * b
        vs[i] = B2 * vs[i] + (1 - B2) * np.power(b, 2)

        # Adjusts for iteration
        m_hat = ms[i] / (1 - np.power(B1, it + 1))
        v_hat = vs[i] / (1 - np.power(B2, it + 1))

        # Updates parameters
        a *= (1 - (REG * abs(a))) # Regularization
        a -= K * m_hat / (np.sqrt(v_hat) + EPS)
        i += 1
```

To demonstrate the Adam optimiser works, I plotted the change in the values of some randomly selected weights during learning. The Adam optimised values are far less erratic than learning with the steepest gradient approach.

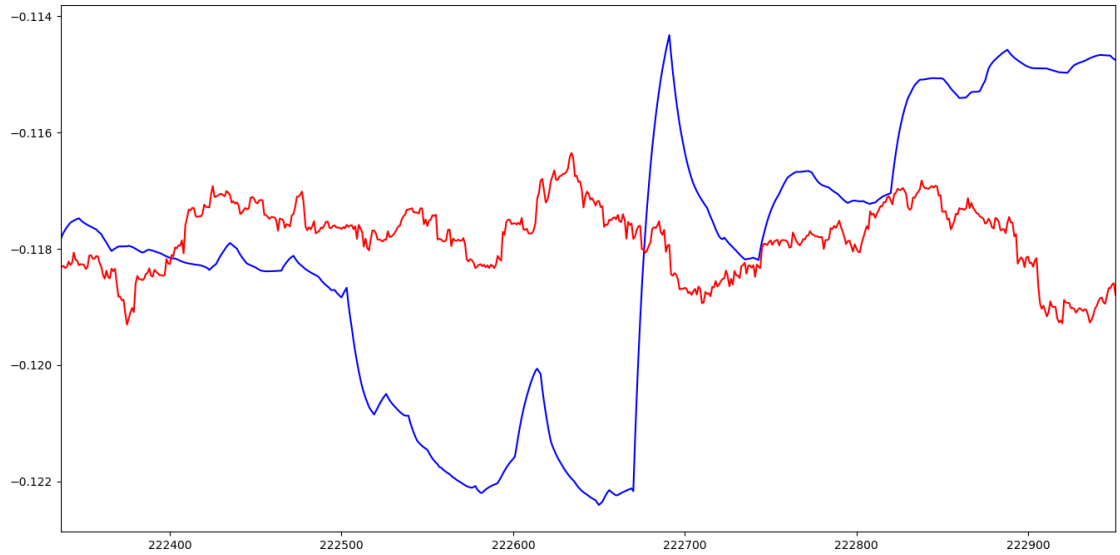


Figure 22: Randomly selected weights values over time, with red being normal and blue with Adam. Adam is more stable.

The Adam optimiser was very effective for me, speeding learning up by 4 times. An interesting note is that the Adam optimiser made the first layer activations look very different. With Adam,

a parameter who's gradient is consistent and high will change by the same amount as a parameter who's gradient is just as consistent but low. As a result, parts of the images which are sparse of data (the edges) have their associated weights increase significantly. As a result the weights at the edge are much higher than at the centre, which is not ideal. It implies that the edge pixels are more significant for determining a number, which is not true.

To fix this, you can add slight random noise to each image so that they are still recognisable but that network can determine that the edge pixels are irrelevant for determining the character. I added a normally distributed numbers with a standard deviation of 0.1 to each pixel of each image and the resulting first layer activations were without the high weights at the edges, as desired.

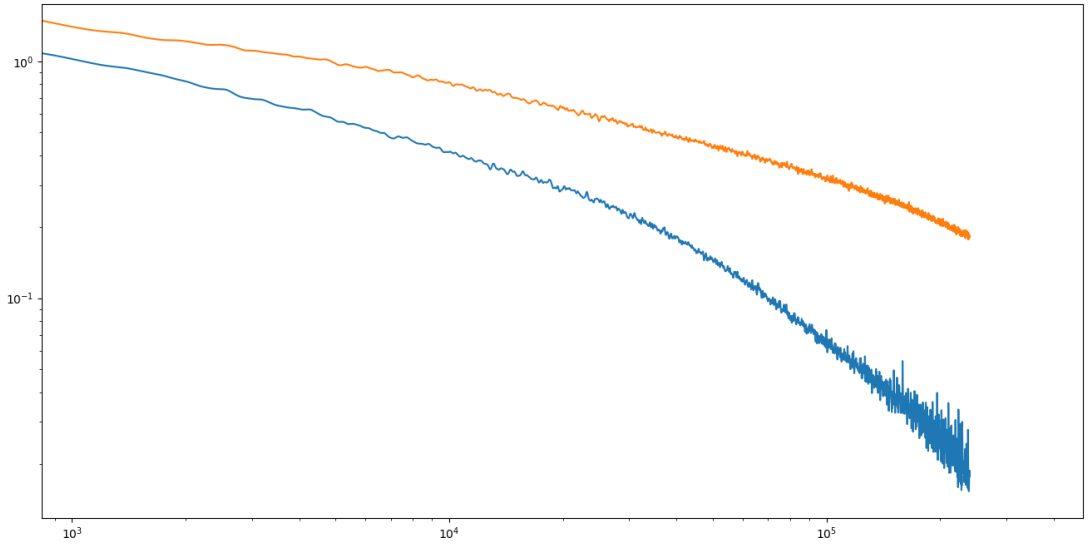
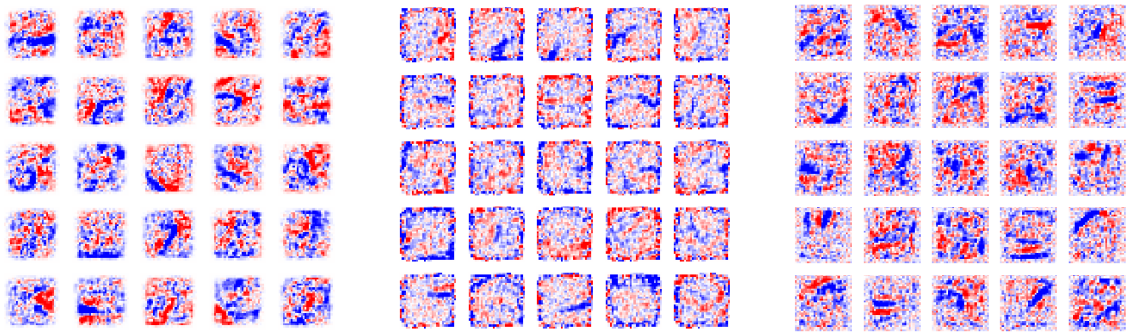


Figure 23: Log-Log axis showing Adam (Blue) requiring 4 times fewer iterations to reach the same cost function than before.



Non-Adam Adam Adam with random noise

Figure 24: Adam first layer activations are greater at the edges, which is not ideal. Adding random noise to the images fixes this and returns the characteristic patters of number recognition.

5 Convolutional Neural Network

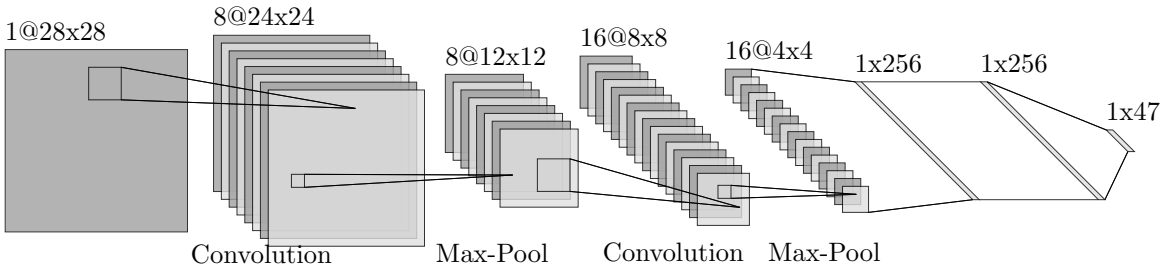


Figure 25: Convolutional Neural Network architecture [2]

Modern deep learning involving computer vision is completed using convolutional neural networks. These use a series of convolutions of the image and filters to extract meaningful features, which can then be used to decide what is in the image. The main advantage of the CNN over the classic fully connected neural network (FCNN) is one of space and speed. The memory required to store the parameters of the network is proportional to the fourth power of the resolution of an image. Therefore memory storage and back-propagation speed will become limiting factors quickly.

A CNN works in three stages:

- Convolution - The image (or multiple feature maps) is convolved with a kernel, which extracts a feature of the image, for instance a vertical edge. Multiple kernels may be used on one feature map or, to speed the algorithm up, you can do a depth-wise separable convolution. This dramatically reduces the number of computations.
- Non linearity - For any neural network, non-linearity is required for complex decision making. I used a Rectified Linear Unit (ReLU), which maps any negative values to 0 and any positive values to itself.
- Max Pooling - To reduce the size of the image quickly, the image is simplified by taking the average or maximum of a local group of pixels and generating a new image. I used 2x2 max pooling to reduce the size of the image by 4 times after every CNN layer.

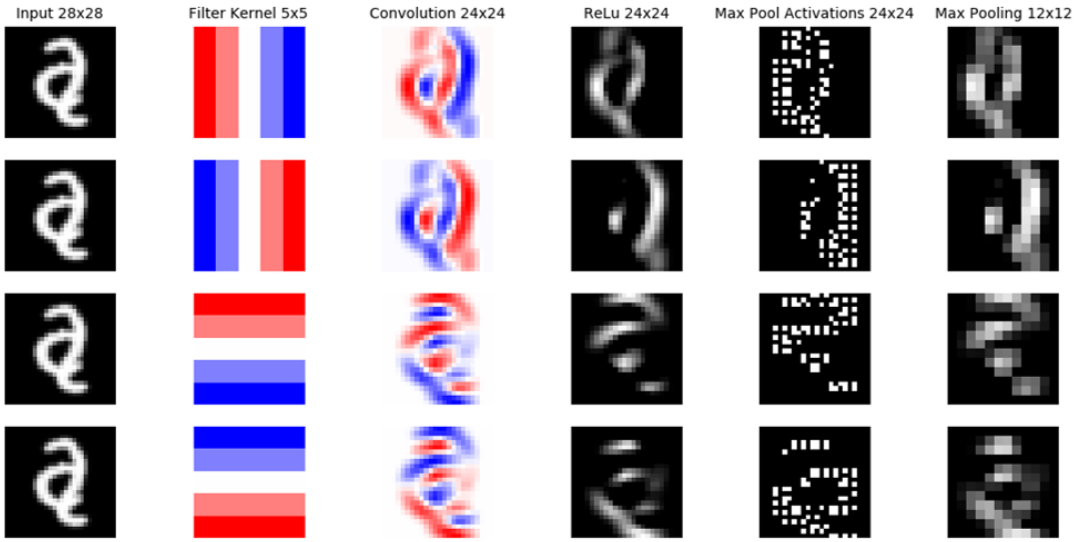


Figure 26: The key processes in a Convolutional Neural Network

Instead of convolving a 3D filter with a 3D set of feature maps to generate a new feature map, it is faster to convolve each input channel with their own 2D kernel, then use a 1x1 channel-wise kernel to create the new feature map. More output feature maps can then be generated by using different 1x1 kernels, without the need to convolve from scratch again. This is known as Depth-wise Separable Convolution and saves computation time while only slightly reducing the ability of the network to extract features [6].

To implement this in code, it is important to note that there are 4 dimensions to be dealt with: batch-wise (all the images in a batch), channel-wise (all the feature maps of an image) and the x and y axes of the feature maps themselves. As depth-wise separable convolution means that kernels will act on only one channel, it makes sense to permute the axes such that the channel dimension is at the start. So the 4D tensors throughout the network are ordered Channel-Batch-X-Y (CBXY).

For back-propagation, we need to cache the pixels triggered during max pooling as these are the only ones which influence the cost function, so they influence the derivatives. We also need to cache the feature maps before and after convolution throughout the network to calculate the derivatives of the 2D kernels and 1x1 depth-wise kernels, which themselves will be calculated with convolutions.

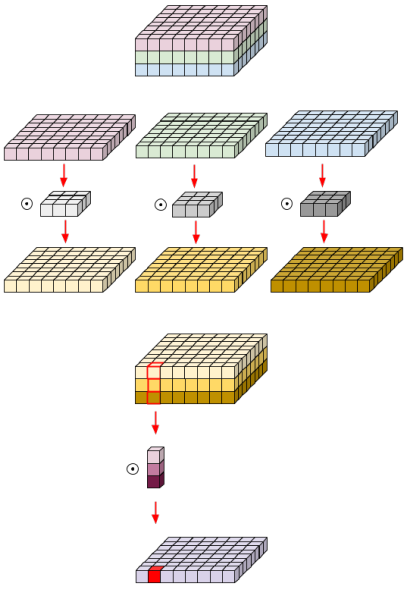


Figure 27: Depth-wise separable convolution [6]

Convolutional Neural Network forward

```
# Pass forward through the convolutional neural network
def cnn_forward(input_tensor, filt, depth_filt):

    # Depthwise seperable convolution
    conv = np.array([signal.convolve(fm, k, mode='valid')
                     for fm, k in zip(input_tensor, filt)])
    y = np.einsum('ij,jklm->iklm', depth_filt, conv)

    # ReLU and max-pooling, caching the activation tensors
    relu = y*(y>0.001)
    a, b, M, N = relu.shape
    mxpl = relu.reshape(a, b, M//2, 2, N//2, 2).max(axis=(3, 5))
    activ = np.repeat(np.repeat(mxpl, 2, axis=2), 2, axis=3) == y

    return conv, mxpl, activ
```

5.1 Convolution Back-propagation

The forward convolution process is:

$$\mathbf{C}_i = \mathbf{M}_{0i} * \mathbf{S}_i$$

Where $\mathbf{M}_i$ is a separated input feature map, $\mathbf{S}_i$ is a separable kernel and $\mathbf{C}_i$ is the outputted convolution. Then there is a depth-wise convolution:

$$\mathbf{Y} = \mathbf{C} * \mathbf{D}$$

Where $\mathbf{C}$ is the combined tensor of the previous convolutions, $\mathbf{D}$ is a 1x1 depth-wise kernel and $\mathbf{Y}$ is the outputted convolution. We then ReLU and max-pool to get the next layer:

$$\mathbf{M}_1 = max(0, \mathbf{Y}) \circ \mathbf{A}$$

Where $\mathbf{A}$ is an activation tensor which is 1 for values which are passed through in max-pooling and 0 otherwise. Passing backwards, we find $\frac{\partial G}{\partial \mathbf{Y}}$ by doing the Hadamard product of the activation tensor $\mathbf{A}$ and $\frac{\partial G}{\partial \mathbf{M}_1}$, where $\mathbf{M}_1$ is the repeated ("un-max-pooled", if you will) input of the next layer.

$$\frac{\partial G}{\partial \mathbf{Y}} = \mathbf{A} \circ \frac{\partial G}{\partial \mathbf{M}_1}$$

Having experimented manually with smaller examples, I discovered that $\frac{\partial G}{\partial \mathbf{D}}$ is the element-wise multiplication and sum of the input ($\mathbf{C}$) and $\frac{\partial G}{\partial \mathbf{Y}}$. This arises from the fact that the multiple depth-wise kernels effectively as a matrix, where each element multiplies a 2D feature map (3D if batch dimension is considered). To code this I used Einstein summation notation (in Python `np.einsum`) as it is easy to manipulate for complex tensor multiplications such as these.

$$\frac{\partial G}{\partial \mathbf{D}}_{ji} = \mathbf{C}_{iklm} \frac{\partial G}{\partial \mathbf{Y}}_{jklm}$$

To progress backwards through the network, we must then find $\frac{\partial G}{\partial \mathbf{C}}$. This is simply the matrix multiplication of the transpose of the depth-wise kernel matrix and $\frac{\partial G}{\partial \mathbf{Y}}$.

$$\frac{\partial G}{\partial \mathbf{C}}_{jklm} = \mathbf{D}_{ij} \frac{\partial G}{\partial \mathbf{Y}}_{iklm}$$

To find the derivative with respect to the separable filter $\mathbf{S}$, we separate $\frac{\partial G}{\partial \mathbf{C}}$ into channels $\frac{\partial G}{\partial \mathbf{C}_i}$ and correlate with the input feature map $\mathbf{M}_i$. However, as the forward operation was a convolution with the separable filter being the rotated kernel, we rotate this correlation by 180 degrees about the x-y plane.

$$\frac{\partial G}{\partial \mathbf{S}_i} = \mathbf{M}_i \otimes \frac{\partial G}{\partial \mathbf{C}_i}; (x, y) \rightarrow (-x, -y)$$

To progress backwards to the previous layer, we need to be able to find the derivative of the cost function with respect to the layer inputs ($\frac{\partial G}{\partial \mathbf{M}}$). This derivative is a "full" correlation between the separable filter kernel $\mathbf{S}_i$ and $\frac{\partial G}{\partial \mathbf{C}_i}$. The separable layers are then combined back together to find $\frac{\partial G}{\partial \mathbf{M}}$. There is no need the rotate this correlation as the input feature map was effectively stationary in the forward convolution.

$$\frac{\partial G}{\partial \mathbf{M}_i} = \frac{\partial G}{\partial \mathbf{C}_i} \otimes \mathbf{S}_i$$

This process is not required for the first layer as we do not need to know the derivatives at the input of the CNN.

Convolutional Neural Network backward implementation

```
# Pass backward through the convolutional neural network
def cnn_backward(conv_deriv, activ, conv, depth_filt, input_tensor, filt=0):

    # Find derivatives at output of layer, different if last layer before FCNN
    if filt:
        output_deriv = np.array([signal.correlate(cd, k, 'full')
                                 for cd, k in zip(conv_deriv, filt)])
    else:
        output_deriv = conv_deriv

    # Find depthwise filter and separable filter derivatives
    y_derivs = np.repeat(np.repeat(output_deriv,2,axis=2),2,axis=3) * activ
    depth_deriv = np.einsum('iklm,jklm->ji', conv, y_derivs)
    conv_deriv = np.einsum('ij,iklm->jklm', depth_filt, y_derivs)
    filt_deriv = np.array([np.rot90(signal.correlate(fm, cd, 'valid'),2,(1,2))
                           for fm, cd in zip(input_tensor, conv_deriv)])

    return depth_deriv, conv_deriv, filt_deriv
```

5.2 Object oriented programming approach

As the convolutional neural network is more complex than a fully connected neural network, I thought it appropriate to use an object oriented approach when programming it in Python. At the centre of it is a `Convolutional_NN` class, with properties such as the learning rate and layer architecture. There are several methods as well, such as initialising the parameters and executing the learning process. The main change I made here was adding a `check_inputs` method which asserts that the various hyper-parameters which define the network are compatible. For example, the layer, filter and kernel sizes much all be positive integers and the final layer must have 47 outputs, as this is the number of classes we are categorising.

`Convolutional_NN` class and method to check inputs

```
class Convolutional_NN:
    def __init__(self, batch_size, filter_sizes, kernel_sizes, fc_layers,
                 learning_rate, iterations, regular_coef=0.0, adam=False):
        self.b_size = batch_size
        self.f_sizes = filter_sizes
        ...

    # Check network hyper-parameters are of the correct type and size
    def check_inputs(self):

        # Batch sizes
        assert type(self.b_size) is int, "Batch Size must be integer"
        assert self.b_size > 0, "Batch size must be positive"

        # Filter sizes
        assert type(self.f_sizes) is tuple, "Filter sizes must be a tuple"
        for f in self.f_sizes:
            assert type(f) is int, "All filter sizes must be integers"
            assert f > 0, "All filter sizes must be positive"

        # Kernel sizes
        assert type(self.k_sizes) is tuple, "Kernel sizes must be a tuple"
        for k in self.k_sizes:
            assert type(k) is int, "All kernel sizes must be integers"
            assert k > 0, "All kernel sizes must be positive"
            assert k % 2 == 1, "All kernel sizes must be odd"

        # Fully connected layers
        assert self.layers, "Must input fully connected layers tuple"
        assert type(self.layers) is tuple, "Layers must be tuple"
        for l in self.layers:
        ...
        print("Hyperparameters OK")
```

To initialise and run the network, the parameters of the network are set when creating the class and then the `learn` method is called:

Running the `Convolutional_NN` class

```
nn1 = Convolutional_NN(batch_size = 64,
                       filter_sizes = (8, 16),
                       kernel_sizes = (5, 5),
                       fc_layers = (256, 256, 47),
                       learning_rate = 10 ** -3,
                       iterations = 1000,
                       regular_coef = 0.0,
                       adam = True,
                       elastic = False)
nn1.learn()
```

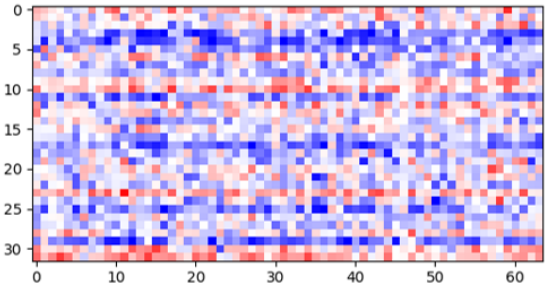
5.3 Batch Normalisation

Batch normalisation is the process of making a node's value normally distributed, with some mean and standard deviation. Ordinarily, it is common for nodes to vary by different amounts for images in a batch. This can lead to learning difficulties where the steepest gradient is in a direction which may go 'down the valley and up the hill' too quickly. We want the nodes' derivatives to behave similarly as we treat them the same in the learning process.

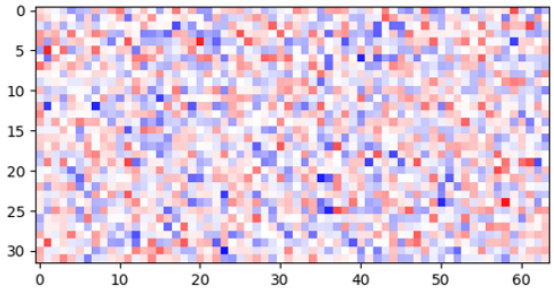
Batch norm works by taking the batch-wise mean away from the values of nodes and then dividing by the standard deviation. This also does away with the need for biases as the mean value for a node in the batch is always 0.

$$\hat{\mathbf{X}} = \frac{\mathbf{X} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

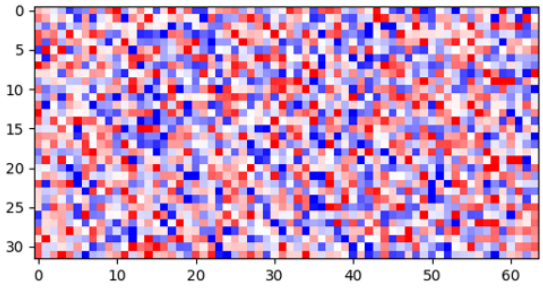
Where $\mathbf{X}$ is the matrix input to a layer, $\boldsymbol{\mu}$ is a vector of batch means, $\boldsymbol{\sigma}$ is a vector of batch standard deviations and $\hat{\mathbf{X}}$ is the Batch Norm output.



(a) Batch Norm input, note batch-wise streaks



(b) Adjusting mean



(c) Dividing by standard derivation

Figure 28: Batch Normalisation process

The right images show the Batch Norm process. Firstly we have the input - this was generated by multiplying a random weight matrix by a batch of images. Note that in the horizontal batch-wise direction there are clear streaks of higher or lower values, which we want to correct for. Then the mean of each node is subtracted. In some batch-wise direction (eg. 8 and 9) the values deviate less than in others. This is fixed in the bottom image, which has divided each node value by the batch standard deviation. Now there are much more more distinct, *meaty* values which will be easier to learn from.

Batch Norm forward implementation

```
if BN:
    sigma = np.std(ith_layer, axis=1)[:, None]
    sigmas.append(sigma)
    ith_layer = (ith_layer - np.average(ith_layer, axis=1)[:, None]) / sigma
    vals_hat.append(ith_layer)
```

The Batch Norm derivative is:

$$\frac{\partial G}{\partial \mathbf{X}} = \frac{n\frac{\partial G}{\partial \hat{\mathbf{X}}} - \sum_b \frac{\partial G}{\partial \hat{\mathbf{X}}} - \hat{\mathbf{X}} \sum_b \frac{\partial G}{\partial \hat{\mathbf{X}}} \hat{\mathbf{X}}}{n\boldsymbol{\sigma}_b}$$

Where n is the number of images in a batch and b represents a summation in the batch-wise direction [7].

Batch Norm backward implementation (bd are node derivatives)

```
if BN:
    a = BATCH_SIZE * bd
    b = np.sum(bd, axis=1)[:, None]
    c = vals_hat[-i] * np.einsum('ij,ij->i', vals_hat[-i], bd)[:, None]
    bd = (a - b - c) / (BATCH_SIZE * sigmas[-i])
```

Unfortunately I didn't see any performance benefits from batch normalisation. I am unsure why this might be, but it may be because the inputs of the neural network (pixels) all have the same significance so the nodes are already approximately distributed with a similar standard deviation.

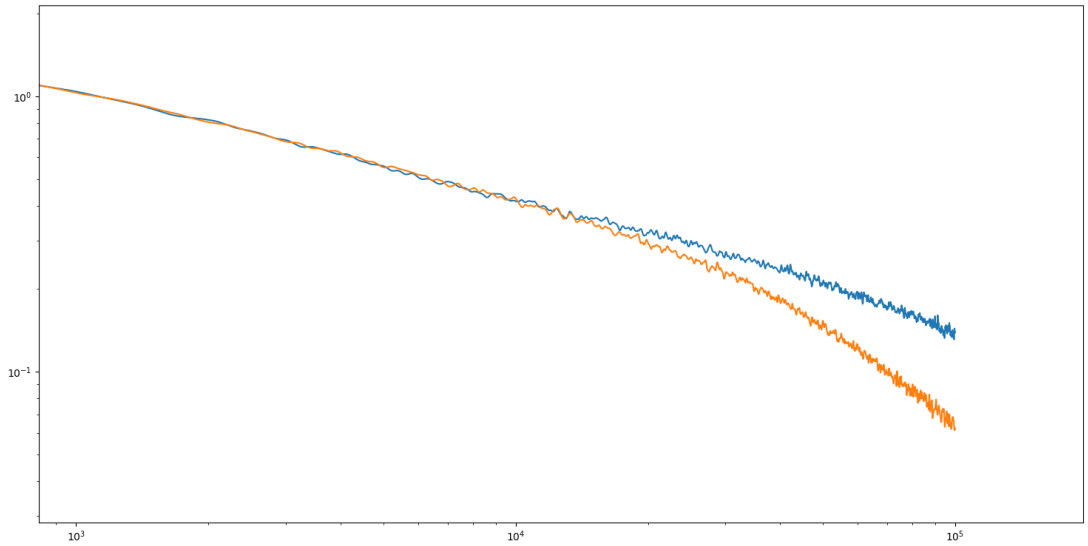


Figure 29: Batch Norm (blue) vs normal learning (orange)

6 Conclusions

- Neural networks can effectively memorise characters. As the characters are simple, just 28x28 pixels, a neural network can quickly learn them to a reasonable degree of accuracy. However, it is important the inputted training data it correctly labeled as the network will still be able to learn the incorrect training data, not realising that it is false. This will reduce its accuracy on previously unseen images.
- Minimising a cost function does not necessarily improve accuracy. If a network is unrestricted to reduce its cost function, it may over-fit its training data and extrapolate too strongly to other areas of the input sample space. Methods such as regularisation and elastic distortion can help to improve accuracy when there are no further gains to be made with the cost function.
- Convolutional neural networks work well on images, however for such small pictures a fully connected network is faster and just as competent. Modern computers can easily hold the required memory to store weights and biases in RAM, so it remains fast. Further, my computer was much faster at multiplying matrices than performing convolutions. Convolutions become relatively faster when the image you are convolving is larger because the Fast Fourier Transform (FFT) is more effective.
- Batch normalisation does not always work. It is most suited to problems where the inputs represent very different things, for instance age and height.

References

[1] 3 Blue 1 Brown (Grant Sanderson)
Neural networks
November 2017
https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

[2] Alex Lenail
Publication-ready NN-architecture schematics
http://alexlenail.me/NN-SVG/

[3] Dr Richard E. Turner
Deep Learning 2P8 lectures
May 2020

[4] Chris Crawford
EMNIST (Extended MNIST) - An extended variant of the full NIST dataset
December 2017
https://www.kaggle.com/crawford/emnist

[5] Diederik P. Kingma, Jimmy Ba
Adam: A Method for Stochastic Optimization
January 2017
https://arxiv.org/abs/1412.6980

[6] Ely Bendersky
Depthwise separable convolutions for machine learning
April 2018
https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/

[7] Kevin Zakka
Deriving the Gradient for the Backward Pass of Batch Normalization
Spetember 2016
https://kevinzakka.github.io/2016/09/14/batch_normalization