MLP Coursework 2 Report

s1311631

November 24, 2016

1 *Training and Validation performance variation with model depth*

In this experiment we compare both the overall performance against the validation set and the relative performance (i.e. degree of over-fitting or lack of generalisation) between the *training* and *validation* datasets.

The network model is a simple one, chosen mainly to eliminate further factors introduced by variable learning rates or other such optimisations, and also to exacerbate the variation induced in performance by altering the number of model layers. This simple model is composed of *Affine Transform* layers interleaved with *Sigmoid Layers*, following the 'vanilla' *Gradient Descent* learning rule.

The number of epochs each model sees is kept with the same value of 200, as is the learning rate η (which is held at 0.1[1])

[1] Learning rate $\eta = 0.1$ was a generally good/safe value as evidenced by the experiments in the first assignment.

1.1 *Experimental Setup*

The number of (Affine) layers is varied from 1 to 8 with increments of 1. The regime for performing the experiment is in source files `experiments/expt1.py` and `experiments/conf/expt1.json`. No 'control' experiment is run though we can compare performance with other models trained over the same data with different architectures. (For what it's worth, a model depth of 3 was used for all the previous assignment's experiments)

Learning Rate hyperparmeter $\eta = 0.1$, models have first layer *Input Data Dimension* $= 784 \rightarrow$ *Hidden Dimension* $= 100$, and final layer with *Output Dimension* $= 10$. The exceptions to this is the single layer network which has no hidden layers.

1.2 *Results*

identifier	time*	valid_err	train_err	valid_acc	train_acc
expt1_1layer	52	0.2580	0.2430	0.9291	0.9329
expt1_2layer	169	0.0835	0.0465	0.9755	0.9891
expt1_3layer	234	**0.0799**	0.0259	0.9767	0.9952
expt1_4layer	249	0.0976	0.0115	0.9762	0.9987
expt1_5layer	372	0.1354	**0.0071**	0.9699	**0.9993**
expt1_6layer	444	0.1787	0.0231	**0.9620**	0.9957
expt1_7layer	505	2.3014	2.3004	0.1064	0.1136
expt1_8layer	478	2.3022	2.3012	0.1064	0.1136

Figure 1: The final accuracies and errors against the validation set at epoch 100 of each model in the experiment. The best value for each metric in the experiment is shown in **boldface**. All values are to 4 decimal places.

*Note that runtimes are for relative comparison only, as experiments were performed in parallel on heterogeneous hardware.

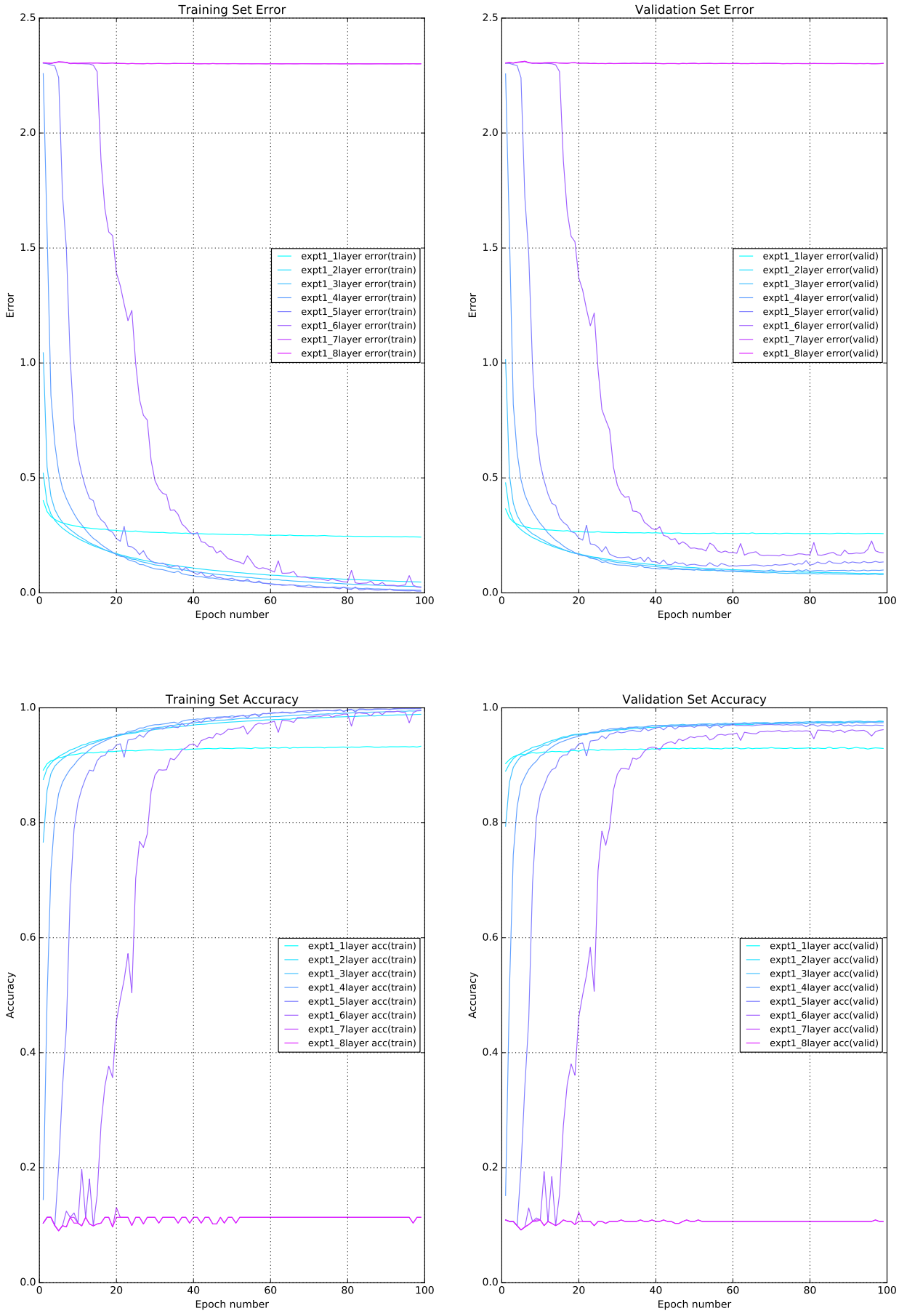

Figure 2: Accuracies **top** and Errors **bottom** over the training set for the **left** and the validation set **right** for the *Variable Depth Models*

1.3 *Observations*

What's immediately obvious from the plots is that for models with more than 6 layers training fails, with the model that had 6 layers showing bouncy and initially slow learning. The single layer network (`expt1_1layer`) shows fast training but high error and low accuracy as one would expect, with a single affine transform not being expressive enough to fully capture all the features in the data.

Models deeper than 6 layers have accuracies and errors basically unchanged from their initialisation values in epoch 0 to 100, and have failed to learn over the data. One hypothesis about the cause of this is the *Vanishing Gradient Problem*[2] – Even though these networks aren't really that deep, as the error is back propagated to the earlier layers, the error assigned to each weight approaches zero and therefore weight updates become ineffective or even cause floating point underflow.

[2] Glorot, X. and Bengio, Y., 2010, May. *Understanding the difficulty of training deep feedforward neural networks.* In AISTATS (Vol. 9, pp. 249-256).

The Sigmoid layer's gradient (derivative) varies over the range $(0, 0.25]$. It is given as shown in Equation 1:

$$\frac{d}{dx}\frac{1}{1+e^{-x}} = \frac{e^x}{(e^x+1)^2} \tag{1}$$

To investigate whether *Vanishing Gradient* is the cause, the 8 layer `expt1_8layer` experiment is re-run, with additional statistics capturing the *argmax* (largest member gradient) as we back propagate for each layer (both *Sigmoid* and *Affine Transform*). The progression of these gradients are plotted in Figure 3 over the course of training (100 epochs).[3]

[3] Note that the 8 layer model has 14 gradient series, as there are 8 *Affine* layers and 6 *Sigmoid* layers.

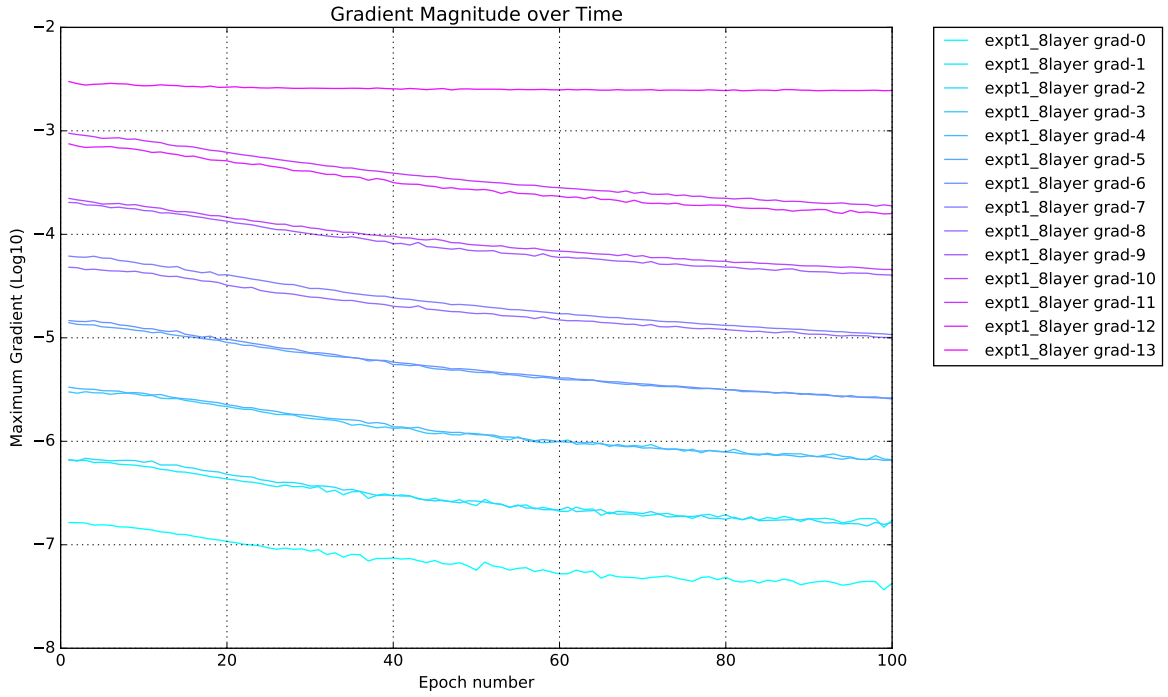

Figure 3: Vertical axis is the base 10 logarithm of the maximum gradient, horizontal is epoch. grad-0 is the input layer, even layers are *Affine Transforms* and odd are *Sigmoid*

As evidenced by Fig. 3, at each model layer pair the gradient looses roughly an order of magnitude – the output layer being in the region 1×10^{-2} and the input layer at 1×10^{-7}. Interestingly, the output layer's gradient is a fairly constant throught the experiment.

In contrast to this, the `expt1_4layer` has the following plot over it's layers (Fig. 4)

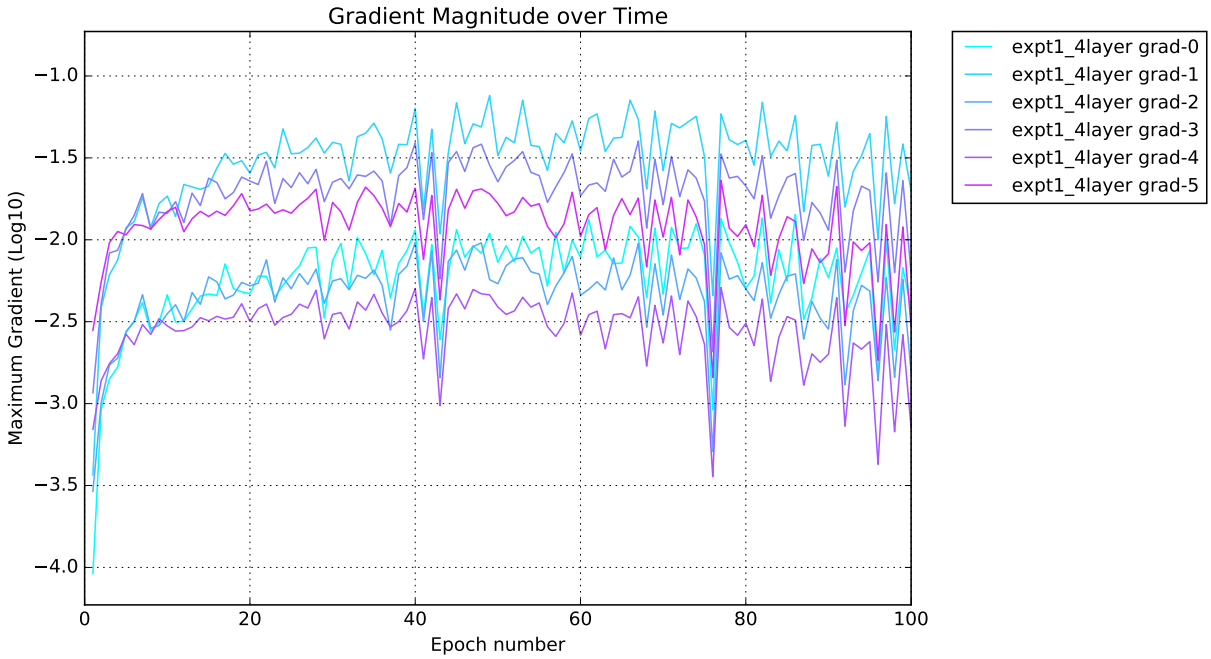

Figure 4: Vertical axis is the base 10 logarithm of the maximum gradient, horizontal is epoch. grad-0 is the input layer, even layers are *Affine Transforms* and odd are *Sigmoid*

There is notably less difference between the input, hidden and output layers, and all the *Affine* layers are grouped, with the *Sigmoids* grouped below. Additionally, the magnitudes are in the 1×10^{-2} region, much larger than the 1×10^{-7} in the 8 Layer Model. The magnitudes show significantly more movement epoch to epoch as well. This 4 Layer Model trained to an validation set error of 0.0976 and accuracy 0.9762 over the 100 epoch regime.

Given these differences, I conclude that the overly deep models that failed to train did so as a result of vanishing gradients, and that increasing the depth of a model (to an extent) can improve accuracy, in this case achieving a maximum accuracy of 0.9993 for a 5 layer net.

2 *Data Augmentation*

Some of the more complex models developed in the first assignment (*RMSProp*, *AdaGrad*, Fig.5) showed significant over-fitting to the training data and as such developed a growing error with respect to the validation data as training progressed. We posit that a countermeasure for this may be data augmentation – creating additional training examples from the existing set by applying a transformation [4] to the data themselves in a `DataProvider`, thereby artificially growing (i.e. augmenting) the training set.

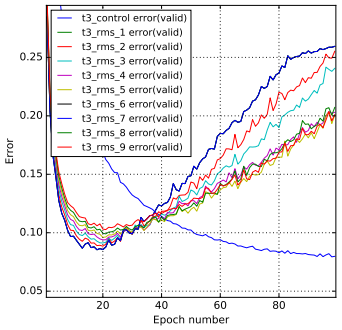

Figure 5: Subplot illustrating the divergence away from the minimum achieved error as training continued for the *RMSProp* Experiments from Assignment 1.

2.1 *Experimental Setup*

We train three models, one as a *control* or reference model that has no deformation applied to it's input data, and another two that have 25% of their input images rotated or blurred, respectively. The magnitude of deformation, which can be though of as how far in feature-space this new augmented data-point is from the original should be relatively short, so as to avoid introducing examples that are very hard to classify or have been changed so much that they are no longer representative of the class they belong to. Figure 6 shows a sample batch with a *Gaussian Blur* (specifically, a `scipy.ndimage.filters.gaussian_filter`) augmentation applied to 25% of the images.

[4] This could either be *Affine*, as in to preserve parallel relationships (semantics), rotational or a kernel convolution (e.g. a *Gaussian* blur).

For example, a 6 rotated π^r becomes a 9, but will still be labelled a 6; this is not a useful training example.

Figure 6: A grid of 100 random images drawn from the *MNIST* dataset, with a *Gaussian Blur* applied to a randomly (uniform) sampled subset of 25% of the images; The blur distance is selected again by a uniformly distributed random number drawn from $[-1, 1]$.

Left shown in greyscale, and **Right** a perceptually uniform colourspace which helps to highlight the effect of the blur. Apologies, I am aware it's visually offensive.

We will compare these three model species to each other, and also experiment over some of the possible parameters, namely the interval the uniform distribution draws from, adjusting how aggressive the transform is. The augmentation will always affect 25% of the batch, and is only applied to the training dataset, not the validation set.

The models all have the same architecture, 3 *Affine Transform* layers interleaved with *Relu* layers. We vary the models only on the hyperparameter ϕ (introduced §2.1.1), keeping the learning rate η a constant 0.1.

2.1.1 *Gaussian Blur Filter*

Random members of the input data set X, $x_i \in X$ are *augmented* to produce a new data point $x_i^{'}$. U is the Uniform Distribution, and ϕ is the weight hyperparameter that controls the magnitude, or extent of the augmentation. In the case of the blur is can be though of as proportional to the standard deviation of the filter's kernel.

$$x_i^{'} = \text{gaussian}(\text{U}(-1, 1) \times \phi) \tag{2}$$

2.1.2 *Rotation*

Similarly, a rotation is applied to some random members of the input dataset, X. In this case, ϕ multiplies the angle produced by the random number generated by U, representing the maximum possible deflection.

$$x_i^{'} = \text{rotation}(\text{U}(-1, 1) \times \phi) \tag{3}$$

Code that performs these augmentations is in the source file `mlp/augmentations.py`

2.2 *Results*

identifier	ϕ	**train_err**	**valid_err**	**train_acc**	**valid_acc**
expt2_control	*n/a*	**0.00042**	0.1209	**1.0000**	0.9779
expt2_blur1	0.5	0.00045	0.1187	**1.0000**	0.9788
expt2_blur2	1.0	0.00156	0.1189	0.9996	0.9776
expt2_blur3	1.5	0.00307	0.1138	0.9993	0.9780
expt2_blur4	2.0	0.00500	0.1168	0.9986	0.9780
expt2_rotate1	10.0	0.00435	0.0906	0.9988	0.9818
expt2_rotate2	15.0	0.00710	0.0798	0.9981	**0.9829**
expt2_rotate3	20.0	0.01199	0.0764	0.9968	0.9818
expt2_rotate4	25.0	0.01537	**0.0748**	0.9954	0.9828

Figure 7: The final accuracies and errors against the validation set at epoch 100 of each model in the experiment. The best value for each metric in the experiment is shown in **boldface**.

Note that comparing the values for η between the two different augmentations has no meaning.

All values are to 4 decimal places, 5 in the case of the training error.

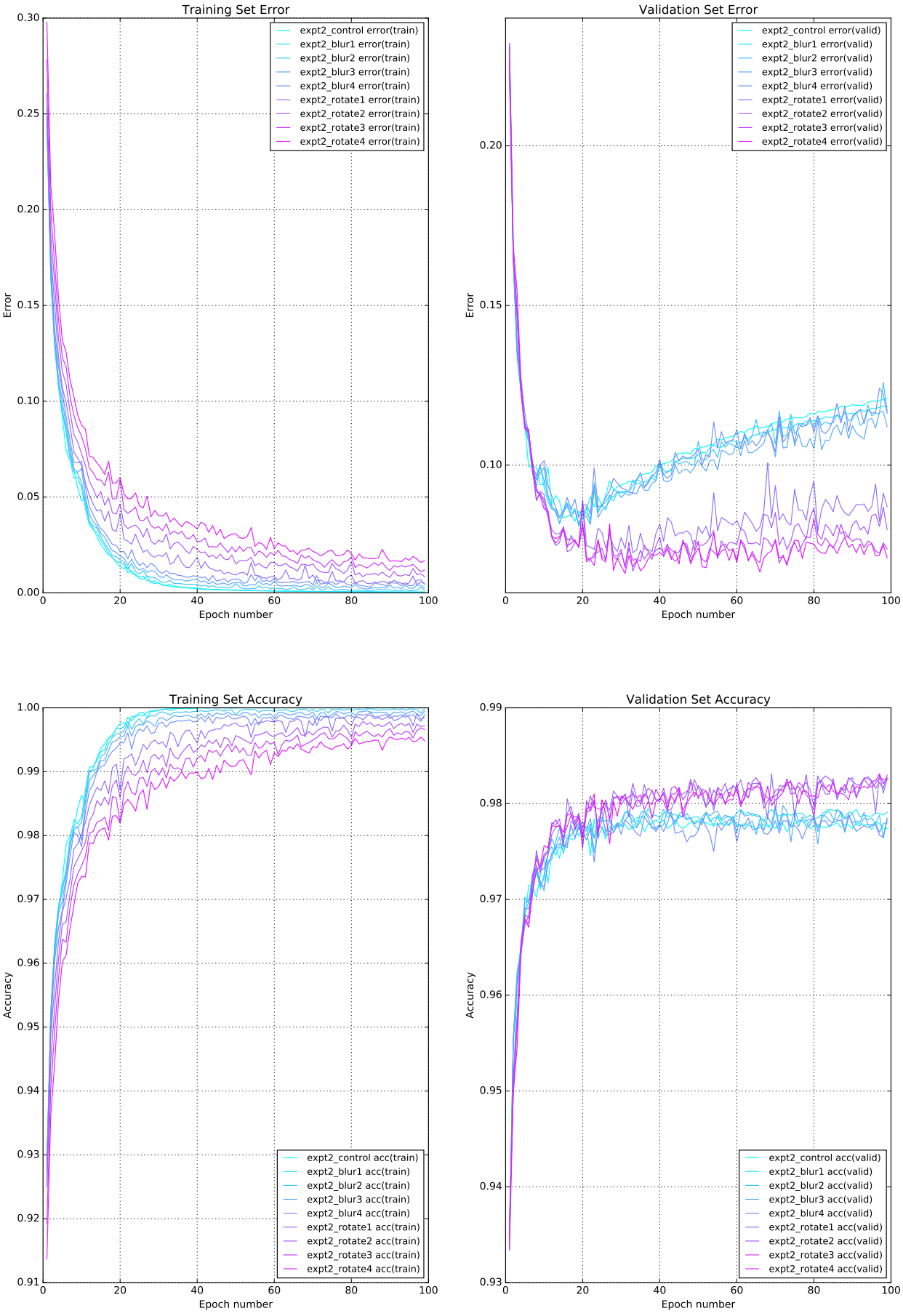

Figure 8: Accuracies **top** and Errors **bottom** over the training set for the **left** and the validation set **right** for the *Data Augmentation Models*

2.3 *Observations*

All models demonstrated an ability to fully capture the features in the data, with training set errors near 0 and accuracies near 1. However, the control model and all the models whose training set was augmented by blurring show strong over-fitting to the training set, with increasingly poor performance when compared to the validation set as training progressed. From these experiments it would appear that rotations out perform the blurs, showing good generalisation (no over-fitting) and high accuracies.

Furthermore, the *more aggressive* blurs performed best[5]. In general, rotations outperforming blurs makes intuitive sense – the whole point of augmenting our data is to evolve reasonable new examples from the existing data that are likely not in the training set but are still representative of the target domain; in the case of MNIST, handwritten digits will be written with varying slant and appear at different rotations within the dataset. Blurs, however are less representative, having the effect of removing an obvious edge from the written characters. Very few data (if any) in MNIST have the appearance of a highly blurred digit. The blurred digits, in effect are a poor attempt at reducing the variance or spurious correlations in the data that would perhaps be better addressed by using a *whitening process* or *principle component analysis*.

[5] `expt2_rotate4` has the lowest validation set error at 0.0748, and a good validation accuracy 0.9828 (best was `expt2_rotate2` at 0.9829)

It also follows logically that over-enthusiastic augmentation (such as flipping a digit over producing a non-number) or augmenting too large a sample of the dataset would have a detrimental effect on performance.

I surmise that the parameter-space explored in this experiment (§2.2) did not encroach on a region where this is a case, so as a follow-up we will take the `expt2_rotate4` experiment and test larger rotations and sample sizes.

2.4 *Further Rotational Augmentation*

These further experiments are designed to push rotational augmentation past it's optimal point. Intuitively, should we aggressively rotate all of the data-points in the training set, then the evolved model should be a poor classifier over both the training and validation sets – only a small amount of the data it would have learnt on would be in the original set.

identifier	ϕ, %	**train_err**	**valid_err**	**train_acc**	**valid_acc**
expt2_control	*n/a*	**0.0004**	0.1209	1.000	0.9779
expt2b_sam20	20	0.0101	0.0814	**0.9974**	**0.9820**
expt2b_sam40	40	0.0253	**0.0697**	0.9925	**0.9820**
expt2b_sam60	60	0.0555	0.0907	0.9821	0.9753
expt2b_sam80	80	0.0737	0.0997	0.9760	0.9703
expt2b_sam100	100	0.0847	0.1048	0.9728	0.9674

Figure 9: The final accuracies and errors against the validation set at epoch 100 of each model in the experiment. The best value for each metric in the experiment is shown in **boldface**.

Fig. 9 results are plotted Fig. 11

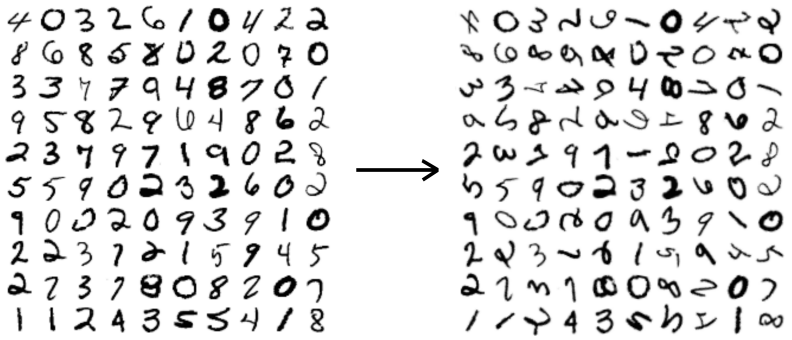

Figure 10: A grid of 100 random images drawn from the *MNIST* dataset, shown **Left** without any augmentation, and **Right** with an aggressive rotational augmentation applied to every digit. This is representative of the training set the model `expt2b_sam100` saw.

2.5 *Further Observations*

The performance of the models seems to be fairly insensitive to the value of ϕ (the magnitude of the transformation applied), with a wide range of values giving adequate accuracies over the validation data. All augmented models performed better than the control model, which over-fit the training data.

Having said all this, the results of this experiment aren't particularly supportive – the `expt2b_sam100` model had a very large amount of augmentation applied, yet is a reasonably good model. It may be the worst of the lot, though a validation accuracy of 0.9674 is not what one would call a failed classifier. Still, augmenting $20 - 40\%$ of the training data in a more gentle manner yields better performance, as per the `expt2b_sam20` and `expt2b_sam40` models.

2.6 *Conclusions on Augmentation*

These experiments imply that a degree of *Data Augmentation* will improve the accuracy of a given model compared to an unaugmented one, all other factors being equal. However, we have seen only a small gain in accuracy, 0.9779 for the control to 0.9820 for the best two augmented models (`expt2b_sam20` and `expt2b_sam40`). Augmentation appears to present more value as a technique for combating over-fitting or compensating for a small training set, though if the transform applied is overly aggressive it will have a negative effect on the learned model's accuracy.

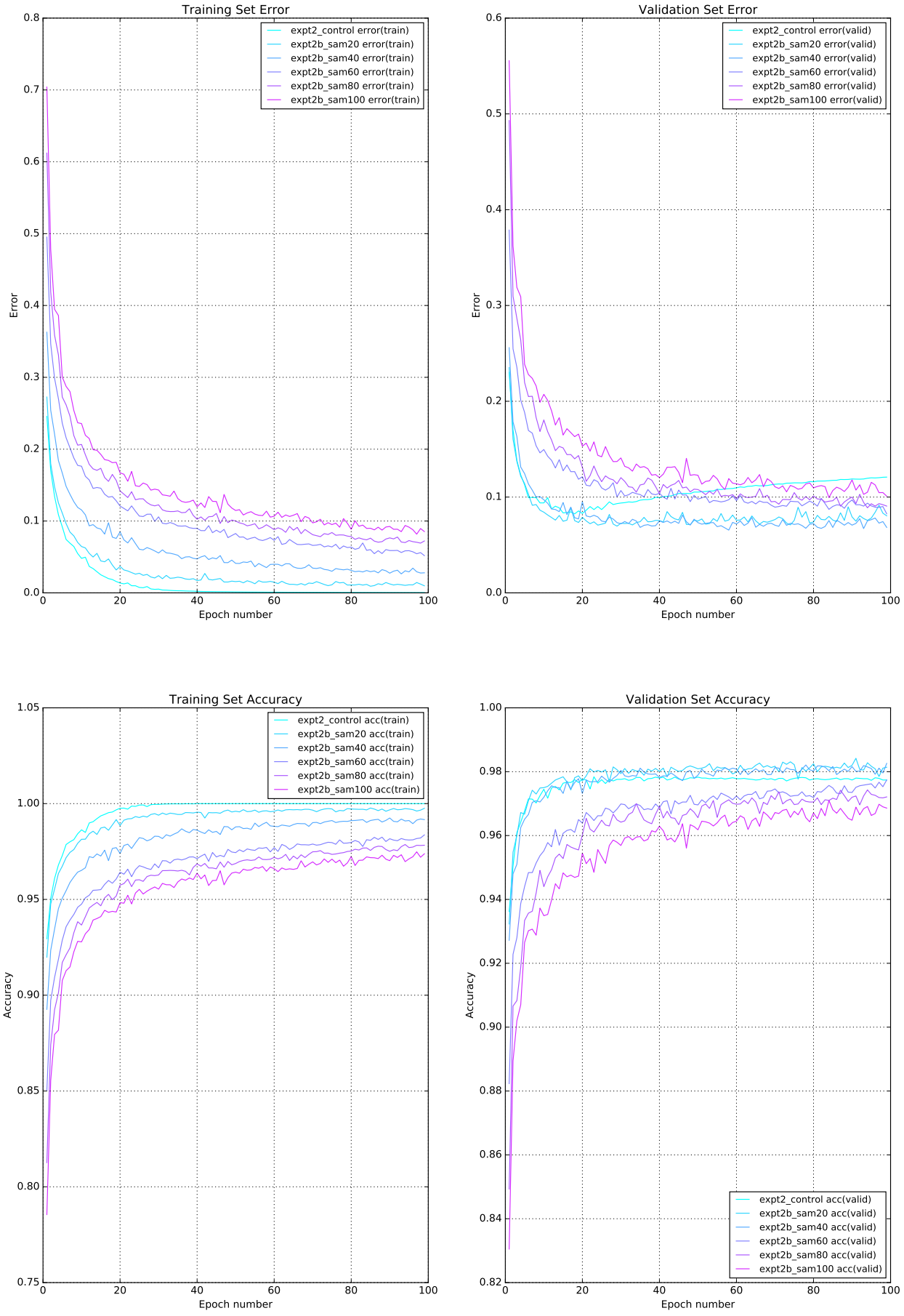

Figure 11: Accuracies **top** and Errors **bottom** over the training set for the **left** and the validation set **right** for the *Further Rotational Augmentation*

3 Batch Normalisation

In this experiment we investigate the efficacy of *Batch Normalisation*[6] for increasing model performance over a variety of metrics, primarily accuracy over the validation data set but also training time and validation error.

This Implementation of `BatchNormalizationLayer` is largely influenced by the framework presented here:

http://cthorey.github.io/backpropagation/

[6] Sergey Ioffe, Christian Szegedy, 2015, Feb.
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
https://arxiv.org/abs/1502.03167 [cs.LG]

3.1 Experimental Setup

A set of batch normalised models with varied learning rate, η are compared with a set of control models with no Batch Normalisation layers, whose learning rates are also varied.

Architecturally, all the models have 3 Affine Transform layers interleaved with Relu Layers:

```
model = MultipleLayerModel([
    AffineLayer(input_dim, hidden_dim, weights_init, biases_init),
    BatchNormalizationLayer(hidden_dim),
    ReluLayer(),
    AffineLayer(hidden_dim, hidden_dim, weights_init, biases_init),
    BatchNormalizationLayer(hidden_dim),
    ReluLayer(),
    AffineLayer(hidden_dim, output_dim, weights_init, biases_init)
])
```

The 'control' models are tagged expt3_ctrlN, and the batch normalised `expt3_bnN`.

identifier	η	train_err	valid_err	train_acc	valid_acc
expt3_ctrl1	0.01	0.0368	**0.0895**	0.9914	0.9735
expt3_ctrl2	0.02	0.0098	0.0934	0.9993	0.9753
expt3_ctrl3	0.03	0.0039	0.1039	0.9999	0.9764
expt3_ctrl4	0.05	0.0014	0.1116	**1.0000**	0.9768
expt3_ctrl5	0.08	**0.0006**	0.1192	**1.0000**	0.9775
expt3_bn1	1.0	0.0007	0.1033	0.9998	0.9806
expt3_bn2	5.0	0.0017	0.1321	0.9993	0.9811
expt3_bn3	10.0	0.0019	0.1956	0.9995	0.9789
expt3_bn4	15.0	0.0011	0.1821	0.9997	**0.9814**
expt3_bn5	20.0	0.0038	0.2552	0.9988	0.9785

Figure 12: The final accuracies and errors against the validation set at epoch 100 of each model in the experiment. The best value for each metric in the experiment is shown in **boldface**.
All values are to 4 decimal places.

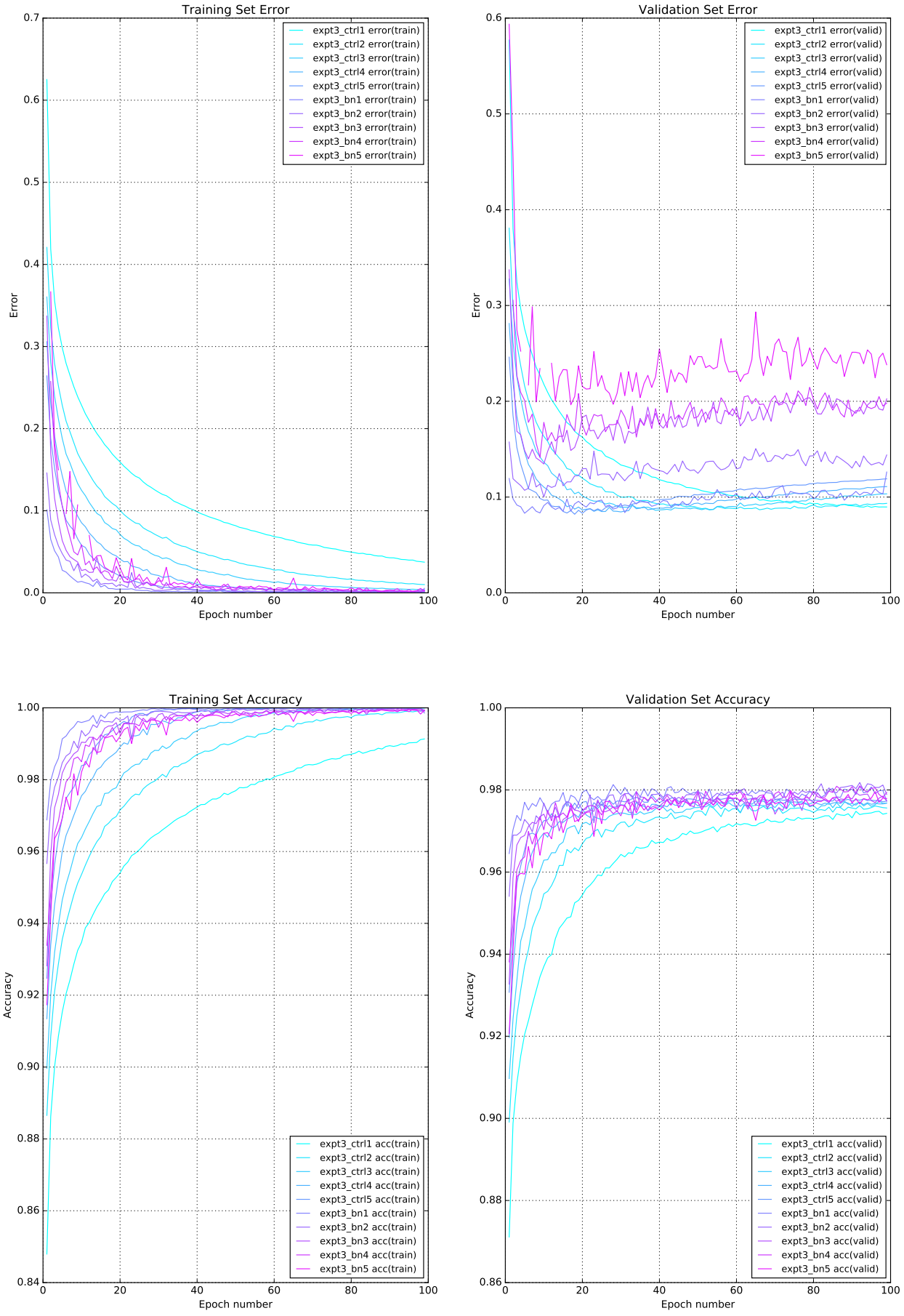

Figure 13: Accuracies **top** and Errors **bottom** over the training set for the **left** and the validation set **right** for the *Batch Normalization* experiments.

3.2 Observations

As expected, the *Batch Normalised* models with their higher learning rates do indeed converge upon their stable accuracies earlier than the reference control models. The Batch Normalisation architecture affords us the ability to use much higher learning rates for accelerated learning, as a result of *internal covariate shift*[7] being mitigated by the per-layer normalisation.

[7] Ioffe, Szegedy, §1, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*

However, while these models train quickly they show significant error and accuracy noise, or bouncing, in the later epochs of the training regime. This may be caused in part by the aggressive learning rate. Accuracy over the validation set in comparison to the control models shows a marginal improvement, up to around 0.9814 (expt3_bn4) compared to 0.9775 for the best control (expt3_ctrl5).

The networks trained with exceptionally high learning rates (expt3_bn5, $\eta = 20.0$) have breaks in the Validation Set Error series (top right, Fig. 13) caused by overflows occurring in the error calculation yielding `NaN`s. With even higher learning rates, these become more frequent and can cause the entire network to fail. The *Batch Normalised* networks also show a trend for having higher accuracies and lower errors over the validation set when the learning rate is lower (which is expected to be the case).

3.3 Batch Normalisation on Deeper Nets

One of the purported advantages of *Batch Normalisation* is that it eliminates the *Vanishing or Exploding Gradient* problem. You'll recall that this was a problem encountered way back in §1.2 and Figures 1 & 2. We will now revisit this problem and apply *Batch Normalisation* to allow the training of an 8 layer network, which was beyond the feasible size for the 'vanilla' net in §1 and examine it's performance and the backwards propagation of error through it's layers, relative to Figures 3 and 4.

We train an 8 layer model with alternating layers *AffineLayer*, *BatchNormalisationLayer* and *SigmoidLayer*. Each triple of these constitutes a model layer. The model's learning rate $\eta = 1.0$.

3.4 Results

identifier	train_err	valid_err	train_acc	valid_acc
expt4	0.000972012	0.110948	0.99966	**0.9800**

Figure 14: The accuracy and error against the validation set at epoch 100 of the 8 layer model in this experiment. The best value for each metric in the experiment is shown in **boldface**.

This shows a drastic improvement over the slow moving gradients from Figure 3 that lost an order of magnitude at each back-propagation stage. This plot is a more complex (due to the larger number of layers) and looks a lot more like it's sibling, Figure 4. From these data it is evident that *Batch Normalisation* does prevent the vanishing gradient problem.

Still, this much deeper network winds up with a validation accuracy of 0.9800 and error 0.1109, a performance that doesn't beat

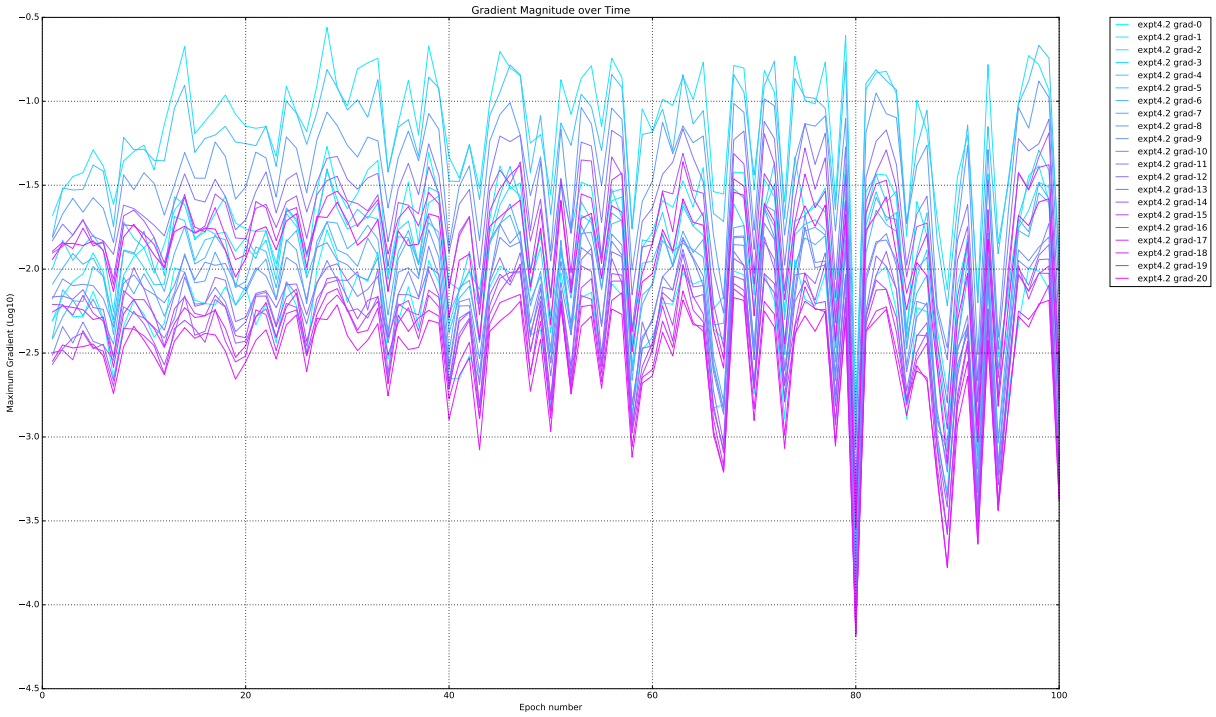

Figure 15: Vertical axis is the base 10 logarithm of the maximum gradient, horizontal is epoch. `grad-0` is the input layer, and the layer numbers from 0 cycle through *Affine Transform*, *Batch Normalisation* and *Sigmoid*.

Note that the layer types are clustered together into two distinct groups.

those from the *Data Augmentations* from §2.2 and indeed is not better than a handful of models elsewhere in this work. Nevertheless, the model trained successfully which is more than can be said for `expt1_8layer` (§1 Fig.1)

3.5 *Conclusions on Batch Normalisation*

With pertinence to the assigned task of training accurate networks, *Batch Normalised* networks show no significant improvement in accuracy over a model without this normalisation; Yet the technique merits consideration for a model's architecture as a result of the increased training speed, and for it's enabling of the use of yet deeper networks.