MLP Coursework 2: Learning rules, BatchNorm, and ConvNets

s1765864

Abstract

The results of this work are threefold. First, it presents three learning rules: stochastic gradient descent, RMSprop, and Adam. These learning rules are compared against one another on training a fully connected 3-layer model to classify images from the EMNIST dataset, with regards to validation accuracy and training time. Different hyperparameters for each method are also explored. Second, the theory of batch normalisation is presented, but its effects could not be investigated due to an error in implementation in the underlying library. Finally, two convolutional networks using one and two convolutional layers are compared against one another. We find that the choice of learning rule has a minor effect on model performance but a significant effect on training time, and that convolutional nets can perform better than shallower fully connected networks.

1. Introduction

This document investigates three questions: how do different learning rules affect training time and model performance, and which hyperparameters perform best on this task? What are the effects of batch normalisation? Do convolutional networks result in better performance than feedforward neural nets with only fully connected networks, without fine-tuning the network architecture and weight initialisation?

The networks were evaluated on the Extended MNIST (EMNIST) Balanced dataset (Cohen et al., 2017), which contains images of handwritten digits and letters. The dataset is dividied into training, validation and test sets, which contain 100,000, 15,800 and 15,800 samples, respectively. Every input is a 28 by 28 pixel monochrome input image, which is represented as a 784-dimensional vector of floating point coordinates ranging between 0.0 and 1.0, representing white and black. Although the 10 digits, 26 lowercase and 26 uppercase characters in total would give 62 classes, the lowercase and uppercase classes of 15 characters were merged to avoid easy misclassification, resulting in 47 target classes. (The characters with merged classes are C, I, J, K, L, M, O, P, S, U, V, W, X, Y, Z.) The target vector uses one-hot coding, i.e. it is 1 on the coordinate of the true class, 0 on other coordinates. The data sets are shuffled randomly after every epoch. Figure 1 represents 25 samples from the dataset, and it also shows that in some cases it is not obvious even for humans to classify the inputs correctly.

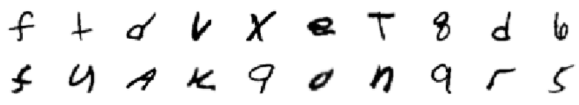

Figure 1. Example data points from the EMNIST dataset

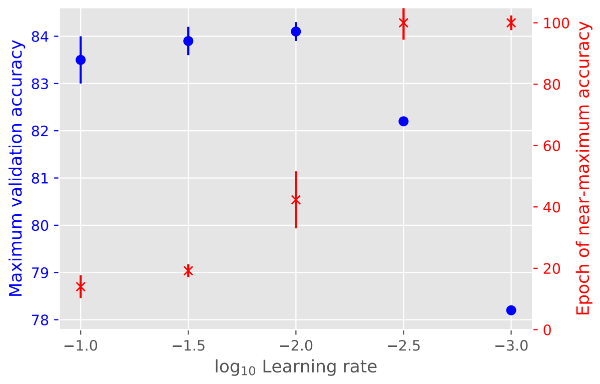

Figure 2. Maximum validation accuracy with SGD at different learning rates, and the epoch when the maximum accuracy was reached. Averaged over 5 experiments.

2. Baseline systems

The baseline system was a neural network with 2 fully connected hidden layers (with 100 hidden units each), ReLU activation functions, trained with stochastic gradient descent (SGD) (Robbins & Monro, 1951) to minimize cross-entropy softmax error. The weights were initialised with Glorot uniform initialisation (Glorot & Bengio, 2010). In Coursework 1 we saw that on the MNIST task such a network architecture leads to as good performance as deeper networks or different activation functions, but takes less time train and its training results are less sensitive to the choice of hyperparameters, which motivated our decision to use it as a baseline for the EMNIST task.

In these baseline experiments the minibatch size was 50, and the learning rate η was chosen from $\{0.1, 0.03, 0.01, 0.003, 0.001\}$; every experiment was run for 100 epochs with 5 different random weight initialisations. Higher minibatch sizes would lead to longer training times but a better estimation of the error function, and

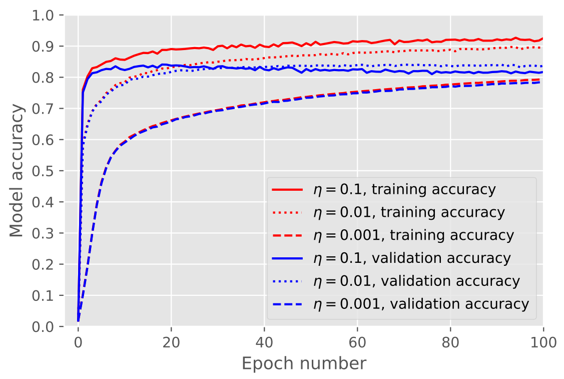

Figure 3. Training and validation accuracy of baseline models trained with stochastic gradient descent, with different learning rates

lower minibatch sizes would update the parameters more frequently but with a worse estimation of the error function (and also worse CPU utilisation due to the smaller batches).

Three example learning curves can be observed in Figure 3, with statistics in Figure 2. We can observe that a learning rate of 0.1 learned quickly but then started overfitting around epoch 10: the training accuracy kept increasing but the validation accuracy started declining. With early stopping, this learning rate leads to good performance quickly. (For example if every trained model is saved and the one with lowest validation accuracy is used for subsequent tests.) The reason is that the "valleys" of the error function are small in this high-dimensional parameter space in comparison with the gradients, therefore even small updates of the parameters can move us uphill on the error function.

On the other hand, a learning rate of 0.003 or lower led to steadily increasing validation accuracy but very slow learning: the system did not reach a local optimum even in 100 epochs, due to the very small updates of the parameters. (Hence the apparently lower performance in the table.) Therefore with SGD, if training time is a great concern, $\eta = 0.1$ is probably the best choice in this setting; if we want to ensure that we reach a good local minimum, $\eta = 0.01$ is preferred.

3. Learning rules

Stochastic gradient descent is a gradient-based iterative method that updates the parameters θ in every iteration by the negative gradient of the error function E, multiplied by the learning rate

$$\theta \leftarrow \theta - \eta \nabla_\theta E.$$

This algorithm has a couple of issues. First, the learning rate is fixed during the whole optimisation procedure, so it doesn't deal well with problems where the gradients of the error function vary on a wide range. (Such as when trying to find the optimum of $\exp(-x^2/0.01)$, starting from $x = 1$.) Second, the learning rate is the same for all parameters, so the algorithm learns inefficiently when the gradient of the error function at the current parameter θ has a high angle to the vector that points from θ to the optimal parameters $\hat{theta}$.

RMSprop is a per-parameter adaptive learning rate method, which fixes both of the above stated problems of SGD (Tieleman & Hinton, 2012). It keeps a running average of the squared gradients over iterations, and divides the update term by the squared root of this average – in effect dividing the updates in each iteration by a running mean of the root mean of the squared gradients; hence the name, RMSprop. This makes parameters with a small gradient use a higher effective learning rate, making them update in higher increments than the parameters that encountered a high gradient in the past few iterations. The decay rate of the moving average is a hyperparameter of this method, with typical values from $\{0.9, 0.99, 0.999\}$. The two equations that describe this rule therefore are the following:

$$r \leftarrow \beta r + (1-\beta)\nabla_\theta E$$
$$\theta \leftarrow \theta - \eta \nabla_\theta E / \sqrt{r + \epsilon},$$

where ϵ is a small number to avoid division by zero when $r = 0$.

The Adam learning rule can be thought of as RMSprop with momentum (Kingma & Ba, 2014). Momentum learning can be thought of like a ball rolling down a hill, where the gradient doesn't affect the position of the ball directly, rather the velocity of the ball is changed by the force acting on it, and this force is proportional to the gradient. Then the position of the ball in the next timestep can be calculated from its velocity.

The simplified parameter-updating procedure is as follows. With θ being the vector of parameters to be optimised, $E(\theta)$ the cost function, and the initial moment vectors $m = 0$ and $v = 0$, we iteratively update the moment estimates

$$m \leftarrow \beta_1 \cdot m + (1-\beta_1)\nabla_\theta E,$$
$$v \leftarrow \beta_2 \cdot v + (1-\beta_2)(\nabla_\theta E)^2,$$

(where $(\nabla_\theta E)^2$ is the element-wise squared gradient), and then using these updated moments, we update the parameters:

$$\theta \leftarrow \theta - \eta \cdot m/(\sqrt{v} + \epsilon).$$

The first and second moment coefficients (β_1 and β_2) and η are hyperparameters of the model.

These moment vectors are biased towards zero, so the actual algorithm corrects this bias by using bias-corrected moments $\hat{m}$ and $\hat{v}$ instead of m and v, as described in Algorithm 1. Also note that in most cases we do not have access to the true error function or its gradient, only to its estimate, namely the error of the current minibatch.

Whereas gradient descent only takes into account the gradient of the error function $(\nabla_\theta E^t(\theta_t))$ at the current timestep and its weight updates are the same irrespective of previous updates, Adam takes into account the first and second moments of the gradient.

Algorithm 1 Adam learning rule

```
Require: $\theta_0$: Initial parameters
  $m, v \leftarrow 0, 0$
  $t \leftarrow 0$ (Iteration counter)
  $\theta \leftarrow \theta_0$
  repeat
    $t \leftarrow t + 1$
    $m \leftarrow \beta_1 m + (1 - \beta_1)\nabla_\theta E$
    $v \leftarrow \beta_2 v + (1 - \beta_2)(\nabla_\theta E)^2$
    $\hat{m} \leftarrow m/(1 - \beta_1^t)$
    $\hat{v} \leftarrow v/(1 - \beta_2^t)$
    $\theta \leftarrow \theta - \eta\hat{m}/(\sqrt{\hat{v}} + \epsilon)$
  until stopping criteria not fulfilled
```

Experimental results with RMSprop. We tested the RMSprop learning rule on the same model as the baseline system in Section 2. The decay term of the moving average was set to $\beta = 0.9$, as suggested in (Tieleman & Hinton, 2012). That report promised that it "speeds up training with mini-batches" – we were interested to see if that allows higher learning rates, or if the same learning rates would allow the system to train quicker. Hence the learning rate η was chosen from $\{0.3, 0.1, 0.03, 0.01, 0.003, 0.001\}$, with each experiment rerun 3 times. Figure 4 shows that with $\eta = 0.003$ or lower, the system learned considerably quicker than SGD with a same learning rate, but RMSprop did not allow as high choices of η. Analysis of the training error curves suggests that after the first epoch the system never reached an optimum with $\eta = 0.01$ or higher (5), and it did not learn anything at all with $\eta = 0.1$ or higher. (The error of the system stayed at the error of random choice, $ln(47) = 3.85$.)

Overall, the learning rule delivered its promise on this simple network architecture: with $\eta = 0.001$, a maximum validation accuracy of 82.7 ± .4% was reached between epochs 6 and 9. (Which is considerably faster than SGD, but slightly worse in performance.)

Experimental results with Adam. We tested the Adam learning rule on the same model as the baseline system in Section 2. The hyperparameters were chosen from all possible combinations of $\beta_1 \in \{0, 0.9\}$, $\beta_2 \in \{0.9, 0.99, 0.999, 0.9999\}$, $\eta \in \{0.1, 0.01, 0.001, 0.0001\}$, and $\epsilon = 10^{-8}$, a minibatch size of 50, and every experiment was run for 100 epochs with two different seeds. The choice of these values was motivated by the original paper, which stated that "good default settings for the tested machine learning problems are $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$".

Adam did not converge at all with $\eta = 0.01$ or higher, and it was common for its accuracy to change from one epoch to the next from 55% to 30% and back.

For $\eta = 0.001$ and $\beta_1 = 0.9$, the learning curves showed an

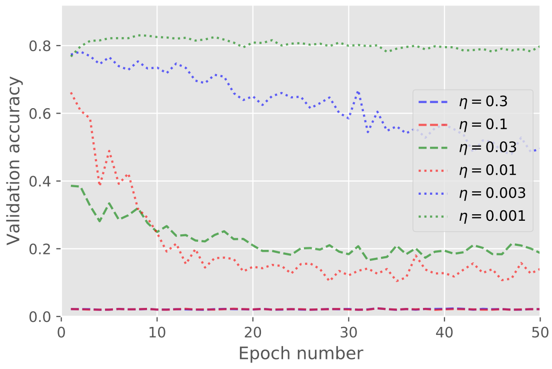

Figure 4. Validation accuracy of RMSprop during training with different learning rates

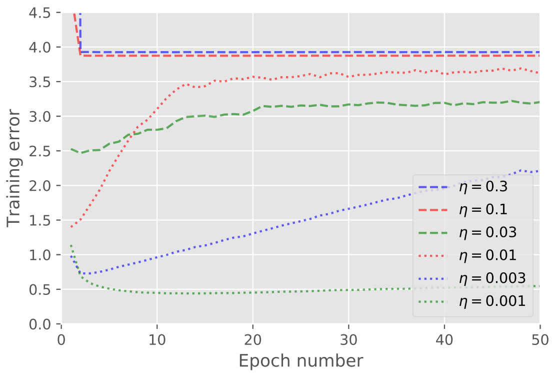

Figure 5. Training error of RMSprop with different learning rates

improvement as β_2 increased, but even with $\beta_2 = 0.9999$ the system reached only 80% accuracy (6. With $\beta_1 = 0.0$, the system also had its best performance with $\beta_2 = 0.9999$, when it reached 83.5 ± 0.3% at epoch 10 to 12, but from there on it started overfitting.

With $\eta = 10^{-4}$ and $\beta_1 = 0.0$, every setting of β_2 led to nearly monotonically increasing validation accuracy, with the maximum accuracy of 83.5% reached at epoch 44 to 52. With $\beta_1 = 0.9$, the system learned very quickly: it reached 83% epoch 6, and the fluctuations in the training error decreased as β_2 increased. The system with $\eta = 0.9999$ was the most successful in overfitting the data; its error curve is shown in Figure 7.

By setting β_1 to zero, the exponential moving average of the first moment turns into an estimate of the gradient based on the error calculated from the current minibatch. With $\beta_1 = 0$, the system learned much slower, suggesting that keeping a longer window of the moving average of first momentums are crucial in this task for speedy updates.

Higher choice of β_2 always led to better performance, also suggesting the necessity of a long window for second mo-

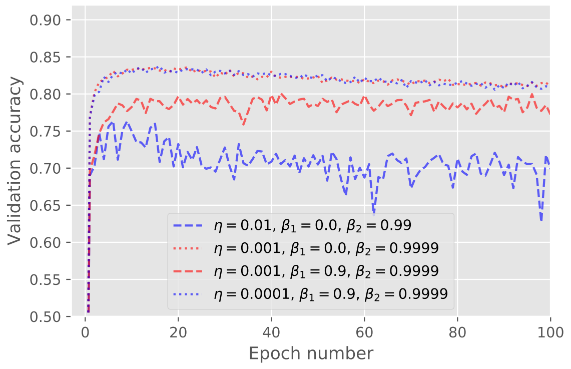

Figure 6. Validation accuracy reached by Adam for different hyperparameters

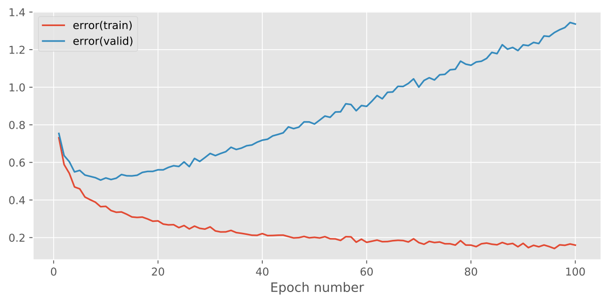

Figure 7. Learning curves of Adam with $\eta = 0.0001$, $\beta_1 = 0.9$, $\beta_2 = 0.9999$

ments.

4. Batch normalisation

Batch normalisation is carried out in a layer which learns the mean and variance of its input distribution and then uses these learned statistics to normalise the input (Ioffe & Szegedy, 2015). It is a stochastic layer, meaning its state changes even during forward propagation during training.

The layer has two parameters β and γ, which are each vectors of the same dimension as the input data, and whose elements are initialised by sampling from a standard normal distribution.

During training, a batch normalisation layer first computes the mean and variance of its input in every minibatch. This is *ideally* done by doing the forward propagation of the whole training data for every previous layer, but in practice these statistics are either calculated as a running mean and variance (Abadi et al., 2016), or just over the current minibatch, to reduce the costs of either memory or computing.

Once the above statistics are obtained, its output $\mathbf{z}$ is calculated as follows. First normalise the input $\mathbf{u}$:

$$\hat{\mathbf{u}} = \frac{\mathbf{u} - \text{mean}(\mathbf{u})}{\sqrt{\text{var}(\mathbf{u}) + \epsilon}},$$

where ϵ is a small number to avoid division by zero when the input takes on exactly one value in a certain dimension at every data point. Then this normalised input $\hat{\mathbf{u}}$ is scaled and shifted by γ and β, respectively, to get the output $\mathbf{z}$:

$$\mathbf{z} = \gamma \odot \hat{\mathbf{u}} + \beta,$$

where the $\mathbf{a} \odot \mathbf{b}$ denotes the Hadamard product of vectors $\mathbf{a}$ and $\mathbf{b}$.

During backpropagation, we update the parameters γ and β, and backpropagate the gradients of the cost function to the original input $\mathbf{u}$.

To batch-normalise a network, a batch normalisation layer is put before the a chosen subset of layers, for example before each nonlinear layer. The original paper (Ioffe & Szegedy, 2015) claims that this technique acts like a regulariser, reducing the need for explicitly setting penalties for the weights, and that it allows the network to train faster and reach higher accuracy.

To test the effect of batch normalisation, we put a batch normalisation layer before every nonlinear layer of the baseline model, resulting in the following architecture, from inputs to outputs:

1. Affine layer with 100 units
2. Batch normalisation layer
3. ReLU
4. Affine layer with 100 units
5. Batch normalisation layer
6. ReLU
7. Affine layer with 10 units.

The affine layers were initialised as before, and the experiments were run with 5 different seeds, with Adam learning with $\eta = 0.0001, \beta_1 = 0.9$ and $\beta_2 = 0.9999$, and a minibatch size of 50.

These models had poorer performance than the baseline systems: the average accuracy reached by them in 100 epochs was 72.3 ± 5.6. As it was later found out, an implementation error in the underlying `mlp` library prevented a learning rule from updating the parameters of a `StochasticLayerWithParameters` object. The error has been fixed, but the experiments were not rerun due to lack of computing capacity.

5. Convolutional networks

Convolutional neural networks were first used successfully in machine learning for image classification in the LeNet1 architecture (LeCun et al., 1990). Convolutional networks build upon three ideas: *local receptive fields*, *weight sharing*, and *pooling* (also called *subsampling*).

In a convolutional layer, every unit is a linear combination of inputs that are close to each other in the 2-dimensional image: this is the local receptive field of the unit, which is usually square-shaped. This enables a convolutional layer to take into account the 2-dimensional quality of images or board games like Go (Clark & Storkey, 2015), instead of operating on the flattened input vectors. The units are grouped into planes, and the weights of each unit in a plane are bound together to always have the same value (this is called weight sharing), so they together form a so-called *feature map* of the input, and the weights of the layer are called its *kernels*. This ensures that units in a layer will be activated when the same feature is present at different positions in the input image, resulting in *translation invariance*.

These make the output **h** of a convolutional layer the convolution of the input **x** and the feature map **w** of the layer (with a bias term b added), followed by a non-linear function f:

$$h_{i,j} = f(\sum_{k=1}^{m}\sum_{l=1}^{m} w_{k,l}x_{i-k+m,j-l+m} + b),$$

or in vector notation:

$$\mathbf{h} = f(\mathbf{w} * \mathbf{x} + b).$$

For example, the convolution of a 28x28 input and a 5x5 kernel would result in 24x24 outputs ($24 = 28 - 5 + 1$).

In practice a convolutional layer has multiple feature maps, and they take multi-channel inputs and produce multi-channel outputs. Thus for a layer with C_{in} input channels and C_{out} output channels:

$$\mathbf{h}^{(out)} = \sum_{in=1}^{C_{in}} f(\mathbf{w}^{(in,out)} * \mathbf{x}^{(in)} + b^{(out)}),$$

with (out) ranging from 1 to C_{out}. Different channels could be for example the red, green, blue channels of an RGB input image, or the result of convolutions of the input with different feature maps in a previous convolutional layer.

The activation function of a convolutional layer could be for example a sigmoid function like in LeNet1 (LeCun et al., 1990), or recently variants of the rectified linear unit (ReLU) have been popular for its faster training times (Nair & Hinton, 2010).

The output of a convolutional layer is usually downsampled with non-overlapping regions, with a typical region size of 2x2. This reduces the size of each layer by removing the precise location of the activations of a feature maps, enabling faster training times. A typical pooling function is max-pooling:

$$h'_{i,j} = max(\{h_{mi+k,mj+l}\}_{k=0,\ l=0}^{m-1,\ m-1}),$$

where m is the size of the pooling region, but sum-pooling and L_p-pooling are also used in some applications (Scherer et al., 2010).

We evaluated the performance of 2 different convolutional neural networks (CNNs). We were interested in whether a network with two convolutional layers has better performance than a network with only one convolutional layer.

The architecture of the network with one convolutional layer looked as follows, from top (input) to bottom (output):

1. Reshape to (1,28,28)
2. Padding (padding: 2)
3. Convolutional layer (output channels: 10, kernel: 5x5, kernels initialised from $U(-0.01, 0.01)$)
4. ReLU
5. Max pooling layer (pooling size: 2)
6. Reshape to $10 \cdot 14 \cdot 14 = 1960$ (flattening)
7. Affine layer (with 576 units)
8. ReLU
9. Affine layer (with 160 units)
10. ReLU
11. Affine layer (with 47 units)

The affine layers all had a Glorot weight initialisation and the bias initialised to 0.

This architecture choice was motivated by the literature (Karpathy, 2017). Two fully connected hidden layers after the convolutional layer seemed a good compromise betwen computing time and memory requirements (more layers) and a sharp decrease in the number of units between neighboring affine layers (less layers). The number of units in the affine layers was chosen so that the ratio of units of neighboring units is nearly the same: $160/47 \approx 576/176 \approx 1960/576$. The number of output channels and kernel size was set in accordance with the suggestion of the coursework (Renals, 2017).

In order to avoid information at the edges of the input being washed away by the convolutions, a padding layer was added before the convolutional layer, which added zero padding aroun each channel.

As in Section 3, the Adam learning rule performed best with fully connected networks with $\eta = 10^{-4}$, $\beta_1 = 0.9$, $\beta_2 = 0.9999$ and minibatch size of 50, so that rule was used for the training of this model.

During forward propagation, for every minibatch input tensor an output tensor of all zeros with the correct size is initialised. Then the procedure uses three `for` loops:

1. over the data points in the minibatch,
2. over the output channels,
3. over the input channels,

and in the innermost loop, the convolution of the corresponding input and kernel is added to the corresponding output channel. (The convolution is calculated by `scipy.signal.convolve2d()`, with the `mode='valid'` setting.) After all iterations are done for every kernel and input, the biases are added and the output tensor returned. Backpropagation of gradients and the calculation of gradients with regards to the kernels are done similarly, but using `scipy.signal.correlated2d()`, to calculate the necessary cross-correlations. The gradients with regards to a bias is the sum of the gradients with regards to the outputs, summed over axes 0, 2 and 3.

The network with two convolutional layers had the following architecture:

1. Reshape to (1,28,28)
2. Padding (padding: 2)
3. Convolutional layer (output channels: 5, kernel: 5x5, kernels initialised from $U(-0.01, 0.01)$)
4. ReLU
5. Max pooling layer (pooling size: 2)
6. Convolutional layer (output channels: 10, kernel: 5x5, kernels initialised from $U(-0.01, 0.01)$)
7. ReLU
8. Max pooling layer (pooling size: 2)
9. Reshape to $10 \cdot 7 \cdot 7 = 490$ (flattening)
10. Affine layer (with 150 units)
11. ReLU
12. Affine layer (with 47 units)

Differences to the previous network are highlighted in red. The reasons for choosing this architecture are the same as they were. The learning parameters were also the same.

One epoch of training on this network took 45% longer on average than with the previous network, and more than 100 times longer than the training of the fully connected networks explored in earlier sections. These models were also relatively quick to train, in that they both reached peak performance under 10 epochs, and 90% of the top performance in a single epoch.

The network with one convolutional layer reached a peak validation accuracy of 87.44%, and already had 83.7% error after the first training epoch. The network with two convolutional layers reached a peak validation accuracy of 87.1%, with 83.2% error after the first epoch. These findings would suggest that the extra convolutional operation does not bring extra performance, but the literature has different findings (LeCun et al., 1990). It would be interesting to see the cause for these differences.

6. Test results

The model without convolutions had the architecture described in Section 2, did have batch normalisation, and was trained with Adam learning rule for 7 epochs with a minibatch size of 50, $\eta = 0.0001$, $\beta_1 = 0.9$, $\beta_2 = 0.9999$, initialised with seed 3. That model had a validation accuracy of 84.2%, and a validation error of 0.493. The accuracy of this model on the test set was 82.22%, with an error of 0.547.

The convolutional model that performed best on the validation set was the one that had one convolution in it, without batch normalisation (the details can be found in Section 5), after 7 epochs of training (the number 7 matching with the one above is just coincidence) – this model had a validation accuracy of 87.44% (and a validation error of 0.363). The accuracy of this model on the test set was 86.76% (with error 0.397).

These results suggest that a convolutional network can achieve better performance on this task than a fully connected network with two hidden layers. This is in line with the original motivation for the use of convolutional nets in image recognition (LeCun et al., 1990).

7. Conclusions

Our experiments show that using RMSprop or Adam can lead to completely trained networks in less than ten epochs on the training set, and to over 90% of the final performance in a single epoch, given the right choice of hyperparameters. However, finding these hyperparameters is not necessarily trivial: apart from the minibatch size, Adam has 3 hyperparameters to be tuned, and the paramaters suggested in (Kingma & Ba, 2014) did not prove to be the best in our comparisons.

As mentioned in Section 4, the effect of batch normalisation could not be tested due to a software error in the underlying library, which prevented the parameters of a batch normalisation layer from updating.

On the other hand, using convolutions on 2D images can lead to significantly better performance, at the cost of greater training time.

Further work could explore the following questions:

- whether the same hyperparameters obtained in Section 3 could be used when training models for different tasks, such as image classification on the CIFAR-10 dataset (Krizhevsky et al.);
- whether batch normalisation or dropout could really result in less overfitting and better performance with fully connected or convolutional neural networks, as suggested in (Ioffe & Szegedy, 2015);
- whether the reason for the better performance of convolutional networks (over fully connected ones) lied not only in the more number of weights: the three

affine layers of the one-convolution network in Section 5 had more than ten times as many weights as the three affine layers of the model described in Section 2.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Clark, Christopher and Storkey, Amos. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*, pp. 1766–1774, 2015.

Cohen, Gregory, Afshar, Saeed, Tapson, Jonathan, and van Schaik, André. EMNIST: an extension of MNIST to handwritten letters. *CoRR*, abs/1702.05373, 2017. URL http://arxiv.org/abs/1702.05373.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, Francis and Blei, David (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/ioffe15.html.

Karpathy, A. Cs231n convolutional neural networks for visual recognition. 2017. URL http://cs231n.github.io/neural-networks-3/.

Kingma, Diederik P. and Ba, Jimmy. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Krizhevsky, Alex, Nair, Vinod, and Hinton, Geoffrey. Cifar-10 (canadian institute for advanced research). URL http://www.cs.toronto.edu/~kriz/cifar.html.

LeCun, Yann, Boser, Bernhard E, Denker, John S, Henderson, Donnie, Howard, Richard E, Hubbard, Wayne E, and Jackel, Lawrence D. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pp. 396–404, 1990.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Renals, S. Machine learning practical: Coursework 2. 2017. URL http://www.inf.ed.ac.uk/teaching/courses/mlp/2017-18/coursework2.pdf.

Robbins, Herbert and Monro, Sutton. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951. doi: 10.1214/aoms/1177729586. URL https://doi.org/10.1214/aoms/1177729586.

Scherer, Dominik, Müller, Andreas, and Behnke, Sven. Evaluation of pooling operations in convolutional architectures for object recognition. *Artificial Neural Networks–ICANN 2010*, pp. 92–101, 2010.

Tieleman, T. and Hinton, G. E. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012. URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.