AIMS Course 4: Machine Learning
Assignment On Automatic Differentiation

Jake Levi

November 2022

1 Introduction

Automatic differentiation refers to an approach to numerical programming in which operations on numbers (or arrays of numbers) are recorded in a computation graph, with the numbers being represented by nodes, and the operations on those numbers being represented by edges. This allows gradients of the results of those computations to be calculated automatically by applying the chain rule along the edges of the computation graph, assuming that the local derivative of each operation in the computation graph is known.

This approach is very powerful, and of particular relevance to the field of machine learning, because once a model and loss function have been expressed in terms of a computation graph, assuming the model and loss function are differentiable, parameters of the model can be optimised with respect to the loss function using gradient-based methods, in which the gradients are calculated using automatic differentiation, without having to derive the gradients explicitly. One popular framework for automatic differentiation is PyTorch [10], which was used during all the experiments described in this report. Further explanation of how automatic differentiation works is provided in appendix B.

2 Multi Layer Perceptron

A multi layer perceptron (MLP) is a simple but powerful machine learning model, which maps an input vector to an output vector by applying successive "layers" of parameterised linear transformations followed by element-wise nonlinear functions to the input vector. The parameters of the linear transformations can be learnt in such a way as to minimise a loss function appropriate to a specific dataset using gradient-based methods, with gradients being calculated using automatic differentiation. For the experiments in this section, various different types of MLPs were trained on the MNIST dataset of handwritten digits [9], using a softmax cross-entropy loss function which is appropriate for multi-class classification tasks.

There are various design decisions to consider when implementing a MLP, including the choice of method used to initialise parameters in the model, and the choice of optimisation method. The parameters of the models used in this section were initialised following [5], by sampling initial weight parameters in each layer from a zero-mean Gaussian distribution whose standard deviation is equal to $\sqrt{2/n_l}$, where n_l is the dimension of the input to the layer, and initialising all bias parameters to zero. This choice of standard deviation ensures that when using rectified linear unit (relu) activation functions, the outputs from each layer do not grow or shrink exponentially. For optimising MLPs, stochastic gradient descent with momentum was used following [13] and [11], in which the parameters w_t at time t are chosen to minimise a loss function f by maintaining an estimate for the momentum b_t of the parameter gradients at each time step, and updating the parameters and momentum as follows (where μ and α are hyperparameters to be chosen):

$$b_t \leftarrow \mu b_{t-1} + \frac{\partial f}{\partial w} \tag{1}$$

$$w_t \leftarrow w_{t-1} - \alpha b_t \tag{2}$$

Figure 1 shows the loss function against time for two different MLPs trained on MNIST for 5 epochs, with both MLPs being trained on both an Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz and an NVIDIA GeForce MX250 GPU for comparison. The figures show the loss function decreasing over time in all four cases, with the GPU performing faster than the CPU for both models, however the speed up offered by the GPU is more significant for the larger model (figure 1b). In fact, the GPU is only 6.8% slower for the larger model, despite having to perform significantly more computation, whereas the CPU is 82.2% slower. Figure 2 shows analogous results for the same two models being trained on a server with Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz and NVIDIA TITAN V GPU, showing

that the server is significantly faster in all cases (both for larger and smaller models being trained on the CPU and the GPU), with the GPU actually being marginally slower for the small model, but marginally faster for the larger model.

Figure 3 shows how the learning curves and final generalisation performance of a MLP trained using gradient descent vary with respect to the momentum hyperparameter (3a), batch size (3b), number of hidden layers (3c), dimension of the hidden layers (3d), and hidden layer activation function (3e). Figure 4 shows the predictions of a trained MLP on different unseen examples of each digit from the test set.

3 Adversarial Examples

Adversarial examples are "inputs formed by applying small but intentionally worst-case perturbations to examples from the dataset, such that the perturbed input results in the model outputting an incorrect answer with high confidence" [4]. Adversarial examples provide a measure of robustness for a given model, because a robust model should be insensitive to adversarial examples. Given a fixed model f, loss function $\mathcal{L}$, input x, label t, adversarial target label $t_{adv} \neq t$, and maximum perturbation α, a simple approach for generating an adversarial example $x_{adv} = x + z$ is to solve the following optimisation problem:

$$\begin{aligned} \underset{z}{\text{Minimise}} \quad & \mathcal{L}\Big(f(x+z), t_{adv}\Big) \\ \text{Subject to} \quad & \|z\|_\infty \leq \alpha \end{aligned} \tag{3}$$

A simple approach for solving this optimisation problem is to iteratively update z_t on time step t as follows:

$$z'_t \leftarrow z_{t-1} - \frac{\partial}{\partial z_{t-1}}\Big[\mathcal{L}\Big(f(x+z_{t-1}), t_{adv}\Big)\Big] \tag{4}$$

$$z_t \leftarrow \alpha \frac{z'_t}{\|z'_t\|_\infty} \tag{5}$$

Using automatic diffentiation makes it trivially straightforward to calculate the gradient in equation 4 and therefore to generate an adversarial example according to equation 3. A plot showing the loss function against time when performing this optimisation procedure using automatic differentiation is shown in figure 5a, starting with the same example of the digit 0 from the MNIST test set and the same trained MLP whose predictions are shown in figure 4. The resulting adversarial example, and the model's predictions for the adversarial example, compared with the original test set example from which the adversarial example was generated, are shown in figure 5b.

4 Sequence Models

Neural networks such as the MLP can be adapted for sequence prediction simply by concatenating the input x_t to the model at time t with a hidden state h_t, using a neural network to calculate both an output from the model y_t and a new hidden state h_{t+1}, and then using the new hidden state as the input hidden state for the next time step, when a new input/output pair is processed by the model. This type of model is known as a recurrent neural network (RNN). A particular type of recurrent neural network is the long short-term memory (LSTM) model [6], which uses gates to learn to control the flow of information from one time step to the next. Using automatic differentiation makes it very straightforward (in principle) to optimise the parameters of a RNN, simply by defining a loss function for the network, averaging that loss function over time steps and over batches of training data within a time step, and then using the automatically computed gradients of the average loss function to optimise the parameters of the model.

Figure 6 shows the loss functions over time while training both an RNN and an LSTM for 8 hours on the complete works of Shakespeare[1], with the objective of minimising the cross-entropy between the models' predictions of the next character and the actual next character, given a substring of text from the training data. This approach also allows the model to generate new strings of text from its predicted distribution over future characters, and an example of these predictions are included in appendix A. Unfortunately, despite training for 8 hours, the models were not able to generate coherent text in the style of Shakespeare. Possible improvements to the models include looking at different parameter initialisation methods (because the parameters were initialised according to [5], which assumes relu activation functions, but the LSTM in particular uses sigmoid and tanh activation functions at the gate outputs), using dropout [12] [3], batch normalisation [7], layer normalisation [1], different optimisers (such as Adam [8]), different network architectures, and other types of regularisation (such as weight decay).

[1] Freely downloaded from https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Section 22.4: topological sort. *Introduction to Algorithms (2nd ed.), MIT Press and McGraw-Hill*, pages 549–552, 2001.

[3] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. *Advances in neural information processing systems*, 29, 2016.

[4] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[8] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[9] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[11] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

A Sample from a RNN trained on the complete works of Shakespeare

koly hi[be. lyyen
sharlsed.
nonk i wins? what soms,
burubtine, eud, whon yours at that pelm
boyd
bleethowerbarvass.
hath deet where wi
my deit witsoney and gedbyour abkres
to, not kerthurg, hitthe. youlls o urlsnous. anclovod abeeksss fours farters thae all leo tay, if thre in i hat in to lmith'ny;
come nof he dius.
traigtoud.
sthenr
biov, werrets, shame;

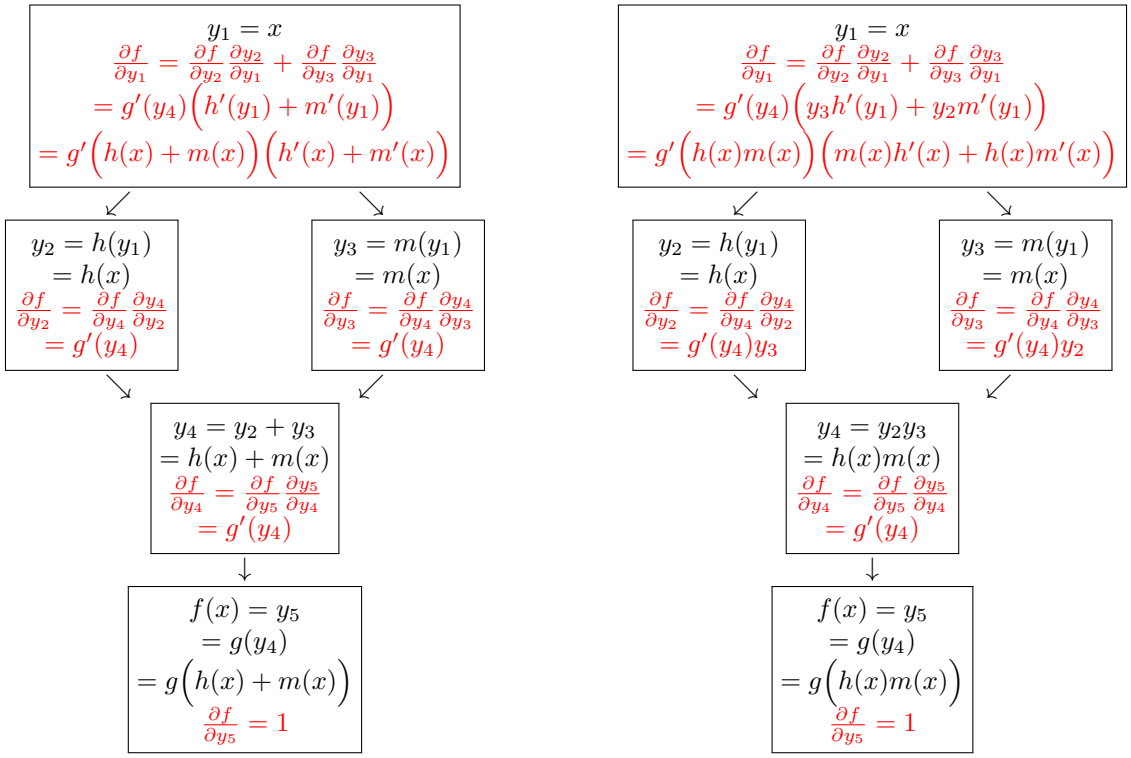
ment ne mond to to
pate hond be, or,
whoulgihe.
sonides of diltl

B Explanation Of Automatic Differentiation

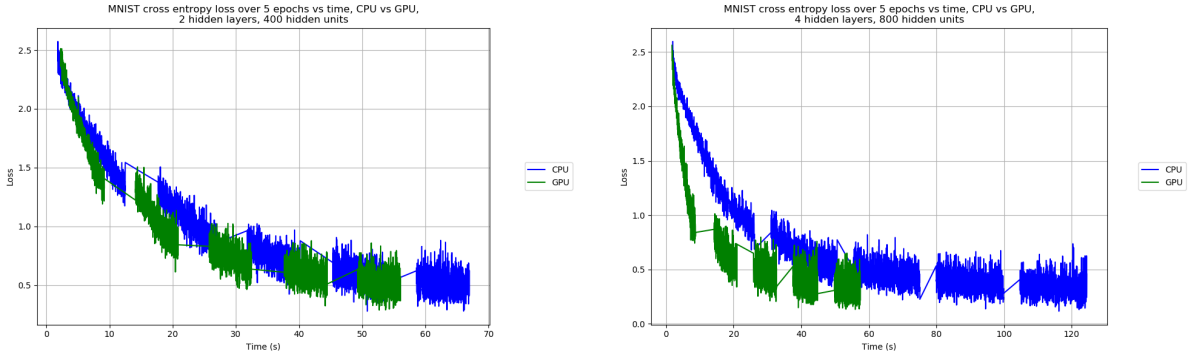
Automatic differentiation is essentially equivalent to repeated application of the (multi-dimensional) chain rule to all nodes in a computation graph with respect to their parents, which can be expressed as the following equation (where $P(x)$ is the set of parents of node x in the computation graph):

$$\frac{\partial f}{\partial x} = \sum_{y \in P(x)} \left[\frac{\partial f}{\partial y} \frac{\partial y}{\partial x} \right] \tag{6}$$

For example, the following computation graphs represent calculating gradients of a function of a sum of functions of a variable x (represented by $f(x) = g\Big(h(x) + m(x)\Big)$) and calculating gradients of a function of a product of functions of a variable x (represented by $f(x) = g\Big(h(x)m(x)\Big)$) respectively:



In practise, this can be applied in three steps. The first step is to set the gradient $\frac{\partial f}{\partial f}$ of the root node f to one and the gradients $\frac{\partial f}{\partial x_i}$ of all other nodes x_i to zero. The second step is to topoligically sort the nodes in the graph (for example using a depth-first search [2]) such that any node only appears after all of its parents. The third step is to iterate through every node y in the topologically sorted sequence of nodes, and for every node y iterate through every child node x of y and add $\frac{\partial f}{\partial y}\frac{\partial y}{\partial x}$ to the gradient $\frac{\partial f}{\partial x}$, until the whole graph has been iterated over, at which point the gradient of every node will be correct.

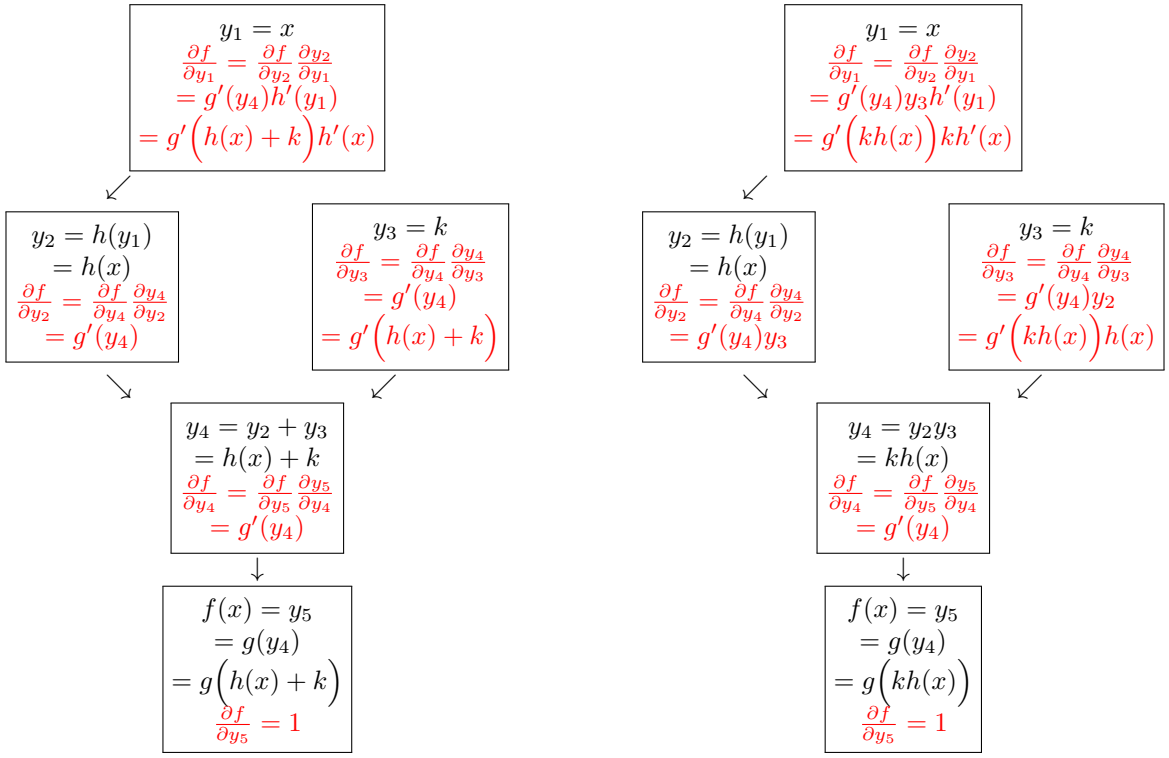


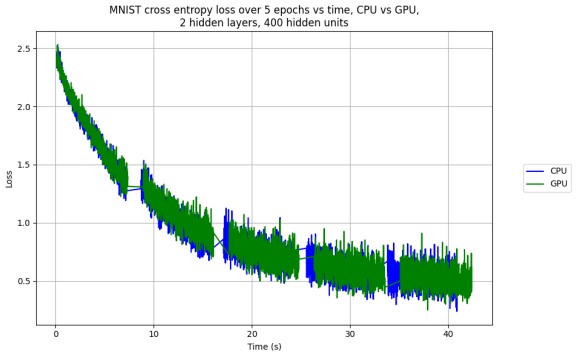
(a) Small model (2 hidden layers, 400 hidden units)

(b) Larger model (4 hidden layers, 800 hidden units)

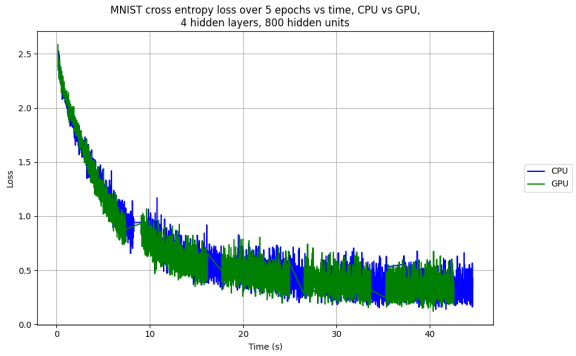
Figure 1: Learning curves for different sized models trained on MNIST over 5 epochs, comparing training times between an Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz and an NVIDIA GeForce MX250 GPU. Gaps in the training curves are due to evaluating test set accuracy once per epoch.

The following computation graphs demonstrate what happens when the function $m(x)$ is replaced by a constant k:



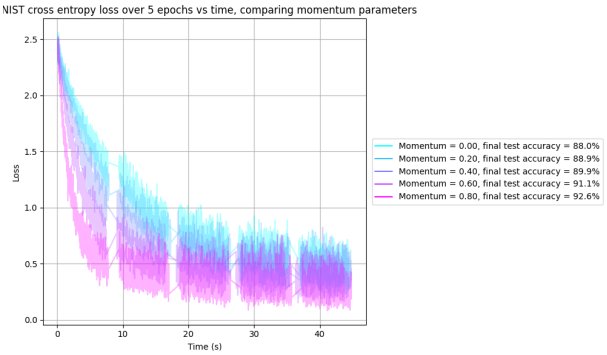


(a) Small model (2 hidden layers, 400 hidden units)

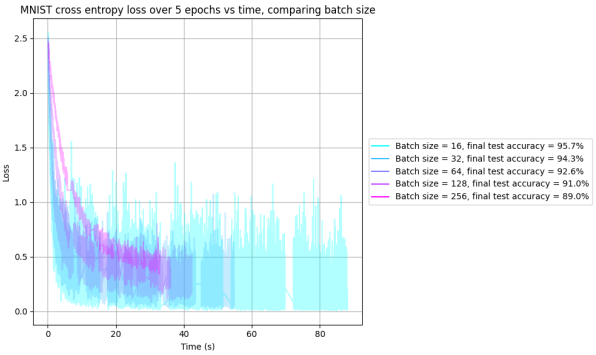


(b) Larger model (4 hidden layers, 800 hidden units)

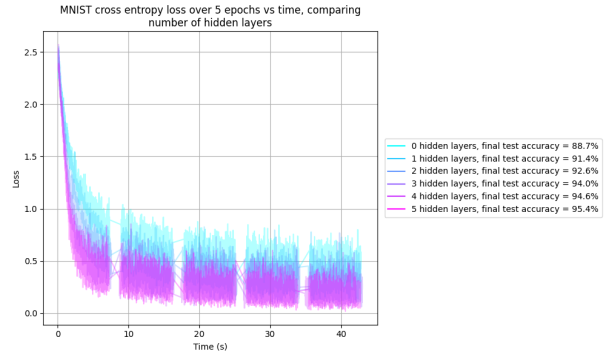
Figure 2: As for figure 1, performed on a server with Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz and NVIDIA TITAN V GPU



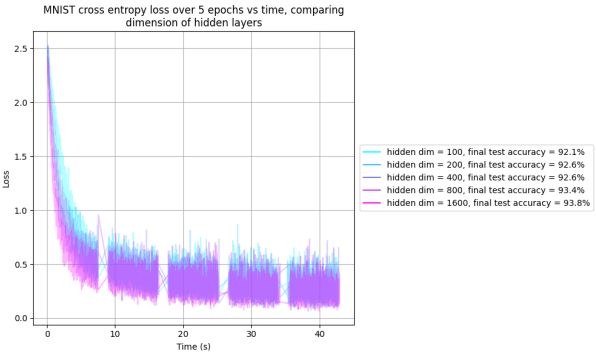
(a) Comparing momentum optimisation hyperparameter



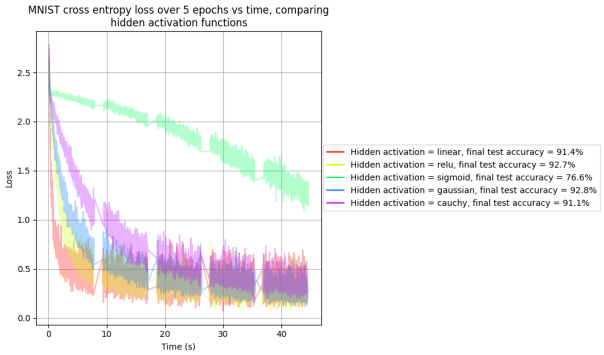
(b) Comparing batch size



(c) Comparing number of hidden layers



(d) Comparing dimension of hidden layers



(e) Comparing hidden layer activation functions

Figure 3: Comparing the effect of different hyperparameters on training a MLP on MNIST. Test set prediction accuracies are included in legends.

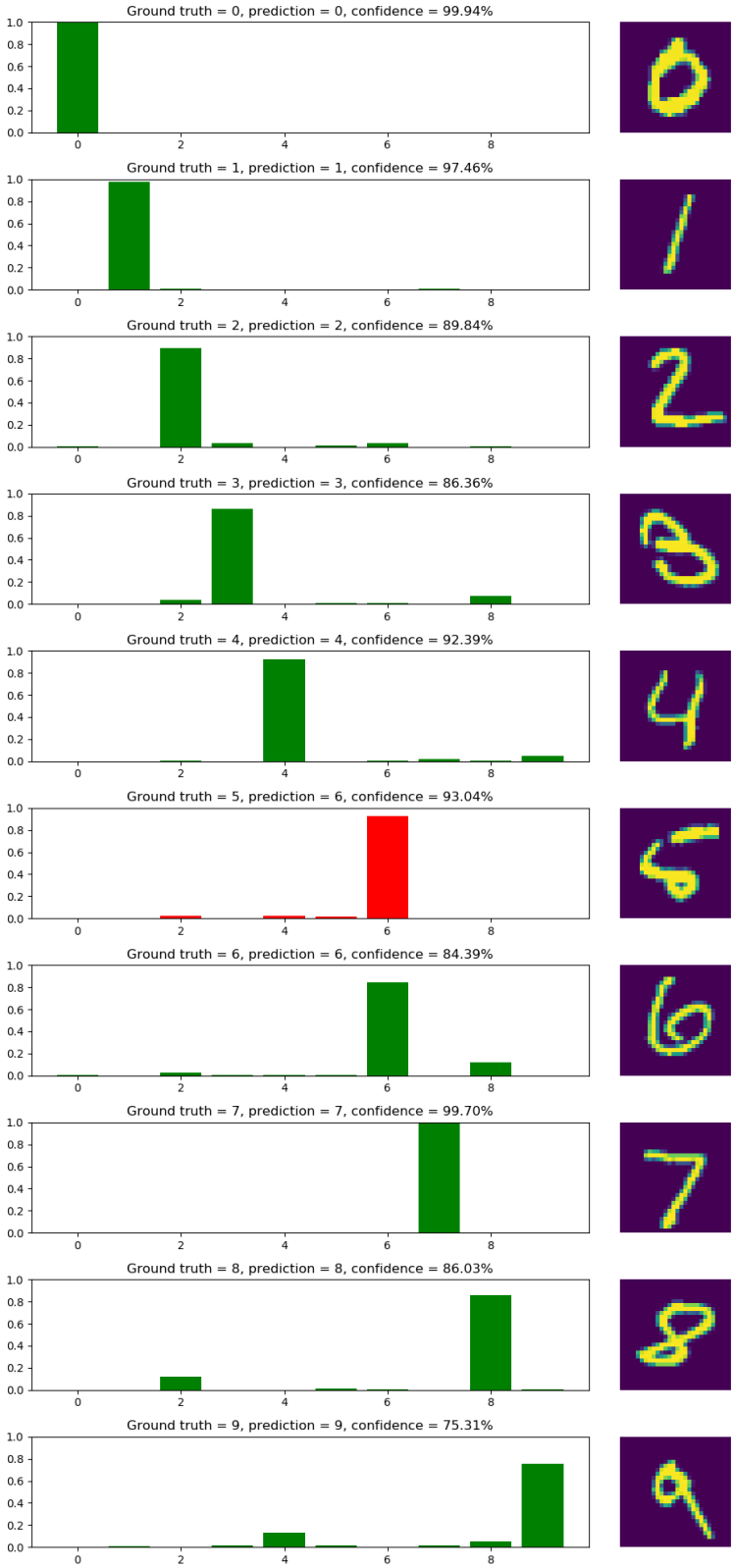
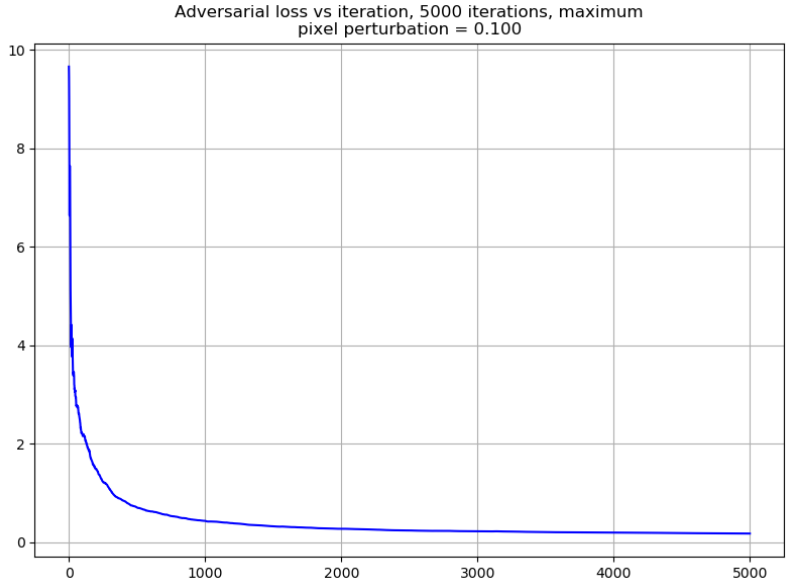
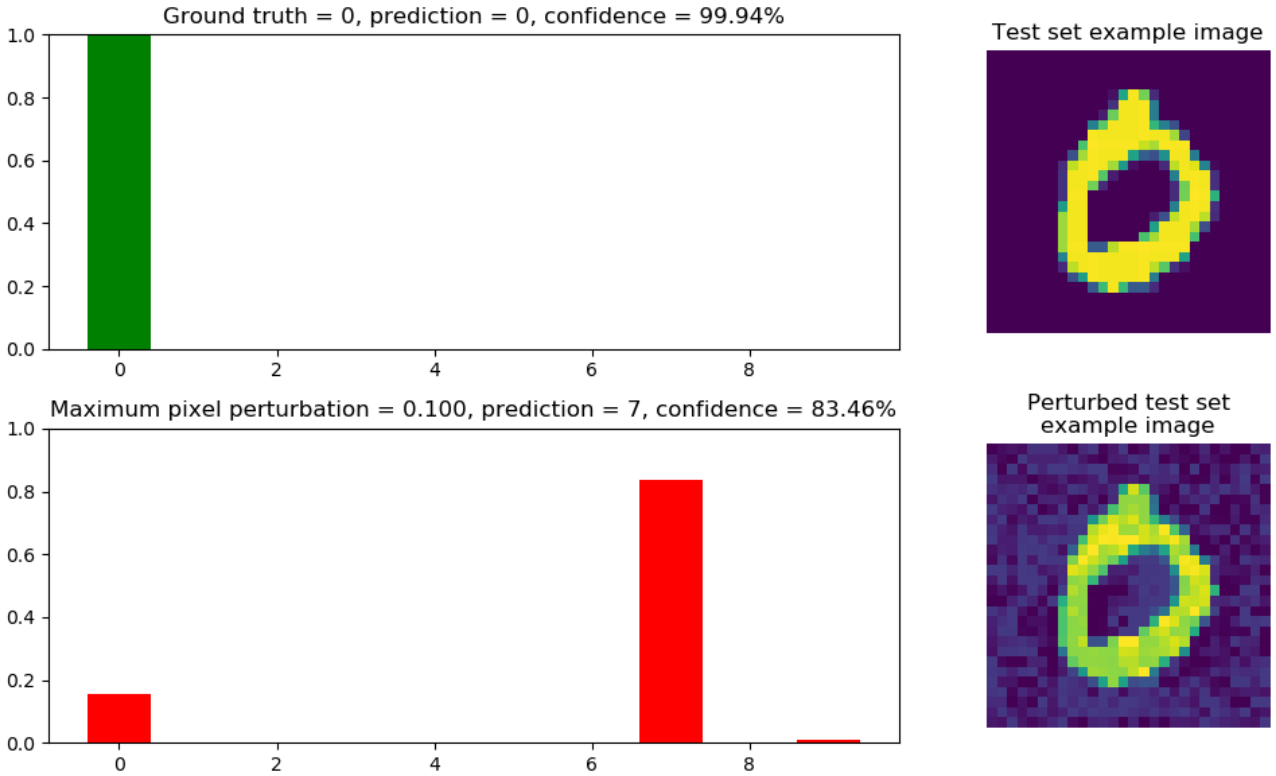


Figure 4: Predictions of a MLP trained on MNIST for 5 epochs on unseen examples of each digit from the test set

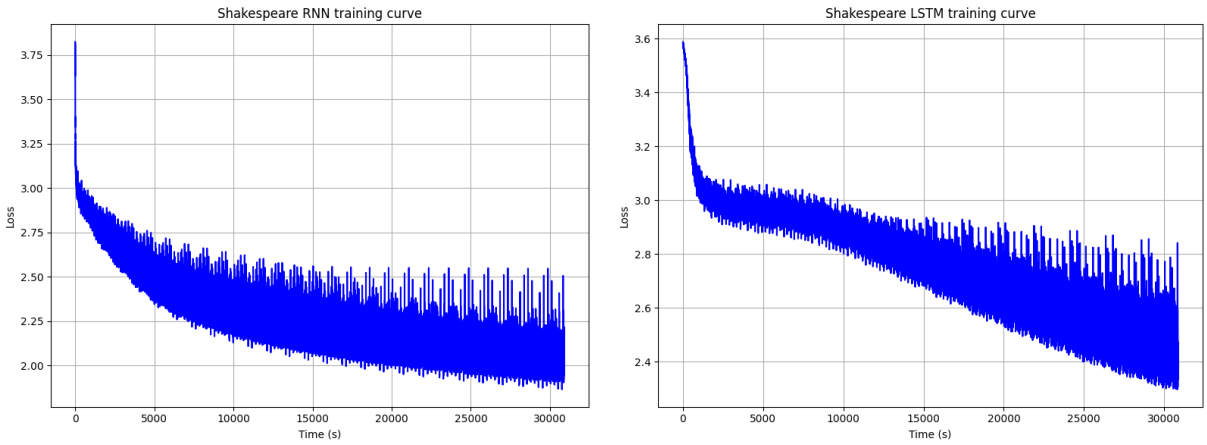


(a) Adversarial loss (equation 3) vs iteration



(b) Trained MLP predictions for adversarial example

Figure 5: Training curve and predictions for an adversarial example



(a) RNN

(b) LSTM

Figure 6: Loss functions over time while training different sequence models on the complete works of Shakespeare, to predict the next character given a string of training data