Deep Learning

USN: 303039534

1 Perceptron

We implement a two-state perceptron with weights $\mathbf{w}$ and inputs $\mathbf{u}$ as described in Dayan and Abbott such that the state v of the perceptron is given by

$$\begin{aligned} +1 \ if \ \ \mathbf{w}\cdot\mathbf{u} - \gamma > 0 \\ -1 \ if \ \ \mathbf{w}\cdot\mathbf{u} - \gamma < 0 \end{aligned} \tag{1}$$

Where γ is a threshold parameter.

We note that for the present task of classifying inputs according to whether their sum is positive or negative, the desired state of the perceptron is undefined in the case where the 10-element input vector has a sum of 0. We can approach this problem in two ways. Either we consider 0 as a separate state and train the perceptron on a 3-way classification problem. Otherwise, we arbitrarily assign state 0 to either the set of positive-sum or negative-sum input vectors. Since a perceptron in its strictest sense must be a binary classifier of binary inputs, we opt for the latter solution and arbitrarily classify zero-sum input vectors as belonging to the +1 state.

We train this perceptron using three different learning rules as also described in Dayan and Abbott.

1. The Hebbian learning rule given in equation 2

$$\mathbf{w} = \frac{1}{N_\mu}\sum_{m=1}^{N_S} v^m \mathbf{u}^m \tag{2}$$

   Here N_μ is the dimensionality of the space, N_S is the number of input patterns, and v^m is the desired output for pattern m with input $\mathbf{u}^m$. Note that this learning rule contains no method for updating γ which is thus fixed at 0 in the present implementation.

2. The perceptron learning rule given in equation 3

$$\mathbf{w} \to \mathbf{w} + \frac{\epsilon_w}{2}(v^m - v(\mathbf{u}^m))\mathbf{u}^m \tag{3}$$

$$\gamma \to \gamma - \frac{\epsilon_w}{2}(v^m - v(\mathbf{u}^m)) \tag{4}$$

   Here ϵ_w is the learning rate, $v(\mathbf{u}^m)$ is the predicted perceptron state for input m, and the other parameters are as above.

3. The delta learning rule given in equation 5.

$$\mathbf{w} \to \mathbf{w} - \epsilon_w \nabla_w E \tag{5}$$

   The penalty function we are trying to minimize is given by

$$E = \frac{1}{2}\sum_{m=1}^{N_S}(v^m - v(\mathbf{u}^m))^2 \tag{6}$$

   Approximating the perceptron step function with a linear function when differentiating, $v(\mathbf{u}^m) = \text{sign}(\mathbf{w}\cdot\mathbf{u}^m - \gamma) \approx \mathbf{w}\cdot\mathbf{u}^m - \gamma$, we can write out the full derivative explicitly as

$$\nabla_w E = -\sum_{m=1}^{N_S}(v^m - v(\mathbf{u}^m))\nabla_w(\mathbf{w}\cdot\mathbf{u}^m) = -\sum_{m=1}^{N_S}(v^m - v(\mathbf{u}^m))\mathbf{u}^m \tag{7}$$

$$\nabla_\gamma E = \sum_{m=1}^{N_S}(v^m - v(\mathbf{u}^m)) \tag{8}$$

   This is entirely equivalent to a batch-version of the perceptron learning rule, and we will therefore observe similar behavior for the two learning rules in our analyses. Experimenting with different transfer functions could also be considered but was not done in the present report.

For the perceptron and delta learning rules, we let $\epsilon_w = 0.01$ for all analyses and terminate learning once the norm of the change in weights is less than 10^{-6} per iteration. Altering these parameters within a reasonable range was not found to affect the performance of the networks.

For an initial investigation of how the three learning rules perform on this problem, we assess converged performance as a function of the number of training patterns N and plot the result in figure 1. The total number of possible input patterns is $2^{10} = 1024$, and varying N from 1 to 400 we observe convergence at $N > 250$. To quantify the performance for each N, we train 10 separate networks on N randomly generated patterns, calculate the fraction of 1000 randomly generated query patterns that are categorized correctly, and average this value over the 10 trials.

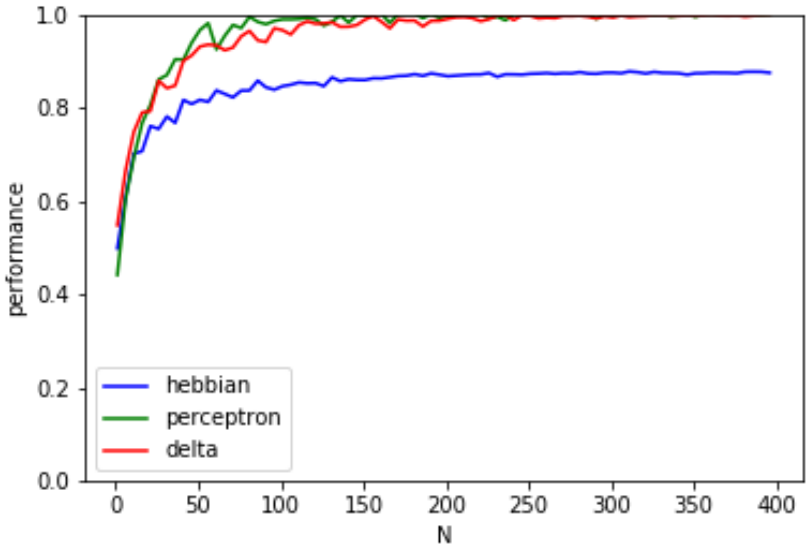


Figure 1: Performance in the sum-classification task as a function of the number of training vectors N for the Hebbian, perceptron and delta learning rules.

As expected from the considerations above, the perceptron and delta learning rules exhibit very similar performance, and both learning rules lead to perfect classification when $N \geq 250$. This is expected since it can be shown mathematically that the perceptron learning rule will converge to a perfect classifier with sufficient training data when the problem is linearly separable (Dayan and Abbott).

As expected, the Hebbian learning rule performs somewhat worse as it does not explicitly optimize our utility function and thus converges to a performance of 87%. The networks all converge to their asymptotic performance after roughly 250 training inputs suggesting that the three methods require similar amounts of training data. However, we can also carry out a comparison of the number of iterations it takes for the networks to achieve this accuracy. For the Hebbian network, the weights are specified in a single step and learning is thus instantaneous. For the perceptron and delta networks, performance as a function of epoch number is given in figure 2 for a network with N=400.

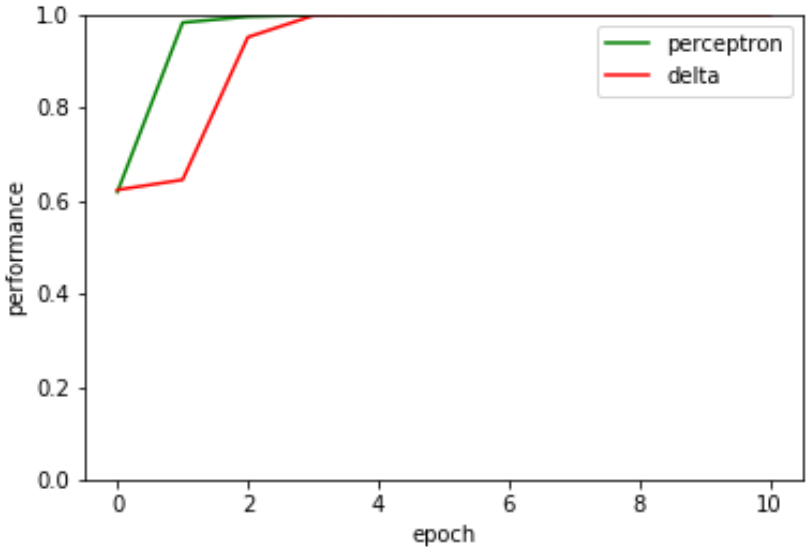
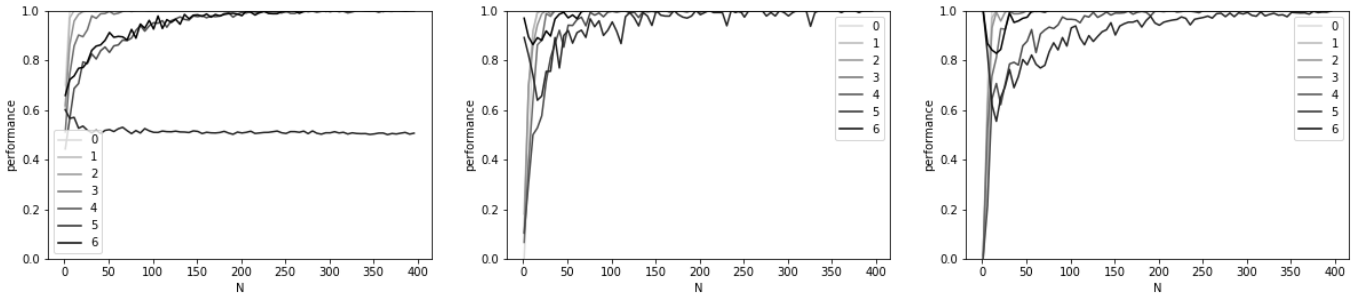


Figure 2: Performance as a function of epoch number with 400 training vectors for the perceptron and delta learning rules averaged over 100 trials.

We see that the perceptron rule has a very high succesrate after a single epoch and consistently convergences to 100% success after 2-3 epochs. On the contrary, the delta learning rule has a worse performance after a single epoch having only performed a single batch update step, but it also converges to 100% success after 2-3 epochs. For smaller training datasets, learning by the delta rule is sometimes faster than the perceptron rule as it is less prone to fluctuations from individual training vectors (not shown).

We can gain further insights into the failure modes of the networks trained by the different methods by assessing their ability to classify inputs specifically when the input vectors contain a particular number n of positive elements and $10-n$ negative elements. The learning rate as a function of training set size in this scenario is given in figure 3.



(a) Hebbian learning (b) Perceptron learning (c) Delta learning

Figure 3: Performance of a perceptron in the sum-classification task depending on the number of positive elements in the 10-element query vector. Learning curves are coloured from lighter to darker as the number of positive elements increases from 0 to 6.

We see that all three learning rules perform the worst when classifying inputs with 5 positive elements, followed by $n = 4$ for the delta and perceptron learning rules with $n = 4$ and $n = 6$ being equally hard for the Hebbian network. The Hebbian network has a performance of only 50% for $n = 5$, but once we realize that the Hebbian network has $\gamma = 0$ this is not surprising. The expected value of a given input element is $\langle u_i^m \rangle = 0$ for $n = 5$ and the expected value $\langle v^m * u_i^m \rangle > 0$ for $n \neq 5$ and identical for all i. We thus expect the Hebbian weights to approach a scaled vector of ones as N increases, which is also observed empirically. With $\gamma = 0$, this leads to correct classification when $n \neq 5$ but random classification of the $n = 5$ case depending only on the noise in the input data. Indeed the overall converged performance of 87% for the Hebbian network arises from a perfect performance on inputs with $n \neq 5$ and 50% success for the 25% of inputs for which $n = 5$. In the perceptron and delta networks, performance is worst for $n = 5$ and $n = 4$ since these are the cases separating the two output categories and thus most easily affected by noise in the training data.

On the basis of the above considerations, we expect that artificially imposing a negative gamma would lead to better performance for the Hebbian network. We show this in figure 4 where we see that we can improve the Hebbian performance to 0.971 when $\gamma = -0.25$ for a network with N=400. If we increase gamma beyond 0.25, performance deteriorates for $n = 4$ and overall performance decreases.

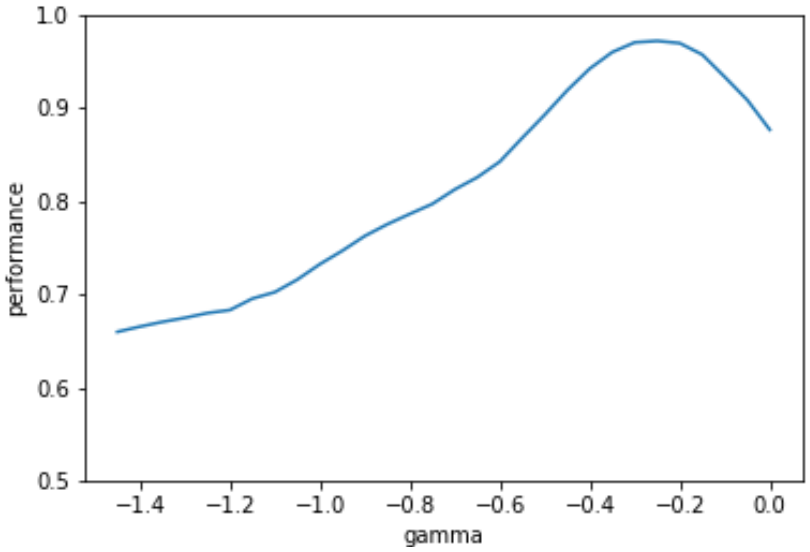


Figure 4: Converged performance of a Hebbian network with N=400 as a function of γ.

Having considered the relatively simple sum-classification task where $v = \text{sign}(\sum_i u_i)$ we now proceed to the more complex task of classifying 10-element input vectors according to the product of their elements $v = \prod_i u_i$. The performance in this task for the three learning rules is given in figure 5 again averaged over 10 separate trials.

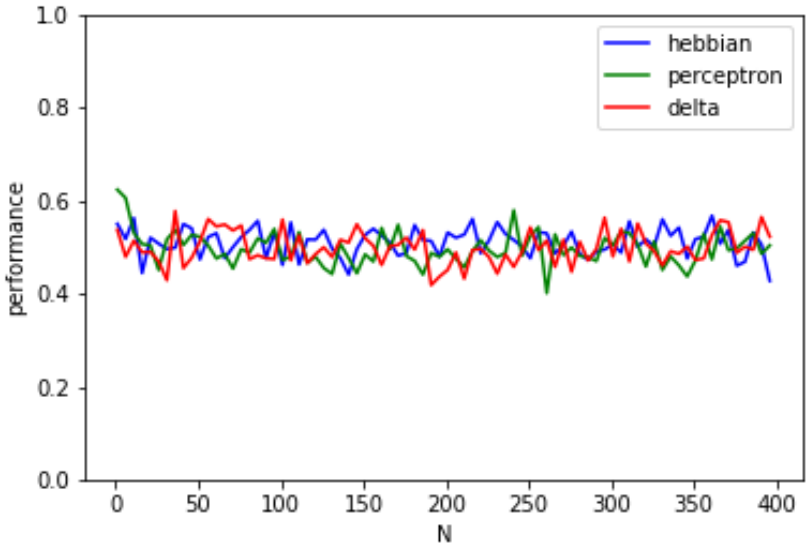
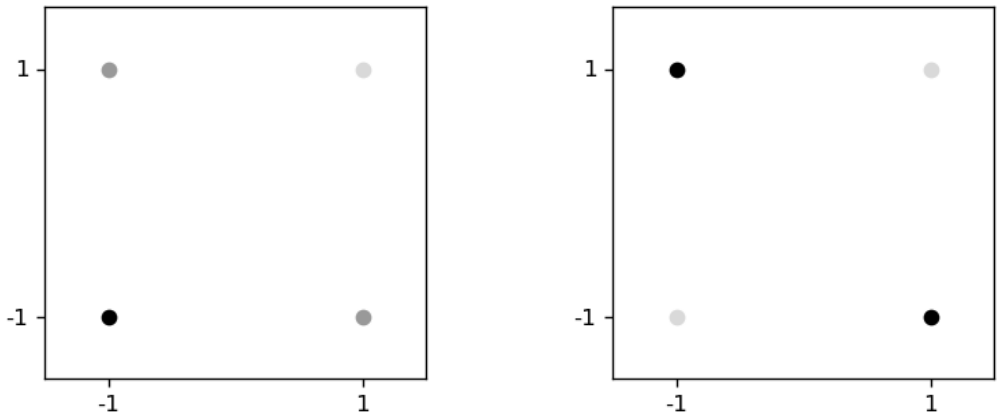


Figure 5: Performance in the product-classification task as a function of the number of training vectors N for the Hebbian, perceptron and delta learning rules.

In this case we see that all three learning rules lead to poor near-random performance. To understand why the parity problem is so much harder than the sum problem, we again turn to the concept of linear separability. If a problem is linearly separable, there exists a (hyper)plane that can succesfully separate all possible inputs into the two output categories. In this case, it is guaranteed that the perceptron learning rule will lead to perfect performance on the training data if trained for long enough. The sum problem is indeed linearly separable; a feature that is illustrated in two dimensions in figure 6a. Here, the axes indicate the values of the two input elements, and the color of the dots indicate whether they should be classified as +1 (light grey) or -1 (black). For the sum problem, the 0-sum vectors are undefined, but we see that they can easily be included in either category and we can still draw a single

line that separates the light grey (and dark grey) dots from the black dots. This defines linear separability in two dimensions.



(a) Sum problem (b) Parity problem

Figure 6: Illustration in two dimensions of the linear separablility of the sum problem and the lack thereof in the parity problem. Axes indicate elements of the two-dimensional input vector, dot color indicates the desired classification as +1 (light grey), -1 (black) or undefined (dark grey) as a function of the input elements.

From figure 6b on the other hand, we see that the parity problem is not linearly separable as we cannot draw a line that separates the black from the grey dots. The best we can achieve here is a 75% succes rate by erroneously including one black dot with the grey dots or vice versa. This linear inseparability generalizes to higher dimensions and explains why the perceptron cannot solve the parity problem. In such non-separable cases, we instead need multi-layer perceptrons which can succesfully transform the non-separable problem into a separable problem and then solve it. This is the topic of the next section.

2 Multi-layer perceptron

To construct a multilayer perceptron, we first relax the constraint that inputs and outputs must be binary. We then add a single hidden layer to give the fully connected network in figure 7 where the dimensionalities are given for the MNIST problem. In the following, we subscript the input, hidden and output layers with i, j, k respectively.

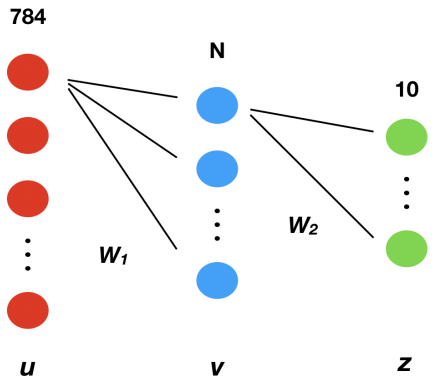


Figure 7: Illustration of the multi-layer perceptron as a fully connected network. Labels above nodes indicate dimensionalities for the MNIST problem, labels below nodes indicate the notation used in the present report.

To implement the multi-layer perceptron, we need a differentiable transfer function and initially choose the tanh function $f(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$ for consistency with the above step function in the single perceptron, which corresponds to the tanh function in the limit of infinite scaling. We will also consider the sigmoidal function $f(x) = \dfrac{1}{1+e^x}$ later in the report.

Denoting the activities of the input, hidden and output layers $\mathbf{u}, \mathbf{v}, \mathbf{z}$ respectively, we have

$$\mathbf{v} = f(\mathbf{u}\mathbf{W_1}) \tag{9}$$

$$\mathbf{z} = f(\mathbf{v}\mathbf{W_2}) \tag{10}$$

Here $\mathbf{W_1}$ is a 784xN matrix and $\mathbf{W_2}$ is an Nx10 matrix for the MNIST problem, and f denotes element-wise application of our transfer function.

In order to train the network, we first define an error function

$$E = \frac{1}{2}\sum_k (t_k - z_k) \tag{11}$$

Here, t_k is a 1-hot representation of the training signal for input k. We then minimize this error using backpropagation. This corresponds to implementing a learning rule where

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{12}$$

For the second set of weights $\mathbf{W}_2$, we can easily calculate the required derivative using the chain rule and equation 10

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - z_k)\frac{\partial z_k}{\partial w_{jk}} = -(t_k - z_k)f'(\sum_j w_{jk}v_j)v_j \tag{13}$$

For the tanh function, $f'(x) = 1 - f(x)^2$, and we can thus use the activities computed in the forward pass to also calculate the weight changes in our backpropagation step since this implies $f'(\sum_j w_{jk}v_j) = 1 - z_k^2$ (for the sigmoidal transfer function, $f'(x) = f(x) - f(x)^2$ instead). Defining $\delta_k = (t_k - z_k)(1 - z_k^2)$ for the tanh transfer function, this gives our second layer learning rule as

$$\Delta w_{jk} = \eta \delta_k v_j \tag{14}$$

For the first layer of weights $\mathbf{W}_1$, we need to apply the chain rule repeatedly and arrive at

$$\frac{\partial E}{\partial w_{ij}} = -u_i f'(\sum_j w_{ij}u_i)\sum_k [(t_k - z_k)f'(\sum_j w_{jk}v_j)w_{jk}] \tag{15}$$

Using the fact that $f'(\sum_j w_{ij}u_i) = 1 - v_j^2$ for the tanh transfer function, this gives

$$\frac{\partial E}{\partial w_{ij}} = -u_i(1 - v_j^2)\sum_k [(t_k - z_k)(1 - z_k^2)w_{jk}] \tag{16}$$

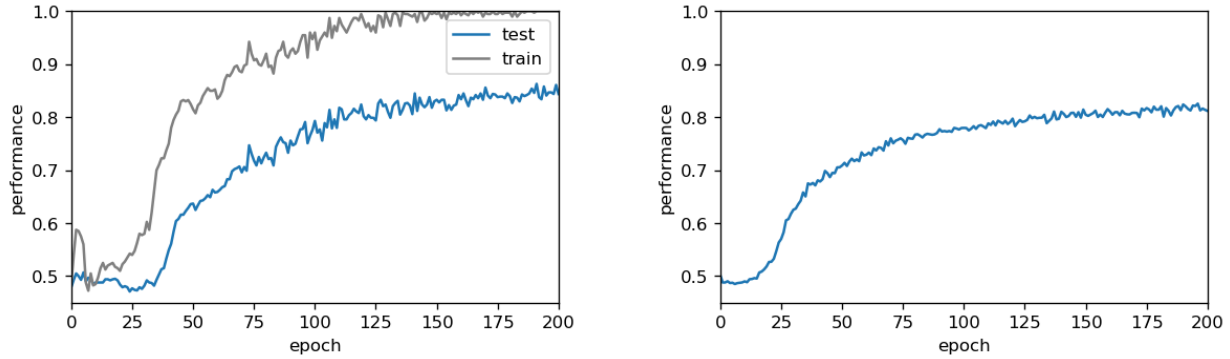
Inserting the above definition of δk we can further define

$$\delta_j = (1 - v_j^2)\sum_k \delta_k w_{jk} \tag{17}$$

This finally gives an update rule for our first layer of weights

$$\Delta w_{ij} = \eta \delta_j u_i \tag{18}$$

We first validate our implementation of the MLP on the parity problem from the single perceptron section. We train the system on 400 training inputs and quantify its classification success on 1000 randomly generated queries as above. For this preliminary analysis, we fix our learning rate at $\eta = 0.05$ and $N = 50$ and include bias units. Defining an epoch as a single pass of elementwise training on the full training dataset, we plot the classification performance as a function of epoch number in figure 8.

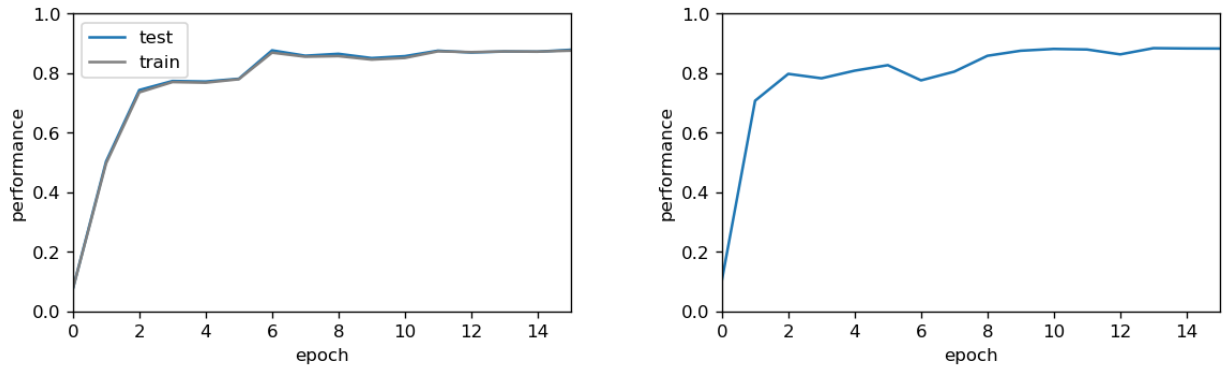


(a) (b)

Figure 8: Performance of a multi-layer perceptron with $N = 50$ and $\eta = 0.05$ in the parity problem. (a) Performance in a single trial for both the test and training data, (b) mean test performance over 10 independent trials.

We see that the multi-layer perceptron succesfully learns the problem to an accuracy of just over 80%, while the performance on the training data goes to 100%. This shows that our implementation does indeed work and that a multi-layer perceptron can solve the parity problem. We will discuss the use of dropout to diminish the training-test performance gap later.

Having validated our MLP, we move on to the MNIST dataset consisting of 60,000 handwritten training digits and 10,000 test digits. We provide the input data as 784-element vectors rescaled such that all elements are between 0 and 1. Our labels are provided as 10-element 1-hot representations of the correct digit and weights are initialized from a Gaussian distribution with $\mu = 0$ and $\sigma = 0.01$. Running a first-pass training with $N = 50$ and $\eta = 0.05$ similar to above, we see that the multi-layer perceptron can also learn the MNIST problem without much parameter optimization (figure 9).



(a) (b)

Figure 9: Performance of a multi-layer perceptron with $N = 50$ and $\eta = 0.05$ in the MNIST problem. (a) Test and training performance in a single trial, (b) mean test performance over 5 independent trials.

For this analysis, the python implementation took an average of 150 seconds per epoch, making an extensive parameter search and optimization challenging. However, converting the same algorithm to julia lead to a 50-fold improvement in computation time to 3.1 seconds per epoch, allowing us to investigate more thoroughly the effect of various network parameters on the performance of the network (note that all training is carried out with a batch size of 1 leading to relatively long epochs).

We start by investigating the effect of the number of hidden units N and the learning rate η in a simple network with no bias units. We carry out a grid search for small N with the result given in figure 10.

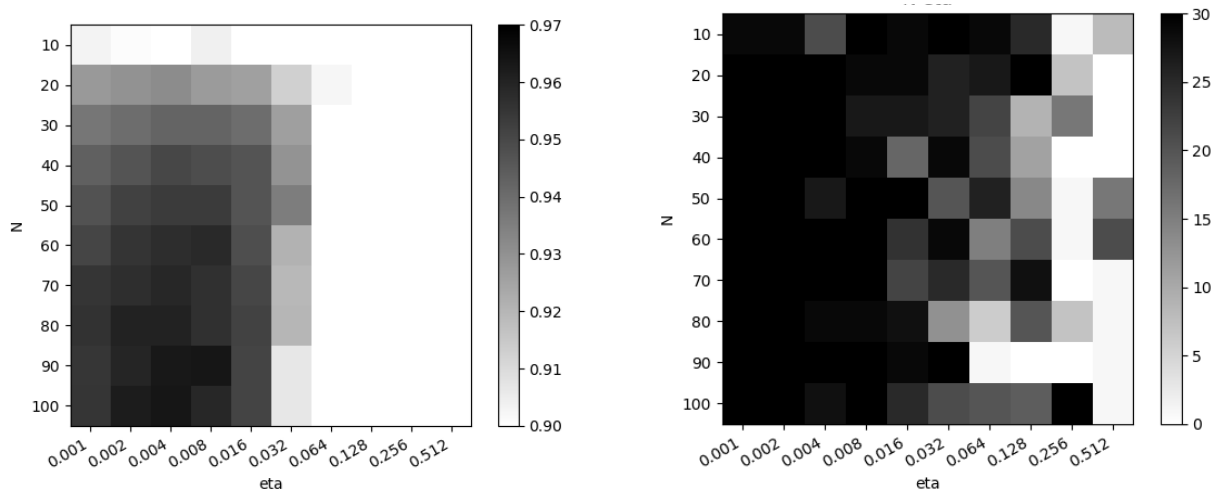


Figure 10: *left*: maximum classification performance as a function of N and η. The colormap has been capped at a lower value of 0.90 to resolve the differences between the more efficient networks. *right*: number of epochs after which the maximum performance is observed. The training sessions were capped at 30 epochs. Performances were averaged over 3 independent networks.

We firstly note that an average performance of $\approx 95\%$ is achieved for a range of different parameters, indicating that even a very simple and relatively small network can achieve good performance in the MNIST classification problem. We also note that the optimum performance is generally achieved after 30 epochs with small learning rates, suggesting that further training might lead to additional improvements. Interestingly, there appears to be only little improvement in performance when increasing N beyond 50.

Since we can achieve good performance for a relatively small network with N=50 and $\eta \approx 0.01$, we use these parameter values to generate learning curves for fixed N and variable η and vice versa in figure 11 to get a better understanding of the effect of these parameters on our network.

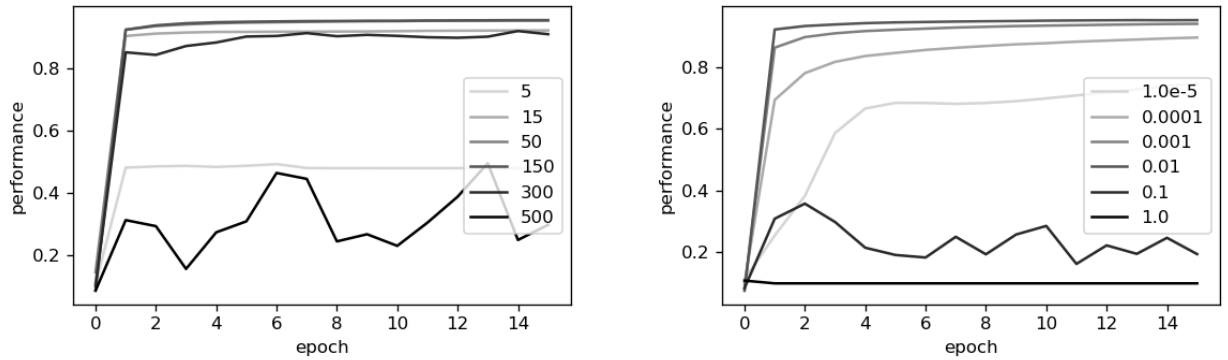


Figure 11: Learning curves for variable N (left) and η (right) with the reciprocal parameter fixed at $\eta = 0.01$ and N=50 respectively. Variable parameters are given in the legends with darker lines corresponding to higher parameter values. Performances are averaged over 3 trials.

We see that as N increases, the converged performance initially increases followed by a plateau of similar performance for N=15-300 (note that the N=50 and N=150 curves coincide). Finally, performance drops catastrophically when the network size becomes too large compared to our learning rate. Together with figure 10 this suggests that increased network size only leads to improved performance if the learning rate has been optimized.

When our learning rate increases, we similarly observe an initial increase in converged performance for a range of η as well as an increase in the rate of learning. However, once η becomes too high compared to N, we observe a catastrophic decrease in performance as the network jumps around the energy landscape rather than descending monotonously.

Taken together, these observations suggest that we can improve performance by increasing network size, but this requires individual optimizations of the learning rates for different network sizes and also leads to increased computational costs since larger networks have more parameters and also require smaller learning rates and thus exhibit slower learning. Additionally, we expect that if the networks become large enough relative to our training data, we will observe overfitting and a decrease in test performance. Given the present computational and algorithmic limitations, this has not been observed.

Based on these investigations, it is clear that we can achieve sensible performances with no bias units, but we now proceed to investigate the effect of bias units for the input and hidden layers. Bias units are additional nodes with a constant value of 1, fully connected to the next downstream layer. These nodes allow us to shift the function we are fitting and thus learn more complex datasets. We therefore expect the addition of bias nodes to improve performance, and indeed it is somewhat surprising that we are able to learn the MNIST dataset with no bias nodes in the first place.

Surprisingly therefore, it turns out that adding bias units does not lead to an observable performance increase for our network with N=50 and $\eta = 0.01$ (figure 12a). One possibility is that this lack of effect is due to us using an antisymmetric transfer function, while in the case where $f(x = 0) \neq 0$, shifting the fitted function could become more important. We therefore modify our MLP to be able to use the sigmoidal transfer function, but still do not observe a difference between networks with and without bias units (figure 12b). Finally, we investigate whether the lack of effect is due to the particular representation of the data that we use such that shifting the input data might necessitate bias units. We therefore shift all input vectors by 0.625 and train the network on this new input dataset (figure 12c).

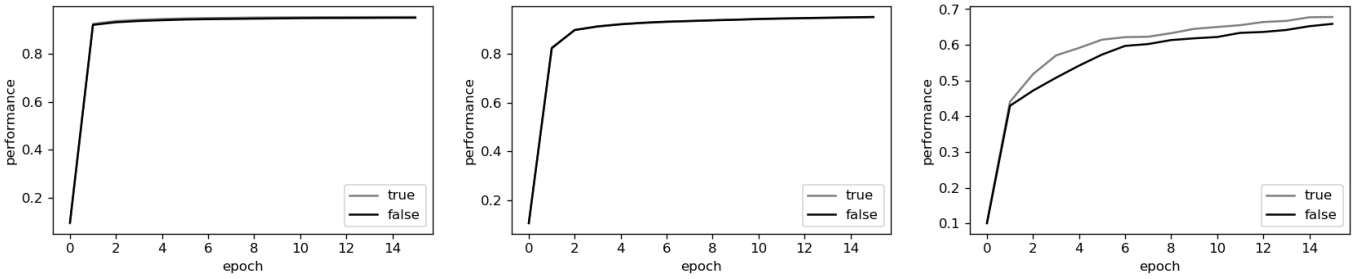


Figure 12: Performance of the MLP on the MNIST dataset with (grey) and without (black) bias units. (a) performance with a tanh transfer function, (b) performance with a sigmoidal transfer function (c) performance with a tanh transfer function after shifting the input data by 0.625 (mean of 30 trials).

We see that the inclusion of bias units does indeed lead to slightly improved performance for this shifted dataset, indicating that the importance of bias units depends on the particular input dataset in question. To investigate this further, we therefore run a similar comparison on the parity problem and see that there is a very large difference in performance between networks with and without bias units for this particular problem (figure 13).

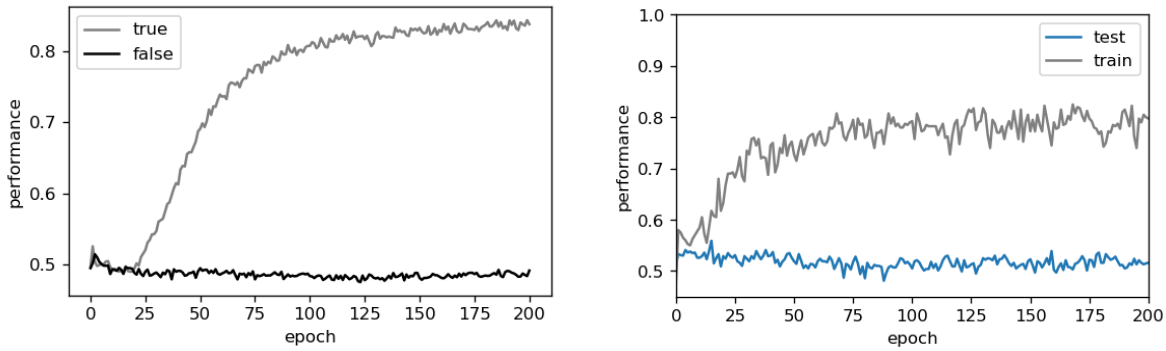


Figure 13: Importance of bias units for the parity problem. (a) mean performance with (grey) and without (black) bias units on the test data over 10 trials. (b) performance on the test (blue) and train (grey) data for a single trial with no bias units. N=50 and $\eta = 0.05$ for both plots.

In this case, the network completely fails to solve the parity problem in the absence of bias units, with the asymptotic performance being even worse than random guessing. On the contrary, the network with bias units learns the problem with an asymptotic test performance greater than 80%. We thus conclude that bias units can greatly improve performance for certain types of input, while it is relatively unnecessary if the data has been appropriately preprocessed. However, in general the inclusion of bias units does lead to improved performance with little increase in computational cost, and we therefore include them for all subsequent analyses.

We also see from figure 12b that the performance with the sigmoidal transfer function is very similar to what we observed for the tanh transfer function and therefore provide a direct comparison in figure 14 showing that the asymptotic performance is indeed similar although the tanh network learns more quickly.

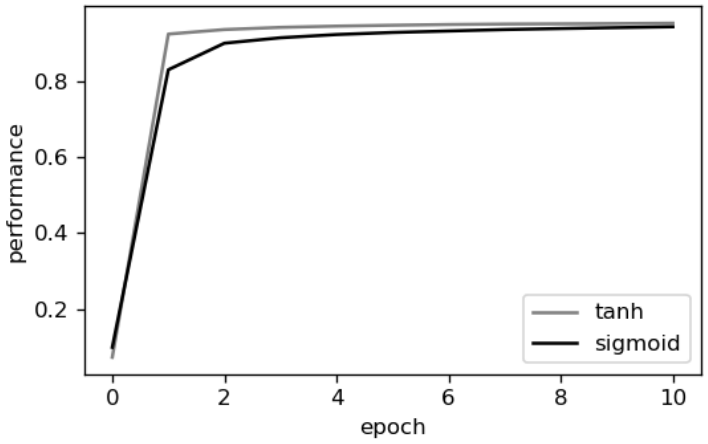


Figure 14: Performance with tanh (grey) and sigmoidal (black) transfer functions for a network with N=50, $\eta = 0.01$, including bias units, and averaged over 3 trials.

Although we have predominantly investigated small network above, we note that LeCun et al. (1998) achieved MNIST error rates of 4.7% and 4.5% with 300 and 1000 hidden units respectively. For the purpose of investigating the classification efficiency of the MLP on the MNIST data, we therefore train a new network with 300 nodes for 20 epochs with a learning rate of 0.005 (optimization not shown) after which the performance has reached 96.2% (figure 15). This corresponds to an error rate of 3.8% and is thus better than both LeCun's 300-node and 1000-node networks.

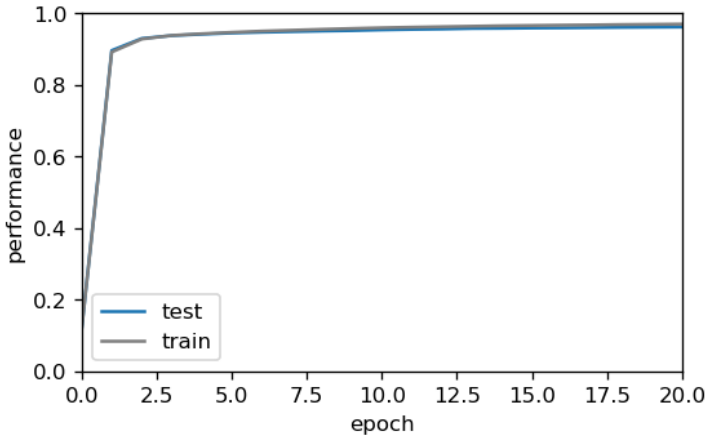
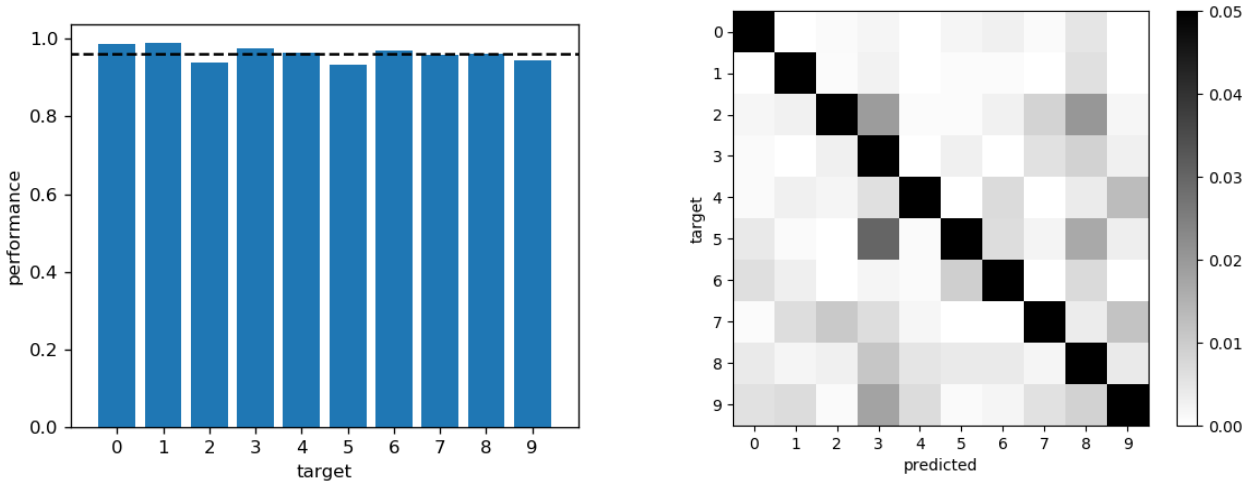


Figure 15: Learning curve for a network with 300 hidden nodes and $\eta = 0.005$. Note that even for this larger network, we do not overfit the data.

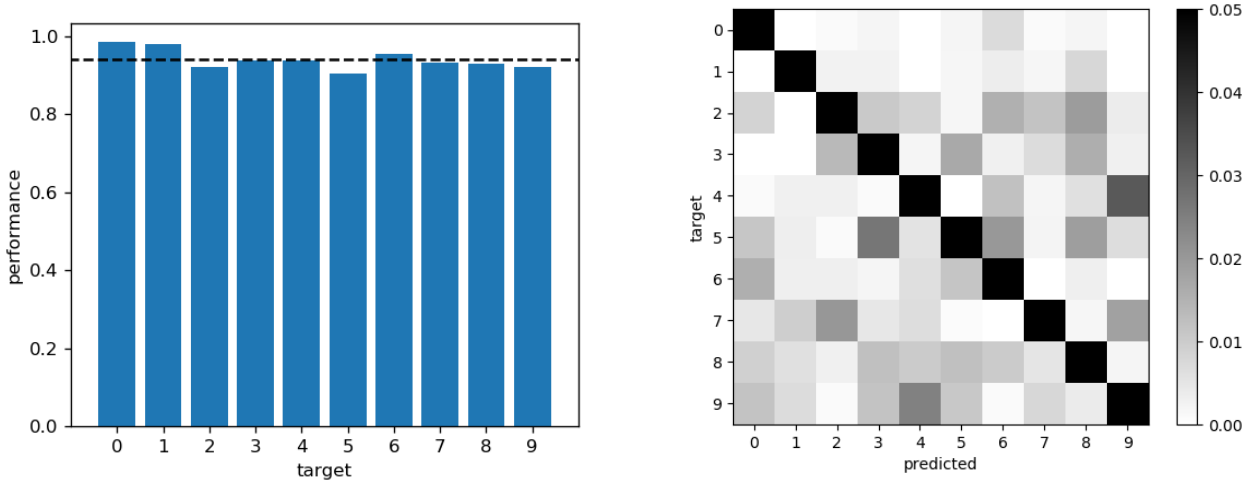
After training the network, we can look at the types of errors it makes to see if there are any particular classifications it struggles with. We therefore plot a histogram of the performance for the 10 different digits in figure 16a and see that the performance for different digits does vary significantly. Notably, performance is very low for '2' and '5' while it is very high for '0' and '1'. To understand why this is the case, we further plot a cross-prediction matrix in figure 16b. This shows for each target digit (rows) how often the network classifies this digit as each of the 10 digits (columns).



(a) (b)

Figure 16: (a) Performance for each of the 10 digits. Dashed line indicates mean performance across test data. (b) Cross-prediction showing how often a given number is predicted for each correct digit. Note that the colormap is capped at 0.05 so we cannot resolve the diagonal elements. However, these values are given in figure (a), and the low cap allows us to better distinguish the off-diagonal elements.

Notably, '2' and '5' are both commonly confused as '3' and '8' but not vice versa, leading to the poor performance for these digits observed in figure 16a. In general, we see that many other digits are commonly misclassified as '3' and '8'. This suggests that these two digits are polymorphic in the training data and the corresponding classes therefore occupy a large volume of input parameter space. These observations are similar to what we see using a sigmoidal transfer function (figure 17) or different N and η (not shown), suggesting that they are features of the data rather than the particular network instance.



(a) (b)

Figure 17: (a) Performance for each of the 10 digits with a sigmoidal transfer function. (b) Cross-prediction showing how often a given number is predicted for each correct digit.

To understand how the network manages to correctly identify the digits, we train a network with 10 hidden units for 20 epochs, achieving a performance of 90.8%. We then plot in figure 18 for each hidden unit the weights from the input layer to that unit, reshaped to correspond to the input images. This shows the features that lead to activation of each of the hidden units.

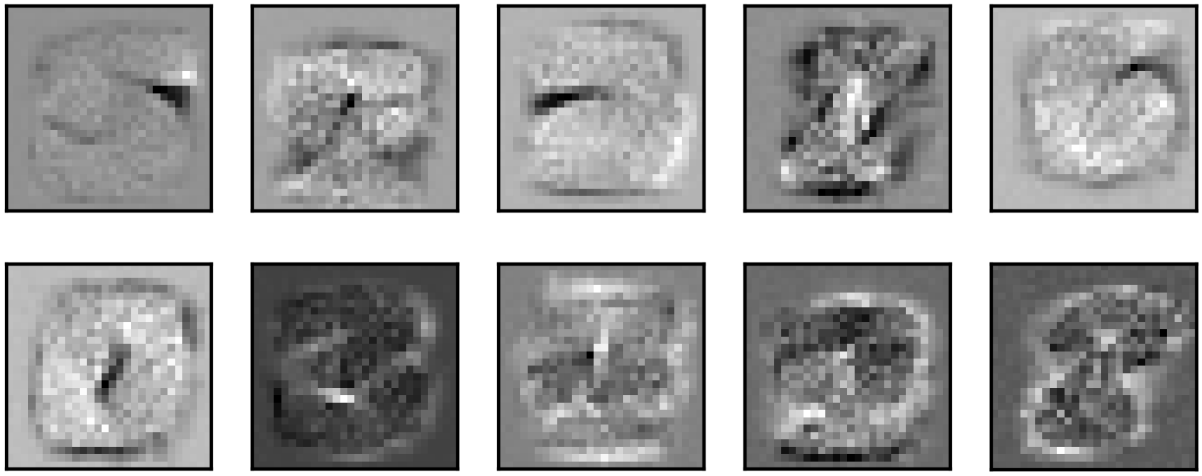


Figure 18: Input weights for each hidden unit in a 10-unit network, reshaped to resemble the input images before flattening.

We see that in the case of N=10, there are recognizable patterns already in the hidden layer which e.g. contains nodes resembling the numbers 0 and 3 (bottom left). Since many weights and all activations are less than 0, much of the network operates in the linear regime of the tanh function, and we can therefore get an idea of the activation of the output layer by multiplying the $\mathbf{W}_1$ and $\mathbf{W}_2$ matrices. We plot the resulting pseudoweights from the input layer to the output layer in figure 19.

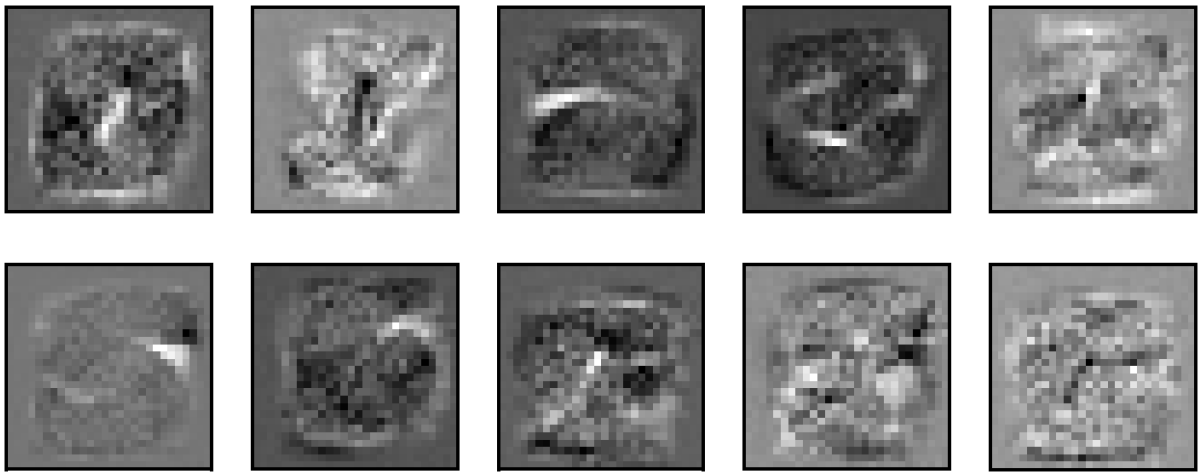


Figure 19: Rowplot of $\mathbf{W}_1 \cdot \mathbf{W}_2$ in a 10-unit network, reshaped to resemble the input images before flattening.

We see that some of these units resemble those in the hidden layer, and a manual inspection of the weights confirms that these have a single dominant input from the hidden layer. Other output units are combinations of hidden units with e.g. output unit 5 corresponding to the number '4' having coefficients of -0.8 from both the first and tenth hidden units and near-zero weights from all other hidden units. In general, we see that many of the activations resemble the digits we are trying to predict with exceptions including '8' - a digit we already know to be rather 'diffuse' in input space from our cross-prediction matrix above.

For larger network sizes, the output units are increasingly made up of combinations of hidden units which are used

as a basis for classification to leverage the increased dimensionality of parameter space. This is expected since we would otherwise achieve comparable performance with single-layer networks. Rather than observing hidden units with activations resembling actual digits, we therefore also see that the activations can now resemble 'features' such as circles or edges. As an example, the first 10 hidden units of a trained network with N=300 have been plotted in figure 20.

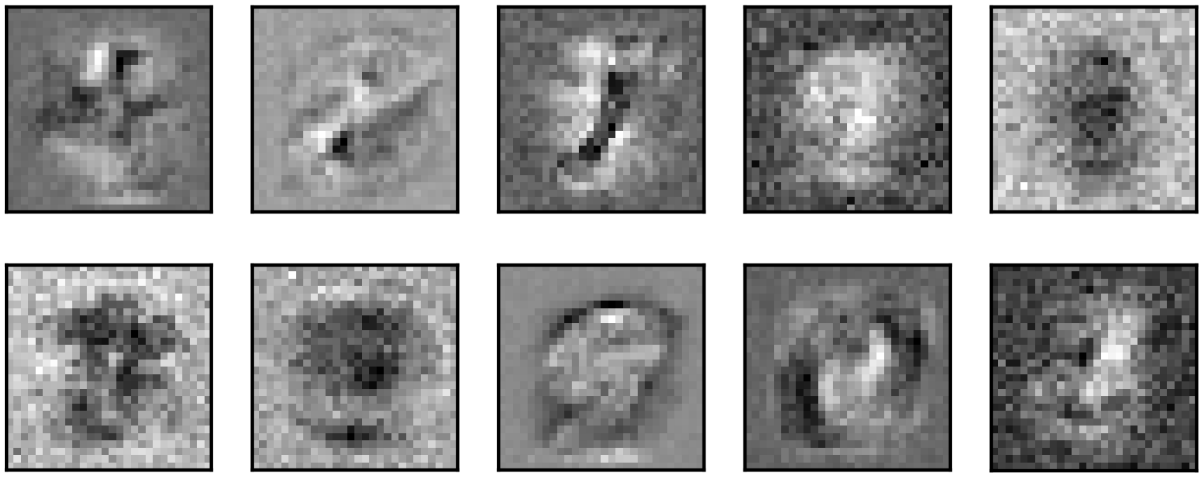


Figure 20: Input weights for each of the first 10 hidden units in a 300-unit network, reshaped to resemble the input images before flattening.

We thus conclude that the hidden units contribute to solving the problem in small networks by responding to one or a few digits, while the hidden units in larger networks correspond to particular features and digits are classified as particular combinations of these features by the second layer of weights.

Finally we return to the problem of overfitting, which is commonly solved using dropout as described by Srivastava et al. (2014). As we see from figure 15, overfitting is not a problem in the present case of 60,000 input data points and only a few hundred hidden nodes. However, if we increase the hidden layer size to the point where overfitting might be a problem in this relatively shallow network, the computational time becomes rather overwhelming for the present implementation.

Instead of resorting to keras or pytorch, we therefore choose to downsample the original training data to investigate the effect of dropout. The downsampling of the input data serves two purposes; firstly it decreases the epoch length for a fixed hidden layer size, and secondly it decreases the hidden layer size needed to overfit the data. We thus achieve an almost quadratic improvement in the time needed to investigate the effects of dropout.

We downsample the input data to just 200 data points and use 1000 random data points as our test data. We then train a network with N=1000 using a sigmoidal transfer function either with or without dropout and plot the resulting learning curves in figure 21. In this case, we use a dropout rate of 10% in the input layer and 40% in the hidden layer since this was found to work well.

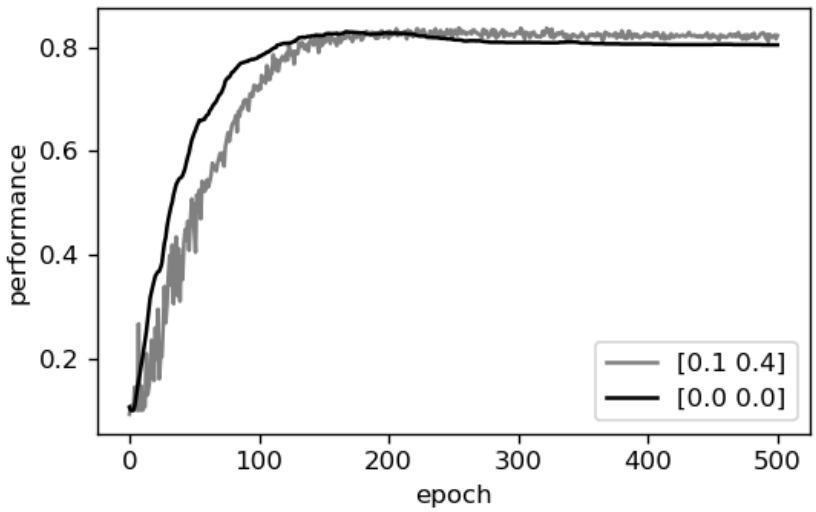
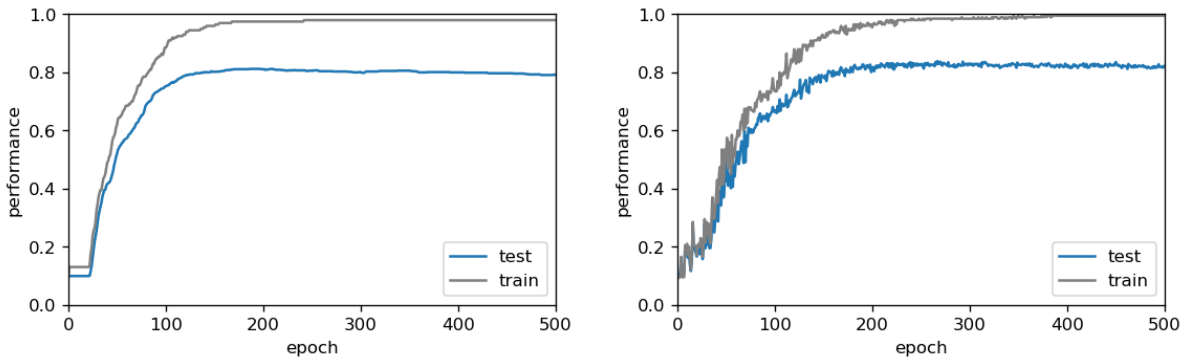


Figure 21: Learning curves for a network with 1000 hidden nodes and $\eta = 0.02$ with input data downsampled to 200 data points and averaged over 3 trials. Legend indicates dropout rate for the input layer and hidden layer.

We see that without dropout, training is initially faster than with dropout but peaks at a test performance of 82% after 180 epochs following which the training performance keeps increasing while the test performance decreases to 80% due to overfitting of the input data. This is further illustrated in figure 22a. When we include dropout, training is initially slower. However, the test performance continues to increase together with the training performance, and we therefore converge to a better test performance of 83% after 500 epochs. This is further illustrated in figure 22b. On a side note, it is also rather remarkable that we can achieve 83% efficiency with only 200 training data points.



(a) (b)

Figure 22: Learning curves for a network with 1000 hidden nodes and $\eta = 0.02$ on input data downsampled to 200 digits with (right) and without (left) dropout.

To get a better idea of the effect of the rate of dropout, we fix the dropout rate at 10% for the initial layer as above and plot the learning curve for a range of different second layer dropout rates in figure 23. We see that as the dropout rate increases, the rate of learning becomes progressively slower. However, the converged performance increases slightly as the second layer dropout rate increases from 0 to 0.6. For a large dropout rate of 0.8, learning becomes very noisy and performance seems to deteriorate, although longer training sessions might alleviate this.

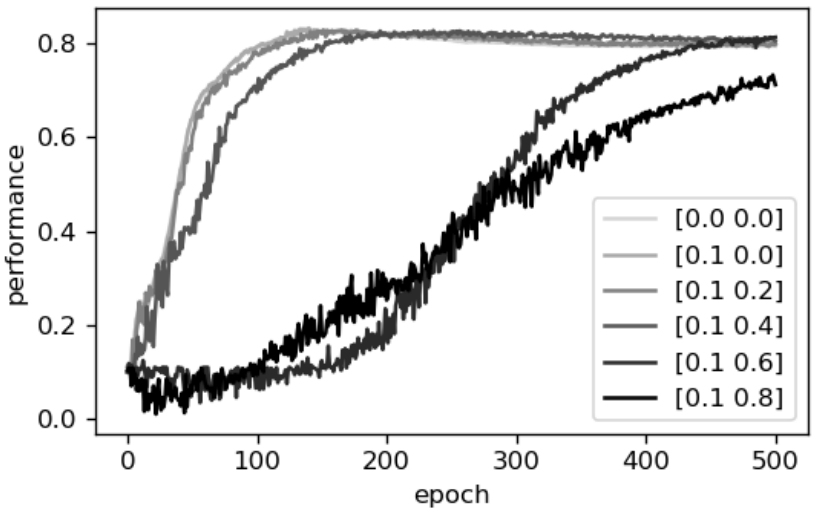


Figure 23: Learning curves for a network with 1000 hidden nodes and $\eta = 0.02$ with input data downsampled to 200 data points. Legend indicates dropout rate for the input layer and hidden layer.

In summary see that dropout limits overtraining by changing the combinations of weights that are trained together in each iteration. This leads to noisier and slower training but can converge to a better performance when the network size is very large compared to the training dataset. In the present case, overfitting is not very pronounced and dropout thus leads to only a modest increase in test performance, but in other contexts with more compute power and larger networks, dropout or other regularization methods can be essential to train a functional network.

3 Critique of deep learning

Gary Marcus, Professor of Psychology and Neural Science at New York University, published an opinion piece on arXiv in early 2018 discussing shortcomings of current approaches in deep learning and AI. The article, *Deep Learning: A Critical Appraisal*, starts by assessing the current state of the field and proceeds to outline 10 limitations of deep learning. In the following, I will outline the main points and arguments made by Marcus while simultaneously commenting on his ideas using examples from the literature. I will then outline the debate with others in the AI community that this article initiated.

In the article, Marcus sets the scene by defining deep neural networks as multi-layer networks trained by backpropagation to learn an input-output mapping in a supervised learning task, and he describes how deep learning has come to dominate the AI field since 2012. He proceeds to highlight the importance of convolutional neural networks (LeCun et al. 1989), and lists speech recognition, object recognition, and video & board games as fields where deep learning has achieved significant advances, pointing out that with enough data and computational power, deep networks could possibly solve any desired task. Marcus then returns to the real world of limited data and compute power which requires AI systems to generalize, and he outlines the following ten challenges faced by deep learning in this respect.

1 *Deep Learning is 'data hungry'*
Marcus points out that while humans are able to generalize from descriptions and a few examples using abstract relations, neural networks generally rely on large annotated datasets without the use of abstract representations. This is exemplified by the challenges faced by convolutional neural networks in generalizing to novel viewpoints in an artificial arena. However, this has since been adressed with good results by DeepMind (Eslami et al 2018) while major advances have also been made in learning from sparse data (Santoro et al. 2016). It is thus not clear that these challenges cannot be adressed by deep networks with augmented architectures.

2 *Deep learning systems are conceptually shallow*
While the actual architectures are 'deep' (i.e. contain multiple layers), Marcus argues that neural networks have no deep understanding of the tasks at hand. This is exemplified by DeepMind's Atari system which fails when the Y-coordinate of the paddle is altered since it has not learned that the ball must hit the paddle but merely how the paddle should move depending on where the ball is. I agree with Marcus that this lack of generalization between related tasks appears to be a significant shortcoming which deep learning does not have an obvious solution to.

3 *Deep learning struggles to represent hierarchical structure*
In e.g. human language processing, we understand that there is a hierarchy to the sentences we use and hear. On the contrary, a neural network assigns each character the same weight and the same role in the input layer. This is framed as a shortcoming when it comes to generalizing, where novel sentence or task structures can lead to poor performance. However, it is worth noting that in human language processing, there is also no hierarchy in the 'input layer' (e.g. in auditory nerves). Instead we learn the hierarchy by being exposed to language, similar to how a neural network might represent sentence structure in its deeper layers after training.

4 *Deep learning struggles with open-ended inference*
Marcus argues that deep learning systems are less intelligent than humans as they cannot draw inferences from data in different domains to what they are trained on. As an example, he points out that humans can draw conclusions about the intentions of a character in a text based only on indirect dialogue. However, in this example Marcus is ignoring the fact that humans are exposed to many years of speech and action data, allowing us to construct both a prior of likely actions and a mapping between speech, feelings and actions. On the contrary, the systems he is comparing with are trained with a specific dataset that is much narrower in scope than our real-world experiences.

5 *Deep learning is not transparent*
Marcus points out that deep learning is essentially a black-box method, particularly for more complex architectures, and that this can pose problems when they are used in real-world applications with potentially fatal consequences such as medical diagnosis. This could pose a significant challenge to the AI field since it means that even if novel algorithms improve succesrates over human subjects, it is not clear who should be liable, if any, when they fail.

6 *Deep learning has not been integrated well with prior knowledge*
Again, Marcus argues that 'prior knowledge' should be explicitly encoded - e.g. in the case of learning the physics of a falling tower, one might encode Newton's law. Marcus thus argues for a hybrid approach where we combine symbolic laws and manipulations with connectionist learning under the assumption that the system cannot learn these laws purely from data.

7 *Deep learning cannot distinguish causation from correlation*
it is argued that a network can learn e.g. a correlation between height and vocabulary, but not the underlying latent

factors of growth and development. However, this is again robbing the network of much circumstancial data. If instead the network was given the height, weight, vocabulary, hair colour etc. over time for just a handful of people, it would likely be able to infer age as the most important predictor of vocabulary, and this is the type of data that humans have had access to throughout their life.

8 *Deep learning presumes a stable world*
Marcus argues that deep learning performs best when the rules governing a system of interest are invariant as in e.g. Go, whereas these algorithms are worse at predicting fluctuating systems such as the stock market or epidemiological data. However, this is not necessarily a fundamental flaw of the algorithm as much as it is an inherent difference in diffulty of the problem at hand.

9 *The answers of deep learning cannot always be trusted*
Here, Marcus points out that the results of a deep neural network can be severely affected by small changes to the inputs, as has e.g. been shown for stop signs in Eykholt et al. 2018. This leaves the systems vulnerable to adversial attacks if they are widely implemented in public. This is a known problem in the field, and the robustness of deep networks will likely need to be improved for certain industrial and public applications (Carlinin & Wagner 2016).

10 *Deep learning is difficult to use for engineering complex systems*
Building on the previous points, Marcus argues that while it is easy to design deep neural networks for simple tasks, it is hard to use them as part of a larger and more complex system due to their unpredictability in the face of novel data. Instead, deep learning is currently more commonly used for stand-alone tasks with less downstream use of the output data.

In essence, Marcus is arguing that deep learning systems are faulty because they cannot generalize outside of their training space. He concludes that deep learning is therefore not the answer in the quest for general artificial intelligence (GAI) and that this will require hybrid approaches with symbolic manipulation. While AI thus does have the potential to solve a wider range of problems, Marcus believes that the current overemphasis on deep learning means that "AI could get trapped in a local minimum, dwelling too heavily in the wrong part of intellectual space". He argues that because of these limitations, the present hype of machine learning is out of touch with reality and risks causing a second "AI winter" when expectations cannot be met in the coming years. Instead, Marcus proposes a view where deep learning is one tool in a general toolbox of AI methods, used only in cases where large datasets are available and the desired output fits neatly into a particular set of categories. For other types of problems, he proposes four alternative approaches that may lead to new insights.

1 *Unsupervised learning*
Marcus suggests that both unsupervised clustering of data and unsupervised temporal predictions using e.g. LSTM networks can help move the field forward. However, these approaches do not reduce the need for large datasets for training; they simply require a different type of data. Marcus therefore also wants to develop systems that "set their own goals and do reasoning and problem-solving at this more abstract level". In this context it is somewhat suprising that he does not comment on the development of GANs (Goodfellow et al. 2014) which is a semi-supervised form of learning where the system does indeed learn to achieve a goal by adversarial 'problem-solving', although the goal is still pre-specified.

2 *Symbol manipulation and hybrid models*
As mentioned, Marcus believes that by incorporating symbolic manipulation to learn fixed rules in parallel with the connectionist neural network mappings, we can improve generalizability.

3 *More insights from cognitive and developmental psychology*
Marcus argues that we should draw more inspiration from human cognition, but while he claims that this can help with 'symbolic manipulation' and 'common sense knowledge', he provides little insight as to how one might go about implementing these concepts. In my opinion, there may be more insight to be gained from the field of neuroscience than psychology since an increased understanding of real-world neural implementations may provide new inspiration for algorithmic structures. An example of this is the recent development of networks with 'complex synapses' inspired by biological systems, which can retain a memory of previously learned tasks to enable continous learning of multiple tasks without catastrophic forgetting (Kaplanis et al. 2018).

4 *Bolder challenges*
Finally, Marcus argues that we should set bigger goals for what to achieve with artifical intelligence, including general reasoning and learning from instructions rather than simple classification tasks. I doubt many people disagree that these goals are desirable, but how to best achieve them is still an open question.

Many of the points made by Marcus relate to the lack of 'common sense' and 'reasoning' of deep networks. However,

the examples that Marcus give such as deciding *Who is taller, Prince William or his baby son Prince George* are solvable by humans only due to the immense amount of prior data that we have from all of our real world experiences. For example, the human eye is estimated to process up to 1000 FPS at 576 megapixels. If we imagine a retinal image to consist of 8-bit color graphics and assume that I sleep 8 hours a day, I have thus seen $2 \cdot 10^{21}$ bits of visual data during my life corresponding to more than 250 exabytes of data - and we can add to that data from all of the auditory and tactile stimuli I have been exposed to. Furthermore, when I started learning from these 250 exabytes of data, I already had a hardwired network pre-programmed by evolution to learn tasks that will be useful as a human, whereas neural networks often start from random connectivity. It is therefore not entirely surprising that I can reason better than an algorithm trained on 10^6 datapoints. Humans have thus processed incredible amounts of data to form prior distributions for all of our inferences, and I am not convinced that we will ever be able to construct algorithms with 'common sense' without such diverse and extensive training data. How data from such diverse sources is integrated in the brain is an entirely different question, and one that poses a different kind of challenge to artificial neural networks. This is not to say that hybrid approaches with symbolic manipulations cannot be useful in aritificial intelligence, but as emphasized above I think Marcus is quick to dismiss deep learning before we have tapped its full potential.

Throughout the article, Marcus highlights the challenges of deep learning while critisizing the AI community for being too narrow in scope and focusing too much on deep learning. However, there is already a general consensus in the field that deep learning is not sufficient for general artificial intelligence, but that it is a small step in the right direction. In that context, it makes sense to investigate deep learning as a step forward from previous symbolic approaches before moving on to general reasoning. This is in line with the vast majority of academic research which is a cumulative process where results in simpler areas pave the way for investigations into more complex problems.

Marcus therefore also faced a backlash from the rest of the AI community after publishing this article, with much of the debate taking place on twitter with very established names in AI such as Yan LeCun, Thomas Dietterich and others. Yann LeCunn is the inventor of convolutional neural networks and head of facebook AI, while Thomas Dietterich is AAAI co-chair, NIPS chair, and often considered one of the founders of machine learning. In contrast, Gary Marcus devotes a large proportion of his time to popular science and science communication in addition to having his primary research interests in music and psychology. As such, he only has very little actual research experience in the field of deep learning or artificial intelligence more broadly, having published no original research in the field and only a few pieces of original work in cognitive science more generally in the past 10 years.

It is thus not entirely surprising that counter-arguments were quickly put forth by the AI establishment, with one comment by Dietterich reading "Disappointing article by @GaryMarcus. He barely addresses the accomplishments of deep learning (eg NL translation) and minimizes others". In other words, while Marcus claims that the AI community is downplaying the limitations of deep learning, the AI community claims that Marcus is downplaying the strengths of deep learning. While there is thus general agreement that deep learning is only part of the puzzle in achieving GAI and that other approaches will be needed as well, there is a fierce clash of rhetoric between Marcus and others.

This is exemplified by a reply from LeCun to one of Marcus' comments, where he states that
"1. Whatever we do, DL is part of the solution.
2. Hence reasoning will need to be compatible with DL.
3. That means using vectors instead of symbols, and differentiable functions instead of logic.
4. This will require new architectural concepts."
This is somewhat similar to the views of Marcus above, although there is a distinct disagreement on the role that symbolic manipulation will play in the future of AI. Marcus believes that symbolic manipulation will be part of the solution to GAI whereas LeCun and others believe that we will have to generalize beyond symbolic manipulations to keep everything in a differentiable and thus optimizable framework. In this view, symbolic rules will emerge as the result of such an optimization, and humans merely formalize it for convenience. This is similar to the view of Geoffrey Hinton who likened the hybrid approach to "developing electric cars by injecting pretrol into the engine" at the 2018 G7 conference in Montreal where he argues that such hybrid methods will hamper the progress of data-driven AI.

While there is thus a clash of rhetoric and beliefs between Gary Marcus and the AI community, what Marcus does provide is an outside view of the AI field from a person still familiar with the research taking place, but whose career is not built around deep learning. This is an important perspective to keep in mind if we are to make fundamental changes rather than merely exploring the 'same local minimum' as Marcus puts it. While dialogue between algorithmic developers, neuroscientists, and psychologists will thus continue to be important for the development of AI methods, hopefully this will take a more friendly and collaborative tone than that currently exemplified by Marcus, LeCun, and others in the field.

Carlini & Wagner "Towards Evaluating the Robustness of Neural Networks" *arXiv* (2017)
Eslami et al. "Neural scene representation and rendering" *Science* (2018)
Eykholt et al. "Robust Physical-World Attacks on Deep Learning Visual Classification" *arXiv* (2018)
Kaplanis et al. "Continual Reinforcement Learning with Complex Synapses" *arXiv* (2018)
Goodfellow et al. "Generative Adversarial Networks" *NIPS proceedings* (2014)
LeCun et al. "Deep learning". *Nature* (2015)
Santoro et al. "One-shot Learning with Memory-Augmented Neural Networks" *arXiv* (2016)

Appendix

```
#for for implementing a multi-layer perceptron with variable learning rate,
#hidden layer size (single hidden layer), transfer function (tanh/sigmoid)
#dropout rate and inclusion of bias units.

using PyCall, PyPlot, LinearAlgebra, Distributions,
    Random, DelimitedFiles, LaTeXStrings, StatsBase, Statistics, MultivariateStats
@pyimport pickle

##pickle functions courtesy of Rob Blackwell
#https://gist.github.com/RobBlackwell/10a1aeabeb85bbf1a17cc334e5e60acf
function mypickle(obj, filename)
    out = open(filename,"w")
    pickle.dump(obj, out)
    close(out)
 end

function unpickle(filename)
    r = nothing
    @pywith pybuiltin("open")(filename,"rb") as f begin
        r = pickle.load(f)
    end
    return r
end

function heatmap(results, xs, ys; xlab="", ylab="",
    filename="default", Title="default", vmin="default", vmax="default")
    #given a matrix of z values and lists of x,y values
    #plots a heatmap
    figure()
    if vmin == "default"
        imshow(results, cmap=ColorMap("gray_r"), vmin = minimum(results),
                vmax = maximum(results))
    else
        imshow(results, cmap=ColorMap("gray_r"), vmin = vmin,
                vmax = vmax)
    end
    print("plotted")
    xticks(0:(length(xs)-1), xs, rotation=30)
    yticks(0:(length(ys)-1), ys)
    colorbar()
    xlabel(xlab)
    ylabel(ylab)
    title(Title)
    savefig(filename, bbox_inches = "tight")
    close()
end

function calc_vs_zs(us; w1s=w1s, w2s=w2s, bias = false, dropout = [0 0],
                    transfer="tanh")
    #for a given input, calculates hidden and output activities
    if transfer == "tanh"
        vs = tanh.(us * w1s) #calculate hidden layer values with tanh transfer
    elseif transfer == "sigmoid"
        vs = 1 ./ (1 .+ exp.( -us * w1s )) #sigmoidal
    else println("transfer function not recognized")
    end
```

```
    if bias vs = [vs 1] end
    if dropout[2] > 0 #dropout on layer 2
        n = length(vs)
        inds = randsubseq(1:n, dropout[2])
        vs[1, inds] .= 0
        vs *= 1/(1-dropout[2]) #rescale
    end
    if transfer == "tanh"
        zs = tanh.(vs * w2s) #calculate output values with tanh
    elseif transfer == "sigmoid"
        zs = 1 ./ (1 .+ exp.( -vs * w2s )) #sigmoidal
    end
    return vs, zs
end

function backpropagate(us, vs, zs, ts; eta = 0.1, w1s=w1s, w2s=w2s, N=N,
                       transfer="tanh")
    #update weights for second layer
    Nu = length(us)
    Nuout = size(w1s)[2] #different from Nv if bias units
    Nv = length(vs)
    Nz = length(zs)
    deltaks = zeros(Nz) #store deltas for last layer
    dw2s = zeros(Nv, Nz)
    for k = 1:Nz #for each output unit
        if transfer == "tanh"
            delta = (1 - zs[k]^2) * (ts[k]-zs[k]) #calculate delta tanh
        elseif transfer == "sigmoid"
            delta = (zs[k] - zs[k]^2) * (ts[k]-zs[k]) #sigmoidal
        end
        deltaks[k] = delta
        for j = 1:Nv #for each hidden unit
            dw2s[j, k] = eta * delta * vs[j] #update rule
        end
    end
    w2s += dw2s #modify weights

    #update weights for first layer
    deltajs = zeros(Nuout)
    dw1s = zeros(Nu, Nuout)
    for j = 1:Nuout #for each hidden unit
        if transfer == "tanh"
            delta = (1-vs[j]^2) * sum(deltaks .* w2s[j,:]) #calculate delta tanh
        elseif transfer == "sigmoid"
            delta = (vs[j]-vs[j]^2) * sum(deltaks .* w2s[j,:]) #sigmoidal
        end
        deltajs[j] = delta #store deltas
        for i = 1:Nu #for each input unit
            dw1s[i,j] = eta * delta * us[i] #update rule
        end
    end
    w1s += dw1s #modiy weights
    return w1s, w2s
end

function train_round(;ninput = ninput, train_ts=train_ts, train_us=train_us,
```

```
                w1s=w1s, w2s=w2s, eta=0.1, N=N, bias=false, dropout=[0 0],
                transfer="tanh")
    #trains a single epoch of the neural network using backpropagation
    nus = size(train_us)[2]
    nzs = size(train_ts)[2]
    for n = 1:ninput #for each image
        ts = reshape(train_ts[n,:], 1, nzs) #target, hvec
        us = reshape(train_us[n,:], 1, nus) #input, hvec
        if bias us = [us 1] end
        if dropout[1] > 0 #dropout on layer 1
            n = length(us)
            inds = randsubseq(1:n, dropout[1])
            us[1, inds] .= 0
            us *= 1/(1-dropout[1]) #rescale
        end
        vs, zs = calc_vs_zs(us, w1s=w1s, w2s=w2s, bias=bias, dropout = dropout,
                            transfer=transfer) #get output
        w1s, w2s = backpropagate(us, vs, zs, ts, w1s=w1s, w2s=w2s, eta=eta, N=N,
                            transfer=transfer) #update weights
    end
    return w1s, w2s
end

function test_all(;test_ts = test_ts, test_us = test_us,
                train_ts=train_ts, train_us = train_us,
                ntest=ntest, w1s=w1s, w2s=w2s,
                bias = false, test=true, loss=false, transfer="tanh",
                Plot = false, filename="test")
    #test performance on all test data
    #if loss, calculates a mean square error instead
    #if Plot, plots performance for individual digits
    if test #consider test data
        testinput = test_us
        testoutput = test_ts
    else #consider training data
        testinput = train_us
        testoutput = train_ts
    end
    ts = [coords[2] for coords in argmax(testoutput, dims=2)] #hardmax function
    ntest, nus = size(testinput)
    correct = 0
    cost = 0
    results = zeros(10, 10) #store prediction results
    for i = 1:ntest #for each datapoint
        target = ts[i]
        us = reshape(testinput[i,:], 1, nus) #input data
        if bias us = [us 1] end #add bias unit
        vs, zs = calc_vs_zs(us, w1s=w1s, w2s=w2s, bias=bias, transfer=transfer)
        if loss #calculate loss function
            cost += sum((zs.*(1.08/sum(zs)) .- testoutput[i,:]).^2)
        end
        prediction = argmax(zs[:]) #use max function for prediction
        results[target, prediction] += 1
        if prediction == target correct += 1 end #count correct predictions
    end
    if loss return log10(cost) end
    frac_corr = correct / ntest #proportion of correct predictions
```

```
    if Plot
        #plot prediction results
        results ./= sum(results, dims=2)
        println(results)
        println("performance: ", frac_corr)
        #plot heatmap of cross-prediction
        heatmap(results, 0:9, 0:9; xlab="predicted", ylab="target",
            filename="figures/"*filename*"_predictions.png", Title="",
            vmin=0, vmax=0.05)

        figure(figsize=(5,4)) #plot barplot of performance
        bar(0:9, [results[i,i] for i =1:10])
        xticks(0:9, [string(lab) for lab in 0:9])
        xlabel("target")
        ylabel("performance")
        hlines([frac_corr],-2,12, "k","--")
        xlim(-1,10)
        savefig("figures/"*filename*"_predictionhist.png", dpi=120)
        close()
    end
    return frac_corr #return prediction performance
end

function train(; nepoch = 100, fname = "test", w1s=w1s, w2s=w2s, eta=0.1, N=N,
                 Plot=true, bias=false, return_weights = false, dropout=[0 0],
                 transfer="tanh", train_ts=train_ts, train_us=train_us)
    #trains the network for nepoch epochs and plots a graph of performance'''
    println("training a network with "*string(N)*
            " hidden units and learning rate "*string(eta)*
            " for "*string(nepoch)*" epochs. bias "*string(bias)*
            ", transfer "*transfer)
    n = 0 #number of epochs
    performance = zeros(nepoch+1) #store test performance
    frac_corr0 = test_all(w1s=w1s, w2s=w2s, bias=bias, transfer=transfer)
    println("now at n="*string(n)*
            " performance="*string(round(frac_corr0, digits=2)))
    performance[1] = frac_corr0

    performance_train = zeros(nepoch+1) #also calculate training performance
    performance_train[1] = test_all(w1s=w1s, w2s=w2s, bias=bias, test=false,
                                    transfer=transfer)

    while n < nepoch #for each epoch
        n += 1
        oldw1s = copy(w1s)
        oldw2s = copy(w2s)
        t = time() #also calculate time per epoch
        w1s, w2s = train_round(w1s=w1s, w2s=w2s, eta=eta, N=N, bias=bias,
                               dropout=dropout,
                               transfer=transfer) #train a single epoch
        t = time() - t
        err = norm(w1s-oldw1s)+norm(w2s-oldw2s) #how much did weights change?
        #quantify how good we are
        frac_corr = test_all(w1s=w1s, w2s=w2s, bias=bias, loss=false,
                             transfer=transfer)
        frac_train = test_all(w1s=w1s, w2s=w2s, bias=bias, test=false, loss=false,
```

```
                                transfer=transfer)
        println("now at n="*string(n)*" test="*string(round(frac_corr, digits=3))*
                "  train="*string(round(frac_train, digits=3))*
                "  time="*string(round(t, digits=2)))
        performance[n+1] = frac_corr #store results
        performance_train[n+1] = frac_train
    end

    if Plot
        #plot test and training performance over epochs
        mypickle(performance, "pickled_data/"*fname*"_performance.pickled")
        figure(figsize = (5,3))
        plot(0:nepoch, performance)
        plot(0:nepoch, performance_train, "0.5")
        xlabel("epoch")
        ylabel("performance")
        xlim(0, nepoch)
        if size(train_us)[2] == 10 ylim(0.45, 1)
        else ylim(0,1) end
        legend(["test", "train"])
        savefig("figures/"*fname*"_performance.png", dpi=120, bbox_inches="tight")
        close()
    end
    if return_weights return performance, w1s, w2s end
    return performance
end

function repeat_performance(;N = N, ntrial = 10, nepoch = 100, fname = "test",
                    w1s=w1s, w2s=w2s, eta=0.1, Plot = true, bias=false,
                    train_us=train_us, transfer="tanh", dropout = [0 0])
    #train ntrial nets and quantify performance
    performances = zeros(ntrial, nepoch+1)
    for i = 1:ntrial #for each trial
        println("\n\nNew trial: "*string(i))
        ninput, nus = size(train_us)
        nzs = size(train_ts)[2]
        d = Normal(0, 0.01)
        if bias
            w1s = rand(d, nus+1, N) #random initial weights
            w2s = rand(d, N+1, nzs) #random initial weights
        else
            w1s = rand(d, nus, N) #random initial weights
            w2s = rand(d, N, nzs) #random initial weights
        end
        #find learning curve
        performance = train(; eta=eta, nepoch = nepoch, fname = "test",
                            w1s=w1s, w2s=w2s, N=N, Plot=false, bias=bias,
                            transfer=transfer, dropout = dropout)
        performances[i, :] = performance #store performance
    end
    performance = mean(performances, dims = 1)[:] #report mean performance
    return performance
end

function repeat_train(;N = N, ntrial = 10, nepoch = 100, fname = "test",
                    w1s=w1s, w2s=w2s, eta=0.1, Plot = true, bias=false,
                    train_us = train_us, train_ts = train_ts, transfer="tanh",
```

```
                    dropout = [0  0])
    #repeat training and return only maximum performance
    performances = zeros(ntrial, nepoch+1)
    for i = 1:ntrial #for each trial
        println("\n\nNew trial: "*string(i))
        ninput, nus = size(train_us)
        nzs = size(train_ts)[2]
        d = Normal(0, 0.01)
        if bias
            w1s = rand(d, nus+1, N) #random initial weights
            w2s = rand(d, N+1, nzs) #random initial weights
        else
            w1s = rand(d, nus, N) #random initial weights
            w2s = rand(d, N, nzs) #random initial weights
        end
        #calculate learning curve
        performance = train(; eta=eta, nepoch = nepoch, fname = "test",
                            w1s=w1s, w2s=w2s, N=N, Plot=false, bias=bias,
                            transfer=transfer, dropout = dropout)
        performances[i, :] = performance
    end
    performance = mean(performances, dims = 1)[:] #find mean

    if Plot #plot result
        figure(figsize = (4,2))
        #figure(figsize = (5,3))
        plot(0:nepoch, performance)
        xlabel("epoch")
        ylabel("performance")
        xlim(0, nepoch)
        if size(train_us)[2] == 10 ylim(0.45, 1)
        else ylim(0,1) end
        savefig("figures/"*fname*"_performance.png", dpi=120, bbox_inches="tight")
        close()
    end

    maxperf, maxepoch = findmax(performance) #find maximum performance
    maxepoch -= 1 #discount zero epoch performance
    return maxperf, maxepoch #return maximum performance and when it occurred
end

function plot_image(us; fname="test")
    #plot our input as a greyscale image
    image = Array(transpose(reshape(us, 28, 28)))
    heatmap(image, [], []; xlab="", ylab="",
    filename="figures/"*fname*".png", Title="")
end

function plot_weights(ws; fname="test")
    #plot weights to our hidden layer
    n1, n2 = 2, 5
    fig, axes = subplots(n1, n2, figsize=(n2,n1))

    for i = 1:(n1*n2)#size(ws)[2]
        println(i)
        image = Array(transpose(reshape(ws[1:784,i], 28, 28)))
        ax = axes[Int(floor((i-1)/n2)+1), (i-1)%n2+1 ]
```

```
        ax[:imshow](image, cmap=ColorMap("gray_r"), vmin = minimum(image),
                vmax = maximum(image))
        ax[:set_xticks]([], [])
        ax[:set_yticks]([], [])
    end
    savefig("figures/"*fname*"_weightfig.png", dpi=480, bbox_inches="tight")
    close()
end

data = unpickle("MNIST/data.pickled") #unpack our training and test data
train_us, train_ts, test_us, test_ts = data[1], data[5], data[2], data[6]

parity = false #if parity, work on the parity task!
if parity
    n1 = 400
    n2 = 1000
    train_us = rand([1, -1], (n1, 10)) #random training data
    train_ts =  zeros(n1, 2) #find real answers
    ts = prod(train_us, dims=2)
    train_ts[ts[:,1].==1, 1] .= 1
    train_ts[ts[:,1].==-1, 2] .= 1

    test_us = rand([1,-1], (n2, 10)) #random test data
    test_ts =  zeros(n2, 2) #find real data
    ts = prod(test_us, dims=2)
    test_ts[ts[:,1].==1, 1] .= 1
    test_ts[ts[:,1].==-1, 2] .= 1
end

subsample = true #subsample our training data for overfitting and dropout
if subsample
    inds = rand(50000:60000, 1000)
    test_us = train_us[inds, :] #random test data
    test_ts = train_ts[inds, :]
    inds = rand(1:50000, 200)
    train_us = train_us[inds, :] #random training data
    train_ts = train_ts[inds, :]
end

shift = false #if shift, shift our data to zero mean
if shift
    train_us = train_us .+ 0.625 #.- mean(train_us, dims=2)
    test_us = test_us .+ 0.625 #.- mean(test_us, dims=2)
end

ninput, nus = size(train_us) #number of inputs and input units
ntest = size(test_us)[1] #store number of test datapoints
nzs = size(train_ts)[2]

N = 10 #number of hidden units
eta = 0.01 #learning rate
nepoch = 20 #number of epochs to train for
bias = true #do we include bias units?
transfer = "tanh" #specify transfer function (tanh or sigmoid)

d = Normal(0, 0.01) #start with random weights in linear regime
if bias
```

```
    w1s = rand(d, nus+1, N) #random initial weights
    w2s = rand(d, N+1, nzs) #random initial weights
else
    w1s = rand(d, nus, N) #random initial weights
    w2s = rand(d, N, nzs) #random initial weights
end

fname="test_N"*string(N)*
        "_eta"*string(eta)[3:end]*
        "_nepoch"*string(nepoch)*
        "_transfer_"*transfer
if parity fname = fname*"_parity" end

#run single trial and plot performance
performance, w1s, w2s = train(eta=eta, nepoch=nepoch, fname=fname, bias=bias,
                        return_weights = true, transfer=transfer)

writedlm("w1s_test_tanh.dlm", w1s) #store out weights!
writedlm("w2s_test_tanh.dlm", w2s)


ntrial = 10 #run ntrial trials and plot average performance
fname = "test_N"*string(N)*
        "_eta"*string(eta)[3:end]*
        "_nepoch"*string(nepoch)*
        "_ntrial"*string(ntrial)*
        "_transfer_"*transfer
if parity fname = fname*"_parity" end
repeat_train(eta=eta, nepoch=nepoch, N=N, ntrial=ntrial,
            fname=fname, bias=bias, dropout=[0.0 0.0], transfer="tanh")

#read weights from N300 network
w1s = readdlm("w1s_N300_tanh.dlm")
w2s = readdlm("w2s_N300_tanh.dlm")
test_all(bias=true, Plot=true) #test predictions and plot

#read weights for 10-neuron network and plot weoghts
w1s = readdlm("w1s_N10_tanh.dlm")
w2s = readdlm("w2s_N10_tanh.dlm")
plot_weights(w1s, fname="w1_N")

 w3s = w1s[1:784,:]*w2s[1:size(w1s)[2],:]
 plot_weights(w3s, fname="w3s_N3") #plot pseudoweights
```

```
#code for exploring the effect of various parameters on performance
include("mlp.jl") #import mlp functions

function heatmap(results, xs, ys; xlab="", ylab="",
    filename="default", Title="default")
    #given a matrix of z values and lists of x,y values
    #plots a heatmap
    figure()
    imshow(results, cmap=ColorMap("gray_r"), vmin = minimum(results),
           vmax = maximum(results))
    print("plotted")
    xticks(0:(length(xs)-1), xs, rotation=0)
    yticks(0:(length(ys)-1), ys)
    colorbar()
    xlabel(xlab)
    ylabel(ylab)
    title(Title)
    savefig(filename, bbox_inches = "tight")
    close()
end

function test_params(param1, vals1, param2, vals2; ntrial=5, bias=false,
                     fname="test", transfer="tanh", dropout = [0 0])
    #given the names of two parameters (param1, param2) and values,
    #performs a grid search and plots a heatmap of performance

    n1, n2 = length(vals1), length(vals2)
    performances = zeros(n1, n2) #store maxiumum performances
    epochs = zeros(n1, n2) #store epoch number for max performance
    for (i, val1) in enumerate(vals1)
        for (j, val2) in enumerate(vals2)
            #for each combination of parameters
            if (param1, param2) == ("N", "eta")
                performance, epoch = repeat_train(N=val1, eta=val2, nepoch=nepoch,
                                                  ntrial=ntrial, Plot=false,
                                                  transfer=transfer)
            end
            performances[i,j] = performance #sore results
            epochs[i,j] = epoch
            #print result
            println(param1,": ", val1, "  ",param2,": ",val2,
                    " performance: ", performance, " epoch: ", epoch )
        end
    end
    #plot results
    heatmap(performances, vals2, vals1; xlab=param2, ylab=param1,
        filename="figures/"*fname*"_perf.png", Title=param1*" "*param2) #plot
    heatmap(epochs, vals2, vals1; xlab=param2, ylab=param1,
        filename="figures/"*fname*"_epochs.png", Title=param1*" "*param2) #plot
end

function compare_learning_curves(param, vals;ntrial=5, nepoch=25, fname="test",
                                 bias=true, N=50, eta=0.05, transfer="tanh",
                                 dropout = [0 0])
    #For a given parameter param, try all values in vals and plot
    #learning curves. All other parameters are as given in the function
    #specification
```

```
    figure(figsize = (5,3))
    for (i, val) = enumerate(vals) #for each parameter
        println("new val:", val)
        #specify new parameter
        if param == "N"
            N = val
        elseif param == "eta"
            eta = val
        elseif param == "bias"
            bias = val
        elseif param=="transfer"
            transfer = val
        elseif param=="dropout"
            dropout = val
        end
        #run simulation
        performance = repeat_performance(N=N, eta=eta, nepoch=nepoch,
                                         dropout=dropout, ntrial=ntrial,
                                         Plot=false, bias=bias, transfer=transfer)
        #plot result
        plot(0:nepoch, performance, color=string(1-i/length(vals)))

    end
    xlabel("epoch")
    ylabel("performance")
    legend([string(val) for val in vals])
    savefig("figures/"*fname*".png", dpi=120, bbox_inches="tight")
    close()
end


#Perform grid seach
#test_params("N", [10;20;30;40;50;60;70;80;90;100],
#            "eta", [0.001;0.002;0.004;0.008;0.016;0.032;0.064;0.128;0.256;0.512],
#            fname="test_N_eta")

#test effect of learning rate
#compare_learning_curves("eta", [0.00001; 0.0001; 0.001; 0.01; 0.1; 1.0],
#            fname="testeta", N=50, nepoch=15, bias=false, ntrial=3)

#test effect of hidden layer size
#compare_learning_curves("N", [5; 15; 50; 150; 300; 500], fname="testN",
#            eta=0.01, nepoch=15, bias=false, ntrial=3)

#test effect of bias (change parameters as appropriate)
#compare_learning_curves("bias", [true; false], fname="test_bias_shift625",
#                eta=0.01, ntrial=30, nepoch=15, N=50, transfer="tanh")

#test effect of bias on parity problem (need to specify in mlp.jl)
#compare_learning_curves("bias", [true; false], fname="test_bias_parity",
#                eta=0.075, ntrial=10, nepoch=300, N=50, transfer="tanh")

#compare transfer functions
#compare_learning_curves("transfer", ["tanh"; "sigmoid"], fname="test_transfer",
#                ntrial=5, nepoch=10, N=50)
```

```
#test effect of dropout on downsampled data
#compare_learning_curves("dropout", [[0.1 0.4], [0 0]], transfer="sigmoid",
#                        fname="test_dropout", ntrial=10, nepoch=300, N=50, eta=0.075)
compare_learning_curves("dropout", [[0 0], [0.1 0.0], [0.1 0.2], [0.1 0.4],
                                    [0.1 0.6], [0.1 0.8]], transfer="sigmoid",
                                    fname="test_dropout_many", ntrial=3,
                                    nepoch=500, N=1000, eta=0.02)
```

```
:( """
Code for implementing a single binary perceptron with hebbian, delta
and perceptron learning rules on the sum and parity problems.
"""

import numpy as np
import matplotlib.pyplot as plt
import copy

class perceptron:
    '''class for implementing a single binary perceptron'''

    def __init__(self):

        self.ws = np.zeros(N) #weights
        self.gamma = 0 #threshold
        return


    def train(self, N = 100, method = 'heb', Print = False, task = 'sum', query=False):
        '''train perceptron on N random patterns. method specifies training method
        task specifies sum vs product of vector for classification'''

        train_data = np.random.choice([1,-1], (self.dim,N)) #training patterns
        if task == 'sum': vs = np.sign(np.sum(train_data,0)+0.01) #ground truth
        elif task == 'product': vs = np.sign(np.prod(train_data, 0))
        else:
            print('task not recognized')
            return

        self.data = train_data #store training data
        self.truth = vs #store ground truth

        if method == 'heb':
            self.train_heb()
        elif method == 'perceptron':
            if query: performance = self.train_perceptron(query=True)
            else: self.train_perceptron(query=False) #return per-iteration perf
        else:
            if query: performance = self.train_delta(query=True)
            else: self.train_perceptron(query=query)

        if Print: print(self.ws)
        if query: return performance
        return

    def train_heb(self):
        '''simple equation from TN'''
        self.ws = np.sum(self.data * self.truth, 1)/len(p.truth)
        return


    def train_perceptron_round(self, epsilon, thresh = 10**(-6), Print=False):
        '''train a single epoch with perceptron learning'''
        oldws = copy.copy(self.ws)
        for i in range(self.data.shape[1]): #iterate through inputs
            u = self.data[:,i] #input vector
```

```
            #update weights
            self.ws += epsilon/2*(
                        self.truth[i] -
                        np.sign(np.dot(self.ws,u)-self.gamma)) * u
            #update gamma
            self.gamma -= epsilon/2*(
                        self.truth[i] -
                        np.sign(np.dot(self.ws,u)-self.gamma))
        err = np.sum(np.abs(self.ws-oldws)) #test for convergence
        if Print: print('err:', err)
        if err < thresh: #converged
            return False
        else: return True #should still train

    def train_perceptron(self, epsilon = 0.01, nlim=1000, query=False):
        '''train perceptron using the perceptron learning rule.
        epsilon is learning rate'''
        self.ws = np.zeros(self.dim) #initialize at zero
        train = True
        n = 0
        queries = [self.queries(1000)] #store performance throughout training
        while train and n < nlim:
            #train until convergence
            n += 1
            train = self.train_perceptron_round(epsilon)
            if query:
                queries.append(self.queries(1000))
        if query: return queries
        return

    def train_delta_round(self, epsilon, thresh = 10**(-6), Print=False):
        '''train a single epoch with delta learning'''
        oldws = copy.copy(self.ws)
        delta = np.zeros(len(oldws))
        dgamma = 0
        for i in range(self.data.shape[1]):
            u = self.data[:,i]
            delta += (self.truth[i] -
                        np.sign(np.dot(self.ws,u)-self.gamma)) * u
            dgamma += (self.truth[i] -
                        np.sign(np.dot(self.ws,u)-self.gamma))

        self.ws += epsilon*delta #batch learning from all inputs!
        self.gamma -= epsilon*dgamma #update gamma
        err = np.sum(np.abs(self.ws-oldws))
        if Print: print('err:', err)
        if err < thresh: #converged
            return False
        else: return True #should still train

    def train_delta(self, epsilon = 0.01, nlim = 1000, query=False):
        '''train perceptron using the delta learning rule.
        epsilon is learning rate'''
        self.ws = np.zeros(self.dim) #initialize at 0
        train = True
        n = 0
        queries = [self.queries(1000)] #store performance
```

```
    while train and n < nlim: #train until convergence
        n += 1
        train = self.train_delta_round(epsilon)
        if query:
            queries.append(self.queries(1000))
    if query: return queries
    return


def query(self, u):
    '''query perceptron with a single input and update state'''
    v = np.sign(np.dot(self.ws, u)-self.gamma)
    self.state = v
    return v

def queries(self, nquery, plusses='random'):
    '''query perceptron with nquery queries and return performance
    plusses specifies how many input components should be +1'''
    if plusses == 'random': queries = np.random.choice([-1, 1], (10,nquery))
    else:
        queries = np.zeros((10, nquery))
        pool = [1 for i in range(plusses)]+[-1 for i in range(10-plusses)]
        for i in range(nquery):
            #put elements in random order
            queries[:,i] = np.random.choice(pool, 10, replace=False)

    truth = np.sign(np.sum(queries, 0)+0.001) #target
    vs = np.sign(np.dot(self.ws, queries)-self.gamma) #predicted
    performance = sum(np.array(vs) == np.array(truth))/nquery

    return performance


def test_performance(self, method = 'heb', Ns = np.arange(1,400, 5), task = 'sum',
                     ntrial = 20, nquery=1000, Plot=True, plusses='random'):
    '''test mean performance for a given method of training
    plusses is nubmer of +1s in input'''
    performances = []
    for N in Ns: #try different numbers of training vectors
        performance = []
        for j in range(10): #10 separate trials
            self.train(N=N, method = method, task = task)
            for i in range(ntrial): #this is silly
                performance.append(self.queries(nquery, plusses=plusses))
        performance = np.mean(performance)
        performances.append(performance)

    if Plot: #plot result
        plt.figure()
        plt.plot(Ns, performances)
        plt.xlabel('N')
        plt.ylabel('performance')
        plt.ylim(0.4,1)
        plt.show()

    return Ns, performances
```

```
def test_by_plus1(self, method = 'heb', fname='figures/test.png', task = 'sum'):
    '''test performance for different input compositions'''
    plt.figure()
    nmax = 6
    for nplus in range(nmax+1): #number of +1 vs -1
        print('new nplus:', nplus)
        #train network and test performance
        Ns, performances = self.test_performance(method = method, Plot=False,
                                                 plusses=nplus, task=task)
        #plot result
        plt.plot(Ns, performances, color = str(1-(nplus+1)/(nmax+1)), linestyle='-')
        print(sum(np.array(performances) < 0.95))
    plt.legend([str(val) for val in range(nmax+1)])

    plt.xlabel('N')
    plt.ylabel('performance')
    plt.ylim([0,1])
    plt.savefig(fname)
    plt.show()

def compare(self, fname='figures/compare.png', task = 'sum'):
    '''run and plot comparison of the three learning rules'''
    plt.figure()
    cols = ['b', 'g', 'r']
    for i, method in enumerate(['heb', 'perceptron', 'delta']):
        print('new method:', method)
        #test performance for each method
        Ns, performances = self.test_performance(method = method, Plot=False,
                                                 task = task)
        #plot result
        plt.plot(Ns, performances, color = cols[i], linestyle='-')
        print(sum(np.array(performances) < 0.95))
    plt.ylim([0,1])
    plt.legend(['hebbian', 'perceptron', 'delta'])
    plt.xlabel('N')
    plt.ylabel('performance')
    plt.savefig(fname)
    plt.show()

def compare_learnrate(self, fname='figures/compare_learnrate2.png'):
    '''compare rate of learning for perceptron and delta rules'''
    plt.figure()
    cols = ['g', 'r']
    for i, method in enumerate(['perceptron', 'delta']):
        print('new method:', method)
        allperfs = np.zeros((100, 1001))
        for j in range(100): #average over 100 trials
            performances = np.zeros(1001)
            perf = np.array(self.train(method = method, query=True, N=400))
            n = len(perf)
            performances[:n] = perf
            performances[n:] = perf[-1]
            allperfs[j, :] = performances
        performances = np.mean(allperfs, 0)
        #only plot first 10 epochs
        plt.plot(range(11), performances[:11], color = cols[i], linestyle='-')
        print(sum(np.array(performances) < 0.95))
```

```
        plt.ylim([0,1])
        plt.legend(['perceptron', 'delta'])
        plt.xlabel('epoch')
        plt.ylabel('performance')
        plt.savefig(fname)
        plt.show()

    def test_gamma(self, fname='figures/testgamma.png'):
        '''test effect of gamma on Hebbian learning performance'''
        perfs = []
        gammas = np.arange(0, -1.5, -0.05) #try different values
        for gamma in gammas:
            newperfs = []
            for i in range(100): #average over 100 trials
                self.train(N = 400, method = 'heb', Print = False, task = 'sum', query=False)
                self.gamma=gamma
                newperfs.append(self.queries(1000, plusses='random'))
            perfs.append(np.mean(newperfs))
        #plot figure
        plt.figure()
        plt.plot(gammas, perfs)
        plt.ylim([0.5,1])
        plt.xlabel('gamma')
        plt.ylabel('performance')
        plt.savefig(fname)
        plt.show()
        print(gammas)
        print(perfs)

def plot_sum_prod():
    '''generates 2d figures of the linear (in)separability of the sum
    and parity tasks'''

    #sum task
    coords = [[1,1], [-1,-1], [1,-1], [-1,1]]
    plt.figure(figsize = (3,3))
    for coord in coords:
        if sum(coord) > 0:
            plt.plot(coord[0], coord[1], marker = 'o', color='0.85')
        elif sum(coord) < 0:
            plt.plot(coord[0], coord[1], marker = 'o', color = '0')
        else:
            plt.plot(coord[0], coord[1], marker = 'o', color='0.6')
    plt.xticks([-1,1], [-1,1])
    plt.yticks([-1,1], [-1,1])
    plt.xlim(-1.5,1.5)
    plt.ylim(-1.5,1.5)
    plt.savefig('figures/sum_2d.png', dpi = 120)
    plt.show()

    #parity task
    plt.figure(figsize = (3,3))
    for coord in coords:
        if coord[0]*coord[1] > 0:
            plt.plot(coord[0], coord[1], marker = 'o', color='0.85')
        elif coord[0]*coord[1] < 0:
            plt.plot(coord[0], coord[1], marker = 'o', color = '0')
```

```
        else:
            plt.plot(coord[0], coord[1], marker = 'o' , color='0.6')
    plt.xticks([-1,1], [-1,1])
    plt.yticks([-1,1], [-1,1])
    plt.xlim(-1.5,1.5)
    plt.ylim(-1.5,1.5)
    plt.savefig('figures/prod_2d.png', dpi = 120)
    plt.show()


plot_sum_prod() #generate 2d separability figs

p = perceptron()
p.test_gamma() #test effect of gamma
p.compare_learnrate() #investigate learning rates

#compare performances oof learning rules in the two tasks
p.compare(fname='figures/compare_sum.png')
p.compare(fname='figures/compare_prod.png', task='product')

#quantify performance by input vector composition for all three learning rules
p.test_by_plus1(method='delta', fname = 'figures/nplus_delta.png')
p.test_by_plus1(method='perceptron', fname = 'figures/nplus_perceptron.png')
p.test_by_plus1(method='heb', fname = 'figures/nplus_heb.png')
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Code for processing and pickling MNIST data to avoid having to
read parge CSV files every time we do anything
"""

import numpy as np
import pickle

image_size = 28 # width and length
labels = 10 #  number of different labels
image_pixels = image_size ** 2
path = "MNIST/"
train_data = np.loadtxt(path + "mnist_train.csv", delimiter=",")
test_data = np.loadtxt(path + "mnist_test.csv", delimiter=",")

train_imgs = np.asfarray(train_data[:, 1:]) / (255*0.99 + 0.01)
test_imgs = np.asfarray(test_data[:, 1:]) / (255*0.99 + 0.01)
train_labels = np.asfarray(train_data[:, :1])
test_labels = np.asfarray(test_data[:, :1])

lr = np.arange(labels)
# transform labels into one hot representation
train_labels_one_hot = (lr==train_labels).astype(np.float)
test_labels_one_hot = (lr==test_labels).astype(np.float)
# we don't want zeroes and ones in the labels neither:
train_labels_one_hot[train_labels_one_hot==0] = 0.01
train_labels_one_hot[train_labels_one_hot==1] = 0.99
test_labels_one_hot[test_labels_one_hot==0] = 0.01
test_labels_one_hot[test_labels_one_hot==1] = 0.99

with open("MNIST/data.pickled", "wb") as f:
    data = (train_imgs,
            test_imgs,
            train_labels,
            test_labels,
            train_labels_one_hot,
            test_labels_one_hot)
    pickle.dump(data, f)
```