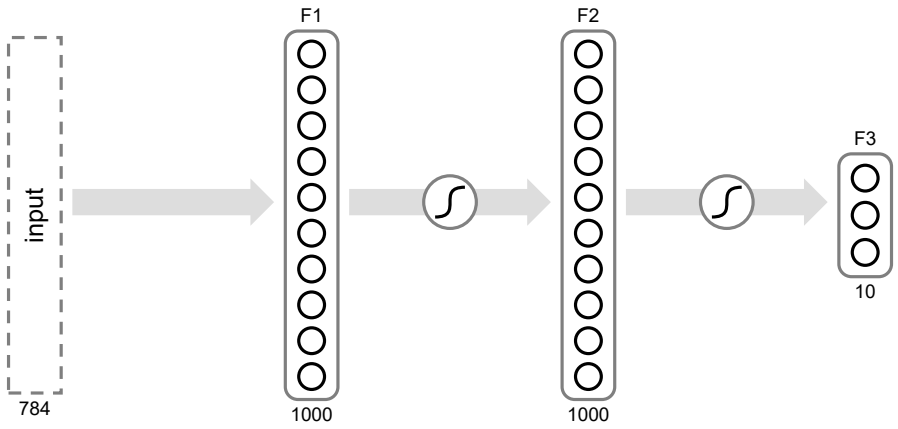
University of Cambridge
Engineering Part IB
Information Engineering Elective

Paper 8: Image Searching and Modelling Using Machine Learning

Handout 2: Multi-Layer Perceptrons



Roberto Cipolla and Matthew Johnson
May 2017

Notebook

You can access an interactive version of this handout at the following URL:

`https://aka.ms/ucepibieep8`

Combining Perceptrons

In the last lecture, we saw that a single perceptron can be a powerful tool for classifying between two classes of input. If we want to move beyond simple two-class problems to distinguish between multiple classes, we can combine perceptrons using what is called a *1 versus all* approach. Each perceptron's weights $\mathbf{w}$ become a row in a matrix $\mathbf{W}$:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x})$$

where $\mathbf{x}$ is the input vector and f is the same step function, applied on a per-index basis on the output vector:

$$f(x_i) = \begin{cases} +1 & x_i \geq 0 \\ -1 & x_i < 0 \end{cases}$$

Each perceptron focuses on a single binary problem: is this input a member of my class, or not? We can then look at multiple outputs and choose the class that corresponds to the classifier that has a positive result. The math stays much the same, though we are going to change the formulation slightly. Instead of including the bias as an extra dimension in the data, we are going to use a common convention in modern neural net architectures and separate it out as a separate *bias* term, $\mathbf{b}$, as follows:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Combining Perceptrons, cont.

Let us look at a sample scenario to see what happens in this new approach:

$$\mathbf{X} = \begin{bmatrix} 0.3 & -0.4 \\ -0.5 & 1 \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} 0.55 & -0.12 \\ -0.32 & 0.5 \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} 0.4 & -0.2 \end{bmatrix}$$

$$\mathbf{WX} + \mathbf{b} = \begin{bmatrix} 0.625 & -0.546 \\ 0.06 & 0.428 \end{bmatrix}$$

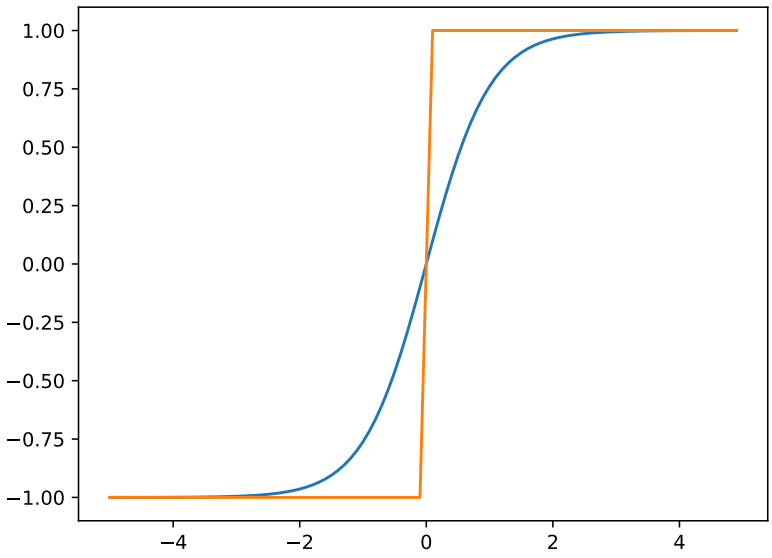
$$\mathbf{Y} = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$$

We can immediately see the problem here: what do we do when two or more perceptrons both give a positive result for the same input? We need a new function that will still be non-linear, but in a "softer" way.

Activation Functions

The step function used by the perceptron is just one member of a class of functions called *activation functions*. There are many others. One that is a close approximation to the step function is the hyperbolic tangent:

$$f(x_i) = \tanh(x_i)$$



Unlike the step function, the hyperbolic tangent is fully differentiable, which will come in helpful later. Just as a reminder, that looks like this:

$$f'(x_i) = 1 - f^2(x_i)$$

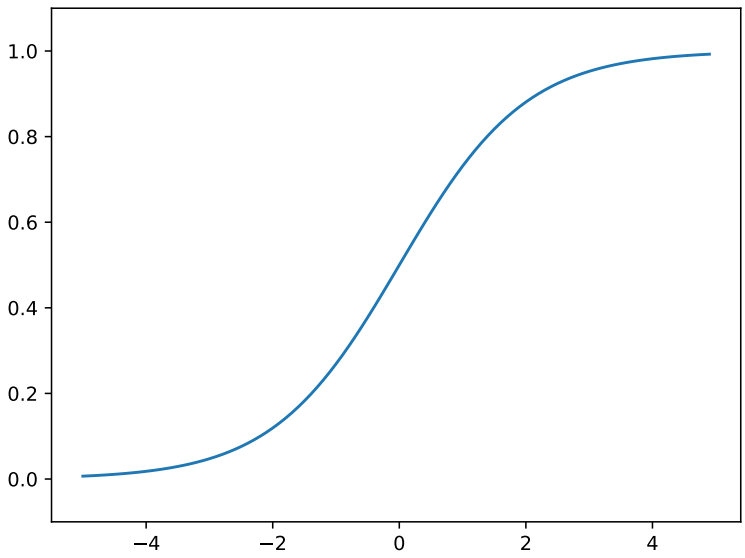
In addition to being roughly linear from -1 to 1, tanh saturates to 1 or -1 as well, thus providing a useful *non-linearity*, which was the driving force behind the use of the step function in the original perceptron.

Activation Functions, cont.

Another commonly used objective function is the *sigmoid* function, which is a special case of the logistic function:

$$f(x) = \frac{1}{1 + e^{-x_i}}$$
$$f'(x) = f(x_i)\,(1 - f(x_i))$$

This function is similar in shape to the hyperbolic tangent, but ranges from zero to one:



Both of these functions have similar characteristics: they are roughly linear from -1 to 1, they saturate (though in the case of the sigmoid, it saturates at 0 and 1). They also both of convenient derivatives. In these lectures, we will use the hyperbolic tangent, but they are both widely used activation functions in neural networks.

Multiperceptron

We now have a new formulation:

$$\mathbf{y} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

This new formulation, which we will call a *multi-perceptron*, has several advantages. In particular, it is fully differentiable, which means we can learn the values of $\mathbf{W}$ and $\mathbf{b}$ via *gradient descent.*

The use of gradient descent to discover the parameters of a network of perceptrons is called *backpropagation*, and was discovered by Paul J. Werbos, who described it in his 1974 thesis [2]. In order to use gradient descent to determine the optimal values for the network we need to define an objective function as a function of the network. The current task we are trying to accomplish is called *classification*, where we try to take an input and classify it as belong to one of several classes. A typical way of performing classification is by generating a probability distribution $P(\mathbf{x})$, which defines a distribution over N classes such that:

$$\sum_{i \in N} P(i) = 1$$

and where the magnitude of $P(\mathbf{x} = i),\ i \in N$ corresponds to the classifier's confidence that $\mathbf{x}$ belongs to class i. A typical choice for a neural network is the Softmax function:

$$P(\mathbf{x} = i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Cross Entropy

The cross entropy is an objective function for neural networks that perform classification. It consists of two components:

$$H(P, Q) = H(P) + D_{\mathrm{KL}}(P||Q)$$

for discrete distributions P and Q, where H is the Shannon entropy:

$$H(P) = -\sum_i P(i) \log P(i)$$

Entropy measures the *information content* of a distribution. This can be thought of as the expectation of the information content given by sampling from the distribution. A perfectly uniform distribution, with equal probability for all events, has maximum entropy. Think of flipping a coin: if it is fair ($P(Heads) = P(Tails) = 0.5$) then we have no way of predicting what the result will be when it is flipped. However, if the coin were weighted, such that $P(Heads) >> P(Tails)$, then flipping it does not give as much information; we knew it was probably going to be heads.

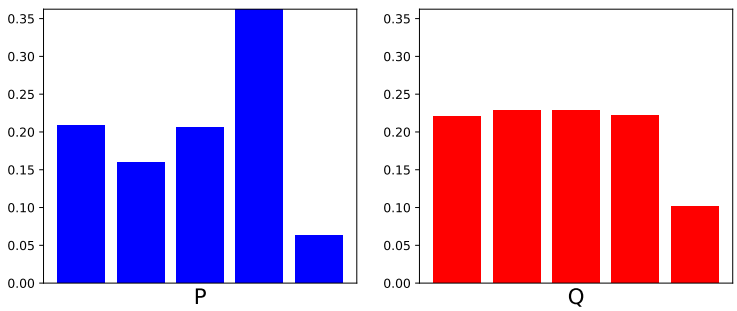
Cross Entropy, cont.

The second term, D_{KL} is the Kullback-Leibler divergence, which is a common measure of how much one distribution diverges from another:

$$D(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

The intuition here is that if P and Q are the same, then the term $\log \frac{P(i)}{Q(i)}$ will be zero and thus the divergence is also zero. We combine them as follows:

$$\begin{aligned} H(P, Q) &= -\sum_i P(i) \log P(i) + \sum_i P(i) \log \frac{P(i)}{Q(i)} \\ &= -\sum_i P(i) \log P(i) + \sum_i P(i) \log P(i) \\ &\quad -\sum_i P(i) \log Q(i) \\ &= -\sum_i P(i) \log Q(i) \end{aligned}$$



In the figures above, the $H(P) = 1.49$, whereas $H(P, Q) = 1.55$. It is always the case that the lower bound of the cross-entropy is $H(P)$.

Loss Function

We can now write the objective function for our entire network, which in this case is the expected classification loss:

$$L(D, \mathbf{W}, \mathbf{b}) = \frac{1}{|D|} \sum_{\mathbf{d} \in D} H(\mathbf{t}, p(\tanh(\mathbf{W}\mathbf{x}_d + \mathbf{b})))$$

where p is the Softmax function described above, D is the dataset, and $\mathbf{t}_i$ is a *one-hot* vector corresponding to the correct label for that data point. In words, we want to minimize the expected cross entropy between the output and the target distribution. Now that we have an objective, we can compute gradients for $\mathbf{W}$ and $\mathbf{b}$ and optimize them directly.

While the above equation looks rather daunting when it comes to computing the partial derivatives for $\mathbf{W}$ and $\mathbf{b}$, you will find that the different functions have been chosen very carefully to make this tractable. First, let's expand the combination of cross entropy and softmax:

$$\begin{aligned} H(\mathbf{t}, \mathbf{p}) &= -\sum_i t_i \log p_i \\ p_i &= \frac{e^{s_i}}{\sum_j e^{s_j}} \\ \mathbf{s} &= \tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) \end{aligned}$$

We want to look at the partial derivatives with respect to s_i. What makes this tricky is that s_i shows up in both the numerator and the denominator, which means we need to index the derivative.

Computing Gradients

The partial derivatives with respect to s_i are:

$$\frac{\partial p_i}{\partial s_i} = p_i(1 - p_i)$$
$$\frac{\partial p_j}{\partial s_i} = -p_i p_j, \; j \neq i$$

Using these derivatives, we can compute:

$$\frac{\partial L}{\partial s_i} = -\sum_k t_k \frac{\partial \log p_k}{\partial s_k} \tag{1}$$
$$= -\sum_k t_k \frac{1}{p_k} \frac{\partial p_k}{\partial s_k} \tag{2}$$
$$= -t_i(1 - p_i) - \sum_{k \neq i} t_k \frac{1}{p_k}(-p_k p_i) \tag{3}$$
$$= -t_i + t_i p_i + \sum_{k \neq i} t_k p_i \tag{4}$$
$$= p_i \left(\sum_k t_k \right) - t_i \tag{5}$$

Remember that $\mathbf{t}$ is a one-hot vector, and so we get the final result:

$$\frac{\partial L}{\partial s_i} = p_i - t_i$$

This may seem surprising, but the functions for the loss and the probability were chosen carefully to end in this way.

Computing Gradients, cont.

The tricky part is over. Now that we have $\frac{\partial L}{\partial s_i}$, we just need the hyperbolic tangent, which as you remember is:

$$s_i = \tanh(h_i) \tag{6}$$

$$\frac{\partial s_i}{\partial h_i} = 1 - s_i^2 \tag{7}$$

At this point, due to the nature of the linear transform, we need to take into account the vector of partial derivatives, which we will denote $\delta = \left\{ \frac{\partial L}{\partial h_i}, \ \forall i \right\}$. We can now compute the interesting derivatives:

$$\frac{\partial L}{\partial \mathbf{W}} = \delta^{\mathrm{T}} \mathbf{x} \tag{8}$$

$$\frac{\partial L}{\partial \mathbf{b}} = \sum_i \delta_i \tag{9}$$

Batching

With perceptrons, we showed a single example to the perceptron at each step. Now that we are using backpropagation, however, we need to do something different. While our goal is to minimize the overall loss, it is often the case that it is either impractical to perform backpropagation for all of the data or even that the gradient obtained from showing everything at once is suboptimal. However, just showing a single example at a time results in the network having trouble make good decisions for all categories.

As a result, the data is typically presented to the network as a (small) batch D_B called a *mini-batch*, which must be selected uniformly at random from D. We need to select the examples at random because otherwise it can cause major issues during training. Imagine if the network was shown only images of zeros in one mini-batch. It would learn something about zeros, but in attempting to adjust it may forget important things it had already learned about twos. In practice, it is better to present a uniform sampling from all categories in the training set, thus allowing the network to learn strategies that are good for all classes, not just a few.

Seeing as we are presenting all examples $\mathbf{x} \in D_B \subset D$, δ is actually defined as:

$$\delta = \left\{ \frac{1}{|D_B|} \frac{\partial L}{\partial h_i}, \ \forall i \right\}$$

Datasets

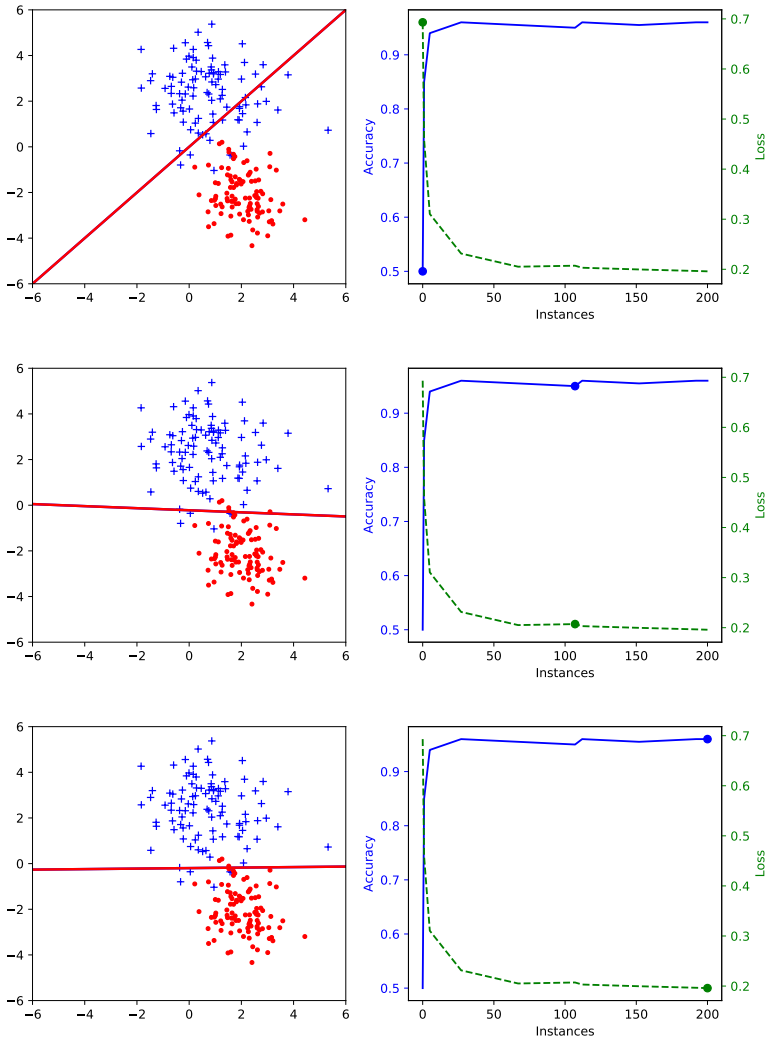
Before we continue, it is important to outline the right and wrong ways of using data. There are three commandments of data science:

1. Thou shalt not train on test data
2. Thou shalt not determine hyperparameters on test data
3. Thou shalt not overfit

These statements stem from the central goal of machine learning: *generalization*. When you train a model, you want to ensure that it can effectively infer information from new, unseen data. This is why you take a portion of your data and set it aside for use as test data. If you train on this data, then you have no way of knowing whether your technique can generalize, and thus your results are meaningless. This is the source of the first commandment. The second commandment is similar: by determining your hyperparameters (like learning rate, number of hidden dimensions) from the test data, you are essentially transmitting information from your test data to your training set *even though* the algorithm does not look at the data directly. For this reason, we set aside a bit of our training data as a *validation* set, which can be used for determining hyperparameters. Finally, there is a temptation to continue training until the results on the training data are as high as possible, which is called *overfitting*. Unless your algorithm is designed carefully to avoid this, you will end up creating a solution that is so specific to your training data that it behaves strangely on the test data.

Training a Multiperceptron

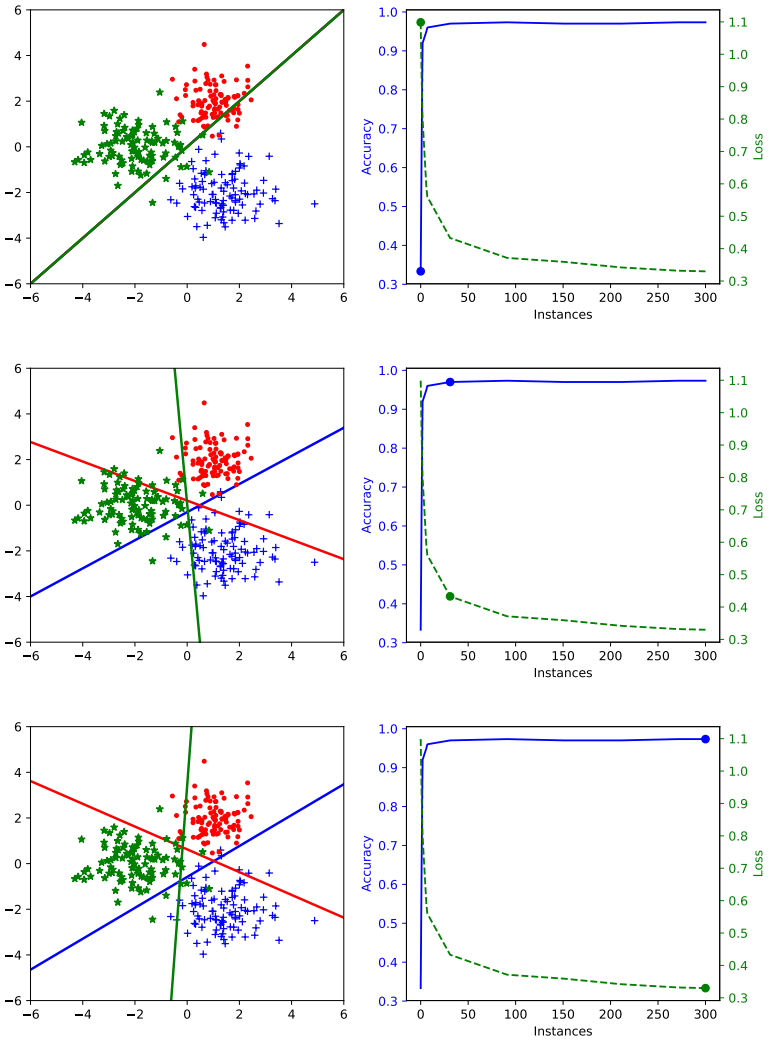
How does this new classifier perform? Below we see three snapshots from a two-perceptron model training on a two-class problem like the one we saw in the last lecture:



We see here both the accuracy and the loss on the same plot. Note how the loss drops as the accuracy increases. This problem is clearly too easy for our new multiperceptron, so we will make it harder.

Three Classes

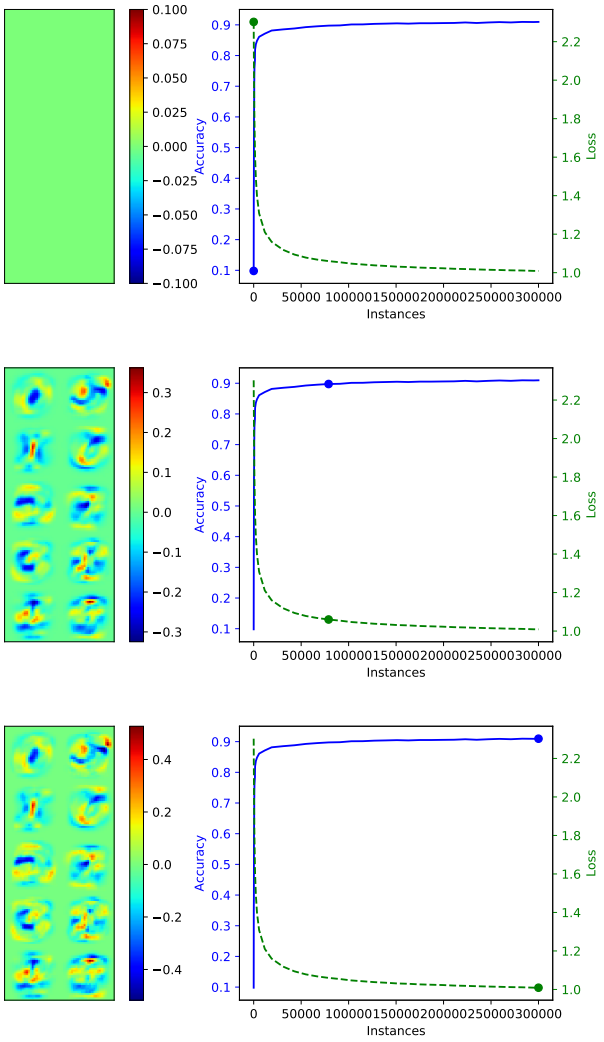
We are now going to do something that was impossible with a normal perceptron: classify inputs into one of three classes. This model has three perceptrons, one per class:



The classifier starts off in much the same way, but very quickly each perceptron (represented by a line in the graph) learns to separate its points from the rest, and is remarkably accurate by the end of training. Time to make things even harder.

MNIST 10-Way Classification

How does our classifier deal with classifying all ten digits of MNIST?



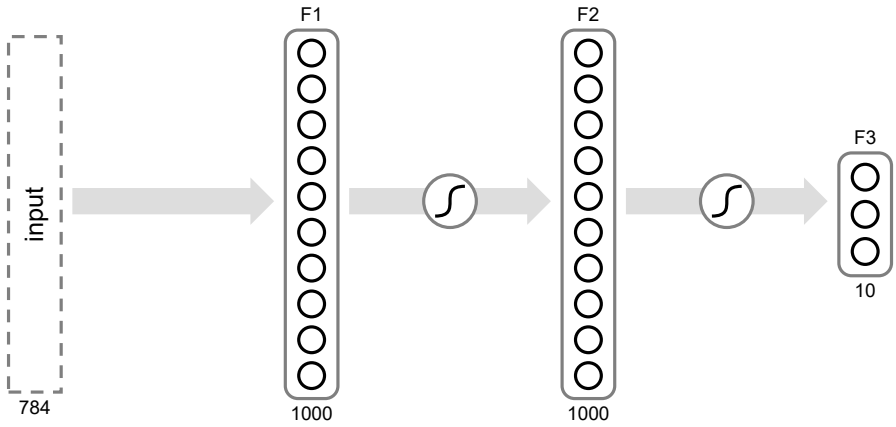
As you can see, each perceptron learns the characteristics of a single digit, as it is only responsible for that single class. However, it seems our network is stalling at a loss of 1 and an accuracy of 0.9. What can we do to make this even better?

Multi-Layer Perceptrons

The same principles that enabled us to train the multiperceptron can be used to combine layers of multiperceptrons together, creating a network known as the *Multi-Layer Perceptron*:

$$\begin{aligned}\mathbf{h}_0 &= \mathbf{W}_0\mathbf{x} + \mathbf{b}_0 \\ \mathbf{s}_0 &= \tanh(\mathbf{h}_0) \\ \mathbf{h}_1 &= \mathbf{W}_1\mathbf{s}_0 + \mathbf{b}_1 \\ \mathbf{s}_1 &= \tanh(\mathbf{h}_1) \\ \mathbf{h}_2 &= \mathbf{W}_2\mathbf{s}_1 + \mathbf{b}_2\end{aligned}$$

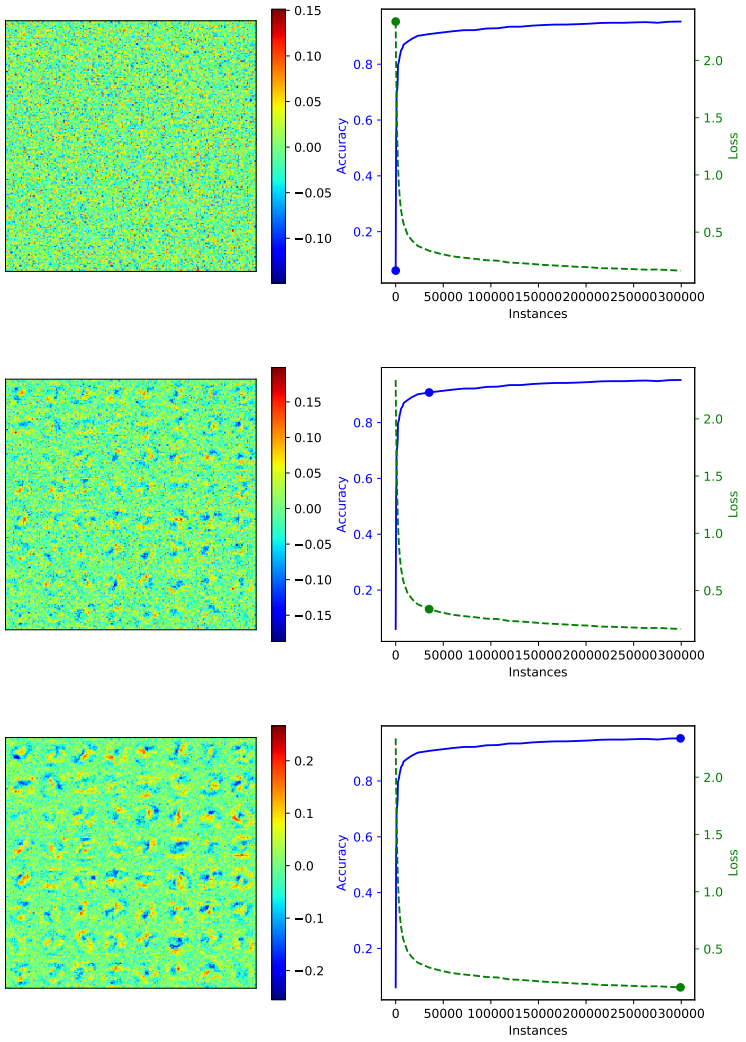
This new network can learn non-linear classification boundaries. We can visualize networks like this using diagrams like the one below:



This is a diagram for a multi-layer perceptron for MNIST with two hidden layers with 1000 dimensions (F1 and F2) followed by a final layer that transforms things into the ten-dimensional output space (F3). The squiggly lines represent non-linearities.

MNIST with MLP

This is what it looks like when we train a simple MLP on the MNIST dataset with one hidden layer that has 64 dimensions:

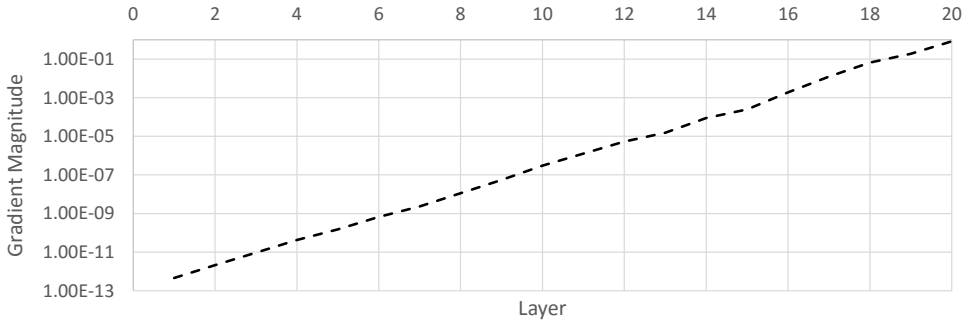


Note how the weights start out as noise, a common initialization strategy for multi-layer perceptrons, and then slowly begin to form structure that looks like bits and pieces of numbers. What is happening is that the network is finding commonalities between different digits like similar strokes or curves and sharing the expertise via the second layer. This new system gets significantly lower loss and a higher accuracy of around 0.95.

Vanishing Gradients

We started with a single perceptron, and then connected them in a single layer to create a multiperceptron. We then looked at how the same techniques can be used to train multiple layers of perceptrons in a multi-layer perceptron via stochastic gradient descent. In doing so we have now reached the state of the art in the 1990s, when research into neural nets entered into a second doldrums due to hardware limitations and the problem of the *vanishing gradient*.

For activation functions like we have seen so far, the value of the function and thus the derivative is between 0 and 1. Since all the derivatives of the non-linearities in a network are multiplied together during backpropagation, the gradient becomes smaller the farther backward it is propagated. For deep networks, it becomes so small that it causes convergence to slow down or stop.

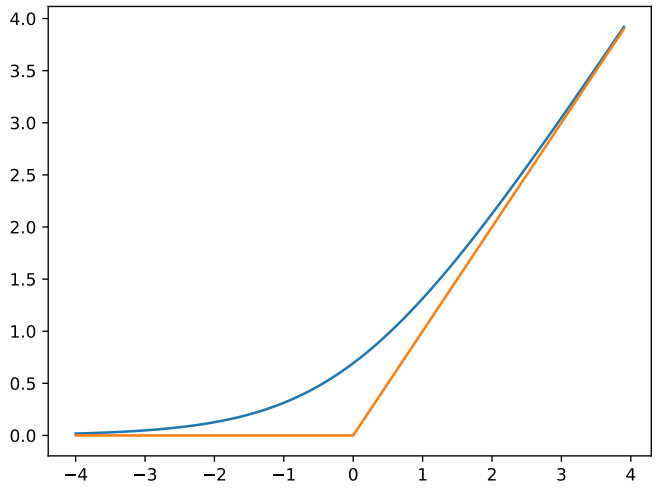


Rectified Linear Units

To overcome the problem of vanishing gradients, a new form of non-linearity was developed: the Rectified Linear Unit, or *ReLU* [1]. ReLUs have a deceptively simple form:

$$f(x_i) = \begin{cases} x_i & x_i > 0 \\ 0 & x_i \leq 0 \end{cases}$$

The ReLU function is an approximation of the softplus function $f(x_i) = \log(1 + e^{x_i})$, as can be seen below:



While at first the discontinuity in the function and the lack of a "squashing" effect would appear to be a problem, in practice it does not have any negative effect on convergence of the network. However, it does have the very desirable effect of entirely reducing the vanishing gradient effect. It does this so nicely (and is so computationally efficient as well) that it is used in the majority of modern deep network architectures. In the next lecture we will look at some of these architectures and a new kind of network operation: *convolution*.

References

[1] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the* 27^{th} *International Conference on Machine Learning*, Haifa, Israel, 2010.

[2] Paul Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences.* PhD thesis, Harvard University, 1974.