CS 181 Spring 2023 Section 4 Notes Solution

1 Neural Networks

1.1 Midterm Skills Checklist

1. Define node, width, activation, architecture
2. Translate from a neural network's graphical representation to analytic formula and vice-versa
3. State the Universal Approximation Theorem definition and explain its signifiance
4. Recast neural networks as regression/classification with a learned feature basis
5. Describe pros and cons of a learned basis
6. Describe effect of changing NN architecture parameters such as width and depth on model complexity
7. Prove neural network regression and classification are non-convex
8. Compute small backpropagation by hand
9. Compute small forward and backward automatic differentiation
10. Describe when it is more efficient to use forward or backward differentiation
11. Describe why NN optimization is difficult
12. Describe three main failure modes of gradient descent and common fixes
13. Explain main challenges of interpreting NNs
14. Describe a few methods of NN interpretation
15. See earlier midterm skills requirements regarding model evaluation, model comparison, and bias-variance trade off

1.2 Takeaways

The key advantage of neural networks is the ability to approximate very complex functions by composing a series of linear transformations with non-linear activation functions. Functionally, they consist of neurons, which are functions that apply an affine transformation using learned weights and non-linear **activation** function to an input vector and outputs a scalar.

In the graphical representation (Figure 1), a **node** is associated with a pre-activation and/or an activation value, and they are the inputs and outputs to neurons. A **layer** consists of all nodes at the same depth in the graphical representation, and the number of nodes in a layer is the **width**. The intermediate layers between the input and output are hidden layers. The **architecture** of a neural network is specified by the number, width, and connectivity of layers and the choice of activation functions at each layer.

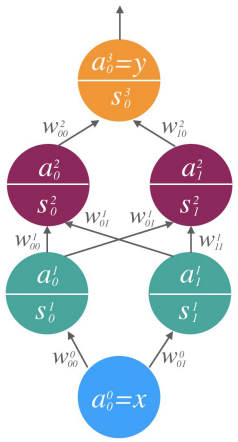


Figure 1: The computational graph for a small 4-layer neural network consists of a layer of input nodes (blue), two layers of hidden nodes (green, maroon), and an output node (orange). The width of the first hidden layer in this example is 2 because there are 2 green nodes. Notice that the information in an architecture specification is exactly the information needed to specifiy the computational graph.

The formal statement of the flexibility of neural networks is a **universal function approximation theorem**: "For any continuous function f defined on a bounded domain I, we can find a neural network that approximates f with an arbitrary degree of accuracy". This means that neural networks can express an extremely wide range of functions.

Neural networks can be thought of learned basis regressors and classifiers. In the case of regression, we can think of the first $L-1$ layers in an L-layer neural network as a basis transformation, and the outputs are linear combinations of these learned features. Likewise, for binary classification, the first $L-1$ layers transform inputs to a feature basis, and the output is a linear combination of these features that are squashed by the sigmoid function to output a probability.

1.2.1 Architecture Concept Question

Describe the effect of increasing the neural network depth and width on the complexity of functions that can be approximated by the neural network. What happens if the activation function is linear?

Solution

As depth increases, more complex functions can be approximated because of increasing non-linearity with each layer (however, there is a minimum width requirement). As width increases you can compose more non-linear functions together at each layer for a similar increase in complexity of functions that can be approximated.

Without non-linear activation functions (for simplicity, let the activation function just be the identity), the model is just a composition of linear transformations, which is still linear. To see this mathematically, suppose the weight matrices are W^0, W^1, W^2 and assume for simplicity there is no bias term. Then, the model $NN(x)$ outputs $NN(x) = I(W^2(I(W^1(I(W^0x))))) = W^2W^1W^0x = Wx$, where W is the matrix product of the three weight matrices and corresponds to a single linear transformation. That is, you get no increase in expressivity with 1 layer or 100 layers if you don't have a non-linear activation function!

End Solution

1.3 Activation Functions

There a wide range of possible non-linear activation functions to choose from when designing neural networks. Among these, the most popular are ReLU, which you will encounter below, and the tanh activation function, which is exactly what it sounds like: $\tanh(z)$. There are many others, including sigmoid and softmax for the final output layers, that researchers decide between when constructing their models.

1.4 Concept Question

What is the difference between the activation functions described here (ReLU, tanh, sigmoid, softmax) and the usage of sigmoid and softmax earlier in logistic regression?

Solution

The purpose of these activation functions is to be non-linear, while earlier the sigmoid and softmax functions were used to transform outputs into probabilities. Thus, sigmoid and softmax both necessarily have range $[0, 1]$, while for instance tanh has range $[-1, 1]$. For neural networks, having a range centered around 0, like tanh, is an advantage because it means there is no systemic bias being introduced to the output values, whereas functions like ReLU, sigmoid and softmax result in all non-negative output values which can limit the expressiveness of the neural network.

End Solution

1.5 Neural Network Optimization

Neural networks have very complex loss landscapes, but empirically, variants on gradient descent can successfully optimize these models to achieve very high (sometimes super-human!) performance on certain tasks. That is, we can update our model parameters $\mathbf{W}$ with some variation of $\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} L$, where L is a differentiable loss function of choice.

1.5.1 Backpropagation Algorithm

However, the complexity of neural networks from composing non-linear transformations leads to a more involved gradient calculation. The chain rule allows us to compute gradients of the loss with respect to the weights in each layer (assuming the non-linear transformations are differentiable at most values), and the **backpropagation algorithm** is an efficient way to compute the numerous partial derivatives efficiently in one forward pass and one backward pass. In the forward pass, the algorithm computes and stores the values at each node for a datapoint, and in the backward pass, the algorithm calculates the gradient of the loss function with respect to each weight.

1.5.2 Using the Chain Rule in Backpropagation

In single-variable calculus, you learned the chain rule:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(x)}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

Notice that the chain rule can be applied recursively! It doesn't matter how many functions of functions there are: we can always use the chain rule to compute a derivative for any component of the function.

This means that if we want to compute a gradient over all the weights in a function, all we need to do is apply the chain rule on each of those weights. If $\hat{y} = f(x)$, the partial derivative of y with respect to any weight can be computed this way, and if we're differentiating a loss function $\sum_{\mathbf{X},\mathbf{y}} (\hat{y} - y)^2$ with respect to a weight, we can just apply the chain rule to $\hat{y}$.

Example. Let's first look at this in small case, where both x and y are one-dimensional. Let $\hat{y} = f(x) = \sigma(v_1 \text{ReLU}(w_1 x))$. Say we want to differentiate this with regard to w_1. The chain rule gives us the following expression:

$$\frac{\partial f(x)}{\partial w_1} = (\frac{\partial \sigma(v_1 \text{ReLU}(w_1 x))}{\partial (v_1 \text{ReLU}(w_1 x))})(\frac{\partial v_1 (\text{ReLU}(w_1 x))}{\partial (\text{ReLU}(w_1 x))})(\frac{(\partial \text{ReLU}(w_1 x))}{\partial (w_1 x)})(\frac{\partial (w_1 x)}{\partial w_1})$$

This may look intimidating, but we actually just need to plug in a bunch of derivatives which we already know and multiply them. Starting from the left, $\frac{\partial (w_1 x)}{\partial x}$ is x since it is a linear function. When we reach the ReLU term, we consider the behavior of the input. When the input is less than zero, there is no change as the input changes, and if the input is greater than zero, ReLU is linear with the input. This means we have two cases:

$$\begin{cases} w_1 x < 0 : \frac{\partial f(x)}{\partial w_1} = 0 \\ w_1 x > 0 : \frac{(\partial \text{ReLU}(w_1 x))}{\partial (w_1 x)} = 1 \end{cases}$$

Notice that the first term is the entire derivative we are trying to compute - if one term in the product is zero, the whole product is zero. For the rest of this example, we're just computing the derivative in the non-zero case. This means that we can ignore all the ReLUs we see in the future, since if $wx > 0$ $\text{ReLU}(wx) = wx$. We then have another linear term, whose derivative is just vx, then we have the sigmoid term. Recall that $\sigma_x(x) = \sigma(x)(1 - \sigma(x))$: this derivative is then just $\sigma_x(x) = \sigma(v_1 w_1 x)(1 - \sigma(v_1 w_1 x))$. We can then multiply all of these terms together to find our

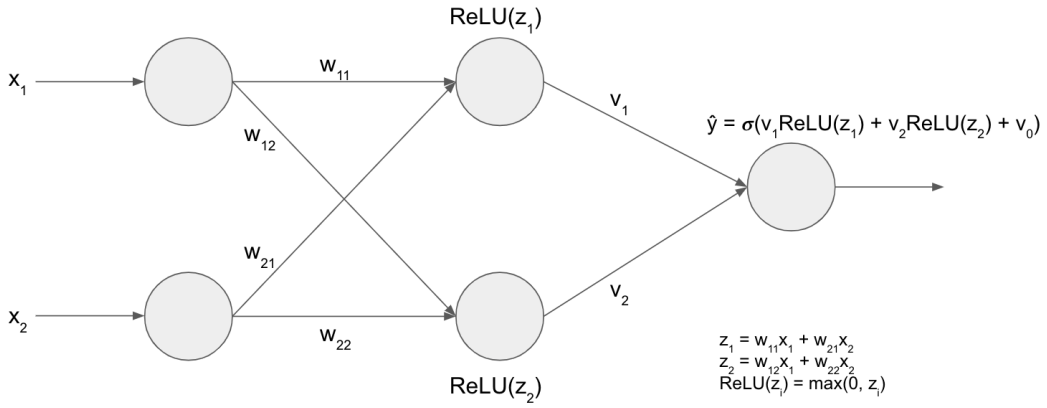
derivative of $f(x)$ with respect to w_1:

$$\frac{\partial f(x)}{\partial w_1} = \begin{cases} w_1 x < 0 : 0 \\ w_1 x > 0 : \sigma(v_1 w_1 x)(1 - \sigma(v_1 w_1 x)) * vx * 1 * x = \boxed{\sigma(v_1 w_1 x)(1 - \sigma(v_1 w_1 x))vx^2} \end{cases}$$

We now have computed a backpropogated derivative of a neural network by hand! Because derivatives are linearly separable, this is the same process you would use on a neural network with many $\mathbf{x}, y$ pairs and many more layers. The fundamental steps are the same: you just have to do them more times.

1.6 Exercise: Backpropagation on a Graphical Neural Network

Consider the following 2-layer neural network, which takes in $\mathbf{x} \in \mathbb{R}^2$ and has two ReLU hidden units and a final sigmoid activation. There are no bias weights on the hidden units.



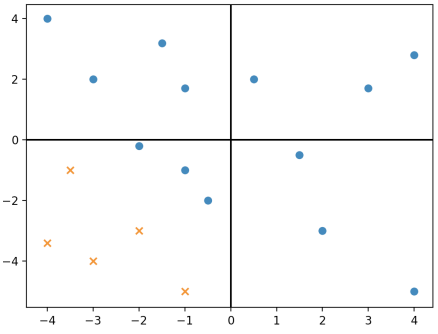
For a binary classification problem with true labels $y \in \{0, 1\}$, we will use the loss function $L = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$.

a. Write out the full function $f(\mathbf{x}) = \hat{y}$ in terms of the weights and non-linearities.

b. Suppose we update our network with stochastic gradient descent on a data point $\mathbf{x} = [x_1, x_2]^T$.
   i. Calculate the gradient of the loss with respect to v_1.
   ii. Calculate the gradient of the loss with respect to w_{11}.

   Note: Make sure to write your final answers in terms of components of $\mathbf{w}, \mathbf{v}$, and $\mathbf{x}$!

c. Consider the classification of data points below. Is it possible that this classification was generated by the set of weights $w_{11}, w_{12}, w_{21}, w_{22} = \{1, 0, 0, 1\}$? Why or why not? What if additional hidden layers were applied to further transform the data (still keeping the specified set of weights fixed)?



c. i. Why is it a bad idea in general to have ReLU as the activation function of the output layer?

   ii. Suppose we want to classify our outputs into 5 categories. Why might it be a bad idea to use the label set $\{1, 2, 3, 4, 5\}$? What could we use instead?

Solution

a.

$$f(\mathbf{x}) = \sigma(v_0 + v_1 \text{ReLU}(w_{11}x_1 + w_{21}x_2) + v_2 \text{ReLU}(w_{12}x_1 + w_{22}x_2))$$

b. i.

$$\begin{aligned}\frac{\partial L}{\partial v_1} &= -(y/\hat{y} + (y-1)/(1-\hat{y})) \cdot \hat{y}(1-\hat{y}) \cdot ReLU(w_{11}x_1 + w_{21}x_2) \\ &= -(y(1-\hat{y}) + (y-1)\hat{y}) \cdot ReLU(w_{11}x_1 + w_{21}x_2) \\ &= (\hat{y} - y) \cdot ReLU(w_{11}x_1 + w_{21}x_2)\end{aligned}$$

ii.

$$\begin{aligned}\frac{\partial L}{\partial w_{11}} &= -(y/\hat{y} + (y-1)/(1-\hat{y})) \cdot \hat{y}(1-\hat{y}) \cdot v_1 \cdot \frac{\partial ReLU(w_{11}x_1 + w_{21}x_2)}{\partial w_{11}} \\ &= -(y(1-\hat{y}) + (y-1)\hat{y}) \cdot v_1 \cdot x_1 \text{ if } (w_{11}x_1 + w_{21}x_2) > 0, \text{ 0 otherwise} \\ &= (\hat{y} - y) \cdot v_1 \cdot x_1 \text{ if } (w_{11}x_1 + w_{21}x_2) > 0, \text{ 0 otherwise}\end{aligned}$$

c. Regardless of whether there are additional hidden layers, this classification could not have been generated by the given weights. As described, all points in the bottom left quadrant would map to the origin, so it is not possible for points of differing predicted label to be in that quadrant.

c. i. If the values entering the ReLU layer are mostly negative, gradients will fail to back-propagate through the network.

ii. The numerical values carry unintended meaning; our model will assume that categories 1 and 2 are similar, whereas 1 and 5 are very distinct. We can fix the problem by using one-hot encoding.

End Solution

1.6.1 Automatic Differentiation

The backpropagation algorithm is a special case of automatic differentiation. The computational graph of most neural networks has an order in which computation proceeds from inputs and results in outputs. We call a node that is an input to a neuron a parent node, and the output node of the neuron the child node. Reverse-mode differentiation starts at the outputs of the network and computes the total derivative of each node with respect to its parents (i.e. values that are transformed into the node we are considering). Forward-mode differentiation starts at the inputs of the network and computes derivatives of each child node with respect to the input node.

1.6.2 Autodiff Concept check

How does the number of inputs and outputs of the neural network affect whether forward differentiation or backward differentiation is more efficient? (Hint: *What is the number of forward-mode differentiation passes needed to calculate the total derivative of the loss with respect to all the inputs? What about using reverse-mode?*)

Solution

If there are fewer outputs than inputs (e.g. image classification), reverse mode differentiation is more efficient because for each output, you can do a single reverse mode pass to get all the derivatives of that output with respect to every input.

Conversely, if there are more outputs than inputs, then you will need to do more reverse-mode passes than forward-mode passes because if you just do forward mode, you do one pass per input to get the derivatives of every output with respect to the input.

To illustrate this with an example, consider the computational graph (Figure 3, credit to Chris Olah's exceptional blog post at https://colah.github.io/posts/2015-08-Backprop/):

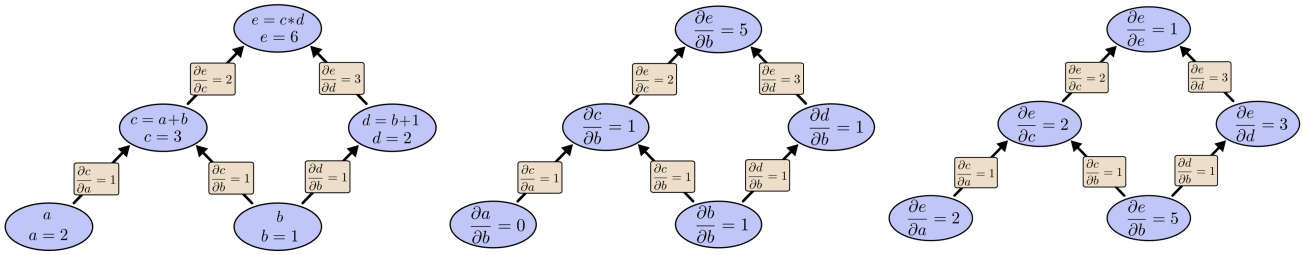


Figure 2: *Left:* The computational graph for a small 3-layer neural network. *Middle:* The calculation of the derivatives with respect to the input b in forward mode, which traverses the computational graph from each input to the output. To get the total derivative of the output, we would have to compute another forward pass for input a. *Right:* Reverse-mode autodifferentiation calculates all the necessary derivatives between the output and inputs in a single pass by starting at the output and progressing in reverse down the computational graph to every input. This reduces computation by a factor 2, but you can imagine with more inputs, the efficiency gain is enormous!

End Solution

1.7 Challenges of Optimization

The loss function of any neural network (with at least 1 hidden layer) that depends only on the output is non-convex!

1.7.1 Non-convexity concept check

One way to see that neural networks are not convex (besides visualizing the loss landscape) is by permuting the nodes and weights of a layer. That is, what happens to the loss and gradient if you swap the places of various nodes in a hidden layer?

Solution

Suppose you've found weights that make the gradient of the loss 0 and your model has achieved a local minimum loss L. Then, if you swap location of two nodes $h_1^{(1)}$ and $h_2^{(1)}$ in layer 1 and you also swap all the weights accordingly, your model will have the same loss L and still have 0 gradient. However, this is a different model with a different set of weights, so your model loss cannot achieve a unique global minimum and the loss is non-convex.

End Solution

This means that we lose nice theoretical guarantees like global optimality and we cannot use well-established, efficient solvers. Practically, we use gradient-based methods to search for a "good" local minimum (i.e. one that is close to the global minimum in loss). However, when the gradient of the loss is 0 in a non-convex problem, you could be at a local minimum or saddlepoint, and there are often regions of very high or very low curvature.

1.7.2 Gradient descent failure modes concept check

What went wrong in each of the figures below? What are possible fixes? What is a third failure mode?

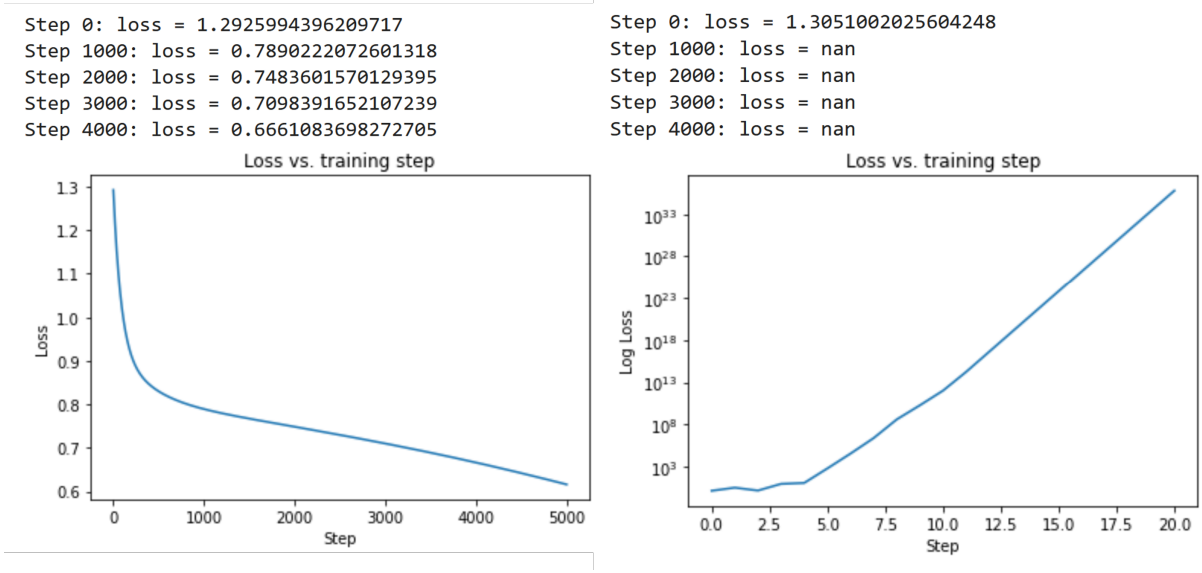


Figure 3: Two training loss curves. Note the y-axis scale!

Solution

For the left figure, we did not train until convergence. Some possible fixes include: 1. increasing learning rate 2. increasing training steps. For the right figure, the loss is diverging, which is often a sign that our learning rate is too large, so the step size is making making our updates overshoot in a very steep region of the loss landscape.

The third failure mode, which we usually cannot detect just by looking at a training loss trace, is being trapped at saddlepoints or local minima. One way to avoid this is stochastic gradient descent, which can jitter the optimizer out of local minima or saddlepoints. Another possibility is randomly initializing training runs from several different sets of weights, and choosing the model that converges to the lowest loss.

End Solution

1.7.3 Addressing the difficulties in optimization

There is a zoo of tricks that researchers have empirically found effective to make training easier. To deal with local minima and saddlepoints, we can add stochasticity and/or momentum terms to the descent steps (see SGD, the Adam optimizer). To find better optima, we can initialize weights randomly in multiple runs. To achieve better convergence, we can tune the learning rate in various ways such as a decay schedule (start with bigger steps and reduce step size as training progresses).

1.8 Interpretability

We often describe NNs as "black-box" models that we have very limited ability to see inside or understand. This makes it difficult to trust models in a variety of ways. A non-exhaustive list is:

1. Can we trust that the model can perform well in deployment?
2. Can we trust a model to be robust to distributional shifts (e.g. inputs that are systematically different than those in training)?
3. Can we trust the model made its decision for the "right" reason or based on undesirable correlations?
4. Is the model informative to human decision makers?
5. Can models capture causality or just pick up correlations in scientific tasks?

In response, ML researchers look for ways to "interpret" or "explain" what a model does under the hood. This umbrella term can range from understanding which parts of inputs were most pertinent to a model's prediction to decomposing a model into intuitively understandable functional parts for a human. Some big ideas include:

1. Explaining predictions by feature importance attribution
2. Built-in interpretability by designing models that we can easily point to various parts of and understand
3. Understanding where decision boundaries lie in classification models
4. Perturbing models to understand when performance breaks down
5. Reverse-engineering the algorithms a deep learning model implements to mechanistically interpret the model

This area of research is very active and an excellent way to make deep learning and machine learning algorithms safer for society.

1.9 Exercise: A Simple NN Classifier

Let's think about a neural network binary classifier with $\mathbf{x} \in \mathbb{R}^2$ ($D = 2$) and with a single two-dimensional hidden layer ($M = 2$) followed by the one-dimensional output layer.
For the non-linear activation function, we use the *ReLU* function defined by the following:

$$\text{ReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Consider a function h defined by the following:

$$h(\mathbf{x}) = \sigma(\mathbf{w}^\mathsf{T}\phi(\mathbf{x}) + w_0) \tag{2}$$

We can decompose h in terms of our neural network classifier's weights where the first (hidden) layer composes the "adaptive basis" and the second (output) layer composes the logistic regression:

$$h(\mathbf{x}) = \sigma(\mathbf{w}^\mathsf{T}\mathbf{ReLU}\left(\mathbf{W}^{(hid)\mathsf{T}}\mathbf{x} + \mathbf{w}_0^{(hid)}\right) + w_0) \tag{3}$$

$$= \sigma\left(\sum_{m=1}^{M} w_{1m}^{(2)}\mathbf{ReLU}\left(\sum_{d=1}^{D} w_{md}^{(1)}x_d + w_{m0}^{(1)}\right) + w_{10}^{(2)}\right), \tag{4}$$

where $\sigma(z)$ is the sigmoid function and in the last equation above, we explicitly separately notate the weights of the output layer

$$\mathbf{w} = \begin{pmatrix} w_{11}^{(2)} \\ w_{12}^{(2)} \end{pmatrix} \in \mathbb{R}^2, w_0 = w_{10}^{(2)} \in \mathbb{R}$$

and of the hidden layer

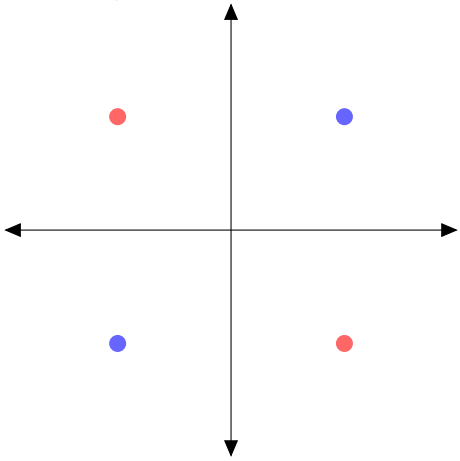
$$\mathbf{W}^{(hid)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} \end{pmatrix} \in \mathbb{R}^{2\times 2}, \mathbf{w}_0^{(hid)} = \begin{pmatrix} w_{10}^{(1)} \\ w_{20}^{(1)} \end{pmatrix} \in \mathbb{R}^{2\times 2}.$$

Then, classifications are made according to $\mathbb{I}_{h(\mathbf{x})>.5}$. Suppose we want to fit the following data:

$$\mathbf{x}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \; y_1 = 1, \quad \mathbf{x}_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \; y_2 = 0$$

$$\mathbf{x}_3 = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \; y_3 = 0, \quad \mathbf{x}_4 = \begin{pmatrix} -1 \\ -1 \end{pmatrix} \; y_4 = 1$$

This looks as follows (blue = 1, red = 0):



1. Why can't we solve this problem with a linear classifier?
2. What values of parameters $\mathbf{W}^{(hid)}, \mathbf{w}_0^{(hid)}, \mathbf{w}$, and w_0 will allow the neural network to solve the problem? Show that your choice of parameters allows the network to correctly classify the data.
   Hint: Think carefully about what the ReLU activation function can do for us. What does it do to various kinds of vectors? Think geometrically.
3. What sort of basis transformation would a logistic regression require to be able to classify this data?
   Hint: How many feature dimensions are there?

Solution

We can solve the problem with

$$\mathbf{W}^{(hid)} = \begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix}, \quad \mathbf{w}_0^{(hid)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\mathbf{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad w_0 = -1$$

How does this solve the problem? Note that

$$h(\mathbf{x}_1) = \begin{pmatrix} 1 & 1 \end{pmatrix} \mathbf{ReLU} \begin{pmatrix} 2 \\ -2 \end{pmatrix} - 1 = 1 \Rightarrow \mathbb{I}_{h(\mathbf{x}_1)>.5} = 1$$

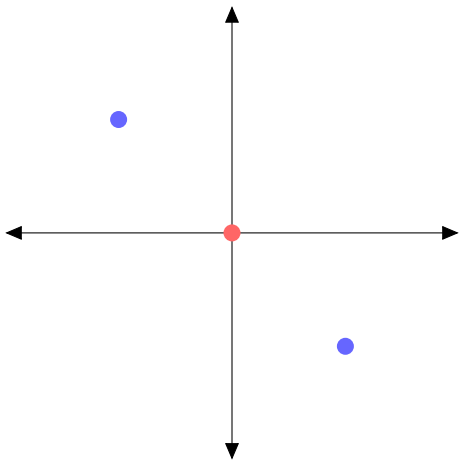
$$h(\mathbf{x}_2) = \begin{pmatrix} 1 & 1 \end{pmatrix} \mathbf{ReLU} \begin{pmatrix} 0 \\ 0 \end{pmatrix} - 1 = -1 \Rightarrow \mathbb{I}_{h(\mathbf{x}_2)>.5} = 0$$

$$h(\mathbf{x}_3) = \begin{pmatrix} 1 & 1 \end{pmatrix} \mathbf{ReLU} \begin{pmatrix} 0 \\ 0 \end{pmatrix} - 1 = -1 \Rightarrow \mathbb{I}_{h(\mathbf{x}_3)>.5} = 0$$

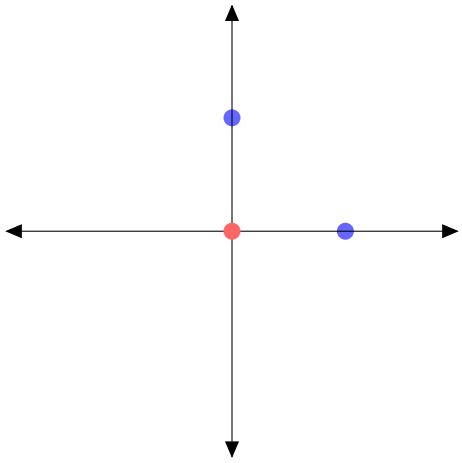
$$h(\mathbf{x}_4) = \begin{pmatrix} 1 & 1 \end{pmatrix} \mathbf{ReLU} \begin{pmatrix} -2 \\ 2 \end{pmatrix} - 1 = 1 \Rightarrow \mathbb{I}_{h(\mathbf{x}_4)>.5} = 1$$

This works because we have a ReLU! Having a nonlinearity in the activation function allows us to find a linear classifier for these examples in the new basis.

What did we do? With $\mathbf{W}^{(hid)}$, we map our data linearly so it looks as follows:



This is great, but still not linearly separable. Applying the ReLU, however, maps this picture to the following one:



Now our data is linearly separable and we can classify the points correctly. Note that the parameters we chose here are not unique (far from it). The important thing is that we transformed our input space well enough to apply a linear classifier.

End Solution